Querying Capability Modeling and Construction of Deep Web Sources

Liangcai Shu[1], Weiyi Meng[1], Hai He[1], and Clement Yu[2]

[1] Department of Computer Science, SUNY at Binghamton,
Binghamton, NY 13902, U.S.A
{lshu, meng, haihe}@cs.binghamton.edu
[2] Department of Computer Science, University of Illinois at Chicago,
Chicago, IL 60607, U.S.A
yu@cs.uic.edu

Abstract. Information in a deep Web source can be accessed through queries submitted on its query interface. Many Web applications need to interact with the query interfaces of deep Web sources such as deep Web crawling and comparison-shopping. Analyzing the querying capability of a query interface is critical in supporting such interactions automatically and effectively. In this paper, we propose a querying capability model based on the concept of atomic query which is a valid query with a minimal attribute set. We also provide an approach to construct the querying capability model automatically by identifying atomic queries for any given query interface. Our experimental results show that the accuracy of our algorithm is good.

Keywords: Deep Web, query interface, querying capability modeling.

1 Introduction

It is known that public information in the deep Web is 400 to 550 times larger than the so-called surface Web [1]. A large portion of the deep Web consists of structured data stored in database systems accessible via complex query interfaces [3] (also called search interfaces). Many Web applications such as comparison-shopping and deep Web crawling [2] require interaction with these form-based query interfaces by program. However, automatic interaction with complex query interfaces is challenging because of the diversity and heterogeneity of deep Web sources and their query interfaces. Automatic interaction includes automatically identifying search forms, submitting queries, and receiving and processing result pages. Some related work that has been carried out by different researchers includes: source extraction and description [4, 8] and deep Web crawling [2, 7, 11, 12].

The focus of this paper is on the automatic analysis of the *querying capability* of complex query interfaces, i.e., we want to find out what kinds of queries are *valid* (acceptable) by any given interface. A typical complex query interface consists of a series of attributes, each of which has one or more control elements like textbox, selection list, radio button and checkbox. A query is a combination of pairs, each consisting of an attribute and its assigned value. Those combinations that are accepted by the query interface syntax are *valid queries*; others are *invalid queries*.

B. Benatallah et al. (Eds.): WISE 2007, LNCS 4831, pp. 13–25, 2007.

Determining whether a query is valid is an important aspect of querying capability analysis. Consider the query interface in Fig. 1. Query {<*Author*, "John Smith">} is valid but query {<*Published date*, (2000, 2006)>} is invalid because this interface requires that at least one of the attributes with "*" must have values. The query with both conditions {<*Author*, "John Smith">, <*Published date*, (2000, 2006)>} is valid.

In this paper, we define a *query* as a set of attributes that appear in the query conditions. Traditional query conditions that include both attributes and values are defined as *query instances*. We propose the concept of *atomic query* to help describe querying capability. An atomic query is a set of attributes that represents a minimum valid query, i.e., any proper subset of an atomic query is an invalid query. For the query interface in Fig. 1, query {*Author*} is an atomic query because any query formed by filling out a value in the *Author* textbox is accepted by the search engine (note that a valid query does not guarantee any results will be returned). We can determine if a given query is valid if we identify all atomic queries for a query interface in advance. We propose a method to identify atomic queries automatically.

The work that is most closely related to our work is [8]. In [8], source descriptions are discussed. The focuses are on what kind of data is available from a source and how to map queries from the global schema to each local schema. In contrast, we are interested in describing the full querying capability of a query interface (i.e., what queries are valid) and constructing the querying capability model automatically. Both the issues studied and the solutions provided in these two works are different.

Fig. 1. Query interface of abebooks.com

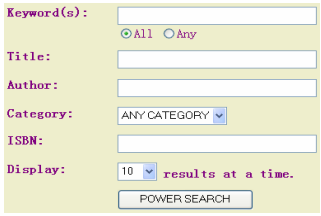

Fig. 2. Query interface of aaronbooks.com

The paper makes the following contributions:

(1). Make a classification of attributes. The attributes are classified into four types: functional attribute, range attribute, categorical attribute and value-infinite attribute.

(2). Propose the concept of atomic query (AQ) and a querying capability model.

(3). Present an algorithm to construct the AQ set that represents querying capability for a given query interface.

(4). Compare different classifiers' performance on result page classification.

The rest of this paper is organized as follows. Section 2 introduces attribute types. Section 3 presents a querying capability model of query interfaces and introduces the concept of atomic query. Section 4 discusses constructing querying capability of a query interface. Section 5 reports experimental results. Section 6 concludes the paper.

2 Attribute Types

We first classify attributes on query interfaces. Different types of attributes may have different functions and impacts on query cost and need to be coped with differently.

On a query interface, an attribute usually consists of one or more control elements (*textbox, selection list, radio button* and *checkbox*) and their labels (in this paper we refer to an attribute by its primary label), and it has a specific semantic meaning. For example, in Fig. 1, attribute *Author* has one textbox element, whereas attribute *Price* has two textbox elements. Many attributes on a query interface are directly associated with the data fields in the underlying database.

There are also *constraints* on a query interface. Constraints are not directly associated with data in the underlying database but they pose restrictions to the attributes involved. For example, *All/Any* in Figure 2 poses restrictions to *Keyword(s)*.

Fig. 3. Query interface of autotrader.com

Fig. 4. Query interface of orbitz.com

Definition 2.1. An attribute is a *functional attribute* if its values, when used in a query, affect only how query results are displayed but not what results are retrieved.

For a functional attribute, different values assigned to it yield the same set of search results, assuming the underlying database is unchanged. Functional attribute *Sort results by* (Fig. 1) determines the order of results while *Results per page* (Fig. 1) and *Display* (Fig. 2) decide how many result records are displayed per page.

From our observation, for a given domain, the total number of functional attributes is a small constant if attributes with the same function but different names are considered to be the same (e.g., *Results per page* in Figure 1 and *Display* in Figure 2). Attribute matching techniques (e.g., [6]) can be used to identify such attributes.

Definition 2.2. If any value assignment of an attribute implies a range of values, the attribute is a *range attribute*.

A range attribute could have one or two control elements. For example, *Published date (min, max)* and *Price (min, max)* in Fig. 1 have two elements. Generally, if an element of a range attribute has value *Null* (no value) or *All*, it means no restriction on this attribute, i.e., there is no restriction on the returned results from this attribute.

Definition 2.3. An attribute is a *categorical attribute* if it has a fixed, usually small, set of distinct values that categorize all entities of the site.

For a categorical attribute, we can usually identify a complete set of its distinct values. Typically, this kind of attributes involves selection lists, such as *Make* and *Model* in Fig. 3, and *Binding* in Fig. 1. A group of radio buttons is equivalent to a selection list if only one of the radio buttons can be selected at a time, e.g., *Search for* in Fig. 3. A categorical attribute may be implemented as a textbox like *ZIP* in Fig.3.

A group of checkboxes can be considered as a special categorical attribute with special semantics. Take *Attributes* in Figure 1 as an example. It has three checkboxes - *First edition, Signed,* and *Dust jacket*. If none of them is checked, it means no restriction with the attribute. If one is checked, say, *First edition*, it means that only *first-edition* books are selected. But if all three checkboxes are checked, it means that *first-edition* or *signed* or *dust-jacket* books are selected, i.e., they are *disjunctive* rather than *conjunctive* whereas most attributes in a query interface are conjunctive.

Definition 2.4. An attribute is a *value-infinite attribute* if the number of its possible values is infinite.

Most attributes in this kind are textboxes, such as *Keywords* and *Title* in Fig. 1. All identification attributes also belong to this type, like *ISBN* in books domain (Fig. 1) and *VIN* in cars domain. Numerical and date attributes belong to this type as well.

Some techniques in [5] can be used in our work to identify and classify attributes into different types. We do not address this issue in this paper.

3 Modeling Querying Capability of Query Interfaces

3.1 Query Expression

In this paper, we assume that a *query* is a conjunctive query over a query interface. In other words, the conditions for all attributes in a query are conjunctive. Based on our observation, almost all query interfaces support only conjunctive queries.

Traditionally, a query condition contains values that are assigned to attributes, like {<*Author,* "Meng">*,* <*Keywords,* "Databases">}. There are three special values - *Null, All,* and *Unchecked*. *Null* is for textboxes, *All* for selection lists and radio buttons, and *Unchecked* for checkboxes. They have the same function – applying no restriction to the attribute. Thereinafter we use *Null* to represent any of these values.

In this paper, we focus on querying capability and each attribute's function in a query condition. We do not care about what value is assigned to attribute unless the value is *Null*. Thus, we do not include values but attributes in our *query expression*. An attribute is precluded from our query expression if its assigned value is *Null*.

Definition 3.1. A *query expression* (or *query* for short) is a set of attributes that appear in some query condition. A *query instance* is a specific query condition consisting of attribute-value pairs with the value valid for the corresponding attribute.

An attribute is assigned with a valid non-*Null* value if and only if it appears in a query expression. For example, query instance {<*Author,* "Meng">*,* <*Keywords,* "Databases">} is represented by query expression {*Author, Keywords*} on the query interface in Fig. 1. Therefore, each query instance corresponds to one query

expression. But one query expression may represent a group of query instances, with the same attributes but not totally the same valid values.

Definition 3.2. A query is *valid* if at least one of its query instances is accepted by the query interface, regardless of whether or not any records are returned.

That a query instance is accepted means that it is executed by the underlying database system. When a query instance is rejected by the query interface, an error message will be returned usually.

3.2 Conditional Dependency and Attribute Unions

Dependency may exist between a constraint and an attribute, e.g., constraint *All/Any* depends on attribute *Keyword* in Fig. 2. We name this dependency as *conditional dependency*. It is a restriction between a non-*functional* attribute and a constraint.

Definition 3.3. Given an attribute A and an constraint C, if any value of C is insignificant unless a non-*Null* value is assigned to A, we say a *conditional dependency* exists between A and C, denoted as $A \rightarrow C$.

Generally the attribute and its dependent constraints appear next to each other. Conditional dependency does not involve functional attributes. We use conditional dependency to define *attribute union* below.

Definition 3.4. Attribute A, its value type T, and constraints $C_1, \ldots, C_n$ ($n>0$) form an *attribute union*, denoted as $AU(A,T,C_1,\ldots,C_n)$, if and only if $A \rightarrow C_i$ ($i=1,\ldots,n$) and no other conditional dependency involves A. In $AU(A,T,C_1,\ldots,C_n)$, T and C_i are optional. And attribute A is named the *core attribute* of the *attribute union*, denoted as CA.

Note that attribute A alone forms an *attribute union* iff there is no conditional dependency involving A. In this case, the *core attribute* is A and the attribute union is denoted by the attribute name A for convenience. In Fig. 2, AU(*Keyword, All/Any*) is an attribute union. Attribute unions do not involve functional attributes.

3.3 Atomic Query

Our observations show that some attributes are necessary to form valid queries.

Example 1. For the query interface in Fig. 1, any valid query must include at least one of the attributes *Author, Title, ISBN, Keywords* and *Publisher*. Other attributes are not required for a valid query.

Example 2. For the query interface in Fig. 4, all of the attributes *From, To, Leave, Return* and *Travelers* are necessary for any valid query.

To describe the function of these attributes in a query, we propose the concept of *atomic query*, a key concept in our querying capability model. If a query is valid, its superset query is also valid. For example, in Fig. 1, if query {*Author*} is valid, query {*Author, Price*} is also valid.

We have stated that a query is a set of attributes. Actually, it also can be considered as a set of *attribute unions*. An *atomic query* is an irreducible set of *attribute unions* that forms a valid query. All supersets of an atomic query are valid and all valid

queries are supersets of some atomic query. Therefore, the identification of atomic queries is the most important part of querying capability analysis.

Definition 3.5. Given a set of attribute unions $S=\{AU_1,\ldots,AU_n\}$ that represents a valid query. S is an *atomic query* iff the query represented by any $T \subset S$ (here $\subset$ denotes proper subset) is not valid. *Atomic query* S is also denoted as $AQ(AU_1,\ldots,AU_n)$.

A query interface may have more than one atomic query. The interface in Fig. 1 has five atomic queries, each consisting of a single attribute: {*Author*}, {*Title*}, {*ISBN*}, {*Keywords*} and {*Publisher*}. The interface in Fig. 4 has one atomic query: {*From, To, Leave, Return, Travelers*} with five attribute unions.

3.4 Describing Querying Capability

We describe querying capability of a query interface using a model similar to the Backus-Naur Form (BNF). Different from BNF, the order of attributes in the query expression is not significant. Any attribute in the expression has a non-*Null* value. Any attribute not in the expression has value *Null*.

There are three special functions used in BNFs in this paper: *AQ, AU* and *ADJ*.

- Function $AQ(O_1, \ldots, O_n)$: if objects $O_1, \ldots, O_n$ form an *atomic query*, the function returns the atomic query. Otherwise, it returns an invalid symbol (note that a derivative rule in BNF does not apply to any invalid symbol).
- Function $AU(O_1, \ldots, O_n)$: if objects $O_1, \ldots, O_n$ form an *attribute union*, the function returns the attribute union; else, it returns an invalid symbol.
- Function $ADJ(A, B)$: if the control elements (or control element groups) A and B are *adjacent* (both left-right and up-down are considered) on the query interface, the function return the combination of A and B. Otherwise, it returns an invalid symbol. Adjacency is determined by some threshold: if the shortest distance between A and B is within the threshold, A and B is considered to be adjacent.

Symbols used and their meanings are as follows:

Q: Query
AQ: Atomic Query, or function *AQ*
AM: Attribute Module, an attribute union or a functional attribute or exclusive attribute (see below)
OQ: Optional Query, a group of attribute unions whose values are optional for a query
AU: Attribute Union, or function *AU*
CoA: Core Attribute, an attribute other than functional attribute
VT: Value Type, the value type of core attribute
C: Constraint
EA: Exclusive Attribute, one of a group of attributes that is exclusive to each other (i.e., only one of them can be used in any query)
ANO: Attribute Name Option, one of the options in a list of attribute names
RA: Range Attribute
CA: Categorical Attribute
VA: Value-infinite Attribute
FA: Functional Attribute
RBG: Radio Button Group, a group of radio buttons
CBG: Checkbox Group, a group of checkboxes
TB: Textbox
SL: Selection List
RB: Radio Button
CB: Checkbox

<Query>::= <AtomicQuery> [<OptionalQuery>]
<AtomicQuery>::=*AQ*(<AttributeModule> [, <AttributeModule>]*)

<OptionalQuery>::=<AttributeModule> [<AttributeModule>]*
<AttributeModule>::=<AttributeUnion> | <FunctionalAttribute> | <ExclusiveAttribute>
<AttributeUnion>::=*AU*(<CoreAttribute>[<ValueType>][<Constraint>]*)
<ExclusiveAttribute>::=<CoreAttribute> <AttributeNameOption>
<AttributeNameOption>::=<SelectionList> | <RadioButtonGroup>
<CoreAttribute>::=<RangeAttribute> | <CategoricalAttribute> | <Value-infiniteAttribute>
<ValueType>::= char | number | Boolean | date | time | datetime | currency
<Constraint>::=<SelectionList> | <RadioButtonGroup> | <CheckboxGroup>
<RangeAttribute>::=<SelectionList> | *ADJ*(<Textbox>,<Textbox>)
| *ADJ*(<SelectionList>, <Textbox>) | *ADJ*(<SelectionList>, <SelectionList>)
<CategoricalAttribute>::=<SelectionList> | <RadioButtonGroup> | <CheckboxGroup>
| <Textbox>
<Value-infinite Attribute>::= <Textbox>
<FunctionalAttribute>::=<SelectionList> | <RadioButtonGroup> | <CheckboxGroup>
<RadioButtonGroup>::= <RadioButton> | *ADJ*(<RadioButton>, <RadioButtonGroup>)
<CheckboxGroup>::=<Checkbox> | *ADJ*(<Checkbox>, <CheckboxGroup>)

The grammar above can be used to determine if a query is valid.

Example 3. The query interface in Fig. 1 has five atomic queries: *AQ*(*Author*), *AQ*(*Title*), *AQ*(*ISBN*), *AQ*(*Keywords*) and *AQ*(*Publisher*). They are defined by the logic of the query interface. There are five *attribute unions* that can be part of any *optional query - Published date, Price, Bookseller country, Binding,* and *Attributes.* And there are two functional attributes that can be part of any *functional query - Sort results by* and *Results per page*. Consider a query Q = {*Author, Title, Binding, Sort results by, Results per page*}. This query contains two atomic queries. Interpretation of the query is not unique. Both *Author* and *Title* are atomic queries. Either of them can be treated as part of an optional query. Fig. 5 is one interpretation tree. A query is valid if there exists at least one way to interpret it.

Several observations:

1. Any attribute module that is part of an atomic query can also be part of an optional query if two atomic queries appear in a valid query.
2. An attribute with a default value *All* is not part of an *atomic query*.
3. For a range attribute, if it reaches its largest range or its value is empty, it has a value *All*, so it is not part of an atomic query. For example, *Year range* in Figure 3, *from 1981 to 2007* means *All*.
4. A non-functional attribute that always has a non-*Null* value automatically becomes part of *atomic query*. An example is *Travelers* in Fig. 4.

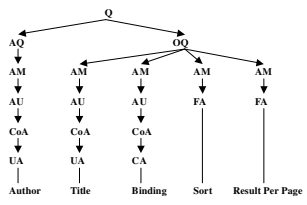

Fig. 5. An interpretation tree

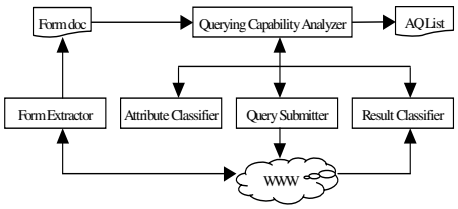

Fig. 6. Overview of the system

4 Querying Capability Construction

4.1 Attributes, Constraints and Control Elements

Constructing the querying capability model of a query interface relies on attribute analysis. Some attributes' functionalities are impacted by constraints.

Constraints and hidden attributes
Constraints generally appear as a radio button group. They are always assigned their default values because different values will not impact querying capability construction, especially for the identification of atomic queries. Hidden attributes are invisible on a query interface but have fixed values assigned to them in each query. The values are included in the form tag on the query interface and can be identified.

Values *Null* and *All*
Values *Null* (for textbox and checkbox) and *All* (for selection list and radio button) mean no constraint on their attributes. An empty textbox means value *Null*.

For attributes with multiple elements, *Null* or *All* values may be assigned in different ways. For example, for attributes with two textboxes to have value *Null*, both textboxes should be assigned *Null* (e.g., *Price* in Fig. 1). To assign an *All* value to an attribute with two selection lists of numerical values (e.g., *Year Range* in Fig. 3), select the smallest value in the first selection list and the largest value in the second one, namely, assign the maximum range.

Type I and type II attributes
From the angle of values *Null* (for a selection list, *Null* implies *All*), any visible non-*functional* attribute can be classified into either of two types as defined below.

Definition 4.1. If a non-functional attribute has no value *Null* in its possible values, it is a *type I attribute* (in most cases, they are selection lists.). If it can be assigned *Null*, it is a *type II attribute* (this type includes textboxes/selection lists with value *Null*.)

Exclusive attributes
Exclusive attributes are generally included in a selection list or a radio button group. Only one of them can be used in any given query. Techniques for identifying exclusive attributes are discussed in [5].

4.2 Constructing Querying Capability Model

4.2.1 Overview
Figure 6 provides an overview of the system. It consists of five parts: *form extractor*, *querying capability analyzer*, *attribute classifier*, *query submitter* and *result classifier*.

For a deep Web source, given its URL, the form extractor retrieves and analyzes the search form from the HTML source of its query interface page, and represents the extracted form as an XML file. With the XML file as input, the querying capability analyzer starts the attribute classifier and then interacts with the query submitter and the result classifier. The query submitter generates a query and submits it to the deep Web source while the result classifier receives result page, analyzes whether the query is accepted, and reports its decision to the querying capability analyzer. Through a

series of query submissions and result analyses, the querying capability analyzer determines the list of the atomic queries for the query interface.

4.2.2 An Algorithm for Searching Atomic Queries

A brute force method for finding atomic queries is to submit a query for every combination of all attributes on the query interface. This is obviously not feasible due to its huge query cost. We now describe a more efficient method.

Definition 4.3. For a query interface, a query including exactly n type II attributes is called a *level n query*.

A query interface with m type II attributes has $\binom{m}{k}$ level k queries, including one level 0 query and m level 1 queries. All type I attributes are included in any query with their default values on the query interface. When a type II attribute is included in a query, it is assigned a non-*Null* value. Fig. 7 shows our algorithm.

The algorithm first recognizes each exclusive attribute as an AQ. If there are no exclusive attributes, it searches for AQs by submitting queries. For any level k query q, if a query that is a proper subset of q (we treat a query as an attribute set) is already in the AQ set, then q cannot be an AQ and should be discarded. Otherwise, submit q, download the result page, and analyze the page. If the page represents acceptance, the query is added to the AQ set of the query interface.

```
Algorithm: identifyAQs(I)
Input: I - a query interface,
       A1 = {a | a is a type I attribute on interface I},
       A2 = {a | a is a type II attribute on interface I}.
Output: L - a set of AQs for I; L's initial value is ϕ.
for each a ∈ A1 ∪ A2 do
        if the value set of a represents a set of exclusive attributes E
                A1 := A1 − {a};
                for each e ∈ E do
                        s := A1 ∪ {e};   // s represents an AQ
                        L := L ∪ {s};
                return L ;
for k = 0 to |A2| do
        Qk := {q | q is a level k query on interface I};
        for each q ∈ Qk do
                if k = 0
                        R := ϕ ;
                else
                        R := {r | r is a level k−1 query, r ⊆ q, r ∈ L};
                if R = ϕ
                        submit query q;                (1)
                        download result page P;
                        if P is an acceptance page     (2)
                                L := L ∪ {q}
Return L
```

Fig. 7. An algorithm for searching atomic queries

In the following, we discuss two important steps in the algorithm: submit a query at line (1) and decide if a result page is an acceptance page at line (2).

4.2.3 Automatic Query Submission

To submit a query is actually to submit a form on a web page. We use the following method to submit a query automatically:

1. Identify attributes to which non-*Null* value will be assigned, and assign values. To assign a non-*Null* value to a selection list is quite straightforward – just select a non-*Null* value from its value list. To do so for checkbox is easy too – just check it. But for textboxes, giving a proper value needs careful consideration. The following two methods can be used to assign non-*Null* value to a textbox.
   - For every domain, keep a table of attributes and some corresponding values.
   - Keep a table of value types and some corresponding values. For example, value 1998 is for year or date, and value 13902 is for zip code.
2. Assign value *Null* to other attributes. For textboxes and checkboxes, just leave them blank or unchecked. For selection lists, select the *Null (All)* value.
3. Assign values to constraints and hidden attributes. Constraints generally are radio button groups; assign default values to them. Hidden attributes have fixed values.
4. Construct URL string with value assignments. A value assignment is a control-name/current-value pair.

4.2.4 Result Page Classification

To construct the querying capability model, we need to submit queries and decide whether the query is accepted. Therefore we must know whether the result page of the query represents acceptance or rejection. The performance of the querying capability construction depends on the accuracy of result page classification.

A typical acceptance page is like Fig. 8 and a rejection page is like Fig. 9.

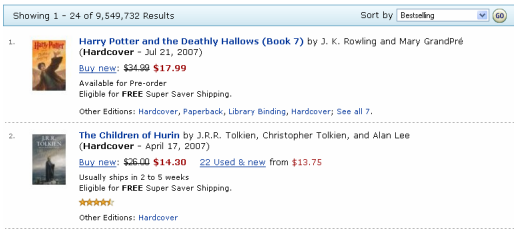

Fig. 8. Part of an acceptance result page of amazon.com

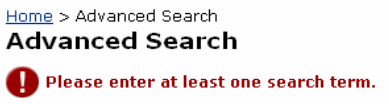

Fig. 9. Part of a rejection result page of abebooks.com

Since a result page is a document, the problem can be treated as a document classification problem. Let's first identify the special features that appear on result pages that represent acceptance and rejection below.

Evidences for Query Acceptance

1. Sequences exist in the result page. Two kinds of sequences may exist with order numbers in a result page, one for result records and the other for result pages (see Fig. 8).
2. Large result page size. A large result page size means this query is probably accepted. From observation, a query is probably accepted if its result page is five times larger than the smallest of a group of result page from the same site.

3. Result patterns for query acceptance. Some patterns imply that the query is accepted, for example, *Showing 1 - 24 of 9,549,732 Results* in title (see Fig. 8).

Evidences for query rejection

1. Error messages for query rejection. When a query is rejected, frequently an error message appears on the result page, such as "*please enter at least one search term*" (Fig. 9), and "*no valid search words"*, etc.
2. The result page is similar to its query interface page. The result page sometimes has the same interface as the query interface page to allow users to enter another query.
3. Small result page size. Rejection pages generally have a relative small size.

In Section 5, we discuss how to build a result page classifier based on the observations above.

5 Experiments

The experiment consists of two parts: (1) build a result page classifier; (2) use the classifier to identify atomic queries.

We use two datasets of deep Web sources from three domains: books, music and jobs. Dataset 1 is for training, i.e., build a classifier. Dataset 2 is for identification using the built classifier. These sources are mainly from the TEL-I dataset at [13].

- Dataset 1: 11 book sources, 10 music sources, and 10 job sources.
- Dataset 2: 12 book sources, 11 music sources, and 10 job sources.

5.1 Build a Result Page classifier

We have discussed evidences for query acceptance and rejection in Section 4.2.4. Based on the evidences, we select five features: length_of_max_sequence (numeric), pagesize_ratio (numeric), contain_result_pattern (boolean), contain_error_message (boolean), similar_to_query_interface (boolean). The class label has two values: *accept* and *reject*.

From Dataset 1, we generated 237 result pages. For each result page, we compute the values for the five features above, and manually assign the class label. The five values plus the class label for the same result page construct one training record. Based on the 237 training records, we build a result page classifier using Weka [14].

We tried four different classifiers, aiming to select the one with the best performance. The four classifiers are C4.5 algorithm of decision tree (C4.5), *k*-nearest neighbors (KNN), naïve Bayes (NBC), support vector machines (SVM).

For training records, we use one part for training, remaining part for validation. If the training ratio (TR) is 0.6, then 60% are used for training, and 40% for validation.

We experimented with four different TRs from 0.5 to 0.8 and the four classifiers. The results are shown in Fig. 10. Classifier C4.5 has the best overall performance. When TR = 0.7, its accuracy is 98.6%. We notice that for all four classifiers, the accuracy reaches their highest when TR = 0.7. When TR is changed to 0.8, the

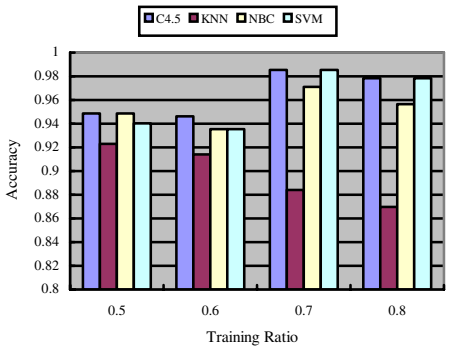

Fig. 10. Comparison for different classifiers

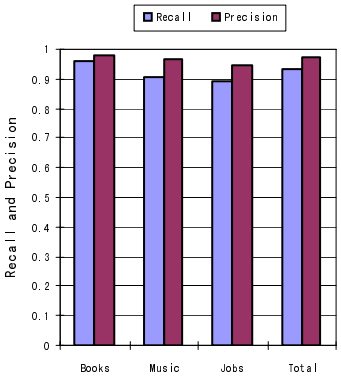

Fig. 11. Average recall and precision

accuracy gets lower due to overfitting. The C4.5 classifier built with TR = 0.7 will be used in the experiment in the later part.

5.2 Metrics for Atomic Query Identification

For a query interface *I*, we determine its actual atomic queries (AQ) by manually submitting queries. After the system finishes AQ identification, we compare the system identified AQs with actual AQs to determine its performance. In this paper, the performance is measured using average *recall* and *precision* over all query interfaces tested.

5.3 Experimental Results

All the results are based on Dataset 2. The accuracy of result page classification affects the recall and precision of AQ identification. Fig. 11 shows the average recalls and precisions for three domains as well as the overall recall and precision. The recall ranges from 89.5% for jobs domain to 95.7% for the books domain while the precisions are significantly higher. This indicates that our method can identify AQs accurately. For every domain, the recall is lower than the precision. This indicates our method for deciding if queries are accepted is conservative. In other words, if there is no strong evidence that a query is accepted, the system tends to treat it as rejected.

6 Conclusions

This paper proposed a method to model the querying capability of a query interface based on the concept of atomic queries. We also presented an approach to construct querying capability automatically. The experimental results show that the approach is practical. The constructed querying model can be useful for all applications that need to interact with a query interface.

Acknowledgment. This work is supported in part by the following NSF grants: IIS-0414981, IIS-0414939 and CNS-0454298.

References

1. Bergman, M.: The Deep Web: Surfacing Hidden Value (September 2001), http://www.BrightPlanet.com
2. Raghavan, S., Garcia-Molina, H.: Crawling the Hidden Web. In: VLDB, pp. 129–138 (2001)
3. Chang, K.C-C, He, B., Li, C., Patel, M., Zhang, Z.: Structured Databases on the Web: Observations and Implications. SIGMOD Record 33(3), 61–70 (2004)
4. Zhang, Z., He, B., Chang, K.C-C: Understanding Web Query Interfaces: Best-Effort Parsing with Hidden Syntax. In: SIGMOD Conference, pp. 107–118 (2004)
5. He, H., Meng, W., Yu, C., Wu, Z.: Constructing Interface Schemas for Search Interfaces of Web Databases. In: Ngu, A.H.H., Kitsuregawa, M., Neuhold, E.J., Chung, J.-Y., Sheng, Q.Z. (eds.) WISE 2005. LNCS, vol. 3806, pp. 29–42. Springer, Heidelberg (2005)
6. He, H., Meng, W., Yu, C., Wu, Z.: Automatic integration of Web search interfaces with WISE-Integrator. VLDB J. 13(3), 256–273 (2004)
7. Wu, P., Wen, J., Liu, H., Ma, W.: Query Selection Techniques for Efficient Crawling of Structured Web Sources. In: ICDE (2006)
8. Levy, A., Rajaraman, A., Ordille, J.: Querying Heterogeneous Information Sources Using Source Descriptions. In: VLDB, pp. 251–262 (1996)
9. Ipeirotis, P., Agichtein, E., Jain, P., Gravano, L.: To Search or to Crawl? Towards a Query Optimizer for Text-Centric Tasks. In: SIGMOD Conference (2006)
10. BrightPlanet.com, http://www.brightplanet.com
11. Bergholz, A., Chidlovskii, B.: Crawling for Domain-Specific Hidden Web Resources. In: WISE 2003, pp. 125–133 (2003)
12. Arasu, A., Garcia-Molina, H.: Extracting Structured Data from Web Pages. In: SIGMOD Conference, pp. 337–348 (2003)
13. Chang, K.C.-C., He, B., Li, C., Zhang, Z.: The UIUC web integration repository. CS Dept., Uni. of Illinois at Urbana-Champaign (2003), http://metaquerier.cs.uiuc.edu/repository
14. Witten, I., Frank, E.: Data Mining: Practical machine learning tools and techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)