Data Extraction from Text Using Wild Card Queries

Haobin Li         Davood Rafiei
Computing Science Department
University of Alberta
{haobin,rafiei}@cs.ualberta.ca

Abstract

We propose a domain-independent framework for extracting facts and relations from natural language text sources, where an extraction task is expressed in a query combining text fragments with some wild cards, and the data that best match the query are extracted. A significance of our querying mechanism is that, despite being simple, it can extract a much wider range of facts than previous techniques; furthermore, our queries can be seamlessly integrated in a relational query language, providing a relational view of text. A given query, however, can be too restrictive and may not extract enough matches. Unlike term queries which can be relaxed by removing some of the terms (as is done in search engines), removing terms from a wild card query is not easy and can ruin its meaning. To address this problem, we propose a rewriting rule language to express alternative rewritings of a given query. Also, to distinguish real facts from false matches, we propose a ranking algorithm that assigns higher weights to promising instances based on the weights of other instances. We develop a prototype that implements our algorithms and conduct experiments comparing the results of our framework with alternative approaches. The experiments show that our approach outperforms an alternative more specific system and a well-known statistics-based method in terms of both precision and recall.

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 phrase “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 focus on a more specific task (e.g. finding course offerings and job postings) and impose a tight restriction on the type of data that can be extracted. Others are only applicable to documents that follow a specific formatting (e.g. wrappers). For example, the KnowItAll [11] 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 “coo of” Y. If a user wants to extract something other than hyponyms and those predefined relations, the system is no longer applicable. A challenge is that the extraction task often is not easy to define or a definition may not be accurate, leading to low precision and recall.

In this paper, 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 with some wild cards, and the result of a query is a ranked list of tuples that match the query. For instance, given the query “% is a car manufacturer”, the output is expected to

---

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006

1 Our query phrases in this paper are in English, but our framework should be applicable to other languages as well.
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. Since the result of a query is a relation, our queries with some syntactic sugar can be integrated in the from clause of SQL 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 Section 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 the relevance of extracted tuples to a query. Our algorithm exploits various semantic relationships that exist between the set of matching tuples of a pattern and also between the set of patterns that extract each tuple. We use the general term pattern to refer to both query and query rewriting. When 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.

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 paper is organized as follows. Section 2 describes both the syntax and the semantics of wild cards, as well as the queries in our framework. An overview of our query evaluation in the context of the Web is given in Section 3. Section 4 discusses the details of our rewriting rules and patterns. Our ranking algorithms are discussed in Section 5. Experimental results are presented in Section 6 and the related work is reviewed in Section 7. We end the paper with conclusions and future work in Section 8.

2 Query-based Interface and Wild Cards

To express extraction tasks concisely and in a flexible manner, we make use of wild cards in our queries. 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. Unlike many of these systems, our introduced wild cards iterate over the domains of parts of speech or other meaningful groupings of natural language words. 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, verbs and 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
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. We assume that the result of a query is ranked and each row is assigned a score as an indication of the level of support the row receives. This score may depend on the size and the coverage of the text collection that is being queried and the set of query rewritings that are being used. Section 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 \( q_1, \ldots, q_k \) 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 \( q_1, \ldots, q_k \); the score of the row for \( q \) is an aggregation of the scores of the row for \( q_1, \ldots, q_k \). 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 [16], etc. Next we discuss how these queries can be evaluated.

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., [8, 6]) 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 our goal in this paper is not to optimize 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 augmented with their similar terms. The word “country” is enclosed by “%”s in the given query. An online system that gives related terms [16] returns “nation” as a related term 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.

In the next step, each query in the query set is passed to a Part-Of-Speech (POS) tagger. 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” where NP stands for a noun phrase; 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 %”. Section 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. 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 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 gives the details of our ranking algorithm. Finally, a sorted list of rows is returned.

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 retrieve very few
or no matches. 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 for-
mats. 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 Section 6 confirm that there is a corre-
lation between the number of rewritings and the pre-
cision of the retrieved results. Also, using multiple re-
lated queries can increase the recall, as it can be seen
from our example in Section 3. 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 → rewrite-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 ex-
panded with all rewritings in the rule body. For each
match, keywords from the query may be remembered
using capturing groups (e.g. parentheses) and the re-
membered values may be recalled using back references
in the rule body. The remembered values can be trans-
formed (e.g. from a singular noun to a plural noun)
before being used in the rewritings. This allows us to
write generic rewritings that can match a large num-
ber 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.
Given the query “countries such as ‘%’, the rule gen-
erates “%, and other countries” and “% is a country”
as possible rewritings.

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

Let $n$ denote the average number of rewritings pro-
duced for each query. If a query uses $k$ star wild cards
and each of these wild cards is replaced with $m$ similar
terms on average, the query expansion would produce
$k \times m$ queries. These queries are expected to be closely
related and 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 preci-
sion 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 vari-
ants. 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 [13]. A sample of hyponym
patterns (some from Hearst’s and some hand-crafted
by us) can be found in Table 1.

<table>
<thead>
<tr>
<th>Table 1: Hyponym patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPI {} “such as” NPI2List</td>
</tr>
<tr>
<td>“such” NPI “as” NPI2List</td>
</tr>
<tr>
<td>NPI {} “especially” NPI2List</td>
</tr>
<tr>
<td>NPI {} “including” NPI2List</td>
</tr>
<tr>
<td>NPI2List “and other” NPI1</td>
</tr>
<tr>
<td>NPI2List “or other” NPI1</td>
</tr>
<tr>
<td>NPI2 “a(n)” NPI1</td>
</tr>
<tr>
<td>NPI2 “is a(n)” NPI1</td>
</tr>
<tr>
<td>NPI1 NPI2</td>
</tr>
</tbody>
</table>

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”; a rewriting of the query is “the light bulb was
invented by %”. 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 2 with the objects
and the verbs of the patterns instantiated to “the light
bulb” and “invent”, respectively.

<table>
<thead>
<tr>
<th>Table 2: Morphological patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPI1 invent the light bulb</td>
</tr>
<tr>
<td>NPI1 invents the light bulb</td>
</tr>
<tr>
<td>NPI1 invented the light bulb</td>
</tr>
<tr>
<td>the light bulb is invented by NPI1</td>
</tr>
<tr>
<td>the light bulb was invented by NPI1</td>
</tr>
<tr>
<td>the light bulb, invented by NPI1</td>
</tr>
</tbody>
</table>

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 ap-
plied to a wide range of queries, it is not hard to find
queries where no generic pattern is applicable or suf-
ficient. 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 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 are work on generating specific patterns automatically [17, 19], and the results are very encouraging. For instance, Lin and Pantel [17] automatically compile a list of over 182,000 classes of related patterns. This collection should be used with some care though 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.

Table 3: Specific patterns for discoverers and their discoveries

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP2 “discovered by” NP1</td>
<td></td>
</tr>
<tr>
<td>NP1 “find” NP2</td>
<td></td>
</tr>
<tr>
<td>NP2 “found by” NP1</td>
<td></td>
</tr>
<tr>
<td>NP1 “uncover” NP2</td>
<td></td>
</tr>
<tr>
<td>NP1 “unearth” NP2</td>
<td></td>
</tr>
<tr>
<td>NP2List “stumble upon” NP1</td>
<td></td>
</tr>
<tr>
<td>NP1 “announce the discovery of” NP2</td>
<td></td>
</tr>
<tr>
<td>NP1 “reveal” NP2</td>
<td></td>
</tr>
</tbody>
</table>

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 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 Min-
ister”. 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.

5 Bringing Order to Results

The data extraction process discussed in this paper 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 [11]. 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 mis-classified as noun phrases, or vice versa.

Since correct extractions are inter-mixed 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 arise in any question answering system. The precision of an answer usually depend 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 [14] 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 pattern² 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. On the Web, we do not usually have access to all pages, and the numbers could be approximated using a search engine collection or a local crawl. There are however some problems with this heuristic. First, all 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. On the Web, in particular, there are 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 document contains the query string \( q \) with \( P(q) \), the probability that a document contains a row \( r \) with \( P(r) \), and the probability that a document 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

\[
MI(q,r) = \log \frac{P(q,r)}{P(q)P(r)}
\]

In some formulations of the mutual information, the above formula is multiplied by \( P(q,r) \) [7]. This measure is used in the past, for instance to evaluate the association between words [9], and also between the instances of a class and a discriminative phrase [11]. In our case, since \( P(q) \) is fixed for a given query, the score of a tuple can be estimated as the ratio \( P(q,r)/P(r) \).

On the Web, 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 fields of the tuple appears in a document. 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. Selecting a single pattern is not guaranteed to achieve a high recall. Also it is not clear how MI can be extended to account for multiple rewritings of a query. In the context of the Web, obtaining hit counts can be costly and may not be reliable (e.g. see [2]).

²The query would not have been issued in the first place if it is assumed otherwise.

**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 subsection presents an adaptation of the HITS algorithm for ranking. Note that the HITS algorithm, 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 Relations 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 1.

![Figure 1: Mutually reinforcing relationship between patterns and tuples](image_url)

An Iterative Algorithm: Let’s associate weight \( w_P(t) \) to each tuple \( t \), and weight \( w_P(p) \) to each pattern \( p \). In an iterative and alternating fashion, the weights can be updated as follows:

\[
w_P(t) = \sum_{\{p|p \text{ extracts } t\}} w_P(p) \quad (1)
\]

\[
w_P(p) = \sum_{\{t|t \text{ is extracted by } p\}} w_T(t) \quad (2)
\]
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.

\[
\text{Iterate}(S_T, S_P, k)
\]

\[
S_T: \text{the set of extracted tuples, } |S_T| = m
\]

\[
S_P: \text{the set of extraction patterns, } |S_P| = n
\]

\[
k: \text{a positive integer}
\]

\[
z^m = (1, 1, 1, \ldots, 1) \in \mathbb{R}^m
\]

\[
z^n = (1, 1, \ldots, 1) \in \mathbb{R}^n
\]

Set \( T_0 := z^m \)

\[
\text{Set } P_0 := z^n
\]

For \( j = 1, 2, \ldots, k \)

1. Apply Eq. 1 to \( P_{j-1} \), obtaining new \( T_j \).
2. Apply Eq. 2 to \( T_{j-1} \), obtaining new \( P_j \).
3. Normalize \( T_j \) so that the length of the vector is 1.
4. Normalize \( P_j \) so that the length of the vector is 1.

End

Return \((T_k, P_k)\)

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

**Theorem 2** 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 \( A^T \) be the transpose matrix of \( A \). \( T^* \) and \( P^* \) are the principal eigenvectors of \( AA^T \) and \( A^TA \) respectively.

The proofs of both theorems can be found elsewhere [14]. 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.

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.\(^3\) 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.

DeWild takes advantage of existing commercial search engines and queries Google and Yahoo (when Google does not respond) via their APIs. 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.\(^4\) 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. As its native ranking, DeWild uses the reinforcing relationships between patterns and tuples, as discussed in Section 5. For comparison purposes, we also implement the other heuristics discussed in the same section.

To find matches for \( * \) wild cards, a publicly available POS tagger called NLProcessor\(^5\) is used to identify the part of speech from the text, so that only noun phrases are extracted. For \( * \) wild cards, our system uses a collection of related words automatically compiled [16] from Wall Street Journal corpus, but it can equally use other collections as well.

Next we report our experiments with DeWild.

6.1 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 [11].

In our first experiment, we used a list of 192 country names, compiled by the US State Department,\(^6\) 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 Sec. 5.2. As is shown in Figure 2-a, DeWild outperforms both heuristics at almost all recall rates. Table 4-a 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 2-b compares

\(^3\)DeWild stands for Data Extraction using Wild cards. The system is available online at devild.ca.ualberta.ca.

\(^4\)Our online demo downloads at most 30 snippets for each query and each rewriting to keep the response time short.

\(^5\)www.infogistics.com/textanalysis.html

\(^6\)www.state.gov/vwwregions/independent_states.html
DeWild to the online system KnowItAll7 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 original 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. 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 4-b shows the extraction patterns which were used after instantiating “US states” in our generic patterns, as well as their weights computed by DeWild. 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 3 shows 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 values less than 0.75. Even for higher recall values, 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 3-b. Both MI-2 and MI-3 perform poorly in terms of precision for all recall values. MI-1 performs good for higher recall values but not so good for smaller recalls, meaning that many incorrect instances show up at the top of the list.

6.2 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 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 4-b and ran DeWild each time with only one of these sets. The precision-recall curve in each case is shown in Figure 4. 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.

6.3 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 [20]; since a QA target consisted of multiple questions, we ended up with a total of 22 questions in the experiment. For each question, we report the number of correct answers given by TREC, the number of answers 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 5.

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 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

---

7 www.cs.washington.edu/research/knowitall
Figure 2: Precision and recall for (a) DeWild, NPages and NPatterns and (b) DeWild, MI and KnowItAll. In both cases, the extraction target is the country names.

Figure 3: Precision and recall for (a) DeWild, NPages and NPatterns and (b) DeWild, MI and KnowItAll when the extraction target is the US states.

Table 4: Patterns that are used to extract country and the US state names, with the weights computed by DeWild in each case.

(a) | Pattern | Weight | (b) | Pattern | Weight |
--- | --- | --- | --- | --- | --- |
| such countries as % | 0.645229 | US states, including % | 0.739794 |
| countries such as % | 0.580203 | US states such as % | 0.526682 |
| countries, including % | 0.434738 | % and other US states | 0.320306 |
| % and other countries | 0.158705 | such US states as % | 0.227648 |
| countries, especially % | 0.127139 | US states, especially % | 0.113638 |
| % is a country | 0.122413 | % or other US states | 0.074993 |
| % or other countries | 0.088431 | % is a US state | 0.046522 |
| %, a country | 0.010305 | US states % | 0.013729 |
| countries % | 0 | %, a US state | 0 |
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 Doets” 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.

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

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 X%” and passed it as a query to DeWild. The top 3 answers returned by DeWild are shown in Table 6. 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 12th-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 Y%” before it was tried. Table 7 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 7, and the remaining one corresponded to the 8th row in the Table. Note that the 5th row in Table 7 is also correct since it is a spelling variation of the first row.

Table 6: Top candidate answers for the question “Who discovered prions?”

<table>
<thead>
<tr>
<th>Result</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanley Prusiner</td>
<td>0.307819</td>
</tr>
<tr>
<td>Scientists</td>
<td>0.408304</td>
</tr>
<tr>
<td>University of San Francisco</td>
<td>0.295243</td>
</tr>
</tbody>
</table>

Table 7: Top candidate answers for the question “Who are the members of the Rat Pack?”

<table>
<thead>
<tr>
<th>Result</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sammy Davis Jr.</td>
<td>0.454338</td>
</tr>
<tr>
<td>Frank Sinatra</td>
<td>0.454338</td>
</tr>
<tr>
<td>Peter Lawford</td>
<td>0.27854</td>
</tr>
<tr>
<td>Joey Bishop</td>
<td>0.252512</td>
</tr>
<tr>
<td>Sammy Davis Jr.</td>
<td>0.252512</td>
</tr>
<tr>
<td>Michelle Griffin</td>
<td>0.252512</td>
</tr>
<tr>
<td>Sammy Davis Jr. playing pool</td>
<td>0.252512</td>
</tr>
<tr>
<td>Dean Martin</td>
<td>0.252512</td>
</tr>
</tbody>
</table>

6.4 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 public domain. The query used for this task was “X% is a senior research scientist at Google”. Due to space limitation, only the top 10 instances and their weights are presented in Table 8.

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 “X% 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 ex-
tra related terms: movie and film. The top 10 results are given in Table 9.

We manually evaluated the extracted results using the Internet Movie Database (IMDB) and concluded that all the results shown in Table 9 were indeed correct movie names, and their release dates were in the summer.

### Table 9: A List of summer movies

<table>
<thead>
<tr>
<th>Result</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Wars</td>
<td>0.279295</td>
</tr>
<tr>
<td>Shrek 2</td>
<td>0.247855</td>
</tr>
<tr>
<td>Spider-Man 2</td>
<td>0.225779</td>
</tr>
<tr>
<td>Harry Potter</td>
<td>0.204052</td>
</tr>
<tr>
<td>Spiderman</td>
<td>0.205189</td>
</tr>
<tr>
<td>Men in Black</td>
<td>0.202629</td>
</tr>
<tr>
<td>Pearl Harbor</td>
<td>0.177852</td>
</tr>
<tr>
<td>Mission Impossible</td>
<td>0.171008</td>
</tr>
<tr>
<td>Van Helsing</td>
<td>0.171008</td>
</tr>
<tr>
<td>Independence Day</td>
<td>0.165511</td>
</tr>
</tbody>
</table>

In one more experiment, we used the query “% is a Canadian writer” to compile a list of Canadian writers. This time, we put together a set of rewritings that were specific to the query. 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 author names which could not be found in one list\(^8\) and 70 names which were not in the other list\(^9\). After combining the two lists, DeWild still reported 58 names which we couldn’t find in the combined list. This experiment shows that our queries can be used to compile a reasonably good list of resources which can be further edited for correctness.

### 7 Related Work

There is a large body of work on question answering. Many systems use a combination of NLP techniques (deep or shallow), learning algorithms and hand-crafted rules to classify the questions and to establish relationships between terms of a question and a possible answer sentence (e.g. [15, 10, 18]). Despite some overlap, there are fundamental differences between our work and the work on question answering. The size of the target set for question answering systems is typically one or only a few, whereas our goal is to use DeWild queries for large-scale data extractions. It is possible to integrate our work within a question answering system if natural language questions can be mapped to DeWild queries.

Large-scale data extraction from the Web has been the subject of various recent work. In particular, Brin [5] suggests an algorithm which takes a small number of examples of a class as a seed set and extracts more examples of the same class. His algorithm learns a set of extraction patterns for each page (or pages with the same URL prefix) that contains some of the examples and use these patterns to extract more tuples from the same page(s). This algorithm does a good job when data is structured in a tabular format but is not expected to work on free text. This is because it is generally unlikely to find more than one example (of the seed set) in a text document such that their surrounding texts are the same. Given a small set of examples, the semantics of the query sometimes is not also clearly defined.

KnowItAll [11] 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 rules which can be instantiated with an input class to produce keywords that must be used to extract the instances of the class. KnowItAll uses co-occurrence statistics, specifically mutual information, to assess the relatedness of each instance. Our approach differs from KnowItAll in several important aspects: First, the query-based interface and the support of wild cards make DeWild more adaptive to different extraction tasks. Second, unlike KnowItAll where a concise description of a class must be given, a DeWild 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 for 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.

Related to our query rewritings is the work on query expansion [4, 3] and query transformation for question answering [1]. Query expansion has shown to be difficult for phrase queries. Also query expansion and transformation techniques are not directly applicable to queries with wild cards. Our queries can benefit

\(^{8}\)www.track0.com/ojvc/authors
\(^{9}\)www.umanitoba.ca/canlit/authorlist
from inverted indexes on terms, phrases, and N-grams and there are already some works in these areas (e.g. [6,8]). Other related work includes the work on Web query languages and wrappers (see [12] for a survey of this area before 1998). These works can be used to extract data from a specific site or a set of pages with similar structures but are not generally applicable to free text. Finally, Google’s fill-in-the-blank is related to our wild cards but is different. Google returns a ranked listed of pages for a fill-in-the-blank search but the ranking is different (and the detail is not published).

8 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.

Our work leads to a few interesting directions. One issue is extractin g 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.

References