Easy web service discovery: A query-by-example approach

Marco Crasso\textsuperscript{1,2}, Alejandro Zunino\textsuperscript{*1}, Marcelo Campo\textsuperscript{1,2}

ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina

Received 26 September 2007; received in revised form 8 February 2008; accepted 12 February 2008
Available online 4 March 2008

Abstract

Web services have acquired enormous popularity among software developers. This popularity has motivated developers to publish a large number of Web service descriptions in UDDI registries. Although these registries provide search facilities, they are still rather difficult to use and often require service consumers to spend too much time manually browsing and selecting service descriptions. This paper presents a novel search method for Web services called WSQBE that aims at both easing query specification and assisting discoverers by returning a short and accurate list of candidate services. In contrast with previous approaches, WSQBE discovery process is based on an automatic search space reduction mechanism that makes this approach more efficient. Empirical evaluations of WSQBE search space reduction mechanism, retrieval performance, processing time and memory usage, using a registry with 391 service descriptions, are presented.

© 2008 Elsevier B.V. All rights reserved.

Keywords: Web services; Web service discovery; Distributed systems; Information filtering; Query-by-example; Information retrieval

1. Introduction

Although the Web was conceived for human use, this infrastructure has evolved to allow computer programs to become major players of the Web. Recently, the trend in software development has been converging towards reusing and composing loosely coupled functionality accessible by the Web, commonly known as services. Service-oriented computing methodology replaces the development of specific software components with a combination of service discovery, selection and engagement [1]. A typical service-oriented architecture (SOA) has three main parts: a provider, a consumer and a registry. A registry provides the foundations for service discovery and selection. Up until now, the software industry has broadly adopted SOA by using Web service technologies [2]. A Web service [3] is a Web accessible software that can be published, located and invoked by using the standard Web infrastructure.

UDDI (Universal Description, Discovery and Integration) is the materialization of the SOA registry component for publishing and discovering Web services. Providers may publish Web services along with their associated metadata in a UDDI registry. For example, providers may manually assign a category to their services from a number

\textsuperscript{*} Corresponding author. Tel.: +54 2293 439682; fax: +54 2293 439681.
\textit{E-mail address:} azunino@exa.unicen.edu.ar (A. Zunino).

\textsuperscript{1} Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).
\textsuperscript{2} Tel.: +54 2293 439682; fax: +54 2293 439681.

0167-6423/S - see front matter © 2008 Elsevier B.V. All rights reserved.
On the other hand, discoverers may look up third-party services or the North applications, i.e. automatically discovering services is not addressed here. Other contributions are summarized below:

service discovery. In this sense, this work aims at making Web service discovery easier for humans and not for services and their descriptions are written to be reused by other developers.

discoverers to ask for a third-party service by providing a natural language description, or a handful set of keywords, for a given query, exploits human pattern matching abilities for writing queries. Moreover, WSQBE allows human discoverers to ask for a third-party service by providing a natural language description, or a handful set of keywords, of its expected functionality. The approach depends on publishers’ use of best practices for naming and documenting services, operations and arguments using meaningful words. In practice, this assumption is usually true, because Web services and their descriptions are written to be reused by other developers.

Although some efforts have focused on assisting Web service discoverers, they have several shortcomings. First, some efforts propose to enhance Web services with semantic Web technologies [12], such as [13–16]. These efforts are very good for dealing with semantically annotated Web services. However, these semantic-based approaches suffer from the typical problems associated with ontologies, namely the high complexity of building them [17,18], the lack of standard ontologies and the absence of public semantically annotated Web services [4]. Second, other efforts attempt to exploit classic information retrieval techniques for assessing the similarity between two Web service descriptions [19–21] without relying on semantics. Basically, these approaches convert available service descriptions and service needs, or queries, into a common representation, and then compare them to find out those with the most similar representations. Though these information retrieval approaches show respectable precision for retrieving relevant services, these one-to-many matching approaches might have problems handling a large number of services.

To cope with the problems mentioned above, we propose WSQBE, an approach for easing the task of discovering suitable Web services. Instead of performing a one-to-many matching, we propose a two-step matching approach that first reduces the search space to a subspace and then compares a query against service descriptions belonging to this small subspace. The idea is to represent Web service descriptions and queries within a classic vector space [22,23], which is partitioned into subspaces corresponding to the categories of the services. This approach makes WSQBE more efficient with respect to previous methods, making it appropriate for dealing with a large number of services. On the other hand, we aim for a programmatic query language that allows users to ask for a service by providing a mere skeleton of its expected operation description or interface. Basically, we aim at allowing a developer to state a query using his/her preferred programming language. The goal is to exempt developers for making an additional effort for searching services by transparently pulling out programmatic descriptions from their service-oriented application source code. This, besides avoiding the problem of knowing which low-level Web service features are important for a given query, exploits human pattern matching abilities for writing queries. Moreover, WSQBE allows human discoverers to ask for a third-party service by providing a natural language description, or a handful set of keywords, of its expected functionality. The approach depends on publishers’ use of best practices for naming and documenting services, operations and arguments using meaningful words. In practice, this assumption is usually true, because Web services and their descriptions are written to be reused by other developers.

The main goal of our method is to ease one essential aspect of the development of service-oriented applications: service discovery. In this sense, this work aims at making Web service discovery easier for humans and not for applications, i.e. automatically discovering services is not addressed here. Other contributions are summarized below:

• A novel combination of text-mining techniques for bridging syntactic differences of Web service descriptions, the classification approach that supports WSQBE search space reduction and techniques for easing query generation.
• WSQBE supports queries expressed as partial Web service descriptions, source code method declarations or natural language descriptions of the expected service or its operations.

---

3 UNSPSC http://www.unspsc.org/.
WSQBE has been validated in four parts. First, by testing its filtering stage on a group of 391 hand-classified Web services. Second, by testing the whole discovery approach with 30 very short descriptive queries, which have been written in a programmatic query language with little effort. Third, by assessing the overhead imposed by our two-step matching process over classic publication and discovery. Finally, by measuring WSQBE scalability, in terms of processing time and memory usage, with the number of published services.

The rest of this paper is organized as follows. The next section surveys the most relevant previous approaches. Section 3 presents WSQBE in detail. Then, its experimental evaluations are reported in Section 4. Finally, Section 5 concludes the paper and describes future work.

2. Related work

During the past few years some research efforts have focused on assisting Web service discoverers. Some of these efforts propose to combine Web services and semantic Web technologies [12]. OWL-S (Ontology Web Language for Web Services) defines three ontologies for describing a Web service model [14]. WSMO (Web Service Modeling Ontology) defines a meta-ontology for modeling Web services [15]. WSDL-S incorporates ontologies into Web service descriptions written in WSDL (Web Service Description Language) [13]. These models allow publishers to associate a Web service with concepts from an ontology. One of the main benefits of describing services with ontologies is that discoverers can have access to an unambiguous definition of each part of a Web service (e.g., inputs, outputs, operations, etc.) and then use semantic matching algorithms [24] to find the required services. By exploiting these unambiguous definitions and semantic matching, software agents can automate the process of seeking, accessing and composing services [25,26,16,27]. Nevertheless, building ontologies has been identified as a costly, time-consuming, tedious and error prone task [17,18]. Furthermore, the lack of standards to integrate or reuse existing ontologies and the absence of public semantic annotated Web services [4] hinder the widespread usage of this approach. Besides, these ontology-based approaches assume that both publishers and discoverers are proficient in semantic Web technologies. In addition, they impose modifications on the current UDDI infrastructure such as semantic description repositories, semantic-aware search engines or protocols for dealing with incompatible information models [28].

Recently, some approaches proposed to exploit classic information retrieval techniques [29]. Basically, the idea is to reduce the problem of discovering relevant services to the well-known problem of finding relevant documents. For example, [19,20,9,30,8,21] adapt the Vector Space Model (VSM) [22] for representing Web service descriptions as vectors and then comparing these representations. Broadly, VSM is an algebraic model for representing text documents in a multi-dimensional vector space, where each dimension corresponds to a separate term (usually single words). As a result, documents having similar contents are represented as vectors located near in the space. Moreover, a query is also represented as a vector. Then, retrieving relevant services is a one-to-many matching technique for finding nearest neighbors in a vector space (VS). As this model relies on words, its retrieval effectiveness depends on developers’ conventions to name and document services. [7] summarizes and compares the most relevant metrics for assessing the similarity between two Web services. The paper reports an evaluation of different approaches with the same corpus of Web services concluding that classic TF-IDF (see Section 3.3), a statistical measure used to evaluate how important a word is to a document, over performed other approaches in most cases. Stroulia and Wang [20] compared a TF-IDF heuristic against a structural-matching heuristic, which is based on analyzing the XML syntax of two service descriptions. Their results have shown that “the information retrieval method performs better than the structure-matching method”. The main drawback of these one-to-many matching approaches is that they might have problems handling a large number of documents, when the number of matching operations is large. Another drawback is that the TF-IDF heuristic requires to assess the occurrences of each term within each available Web service of a VS, then, when publishing a new service, the whole VS must be recalculated. Clearly, recalculating the VS for a large number of services is not desirable [23,31].

Kozlenkov et al. [32] propose a novel query language for Web service discovery based on architectural specifications. This work derives queries from behavioral and structural Unified Modeling Language (UML) design models of SOA systems. These models must be specified by using a UML extension, a.k.a. profile. This profile allows designers to indicate if an operation must be either implemented or delegated to a third-party service. Moreover, by
using this profile designers can specify many conditions about the services and operations that should be discovered (e.g., conditions about the provider of a service, the number of parameters of an operation, etc.). To assess the similarity between a UML-based query and available services, a two-step process is used. In the first step, this approach looks for services with operations that satisfy the conditions of the query and retrieves the specifications of such services. In the second step, a similarity heuristic based on graph-matching is used for finding the operations that best match the query. The authors formulate this stage as an instance of the assignment problem. They implemented it by using a standard algorithm for the assignment problem [33], which solves it in polynomial time: $O(n^3)$. In spite of the expressiveness of the proposed query language, queries might be rather hard to define. On the one hand, service-oriented application designers must be proficient in this UML profile. On the other hand, designs of existent service-oriented applications must be adapted to incorporate the proposed profile and its discovery capabilities.

Zhuge and Liu [34,35] present a flexible approach for Web service retrieval. The flexibility of this approach is twofold. First, it proposes a powerful definition of similarity between services to allow discoverers to look for services that are identical, more specific or more general than their queries. Second, this discovery approach uses a query language whose syntax and semantics are similar to SQL. Therefore, a developer might find this query language straightforward to learn. This approach uses a relational model of Web service data based on the standard Web service schema information present in UDDI. We think that this approach and ours may be combined to improve them. On the one hand, [34] may further reduce the search space by using our automatic classification approach and may exploit the support for natural language queries provided by WSQBE. On the other hand, by incorporating the aforementioned similarity assessment we can enhance the WSQBE filtering stage. Moreover, we can provide another alternative for query specification by incorporating this SQL-like language in WSQBE.

By considering previous approaches for service discovery, we believe that there is a relationship between a developer’s effort and the benefit he/she may obtain from a discovery system. We think that the more effort developers put on: (1) describing their services and (2) describing their service needs, the more accurate discovery results they would obtain. Despite the advantages of ontology-based approaches, namely unambiguous definitions and accurate semantic matching capabilities, these approaches increase developers’ effort with regard to two aspects. First, Web services must be semantically described. Second, discoverers must be trained in ontology description languages to specify queries and to select a service from the list of semantic descriptions associated with several candidates. On the other hand, the aforementioned UML profile allows discoverers to specify very expressive and specific filtering criteria. As a consequence, we might expect short, accurate and specific lists of candidate services in responses. Nevertheless, it might be hard to adopt this profile and its discovery capabilities. Finally, though vector-space-based approaches demand no effort from publishers, in general their accuracy is lower than previous approaches [11]. Moreover, current VSM-based approaches might have performance problems dealing with a large number of services. In addition, if developers do not follow good development practices and give meaningless names to their services, operations and arguments (e.g., “arg0”, “arg1”, “doSomething”), the accuracy of VSM approaches may be low.

We aim to position WSQBE at being useful for developers effortlessly. To do this, we have designed a novel two-step matching method for efficiently assessing Web service similarity. First, by exploiting existing meta-data associated with Web services, WSQBE deduces a group of similar services to reduce the search space. Secondly, WSQBE examines this group to retrieve a ranked list of similar services. This ranked list alleviates discoverers when examining retrieved results, because the most relevant services are usually at the top of the list. Furthermore, WSQBE provides a flexible query language, which exempts discoverers from making an additional effort when specifying their service needs. On the other hand, publishers are unaware of WSQBE existence, since no additional semantic meta-data is required. Therefore, our approach does not require training on new technologies. Similarly to current VSM-based approaches, WSQBE depends on developers following best practices for describing and documenting services. To reduce the impact of this limitation, we present several text-mining preprocessing techniques for dealing with syntactic differences in service descriptions. As we will show, both retrieval accuracy and time complexity of WSQBE are promissory. The next section describes WSQBE.

3. Query-by-example approach for Web service discovery

For the most part, a service-oriented software system comprises a set of functional or logical services with well-defined interfaces used for communication among them. At design time, a SOA software designer may decide that a certain functionality might be either implemented or delegated to a third-party service, i.e., to outsource a Web
In order to reduce the effort for searching third-party services, WSQBE looks for Web services that fulfill a desired service expectation. In essence, by applying the idea of Query-by-example we propose to make easier the task of defining a query for discovering Web services. Fig. 1 illustrates the overall process WSQBE uses for service discovery. Query by Example (QBE) is a method of query creation that allows a user to search for an entire piece of information based on an example in the form of a selected part of that information. This method is useful when users partially know how the desired results are. The cornerstone behind WSQBE discovery approach is that consumers partially know the descriptions of the Web services they want. WSQBE aims at allowing consumers to ask for a third-party service by providing a skeleton of its description, represented as a handful set of words or a partial functional description of the expected interface (step 1 in Fig. 1). Conceptually, WSQBE analyzes this skeleton, i.e., the example, and then looks for services whose descriptions are similar to the skeleton. WSQBE uses a multi-dimensional vector space representation for examples and available services.

WSQBE preprocesses the example, removing irrelevant words and bridging syntactic differences (step 2 in Fig. 1). The resulting set of words is converted to a vector (step 3 in Fig. 1). Then, instead of comparing the example against a large number of services, WSQBE makes a first reduction of the search space by deducing the category of the desired service (step 4 in Fig. 1). To do this, WSQBE uses an automatic classification system. This classifier exploits machine learning techniques to learn from available services, which have been previously categorized and published in UDDI registries by their providers. This reduction of the search space can be interpreted as an automatic version of a typical manual discovery process, which usually comprises two steps: browsing and selection. Subsequently, WSQBE compares the example only against the services belonging to its category (step 5 in Fig. 1). Finally, WSQBE returns those services that are most similar to the example (step 6 in Fig. 1). Algorithm 1 summarizes the different stages of our approach. WSQBE aims at not imposing modifications over the current Web service infrastructure. In particular, WSQBE implements the UDDI Publish and Inquiry specifications, so that it may be placed between users (both publishers and discoverers) and UDDI transparently, as shown in Fig. 1. A publisher uses the UDDI Publish API in order to publish his/her services in a WSQBE registry, and then WSQBE maps the services onto the VS and forwards the request to a UDDI registry. In the same way, WSQBE supplies discoverers with the UDDI Inquiry API and a special API for supporting example-based and natural language queries. The rest of this section will explain in detail the stages of the approach.

3.1. Representing Web service descriptions and queries as vectors

From an information retrieval viewpoint, the data within an information system include two major categories: documents and queries, or the expressions of information need. The key problems are how to state a query and how to identify documents that match that need [29]. The distinction between considering a query to be a document and
1: *procedure* DISCOVER(example)   \(\triangleright\) Returns a list of candidate Web services
2:  String[] stems ← PREPROCESS(example)
3:  double[] vector ← CREATEVECTOR(stems)
4:  Category[] category ← CLASSIFY(vector)
5:  for all service ∈ category[0] do
6:     if cosineSimilarity(vector, service) > Threshold then
7:         APPEND(service, candidates)
8:  end if
9:  end for
10: return candidates
11: end procedure

Algorithm 1. Main steps of WSQBE.

![Fig. 2. Vector space model.](image)

considering it to be different from a document affects the manner in which the retrieval process is modeled. If the query is considered to be a document, then retrieval is a matching process. The backbone of our approach is to use the same representation for both services and queries. As a consequence, the Web service discovery process is reduced to a matching problem.

Matching similar documents is a problem with a long history in information retrieval [38]. Methods based on linear algebra have shown to be suitable alternatives for correlating similar documents [39]. These techniques map documents onto a vector space (VS) [22], in which a vector \(\vec{v} = (e_0, \ldots, e_n)\) represents each document. Each element \(e_i\) represents the importance of a distinct word \(w_i\) for that document. For example, in Fig. 2 (a) we represent a document containing the terms “currency” and “exchange”. Then, similar documents will have similar vector representations. As a consequence, searching related documents translates into searching nearest neighbors in a VS. For example, in Fig. 2 (b) the cosine of the angle \(\Omega\) provides an estimation of how similar two documents are. In our approach, these vectors stand for the descriptions accompanying available Web services. Moreover, by considering a query as a document, WSQBE maps queries onto vectors in a VS also. Therefore, a vector \(\vec{q} = (e_0, \ldots, e_n)\) stands for a particular query and its nearest neighbors in the VS stand for relevant services. A Web service is described by a WSDL document [40], a well structured standard for describing, in a textual manner, a Web service, its operations and arguments. Mostly due to WSDL structured format, accessing a Web service description to extract the importance of each distinct word is considered feasible [41]. As a consequence, mapping a WSDL document onto a vector is also feasible. Nevertheless, two service descriptions which are semantically equivalent may be different from a pure syntactical point of view, and then these services will have different representing vectors [42]. Clearly, this situation could harm our discovery approach performance. To address this problem we have designed some text-mining preprocessing techniques that attempt to bridge syntactic differences by taking into account the particularities of service descriptions. Section 3.2 will explain in detail the stages of our text-mining method.

There are some different similarity calculations for finding near neighbors in a VS [29,23]. One measure that is widely used is the *cosine measure* [38], which has been shown to be better than other similarity metrics in terms of retrieval effectiveness [43]. This measure is developed from the cosine of the angle between two vectors. This approach assumes that two documents with a small angle between their vector representations are related to each
other. As the angle between the vectors shortens, the cosine angle approaches 1, i.e., the vectors are closer, meaning that the similarity of whatever is represented by the vectors increases. We use this measure for retrieving a ranked list of relevant services to a query. To do this, we compute the cosine angle between a query $Q$ and each service $S$, and then sort these results in decreasing order of cosine angles. Formally:

$$cosineSimilarity(Q, S) = \frac{Q \cdot S}{|Q||S|} = \frac{\sum_{i=1}^{T} t_{S,i} \times t_{Q,i}}{\sqrt{\sum_{i=1}^{T} t_{Q,i}^{2} \sum_{i=1}^{T} t_{S,i}^{2}}}.$$ 

The computational complexity of assessing the cosine similarity measure between two vectors is:

$$O(T) = \sum_{1}^{T} 2 \times M + S$$

where $T$ is the number of different terms, $M$ and $S$ are two constants representing the corresponding cost of multiplication and square root operations, respectively. This complexity takes linear time and depends on the number of dimensions of the VS, i.e., the number of different terms $T$. As a consequence, comparing vectors may be a time-consuming task when the number of terms is large. Therefore, instead of comparing an example against all available services, i.e., the whole VS, WSQBE compares the example against a subset of the VS (see Algorithm 1, line 6). This subset represents Web services belonging to the same domain. As a consequence, the services within a domain usually share the same sublanguage [44]. For the purposes of this paper we can informally define a “sublanguage” as a form of natural language used in a sufficiently restricted setting [45]. Typically, a sublanguage uses only a part of the language structures. Therefore, the number of different terms of an individual domain or sublanguage is often lower than the sum of different terms present in each domain [44]. In other words, in a business domain words such as economy, competitive and currencies occur often, while words such as affine, chebyshev and commutative seldom appear. Then, comparing the example against the services associated with a particular domain may reduce $T$, i.e., the number of different terms, thus reducing the number of operations for computing $cosineSimilarity$.

UDDI allows service providers to publish a Web service along with its associated meta-data, for example the name of the category the service belongs to [10]. WSQBE exploits this meta-data to build groups of category related vectors. WSQBE deduces the category of a given query and then compares it only against the vectors that belong to that category. In order to deduce the corresponding category for a query, WSQBE uses a document classifier (see Section 3.3 for further details). As we will see, this classifier represents each category $i$ as a vector $\vec{c}_i$. This vector is generated by computing the average of the vectors that stand for the services belonging to category $i$. The category whose vector is more similar to the query, in terms of cosine similarity, is assigned to the query. Therefore, deducing a proper category for a query takes linear time, $O(CT)$, and depends on the number of available categories $C$ and different terms $T$ in the whole VS.

In essence, WSQBE uses the following two-step matching process:

1. comparing the query against the average of each category to determine the nearest category.
2. comparing the query against the services belonging to the category returned by the previous step.

To clarify the ideas introduced so far we show an example in Fig. 3. It illustrates the complexity of querying a registry with 8 available services divided into 2 categories. Here, we defined complexity as the number of vector comparisons performed. The first row shows the complexity of comparing a query against all vectors, which results in $O(ST)$, with $S$ being the number of available services. On the other hand, the second row shows the complexity of each individual step of our approach, which results in $O(max(CT, S_CT))$ with $C$ being the number of categories, $S_C$ being the maximum number of services per category and $T_C$ being the maximum number of different terms per category. The number of categories $C$ is, at most, equal to $S$, $S \geq C$, the maximum number of services per category $S_C$ is, at most, equal to $S$, $S \geq S_C$, and the maximum number of different terms per category $T_C$ is, at most, equal to $T$, $T \geq T_C$. Clearly, this two-step method may reduce the number of vector comparisons. In fact, if $C + S_C < S$, WSQBE requires less vector comparisons with respect to a one-step approach. The more the number of categories, the less the required vector comparisons. Conversely, if $C + S_C \geq S$, this is, there is at most one service within each category, WSQBE requires only one more vector comparison. In addition, depending on the linguistic characteristics of each sublanguage used in the categories, WSQBE may also require less operations for computing $cosineSimilarity$. 
Text mining, also known as intelligent text analysis, refers to the process of extracting interesting and non-trivial information and knowledge from unstructured text [46]. In general, automatic document classifiers support classification of documents seen as objects that are characterized by features extracted from their contents, where these features must be pulled out using text-mining techniques. In particular, WSQBE uses text mining for extracting features from Web service descriptions. In this context, a feature is a term which is relevant to a particular category of services.

In object-oriented terms, a WSDL document describes a service as an interface, an operation as a method, and a message as a method argument. These arguments are defined abstractly by using the W3C standard XML Schema Definition (XSD) [47]. Optionally, each part of a WSDL document may contain documentation in the form of comments. Fig. 4 shows a WSDL document and its corresponding relevant information (in bold) for retrieval. We will use this example in the rest of the section for clarifying the main stages of our approach.

Web service documentation is mostly comments written by developers, as Sabou et al. [42] assert. In general, these comments are written in English, have a low grammatical quality, punctuation is often ignored and several spelling mistakes and snippets of abbreviated text are present. Also, different development teams implement different services, and these teams may use different coding conventions. For example, they may use different notations, such as Java Bean Notation or Hungarian Notation. Moreover, WSDL allows developers to define the same message in many
encoding styles (rpc/literal, rpc/encoded, document/literal and document/literal wrapped), which may be syntactically different from each other. Broadly, a style deals with how exchanged data is encapsulated by the constructors of XSD. As a result, two services conceived for a particular task may have a syntactically different interface, such as sendMail(ns:email e) and sendMail(xs:string sFrom, sTo, sSubject, sBody).

As we will show in Section 4, these characteristics of WSDL degrade the next phases of WSQBE, namely classification and vector comparison. As a consequence, it is necessary to preprocess WSDL documents before mapping them to vectors. To address these problems we have designed some text-mining preprocessing techniques that take into account the particularities of service descriptions. Fig. 5 shows the overall text-mining process, which is used by Algorithm 1, line 2. We have developed a parser that extracts and preprocesses the textual description within a WSDL file by taking advantage of its well structured format. We have implemented this tool in Java by using the WSDL4J5 and WSIF6 libraries. The tool parses a WSDL document and pulls out the comments associated with the service, the operations and their arguments (second step of Fig. 5).

In the third step, by executing a type enlargement algorithm our approach attempts to bridge the different encoding styles mentioned. Here, if an argument data-type is non-primitive (i.e., is not an integer, string, boolean or float) the tool looks for the definition associated with this composed type and invokes a type enlargement algorithm, which extracts the names of the elements that are encapsulated in the composed type. This algorithm receives an XSD type definition that combines some subtypes by using the XSD encapsulation constructors, e.g., xsd:element, xsd:sequence, xsd:complexType. By using these constructors a subtype is associated with a name, then the algorithm returns these names. For example, in Fig. 4 the argument named currency is defined according to the document/literal wrapped style, then the algorithm must extract the term “currency” from an xsd:sequence construct, which is within an xsd:complexType tag, which in turn is within an xsd:element tag. Instead, by using the document/literal encoding style for defining the same argument, as listed below, the algorithm has to extract the term “currency” from two xsd:elements tags:

```xml
<types>
  <schema>
    <element name="srcCurrencyElement" type="xsd:string"/>
    <element name="destCurrencyElement" type="xsd:string"/>
  </schema>
</types>

<message name="myDocumentStyleArg">  
  <part name="srcCurrency" element="srcCurrencyElement"/>
  <part name="destCurrency" element="destCurrencyElement"/>
</message>
```

The type enlargement algorithm defines how to expand xsd:complexType and xsd:element types, and then defines rules to break down other cases into these base cases. Type enlargement occurs at most at one level of depth, instead of fully recursively, limiting the overall complexity of the algorithm. Arrays of “something” are handled in the same way too. Algorithm 2 describes our type enlargement method. Though this algorithm is simple, it is very useful for mining significant words within complex type definitions.

In the fourth step, by splitting combined words our approach attempts to bridge different naming conventions. In general, developers combine a verb and a noun for denoting the name of an operation, such as getQuote or get_quote.

---

5 WSDL4J http://sourceforge.net/projects/wsdl4j.
1: \textbf{procedure} ENLARGE(type, follow) \>
Returns a String[]
2: \textbf{if} \! follow \textbf{then}
3: \> r ← type.name
4: \textbf{else if} type is complex \textbf{then}
5: \> \textbf{for all} \( s \in \text{type.sequenceElements} \) \textbf{do}
6: \> \> r ← r + s.name
7: \textbf{end for}
8: \textbf{else if} type is element \textbf{then}
9: \> \textbf{for all} \( c \in \text{type.children} \) \textbf{do}
10: \> \> \> r ← r + ENLARGE(c, false)
11: \textbf{end for}
12: \textbf{end if}
13: \textbf{return} r
14: \textbf{end procedure}

Algorithm 2. Type enlargement.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Rule</th>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Beans</td>
<td>Splits when changing text case.</td>
<td>getZipCode</td>
<td>get Zip Code</td>
</tr>
<tr>
<td>Hungarian</td>
<td>Splits when changing text case.</td>
<td>ulAccountNum</td>
<td>ul Account Num</td>
</tr>
<tr>
<td>Special symbols</td>
<td>Splits when either “_” or “-” occurs.</td>
<td>get Quote</td>
<td>get Quote</td>
</tr>
</tbody>
</table>

Table 1 Rules for splitting combined words

Then, every distinct operation and message that follows each naming denomination would be treated as a different word. In order to bridge different naming conventions, our tool searches for combinations of words and splits them into verbs and nouns. We also take into account combinations of more than two words, e.g., for the combined word “getQuoteFor” the tool separately dumps the words “get”, “quote” and “for”. Table 1 summarizes the rules for splitting combined words.

In the fifth step, by removing symbols and stop words our approach attempts to “clean” service descriptions. A stop word is a word with a low level of “usefulness” within a given context or usage. We use a list of about 600 English stop words and a small list of stop words related to the Web services domain, such as request, response, soap and post. In general, removing special characters and stop words is a simple but an useful technique for filtering non-relevant terms [48]. Nevertheless, the stop word elimination process is improved by splitting combined words. Because of developers’ coding conventions, some stop words are not removed. Besides, operations and arguments having similar features are treated as being non-similar. By splitting combined words many stop words are removed, many relevant nouns arise and many words are recognized as being common (non-relevant). Finally, we utilize the Porter Stemming algorithm [49] for removing the commoner morphological and inflectional endings from words, reducing English words to their stems.

We will show the text-mining preprocessing stage of WSQBE with an example using the description of a Web service for finding currency exchange rates (see Fig. 4). Table 2 shows two vectors of word occurrences. The vector on the left, contains the extracted words. Conversely, the vector on the right contains the stems generated by applying the preprocessing techniques. By utilizing our text-mining process we pulled out all relevant stems. For example, by preprocessing the message “GetRateSoapOut” we removed the stop words “get”, “soap” and “out”, while the relevant word “rate” arose. On the other hand, by expanding element data-type definitions we included more occurrences of the relevant words in the result. For example, by expanding the data-type associated with the message “GetRateSoapIn” we derived the stems “src”, “currenc”, “dest” and “currenc” from the combined words “srcCurrency” and “destCurrency”. The inclusion of each occurrence of a relevant word might be important for the performance of the Web service classifier and vector matching method.

In Section 4, we will evaluate how this text-mining process impacts on the performance of WSQBE discovery approach.
3.3. Reducing the search space

To reduce the search space, we propose a document classifier that deduces a category from a number of predefined categories such as business, educational, finance, scientific, etc. Document classification refers to the process of assigning an electronic document to one or more categories based on its contents [50]. Automatic document classifiers support classification of documents seen as objects characterized by features extracted from their contents. In general, when some external mechanism, such as human feedback, provides information on the correct classification for documents, we talk about supervised document classification. A supervised document classification approach consists of two phases: (1) training phase, and (2) classification phase. During the training phase, such a learning system receives a collection of categorized documents and builds a classifier. Then, during the classification or evaluation phase, this classifier deduces one or more categories for a new document, based on features extracted from this document.

The cornerstone behind WSQBE supervised classification approach is the fact that there are dependencies between the category of a Web service and its description. As in a related work [41], WSQBE considers the dependencies between the category of a Web service, the operations and their input and output arguments. In object-oriented terms, WSQBE assumes that:

1. The category of a Web Service depends on its textual comments
2. and its method signatures.

The assumption number (1) has been studied and properly evaluated before [41,42]. On the other hand, according to the mentioned different coding conventions used for defining the signature of a Web service operation, the hypothesis number (2) is not obvious unless we attempt to bridge different encapsulation and naming approaches. In a previous paper [51], we reported several experiments for assessing the degree of dependency between the category and the signature of an operation. These experiments showed that the degree of dependency increases proportionally to the number of times an argument appears in the services of this particular category, but this dependency is offset by how common the argument is in the whole collection. By treating each preprocessed WSDL file as a document we have developed a classifier in the context of Web services. In this way, we first employ the preprocessing method described in Section 3.2 for pulling out all stems contained in each WSDL file. Then, the resulting collection of stems stands for a document associated with a Web service. Second, we have developed a VSM implementation in Java for representing these documents in a multi-dimensional space and building the classifier.

There are some different learning algorithms for automatic classification of documents [38]. Rocchio is a learning algorithm, originally designed to use relevance feedback in querying full-text databases [52], which has been adapted to text classification [53]. The overall phases of this classification algorithm are described subsequently. Initially, every document is represented as a vector \( \vec{v} = (e_0, \ldots, e_n) \). Each element \( e_i \) represents the importance of a distinct word \( w_i \) for that document. A word weighting method is used for measuring this importance. Then, during the training phase, the classifier divides the vector space into subspaces, one for each category \( \vec{c}_i \). A subspace is centered on an average vector, known as centroid, computed as the center of mass of all documents in the category. This vector stands for the documents that belong to category \( i \). Formally:

\[
\vec{c}_i = \alpha \sum_{d \in C_i} \vec{d} \sum_{d \in C_i} \vec{d} - \beta \frac{\sum_{d \in C_i} \vec{d}}{|D - C_i|}
\]
public interface CurrencyConverter {
    double getRate(String srcCurrency, String destCurrency);
}

with $C_i$ being the subset of the documents from category $i$, and $D$ the number of documents of the entire data-set. First, both the normalized vectors of $C_i$, i.e. the positive examples for a class, as well as those of $D - C_i$, i.e. the negative examples for a class, are summed up. The centroid vector is then calculated as a weighted difference of the positive and the negative examples. The parameters $\alpha$ and $\beta$ adjust the relative impact of positive and negative training examples. As suggested by [54], we use $\alpha = 16$ and $\beta = 4$. During the evaluation phase, a new document is represented as a vector and then compared to the vectors associated with all categories by using cosine similarity [38]. Finally, the category which maximizes vector similarity is selected (see Algorithm 1, line 4).

Some efforts have been made on word weighting approaches. A comparison of these alternatives can be found in Salton et al. [55]. WSQBE uses TF-IDF because this heuristic has been shown to be suitable for weighting words present in Web service descriptions [20]. TF-IDF is a word weighting heuristic that combines a document term frequency factor (TF) with a collection-dependent factor, namely inverse document frequency (IDF). TF determines that a word is important for a document if it occurs often in it. On the other hand, words which occur in many documents are rated as less important because of their IDF. Formally: $tfidf = tf \cdot idf$ with:

$$tf_i = \frac{n_i}{\sum_k n_k}$$

with $n_i$ being the number of occurrences of the considered term, the denominator is the number of occurrences of all terms, and:

$$idf_i = \log \frac{|D|}{|\{d : d \ni t_i\}|}$$

where $|D|$ is the total number of documents in the corpus and $|\{d : d \ni t_i\}|$ is the number of documents where the term $t_i$ appears. Despite the good precision of TF-IDF when used with a document classifier [23], this heuristic has a drawback. The $idf_i$ component is a collection-dependent factor, i.e., a global factor, thus the whole vector space has to be recalculated when a new document is added to the corpus. As we will show in Section 4 we have also evaluated our approach by using a simplified heuristic, namely TF (Eq. (1)), that does not have this problem and has little negative impact on WSQBE performance. To sum up, the third line of Algorithm 1, namely PREPROCESS(stems) uses either TF or TF-IDF to convert the list of stems returned by the text-mining process (Fig. 5) to a vector representation.

In Section 4, we will show how the accuracy of the proposed document classification technique for Web services surpasses that of a previous work based on Naïve Bayes [56] and Support Vector Machine [57] techniques.

3.4. Discovery assistance: Minimizing discoverers’ effort

WSQBE aims at hiding Web service technology details from SOA developers. To do this, WSQBE allows discoverers to ask for services by providing an example in the form of a: (1) a natural language description of the expected service or its operations, (2) a functional description of these operations. In particular, WSQBE accepts an object-oriented description of such operations. For example, WSQBE accepts either a natural language description: “Currency Converter”, or the next Java Interface as a query:

Natural language queries provide a “Google-like” inquiry interface, which most people are familiar with. On the other hand, by accepting the abstraction mechanism of a widely used programming language, WSQBE exempts Java developers from putting an additional effort into query creation. In this way, developers are unaware of UDDI existence or published Web services organization. Furthermore, this way of describing a desired operation is very familiar to programmers and relieves them from learning another query language.

Similarly to the syntactic differences that occur in Web service descriptions, we deal with these at the query-side. To reuse the WSDL parser described in Section 3.2 we have implemented a query processor that transforms a Java interface into a WSDL document by using the Java2WSDL\(^7\) library. The current version of Java2WSDL maps the

\(^7\) Java2WSDL http://ws.apache.org/axis/.
methods of a Java interface named “A” onto operations of a Web service named “A”. Although this tool maintains the names of the interface and its methods, the names of the parameters are removed at the translation stage. Then, the resulting WSDL documents do not contain meaningful parameters, however they use a numerical naming convention, e.g., param0, param1, and so on. Therefore, we omitted the evidence associated with parameter names in the queries, which could harm the precision of our vector matching approach because this numerical convention reduces any meaningful different term to a simple “param”.

There are several programming-language-dependent tools that generate a WSDL document from an object-oriented specification automatically, such as Java2WSDL or the Web Services Description Language tool.\(^8\) WSQBE has been designed to easily integrate different translation tools. Therefore, WSQBE can support other programming languages for expressing queries.

After generating the WSDL document that stands for a query, WSQBE uses the text-mining techniques described in Section 3.2, i.e., WSQBE preprocesses the example. WSQBE preprocesses natural language queries avoiding the steps for generating a WSDL document, parsing it and bridging its data-types (steps 1, 2 and 3 in Fig. 5). As a result, WSQBE generates a vector representation from the resulting collection of terms. Finally, the problem of finding services relevant to the example is reduced to looking for similar vectors in the vector space.

In the next section we will show the feasibility of the proposed discovering approach by analyzing its performance when using Java interfaces as queries.

4. Evaluation

This section describes the experimental evaluation of WSQBE. First, we evaluated the accuracy of our classification approach through different tests. Here, we analyzed the impact of our text-mining method on the classification results. Second, we examined the overall performance, in the perspective of accuracy, of our QBE retrieval approach using two term weighting techniques. Third, we took time metrics on the implementation of WSQBE in an attempt to have an assessment of the overhead introduced by its text-mining preprocessor and its classifier with respect to not using it. Finally, we measured how WSQBE behaves when the number of published services grows.

4.1. Classifier evaluation

First, we have shown that our TF-IDF classifier accuracy averages 91% with a tolerance value of 3, this is, considering the first three deductions, and it performs better than a TF classifier and other classification approach named Assam [41] using the same data-set. In the experiments we used a Rocchio classifier with a global term weighing technique (TF-IDF, see Eq. (2)) and a local one (TF, see Eq. (1)). The goal was to assess the computational complexity versus precision trade-off. On the one hand, TF-IDF is well-known for achieving good document retrieval precision, but with a high computational cost for adding new documents. On the other hand, TF can be computed very fast, but precision may be compromised. In addition, we compared these Rocchio classifiers against Assam [41], which is based on two well-known classifiers such as Naive Bayes and Support Vector Machine.

To compare our classification approach against Assam, we reproduced the experiment reported in [41] with the same data-set and evaluation methodology, but using our classifier. Assam was evaluated by its authors with a group of 391 Web services divided into 11 categories, which has been made publicly available.\(^9\) Moreover, its accuracy was evaluated with a tolerance value and by driving a leave-one-out experiment. A tolerance value of \(t\) represents that the correct classification is included in a sequence of \(t + 1\) deductions. Leave-one-out is a cross-validation method for evaluating classifiers [58]. Basically, one observation is arbitrary chosen from the initial sample to form the validation data and the remaining observations are retained as the training data. Therefore, we preprocessed the 391 Web services. Afterward we built the classifiers and conducted a leave-one-out experiment for the TF-IDF classifier and another for the TF classifier. WSQBE TF-IDF classifier accuracy consistently surpassed Assam accuracy in, at least, 9% with two values of tolerance: \(t = 0\) and \(t = 1\). Conversely, WSQBE TF classifier accuracy surpassed Assam accuracy in 2%, with \(t = 0\) and \(t = 1\), but their accuracy converged at \(t = 3\). The results are shown in Fig. 6. Note that these results may vary with different data-sets. In addition, it would be desirable to compare the standard deviation statistic for...
of Assam and WSQBE, however to the best of our knowledge the authors did not report it in [59] or [41]. Also, we have shown that by applying our text-mining process to a group of 391 Web services, WSQBE improves the overall accuracy of the Rocchio TF-IDF classifier in, at least, 5%. To compare the accuracy obtained by using type expansion and verb noun separation versus not using them, we conducted another experiment. We first adapted our text-mining process for omitting the techniques just mentioned. The resulting extracted text differed from the original results in having more unique words per category. For example, there were 791 unique terms in the "business" category after preprocessing, and 2175 words before. This was caused by the inclusion of many irrelevant words, as explained in Section 3.2. The inclusion of such irrelevant words does not contribute to the classifier. On the contrary, it is expected that such words could harm the classifier performance [48]. At this point, we had two collections of documents, the collection of preprocessed Web service descriptions and the collection of Web service descriptions per se. Therefore, we trained two Rocchio TF-IDF classifiers for running tests with each one. Fig. 6 shows the results. Although these results cannot be generalized to other data-sets, our text preprocessing approach does not depend on the data-set used in the evaluations. As a consequence, it is reasonable to expect at least a small precision advantage when using our text preprocessing approach with other data-sets, versus not using it.

4.2. Query-by-example evaluation

Second, we have evaluated WSQBE performance in the perspective of accuracy. Again, we compared TF-IDF with TF, in order to assess the computational complexity versus precision trade-off. In our evaluation, the data-set has 391 Web services divided into 11 categories. Although each category has its own services population, the most populated category has 65 services and there is an average of 40 services per category. Our experiments have shown that by using WSQBE a discoverer selects a proper service from a set of only 10 WSDL documents using TF-IDF and TF indistinctly. In fact, this set is an ordered list where services having a higher confidence in being relevant to the query are located at the top. As a result, he/she sequentially examines, at worst, 10 WSDL documents before finding one relevant service. We measured the average position of the first relevant services within the retrieved results and it was 1.83 and 2.2 using TF-IDF and TF respectively. In consequence, a discoverer examines only 2 WSDL documents on the average case, and 10 WSDL documents on the worst case, for our data-set. Clearly, although these results cannot be generalized to other data-sets, the results obtained are promissory. We conducted 30 experiments to evaluate the precision of our discovery approach. Basically, for each experiment we built a query, i.e., an example, and manually examined the WSDL documents of the retrieved set. A software developer judged the retrieved WSDL documents in response to each example. For these 30 tests we generated the examples in Java, by exploiting the interface abstraction mechanism and the Java2WSDL tool. As we explained in Section 3.4, we omitted the evidence associated with parameter names in the queries. In addition, we did not take into account natural language descriptions in the queries. In fact, we focused on measuring the performance of WSQBE with very short descriptive queries. Table 3 summarizes the size of each query in terms of the number of defined methods and resulting stems, for example the query presented in Section 3.4, namely CurrencyExchanger, comprises 1 method and 6 stems.
A significant detail regarding the evaluation of WSQBE is the definition of "hit", i.e., when a returned WSDL document was relevant to the user. The user determined that if the operations of a retrieved WSDL document fulfilled the expectations declared in the Java example then it was a hit. For example, if he/she expected an operation for converting from Euros to Dollars, then a retrieved operation for converting from Francs to Euros was non-relevant, even though these operations belonged to the same category or they were strongly associated. Conversely, in this case only operations for converting from Euros to Dollars were relevant. This definition of hit makes the validation of WSQBE stronger than a related work [20], which defines that two services are relevant to each other if they belong to the same category.

While there are some different methods for evaluating the performance of a retrieval system, we measured WSQBE performance in terms of the proportion of relevant services in the retrieved list and their positions relative to non-relevant ones. In this sense, we assessed $R$-precision, Recall and Precision-at-$n$ measures. The rest of the section present the results of assessing these three statistics.

### 4.2.1. $R$-precision

One of the most used measures of retrieval performance is $R$-precision. Basically, given a query with $R$ relevant documents, this measure assesses the precision at the $R$th position in the ranking. For example, if there are 10 documents relevant to the query within the data-set and they are retrieved before the 11th document, we have a 100% precision, but if 5 of them are retrieved after the top 10 we have a 50% precision. Formally:

$$R \text{ precision} = \frac{\text{RetRel}_R}{R}.$$ 

We obtained the $R$-precision for each experiment and then we averaged these results. The averaged $R$-precision of the TF-IDF experiments was 72%. This result has shown that WSQBE with TF-IDF included 72% of the relevant services at the top of the list. In fact, WSQBE included 72% of the relevant services before including a non-relevant service. On the other hand, the averaged $R$-precision of the TF test was 66%.

### 4.2.2. Recall

Recall is a measure of how well the engine performs in finding relevant documents [38]. Recall is 100% when every relevant document is retrieved. Formally:

$$\text{Recall} = \frac{\text{RetRel}}{R}$$

where $R$ is the total number of relevant services to a query in the collection. By returning every document in the collection for every query we could achieve good recall, but looking for relevant services in the entire collection would be a cumbersome task. Conversely, we want to achieve good recall in a window of only 10 retrieved services. Therefore, we measured the recall for each test when $\text{RetRel} = \text{RetRel}_{10}$. The averaged Recall of the TF-IDF experiment was 91%. This result has shown that, in general, for every query WSQBE included relevant services in the top 10 retrieved services. On the other hand, the averaged Recall of the TF experiment was 95%.
4.2.3. Precision-at-\(n\)

Precision-at-\(n\) measure allows computing precision at different cut-off points [29]. For example, if the top 10 documents are all relevant to the query and the next 10 are all non-relevant, we have 100% precision at a cut-off of 10 documents but a 50% precision at a cut-off of 20 documents. Formally:

\[
\text{Precision at } n = \frac{\text{RetRel}_n}{n}
\]

where \(\text{RetRel}_n\) is the total number of relevant services retrieved in the top \(n\). We evaluated Precision-at-\(n\) for each experiment with six values of \(n\): \(n = 1, n = 2, n = 4, n = 6, n = 8\) and \(n = 10\). Fig. 7 shows the averaged Precision-at-\(n\) for WSQBE discovery approach. On the one hand, TF-IDF results have shown that 83% of the services at the top of the list were relevant. On the other hand, TF precision-at-1 was 73%. For all the examples there are, at worst, 8 relevant services within this data-set. Besides, there are 10 examples that have only one relevant service associated. Clearly, this particularity of the data-set harms the precision of our approach as \(n\) and the number of retrieved services increases. Note that for larger data-sets with many relevant services for each query, Precision-at-\(n\) is expected to increase with \(n\) until \(n\) is large enough to allow most relevant services to enter the top \(n\) retrieved services.

4.3. Processing time and memory usage

Finally, the averaged overhead introduced by WSQBE when publishing a new Web service has been empirically shown to be 14.34 milliseconds (ms) over doing this task without WSQBE. Moreover, we have shown that the overhead introduced by WSQBE when discovering Web services is 13 ms, on the average case.

On the one hand, to measure the time penalty for publishing a WSDL document we obtained the time required for preprocessing each document of the data-set. The processing time required by UDDI was not measured, because we wanted to analyze how WSQBE impacted over the performance of classic publication and discovery tasks. In order to mitigate any noise introduced by external conditions that could influence this performance test, each WSDL file was preprocessed 10 times. The CPU time demanded by each preprocessing execution was measured and the average was computed. On the other hand, to evaluate the overhead introduced when discovering services we measured the time required by WSQBE to answer the 30 queries described in Section 4.2. Again, we performed 10 executions for each query and then the average was computed.

Both publishing and discovering evaluations were deployed on an Intel Pentium D working at 3.0 GHz with 1.0 GB of RAM. Here, WSQBE and jUDDI\(^\text{10}\) (an open source UDDI v.2 implementation) were running on Sun JVM 1.6.0_02 and Ubuntu Linux 7.10. It is worth noting that both WSQBE and jUDDI along with the data-set were hosted at the same computer, to avoid the overhead introduced by the network. Clearly, in a real word scenario, the Web service registry, publishers and discoverers may be spread over the Internet. Conversely, WSQBE and a UDDI often reside at

\(^{10}\) jUDDI \(\text{http://ws.apache.org/juddi/}\).
the same server machine or, at least, at the same local area network, in which the cost of networking is inexpensive. Then, the cost associated with forwarding a UDDI request from WSQBE can be safely ignored.

Moreover, we evaluated the scalability of WSQBE, under the aforementioned deployment, with regard to memory usage and processing time when the number of published Web services grows. On the one hand, the averaged memory usage of the JVM has been empirically shown to be 9.76 Megabytes (Mb), with a standard deviation of 2.78 Mb. The building block of WSQBE implementation is a “vector”, in which each entry is a weighted term. Then, a VS is a collection of vectors and a partitioned vector space is a collection of VS. In consequence, it is expected that the more the terms, the more the memory usage. In this sense, Fig. 8 (a) shows the correlation between the number of published Web services (x axis) and the number of different stems (y axis), while Fig. 8 (b) shows the memory usage variations. As the reader can see, initially the number of different stems linearly grows with the number of published services. However, the number of stems converges on the sum of different terms of each individual category or sublanguage. With respect to memory usage, the experiments show that WSQBE had a consumption peak of 15.2 Mb with 211 published services. Afterward, the memory usage oscillated between 4 Mb and 15 Mb. As shown in Fig. 8 (b) memory usage was influenced by the underlying JVM mechanism for recycling unused objects (a.k.a. garbage collector).

On the other hand, to measure the impact of the number of published services over the time processing, we arbitrarily picked n services and published them in WSQBE. Then, we measured the time required by WSQBE to answer the aforementioned 30 queries. We measured the processing time for each query with seven values of n:

\[ n = 60, \quad n = 120, \quad n = 180, \quad n = 240, \quad n = 300, \quad n = 360 \text{ and } n = 391. \]

Again, we built 10 data-sets for each value of n. Fig. 9 (all data points on the left and only the average on the right) shows the resulting times for each value of n. As the reader can see, the more the published services, the more the vector comparisons, and therefore the more the processing time for discovering. As expected, the JVM memory management impacted on the variability of the results. This result is coherent with the explanation of time complexity presented in Section 3.1.

4.4. Discussion

Note that in these experiments R-precision, Recall and Precision-at-n were measured taking into account only the first category deduced by the classifier. As a consequence, these metrics are influenced by the precision of the two steps of WSQBE, namely, the classifier and the document retriever. As the tolerance-based results have shown (see Fig. 6), by taking into account the services that belong to the second and third deduced categories, WSQBE might have even better results. As a consequence, WSQBE may be tuned to consider alternative subspaces only if the similarities between the query and the services of the first deduced subspace are below an acceptance threshold. In other words, analyze the next subspace when no similar services have been found within previous subspaces. Besides, this modification is still more efficient than comparing a query against the whole vector space. Obviously, the more the categories considered during a search, the more the number of vector comparisons.
To sum up, the results of the experiments have shown that our classification approach achieves good accuracy. We have shown that Rocchio with either TF-IDF or TF is a proper classification technique in the Web services context for our data-set. In addition, our text-mining techniques have improved the classifier accuracy in, at least, 5% for the same data-set. Also, we have shown that our approach to aid the discovery process effectively reduces discoverers’ effort with regard to two dimensions. First, we have shown that WSQBE assists discoverers by generating a short and accurate list of candidate services, and second, WSQBE makes query specification very easy. Finally, we have empirically shown that the current implementation of WSQBE imposes averaged time penalties of 14.34 ms and 13 ms when publishing and discovering services, respectively, with this data-set and queries. Moreover, we have shown that memory usage depends on the number of dimensions of the VS, which converges on the sum of the different terms of each individual category. For instance, by publishing all the services of the data-set, the averaged memory usage was 9.76 Mb. Finally, we have shown that the discovery time increases linearly as the number of published services grows. Note that the implementation of WSQBE is not optimized, for example, it neither uses indexes nor takes advantage of multi-core processors for comparing vectors in parallel.

5. Conclusions

Distributed applications may gain high levels of interoperability, flexibility and reuse by relying on Web service technologies. However, as the amount of published Web services grows, discovering proper services becomes harder. This paper presented a novel search method for Web services called WSQBE. This method combines current technologies for describing and publishing Web services with information retrieval techniques in order to filter and evaluate available services automatically. This combination has shown that by representing Web services and queries as a collection of vector subspaces, Web service retrieval is computationally efficient, even with many available services. Moreover, reported evaluations have shown that WSQBE alleviates discoverers by narrowing down the number of candidate services from 40 to only 1 in 83% of the queries using a data-set with 391 Web services. In fact, these evaluations confirm that queries can be effortlessly derived from existing service-oriented applications’ source code and, despite WSQBE lightweightness with respect to structural and semantic methods, achieves good precision for discovering relevant services. Though the automatic classifier may introduce errors and categories may overlap, tolerance-based results have shown that by examining the services that belong to several deduced subspaces, classification errors and category overlapping can be mitigated at the expense of increasing the number of vector comparisons. WSQBE has been empirically shown to impose an overhead of 14.34 ms and 13 ms over classic publication and discovery tasks with this data-set, respectively. Besides, scalability experiments have suggested that memory usage converges on a maximum value, which is actually a desired feature in real word applications, and also vector comparisons may be done in parallel to enhance performance.
Although we have evaluated WSQBE with a well-known set of Web services, the reported results may vary with different data-sets. As a consequence, WSQBE should be further tested with other data-sets. We are planning to further test its discovery mechanism with a recently published data-set\(^{11}\) of 365 real Web Services [60].

The main limitation of WSQBE is that it assumes that a corpus of previously classified services is available. This generates the inability for handling dynamic creation of categories without re-building the classifier. To cope with such a requirement, an incremental clustering [61] approach might be more suitable than a classification one. Besides, although WSQBE attempts to bridge syntactic differences, its heuristic for comparing two Web services is still syntactic. For instance, suppose there are two Web service operations for IP location. One takes “ip address” as input and returns “latitude / longitude”. The other takes “hostname” as input and returns “zipcode”. Clearly, WSQBE will place these operations far apart in the VS, deteriorating discovering effectiveness, unless the names of these operations and their documentation near the vectors. As a consequence, the accuracy of the approach depends on publishers’ use of best practices for naming and documenting services, operations and arguments.

This work will be extended in several directions. In the first place, we will extend WSQBE capabilities for query generation. In this sense, we will incorporate parameter descriptions (e.g., names and data-types) into our process for pulling out queries from applications’ source code. We expect that the more expressive queries get, the more accuracy WSQBE would achieve. In the second place, we will evaluate more sophisticated similarity heuristics (e.g., constraint-based, schema matching, similarity degree based, etc.). Although some of these heuristics have a polynomial computational complexity, we think that it could be feasible to employ them on a small number of candidate services. Therefore, we plan to incorporate these heuristics into the third step of WSQBE to match a handful set of services only and further reduce discoverers’ effort. In the third place, we will investigate how to scale WSQBE to handle millions of services by decentralizing its algorithms and integrating it with a peer-to-peer infrastructure.

Finally, we are planning to incorporate WSQBE support for automatically discovering relevant services given a Java interface into JGRIM [62], an infrastructure for easing the development of applications for service-oriented Grids. The goal is to allow developers to invoke Web services as easily as local Java methods, this is, without dealing with Web service technology details, including discovering.

Acknowledgements

This work is partially supported by the National Council of Technical and Scientific Research of Argentina (CONICET). We also thank Cristian Mateos and Pablo Gotthelf who contributed to improving the presentation of this work.

References


\(^{11}\) The QWS Dataset http://www.uoguelph.ca/~qmahmoud/qws/index.html.


