Science of Computer Programming 77 (2012) 355–374

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

WSDL term tokenization methods for IR-style Web services discovery

Chen Wu

ICRAR, The University of Western Australia, Australia

ARTICLE INFO

Article history:
Received 27 October 2010
Received in revised form 11 August 2011
Accepted 16 August 2011
Available online 7 September 2011

Keywords:
Web services discovery
Information Retrieval
String tokenization
Source code mining
Data engineering

ABSTRACT

The IR-style Web services discovery represents an important approach that applies proven techniques developed in the field of Information Retrieval (IR). Many studies exploited the Web Services Description Language (WSDL) syntax to extract useful service metadata for building indexes. However, a fundamental issue associated with this approach is the WSDL term tokenization. This paper proposes the application of three statistical methods for WSDL term tokenization—MDL, TP, and PPM. With the increasing need for effective IR-style Web services discovery facilities, term tokenization is of fundamental importance for properly indexing WSDL documents. We compare our applied methods with two baseline methods. The experiment suggests the superiority of MDL and PPM methods based on IR evaluation metrics. To the best of our knowledge, our work is the first to systematically investigate the issue of WSDL term tokenization for Web services discovery. Our solution can benefit source coding mining, in which a key step is to tokenize names (i.e. terms) of variables, functions, classes, modules, etc. for semantic analysis. Our methods could also be used for solving Web-related string tokenization problems such as URL analysis and Web scripts comprehension.

1. Introduction

Web services discovery is important for service programming activities including reusing, sharing, binding, and composition. The notion of Service-Oriented Architecture (SOA) and loose-coupling relies on the assumption that programmers are able to dynamically discover the most relevant Web services to suit their application needs. Web Services Description Language (WSDL) is a well-adopted standard for describing a Web service's functional capability and technical specifications "on the wire" [1]. While efforts have been made to augment WSDL documents with precise semantics [2], logic and rules [3], removed anti-patterns [4], various annotations [5], and diverse data and model characteristics [6], many existing methods (discussed in Section 3) exploited WSDL documents 'as-is' to discover useful characteristics that facilitate service discovery. In adopting this approach, an increasing number of studies used various Information Retrieval (IR) techniques to search textual service metadata for service discovery. For example, Wang and Stroulia [8] employed the inverted file [7] to index and search natural language description of desired services. Similarly, Platzer and Dustdar [9] used the Vector Space Model (VSM) to build a Web service search engine, and Sajjanhar et al. [10] have attempted to leverage Latent Semantic Analysis (LSA), a variant of VSM, for Web services discovery. All of the above work index service metadata found either in a WSDL file (i.e., the <documentation/> tag) or from a Universal, Description, Discovery, and Integration (UDDI) registry entry. In both cases, service metadata written in English are manually created by service providers.

More recent research (e.g., [11,12]) extracted textual information from the WSDL syntax (e.g., the attribute/value tag) and added them as service metadata. While this method is able to provide more information to IR techniques, it introduces the term tokenization problem (See the problem description in Section 2). Unlike a human natural language, WSDL is an XML-based interface language processed automatically by machines. The naming constraints on 'NMTOKEN' in WSDL

E-mail address: chen.wu@uwa.edu.au.

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.08.001

Table 1
Examples of NMTOKENs in real-world WSDL documents (fetched on 18 April 2011).

	WSDL URL	NMTOKEN	Heuristics tokenization based on naming tendencies
1	http://developer.ebay.com/webservices/latest/ebaySvc.wsdl	GetMyeBayBuyingRequest	Get Mye Bay Buying Request
2	http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl	AWSECommerceService	AWSE Commerce Service
3	https://rds.us-east-1.amazonaws.com/doc/2010-06-28/AmazonRDSv3.wsdl	EC2SecurityGroupOwnerId	EC 2Security Group Owner Id
4	http://api.hi5.com/hi5.wsdl	hi5appProvider hi5AuthToken	hi 5app Provider hi 5Auth Token
5	http://182.72.145.196/Cash2anyoneAPI/cash2anyone1.wsdl	iPhoneDeviceLogoutRequest UpdateeWalletResponse	i Phone Device Logout Request Updatee Wallet Response
6	http://www.lexus.com.au/services/configurator/configurator.wsdl	vehiclePositionIPad	vehicle Position I Pad
7	http://ipaddressinfo.com/ip/whois-service.wsdl	getMD5forIPwhois	get MD 5for I Pwhois
8	http://ws.xwebservices.com/XWebACHDirectory/V1/XWebACHDirectory.wsdl	DirectorySOAP12Port	Directory SOAP 1 2Port

specification follow the XML standard. More specifically, W3C [13] states that the production of NMTOKEN (i.e. Nmtoken) is composed of NameChar defined as:

Letter|Digit|'.'|'-'|'_'|':'|CombiningChar|Extender

which does not include '#x20'—the Unicode character 'WHITE SPACE'. This is because '#x20' is reserved to form multiple Nmtokens, i.e.

Nmtokens ::= Nmtoken (#x20 Nmtoken)* [13].

As a result, one must tokenize XML tag '<operation name="getlasttradeprice">' into 'get', 'last', 'trade', 'price' rather than using the whole NMTOKEN—'getlasttradeprice'. Tokenization here becomes crucial for IR techniques to function properly. This is because most IR methods (e.g., the inverted indexes) do not perform a full-text online searching for efficiency. They use string matching algorithms to look up a well-crafted index containing a set of pre-defined terms prior to the searching process. A query term that was somehow not included in this index will not produce any matches. Suppose a Web service operation name 'getlasttradeprice' was added to the index instead of two separate terms: price or trade. A service discovery request (query) on "trade price", which is often issued by a human (e.g., a software developer) who is used to placing white spaces between two terms, will not be able to find this service even though it is of high relevance.

To solve this problem, Kokash et al. [11,12] used three heuristics (e.g., the pattern of uppercase/lowercase letters as discussed in Section 3.2) to tokenize NMTOKEN strings before indexing. These heuristics are based on the observation that human developers actually follow upper/lower case and the "_" naming convention as suggested in [14]. Interestingly, based on our experience of analyzing some thousands real-world WSDL documents, we found that some heuristics are not so effective. Consider NMTOKENs extracted from eight real-world WSDL documents as shown in Table 1.

In Table 1, WSDL #1, #2, and #5 show that the upper/lower heuristics produces either misleading (e.g. Pad, Phone, Bay) or incorrect (e.g. Mye, Updatee, AWSE) index terms. It appears that heuristics will produce an incorrect tokenization for both iPad and IPad. Like software component APIs, numerous WSDL documents contain digits in their operations, messages, or part names. The Kokash heuristics does not work well in these NMTOKENs as shown in Table 1. For example, for WSDL #3, EC2 (vs. EC and 2Security) is Amazon's Web service. In WSDL #4, hi5 (vs. hi) is the name of a company, and neither 5app nor 5Au makes too much sense for searching purposes. MD5 is the hashing function, which becomes MD and 5for in WSDL #7. The important SOAP version information 12 is somehow lost after tokenization in WSDL #8.

It is true that one can devise extra heuristics based on variations of subtle human naming tendencies to tokenize NMTOKENs shown in Table 1. However, enumerating and predicting all possible naming tendencies can be very challenging (if not impossible). After all, creating NMTOKENs is at the hand of different service developers, who are beyond the control of any tokenization algorithms. This suggests that heuristics based on naming tendencies are at least less reliable and hence sub-optimal to 'decipher' irregular, non-word terms and acronyms abundant in WSDL documents of various types.

To address this issue, we adopted a different approach to the WSDL term tokenization problem. In this paper, we propose the use of statistical models to fit latent, higher-order data regularity that is independent of apparent naming tendencies.

1 The ticker symbol is an input.

2 The trade price is an output.

3 Get the last trade price with the input sticker symbol.

4 Get the last trade price with the output trade price.

5 The stock quote is the last trade price for a stock.

Fig. 1. The Stock text database consisting of five online documents.

To thoroughly evaluate a method that is completely orthogonal to any naming tendencies such as upper/lower cases and delimiters (e.g. "_"), we used a synthetic dataset (sawsdl-tc1 [15]), in which NMTOKENs do not always follow normal human naming tendencies. For example, NMTOKENs such as "BankeraddressSoap", "DroughtreportService" are abundant in this data collection. We believe this dataset is useful in assessing our methods' unique capability, which is absent in existing heuristics-based tokenization methods. Without loss of generality, we also used another real-world dataset (Assam [16]) to compare our methods with two existing baseline methods that utilize heuristics on human naming tendencies.

Based on the problem description in Section 2 and related work in Section 3, we first briefly introduce the overall IR-style Web services discovery approach in Section 4. We then discuss the use of three WSDL term tokenization methods—Minimum Description Length (MDL), Transitional Probability (TP), and Prediction by Partial Matching (PPM) in Sections 5, 6, and 7 respectively. The MDL method incrementally constructs a knowledge base to search for a tokenization scheme with a minimal description length. Different from the online nature of MDL, the TP method derives and uses the global knowledge to compute the transitional probabilities between adjacent terms. The PPM method builds probabilistic models to predict the most probable character sequence. In Section 8, we present evaluation work that compares the service discovery performance between these three methods and two baseline methods. The evaluation shows that MDL and PPM have achieved superior results than the two baseline methods.

The main contribution of this paper is as follows. Given its vitality to effective utilization of IR techniques for WSDL document retrieval, our work fills an important gap between the emergent research in IR-style Web services discovery and the existing body of knowledge in IR. More importantly, by addressing a fundamental issue, our solution could be directly used for source code mining, in which a key step is to tokenize various names of variables, functions, classes, modules, etc. for semantic analysis. Our methods also have the potential to solve other pressing Web-related string tokenization problems such as URL/URI analysis (search engine optimization, social network link analysis, etc.), automated Web scripting comprehension, and so on.

2. Background and motivation

In this section, we provide succinct background of inverted file indexes. We introduce technical details of WSDL term extraction, which produces the input for WSDL term tokenization issue. We then discuss the motivation for WSDL term tokenization.

2.1. Inverted file

Inverted file is widely used for indexing text database. To support efficient information retrieval, the words of interest in the text are sorted alphabetically. For each word, the inverted file records a list of identifiers of the documents containing that particular term. Consider a sample text database *Stock* presented in Fig. 1, which consists of five documents.

The indexer parses these five documents, and produces a set of distinct words for constructing the inverted file. The inverted file has two components—a vocabulary and a set of inverted lists. The vocabulary comprises a collection of distinct words extracted from the text database (see Section 2.2). For each word t, the vocabulary also records: (1) the number (f_t) of documents that contain t, and (2) the pointer to the corresponding inverted list. Each one of the word-specific inverted lists records: (1) a sequence of documents that contain t (notice that each document is represented as a document number d), and (2) for each document d, the frequency ($f_{d,t}$) of t appearing in d. Thus, the inverted list is a list of $< d, f_{d,t} >$ pairs. This information provides essential statistics for applying any IR techniques (e.g. vector space) in the information searching stage. An example of inverted file for the *Stock* text database is shown in Fig. 2.

A complete inverted file in practice often includes many pre-computed statistics such as document norm, etc., and the document identifiers may have been optimized for compression. We do not include such additional information in Fig. 2 for brevity purpose. Based on this inverted file, we sketch a simple searching algorithm in Fig. 3, which takes as input the user query (a list of key words) and produces a ranked list of relevant documents.

Each query term attempts to look up in the vocabulary of the inverted file. If a matching word is found, it reads both f_t (line 80) and the inverted list (line 100) to compute the query term weight (line 90) and the document term weight (line 120) respectively. Both weights contribute to the overall TF–IDF score (line 130) of a document containing that query term. The algorithm then uses the document length to normalize the relevance of a document (line 160–180). It finally returns

vocabulary	f_t	*inverted lists*
a	1	< 5, 1 >
an	2	< 1, 1 >, < 2, 1 >
for	1	< 5, 1 >
get	2	< 3, 1 >, < 4, 1 >
input	2	< 1, 1 >, < 3, 1 >
is	3	< 1, 1 >, < 2, 1 >, < 5, 1 >
last	3	< 3, 1 >, < 4, 1 >, < 5, 1 >
output	2	< 1, 1 >, < 3, 1 >
price	3	< 3, 1 >, < 4, 2 >, < 5, 1 >
quote	1	< 5, 1 >
stock	1	< 5, 2 >
symbol	2	< 1, 1 >, < 3, 1 >
the	5	< 1, 1 >, < 2, 1 >, < 3, 2 >, < 4, 2 >, < 5, 2 >
ticker	2	< 1, 1 >, < 3, 1 >
trade	4	< 2, 1 >, < 3, 1 >, < 4, 2 >, < 5, 1 >
with	2	< 3, 1 >, < 4, 1 >

Fig. 2. The inverted file for the *Stock* text database.

```
10.  input query
20.  output k most relevant documents
30.
40.  score[1..m] ← [0,...0]
50.  for each query term t in query
60.      inverted_file.vocabulary.lookup(t)
70.      if t cannot be found continue
80.      f_t ← inverted_file.getDocFreq(t)
90.      w_q,t ← computeQueryTermWeight(f_t)
100.     inverted_list ← inverted_file.getInvertedList(t)
110.     for each < d, f_d,t > pair in inverted_list
120.         w_d,t ← computeDocTermWeight(f_d,t)
130.         score[d] ← score[d] + w_q,t × w_d,t
140.
150. norm[1..m] ← inverted_file.getPreComputedDocNorms()
160. for each score_d in score[1..m]
170.     if score_d > 0 then
180.         score_d ← score_d / norm[d]
190. find the k largest score_d and return the corresponding documents
```

Fig. 3. A simple searching algorithm.

the k most relevant documents (line 190). Readers can refer to [17] for a thorough understanding of these fundamental IR concepts and techniques (TF–IDF, term weight, etc.) for inverted file indexing and searching.

Line 70 in Fig. 3 states that a query term is ignored if it does not match any words in the vocabulary. This is justifiable because a 'mismatched' term is of little use—it does not hold any information required for executing the subsequent searching (line 80–130). However, since indexing occurs temporally prior to searching, the vocabulary was constructed well before a user query is produced. Therefore, it is possible that the vocabulary does not include query terms that do not appear in the original text database. In this paper, we provide additional tokenization to the original text in order to minimize such a possibility. The main contribution of this paper lies in the novel WSDL term tokenization methods, without which any inverted file-based searching algorithms is under-optimized when being applied for Web services discovery.

2.2. WSDL term extraction

As shown in Fig. 4, a WSDL document is an XML document validated against the WSDL schema [1]. Previous studies [18] in the XML retrieval literature suggested extract content from XML markups before applying IR techniques. Since the name attribute of an XML element represents the semantics for that particular element. Hence, the value (i.e. nmtoken) of attribute name depicted in Fig. 4 is a good candidate for content extraction.

Fig. 5 presents a WSDL document instance. The value (e.g. GetLastTradePrice) of the name attribute for element <operation /> carries useful information implying the purpose of this operation at the lexical level. Similarly, values (i.e. nmtoken) of attributes "name" are extracted from a number of important WSDL elements (marked as bold faces in Fig. 4)—definitions, message, part, portType, operation, input, output, service, and port.

The value (i.e. qname) of attribute element for element part is also extracted for capturing the data structure of the parameters sent to/from the service operations. This forms recursive extraction of underlying data types for this element and/or type. In this example, the value (i.e. body) of attribute name for element part gives little useful information representing the real meaning of the input message part. Nevertheless, values (i.e. TradePriceRequest and TradePrice) of attribute element provide very valuable data in understanding the meaning of two message parts.

The data structure within the WSDL element types and schema reveals more important lexical information about these two message parts through extracting the value of attribute name for element. Thus, in the case of TradePriceRequest, the

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
  <import namespace="uri" location="uri"/>*
  <wsdl:documentation .... /> ?
  <wsdl:types> ?
    <wsdl:documentation .... />?
    <xsd:schema .... />*
  </wsdl:types>
  <wsdl:message name="nmtoken"> *
    <wsdl:documentation .... />?
    <part name="nmtoken" element="qname"? type="qname"?/> *
  </wsdl:message>
  <wsdl:portType name="nmtoken">*
    <wsdl:operation name="nmtoken">*
      <wsdl:input name="nmtoken"? message="qname">?
      </wsdl:input>
      <wsdl:output name="nmtoken"? message="qname">?
      </wsdl:output>
      <wsdl:fault name="nmtoken" message="qname"> *
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="nmtoken" type="qname">*
    <wsdl:documentation .... />?
    <!-- more elements omitted here --> *
  </wsdl:binding>
  <wsdl:service name="nmtoken"> *
    <wsdl:documentation .... />?
    <wsdl:port name="nmtoken" binding="qname"> *
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Fig. 4. A valid WSDL document structure. Adapted from [1].
Source: Adapted from [1].

element value is tickerSymbol. For TradePrice, price is extracted. Thus, by exploring solely lexical information, one can speculate that this Web service takes as input the stock "*ticker symbol*", and returns as output the "*price*" of the corresponding stock. The related operation name GetLastTradePrice also supports this proposition.

The <documentation/> elements contain natural language information that could be used for constructing the text database. However, the reliability of comments or documentation written by humans is concerning as they may be either misleading or obsolete from the actual WSDL interface. Therefore, we have developed a WSDL parser that extracts textual content directly from the interface definition to construct the original text database.

2.3. WSDL term tokenization

As shown in Fig. 5, a WSDL document uses untokenized words and sentences to define WSDL element attributes such as names (bold face fonts). This is due in part to the rules defined in the WSDL standard: all WSDL element names are associated with the W3C Schema attribute data type 'NMTOKEN', which is a mixture of name characters (including letters, digits, combining chars, etc.) but excludes the single white space (#x20) [19]. Furthermore, in practice, most WSDL documents are not created directly by humans but are automatically converted from other high level programming languages (e.g., Java, C#, etc.), in which white spaces are prohibited in variable names. Consider the operation name GetLastTradePrice in Fig. 5. Under the normal text operation, the inverted file will create a new entry for the single word GetLastTradePrice in the vocabulary.

As a result, the inverted file would be something looks like in Fig. 6, in which a partial vocabulary and its associated document frequencies f_t and inverted lists (assuming the document id for this WSDL file is '1') are presented.

Suppose a user issued a query with two key words trade and price. Based on the searching algorithm in Fig. 3 and the inverted file index in Fig. 6, the user will unfortunately miss the sample WSDL document (Fig. 5). A literal string matching

```
……
<types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>

  </message>
……
<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
……
```

Fig. 5. A sample of a partial WSDL document, source [1].

Vocabulary	f_t	*Inverted lists*
…	…	…
body	4	< 1, 2 >, … , … , …
…	…	…
getlasttradeprice	1	< 1, 1 >, …
…	…	…
stockquoteporttype	1	< 1, 1 >, …
…	…	…
tradeprice	3	< 1, 2 >, … , …
tradepricerequest	2	< 1, 2 >, …
…	…	…

Fig. 6. The partial vocabulary for the WSDL inverted file.

between GetLastTradePrice and Trade or Price will always return false for the vocabulary lookup operation (line 60 in Fig. 3). While substring searching may solve the problem theoretically, we would have to introduce more sophisticated in-memory indexes such as suffix tree [20] or suffix array [21] for indexing the vocabulary per se in addition to the existing inverted file index. Moreover, the inverted lists would have to be computed online based on user-generated string patterns. Therefore, we focus on constructing a simple yet effective inverted 'WSDL-file' index without significant augmentation to the inverted file data structure.

To achieve this, we have to tokenize the operation name GetLastTradePrice into four separate terms—Get, Last, Trade, and Price. One may argue that a trivial string pattern discovery (e.g. letter case patterns) would easily tokenize them into meaningful terms. Nevertheless, the tokenization problem deserves formal investigation for two important reasons. Moreover, as shown in Table 1, simple tokenization heuristics based on human naming tendencies may not be able to deal with real-world WSDL NMTOKENs. Therefore, WSDL term tokenization constitutes a pressing research issue.

3. Related work

3.1. IR-based service discovery

Numerous recent efforts have been reported in applying IR for Web services discovery. Here we only survey recent works that have specifically used inverted file indexes for Web services discovery. Readers refer to [23] for a more comprehensive

study on approaches to Web services discovery, [24] for more recent development in IR methods and evaluation frameworks, and in particular [22] for an alternative IR method based on the language model. We identify the limitations of current methods of term tokenization to address the problem defined in Section 2.

Wang and Stroulia in [8] employed the IR techniques in Web services discovery. Their method consists of two parts—information retrieval and structure matching. It is in the former part that the inverted file is used to index and search natural language description of desired services. These descriptions are provided in the WSDL <documentation> tags, which provide text written in human natural languages. The common Term Frequency–Inverse Document Frequency (*TF–IDF*, equivalent to Line 130 in Fig. 3) weighting scheme is used to reflect the relative importance of each word in the service description. Platzer and Dustdar [9] proposed a Web services search engine based on the Vector Space Model. The inverted file index includes human-written service descriptions of already composed services found in distributed UDDI registries. For the traditional TF–IDF weight scheme, they designed a three-step algorithm to merge the document vectors from two UDDI repositories.

Lee et al. [25] proposed a framework for XML Web services retrieval. Situating on top of the UDDI and using WSDL files, this framework applies modified TF–IDF weights using a five-step approach: parsing, tokenization & stemming, labeling, creating signature & indexing, and similarity comparison. The authors did not provide technical details on the tokenization method. They presented a very simple example (i.e. SearchByAuthor → search, by, and author). It is believed that they have used a naive string pattern discovery to tokenize words extracted from WSDL documents.

Kokash [11] proposed three different functions to measure WSDL specification lexical similarity: VSM-based TF–IDF, semantic similarity, and WordNet-based comparison. In the VSM function, the TF–IDF weight of word w_j in WSDL document d_i is computed as

$$weight_{ij} = TF_{ij}IDF_j = \frac{n_{ij}}{|d_i|} \log\left(\frac{n}{n_j}\right),$$

where n_{ij} is the raw frequency of w_j in d_i, $|d_i|$ is the total number of words in document d_i, n is the total number of WSDL documents, and n_j is the total number of WSDL documents in which the word w_j has occurred. In the WordNet-based function, both the query and WSDL concept descriptions are expanded with synonyms obtained from the WordNet lexical database [26]. In the semantic similarity function, a series of linguistic-related steps are performed.

Sajjanhar et al. [10] has attempted to leverage the Latent Semantic Indexing [27], a matrix factorized version of inverted file indexes, to facilitate Web services discovery that can capture the conceptual associations of advertised free texts from two WSDL documents. The terms are extracted from the UDDI description field written in natural language (mainly in English). More recently, Crasso et al. [28] used the TF–IDF technique to facilitate classifying Web services in a reduced search space.

The aforementioned work all employed some information retrieval techniques (e.g. TF–IDF). However, majority of them appeared to neglect the importance of NMTOKEN sequence tokenization in their methods.

3.2. Kokash method

Based on [11], Kokash et al. [12] conducted a more comprehensive experiment which contains 447 Web services divided into 68 groups. To the best of our knowledge, this is the first and only work that explicitly dealt with the WSDL term tokenization problem using the pattern-based method. Kokash et al. [11,12] defined three tokenization heuristics that "work fairly well" [11]: (1) sequences of an uppercase letter and subsequent lowercase letters, (2) sequences of a number of consecutive uppercase letters, and (3) sequences between two non-word symbols. Crasso et al. [28] also used the same heuristics to "bridge different naming conventions".

Although hard-coded heuristics work fine for some WSDL files, their behavior is uncertain if some new WSDL files with different encoding rules are presented. As shown in Section 1 and our experiment, these simple heuristics cannot correctly tokenize many real-world irregular WSDL NMTOKENS that do not strictly follow naming conventions. Since the Kokash method is the only Web services discovery work that explicitly reported the method of term tokenization, we use it as one of the baselines in this paper.

3.3. MMA method

The Maximum Matching Algorithm (MMA) has been used for Chinese word segmentation studies [29,30]. The basic idea is to use an external dictionary to verify the possible word tokens from the unsegmented text. The algorithm starts from the first character in a sequence and reads one character at a time into a 'character sequence' ***cs***. After reading each character, it attempts to find in the dictionary the longest word ***w*** starting with the current character sequence ***cs***. If ***w*** can be also found in the text (i.e., the remaining part of ***w*** also matches the following characters read from the text), the MMA marks a boundary at the end of ***w*** and starts again from the following character using the same longest word matching strategy until it reaches the end of the character sequence. If no matching words can be found in the word list, the first character in the character sequence ***cs*** itself will be identified as a single word. The limitations of the MMA-based method lies in

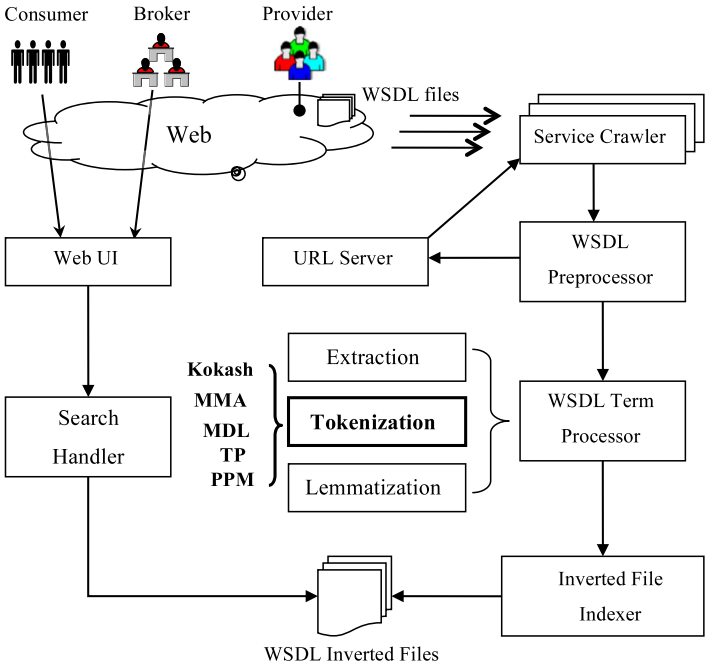

Fig. 7. IR-style service discovery approach.

its heavy reliance on a dictionary, which limits its scope of usage and performance. For instance, an MMA-based method with a dictionary that does not contain bioinformatics jargons may perform poorly when tokenizing bioinformatics Web services. Given its simplicity and popularity in segmenting Chinese words, we use MMA as another baseline, against which we compare the three methods used in this paper.

In this paper, we aim to use statistical models that can automatically learn the complex or latent regularities from NMTOKENS strings without resorting to external dictionaries and explicit rule definitions. To do this, we propose the application of three statistical tokenization methods—MDL-based, TP-based, and PPM-based as summarized in Section 4 (Fig. 7).

4. IR-style service discovery approach

The IR-style Web services discovery approach is illustrated in Fig. 7, in which term tokenization constitutes an important step for WSDL term processing. Initially, service providers deploy their Web services accessible to the public via the Web. In doing so, they also publish a service description, i.e., the WSDL documents, which captures the functional capabilities and technical details (e.g., transport bindings) of a Web service.

These service descriptions can be collected by a number of Service Crawlers, which fetch WSDL files from the Internet. Alternatively, they can also be collected from some well-known service datasets such as the one provided in [31]. Crawlers hand over retrieved WSDL files and associated HTML files to the WSDL Preprocessor for link analysis. This yields a list of new URLs that may point to some new WSDL files. These URLs are assigned to an idle crawler by the URL server. Readers refer to [31] for the details on WSDL crawling algorithm.

All retrieved WSDL files are then passed to the WSDL Term Processor, which (1) parses WSDL files and extracts important data (e.g., operations, messages, data types, etc.), (2) tokenizes extracted content into separate terms (the focus of this paper), and (3) carries out other linguistic tasks such as lemmatization and stop-word elimination, etc. Five tokenization methods (two baselines – Kokash and MMA – and three statistical methods—MDL, TP, and PPM) are used in (2).

The WSDL Processor generates the 'term document', which contains separated words in a flat structure. The term document is transferred to the Inverted File Indexer. The indexer takes as inputs tokenized and lemmatized terms with their associated occurrences information in each document and generates as outputs the compiled data arrangement with pre-aggregated information optimized for fast searching. The data structure of inverted index is consistent with the notion of *term-document* matrix, which consists of term vectors as matrix rows and document vectors as matrix columns. The term vector is sorted that allows fast lookup operation. The data structure of inverted files are based on [7].

The next three sections discuss the technical detail of three tokenization methods—MDL, TP, and PPM.

Table 2
An example: three tokenization schemes with associated description lengths.

Tokenization scheme	Description		Description length
	M	$D\|M$	
Convert from input **tooutput**	1 convert **2 tooutput**	1 3 5 **2**	
Input **tooutput**	3 from 4 data	5 **2**	$(28+5)+10=43$
Convert from data **tooutput**	5 input	1 3 4 **2**	
Convert from input to output	1 convert 2 to	1 3 5 2 6	
Input to output	3 from 4 data	5 2 6	$(28+6)+13=47$
Convert from data to output	5 input 6 output	1 3 4 2 6	
Convert **frominput** to output	1 convert 2 to	1 **7** 2 6	
Input to output	3 from 4 data	5 2 6	$(37+7)+12=56$
Convert from data to output	5 input 6 output **7 frominput**	1 3 4 2 6	

5. Minimum description length (MDL)

5.1. MDL principle

The Minimum Description Length (MDL) principle was initially proposed by Rissanen [32] as a method for statistical model selection. The idea is that, amongst a set of models capturing the regularities in the observed data, choose the one with the shortest description. Formally,

$$M_{mdl} = \text{argmin}_M\,[\text{Length}\,(D|M) + \text{Length}\,(M)] \tag{1}$$

where Length $(D|M)$ represents the length of data description given the selected model, and Length (M) is the length of the model. Note that description of data in effect includes two parts—data description induced by the model and the model description per se. The rationale of MDL is that (1) the model captures all important features regulating the data such that the amount of information needed to represent the data is significantly reduced using these features, and (2) the model itself is concise and easy to describe. A good model thus aims to reduce the total length of these two parts Length $(D|M)$ and Length (M) as shown in Eq. (1). Readers refer to stochastic complexity [33], coding theory [34] and the Kraft inequality [35] for the rationale of Eq. (1)—namely, a shorter data description length corresponds to a bigger probability of the observed data.

Consider the three NMTOKENs to be tokenized: convertfrominputtooutput, inputtooutput, convertfromdatatooutput. Table 2 shows three possible tokenization schemes with their associated description lengths based on Eq. (1).

The left-most column of Table 2 lists three possible tokenization schemes. For each scheme, the middle column shows its data description, which consists of two parts – the model M that generates the observed data and the observed data D given this model M – $D|M$. The total description length is shown in the third column, which includes both Length (M) and Length $(D|M)$. The concept of model M is an abstract one. In this example, M appears as an unknown word dictionary, which generates the three known NMTOKENs by choosing, re-ordering, and concatenating its word entries. (Eq. (1)) then aims to find a dictionary M_{mdl} such that the total description length of M_{mdl} and its resultant NMTOKENs is the shortest compared to those induced by any other possible dictionaries.

The description of the dictionary (i.e. model) M in Table 2 consists of two parts—a set of words V and a set of numbers N, with each number uniquely identifying a specific word in V. For example, in the first tokenization scheme in Table 2, V = {convert, tooutput, from, data, input}, N = {1, 2, 3, 4, 5}. The description length of M is thus the combined lengths of V and N, namely Length (M) = Length (V) + Length (N). For illustration purpose, Length (V) is simply defined as the sum over description lengths of all words in V, where each word has a description length equal to the number of characters in it: for example, Length (convert) = 7. In the above example, Length (V) = Length (convert) + Length (tooutput) + Length (from) + Length (data) + Length (input) = $7+8+4+4+5=28$. Length (N) is defined as the set cardinality $|N|$, thus Length $(N) = |N| = 5$ in the above example.

The description of NMTOKENs $D|M$ in Table 2 is represented as a permutation (i.e. choosing, re-ordering, and concatenating) of elements (repeatedly) drawn from N. The description length of $(D|M)$ is simply the element counts in the permutation. Following the same example in Table 2, a permutation P = [1 3 5 2 5 2 1 3 4 2] consisting of elements repeatedly drawn from N is formed to represent the three NMTOKENs. Since the element counts in P is 10, Length $(D|M) = 10$. Therefore, the total description length = Length (M) + Length $(D|M) = (28+5)+10 = 43$ for the first tokenization scheme as shown in the third column of Table 2. Based on the MDL principle, the first tokenization scheme is the best solution as it gives the shortest description length (43) compared to 47 and 56. Although this is not the "grounded truth" since external knowledge suggests that "to" and "output" are two separate words, the first scheme does its best to shorten the description length by combining "to" and "output" given the observed data (i.e. three NMTOKENs), in which "to" and "output" never appear apart.

Table 3
Length measurement method used in Brent and Cartwright [36].

Length of M		Length of $D\|M$
Total number of characters in M	Total number of identifiers in M	Total expected word length
$\log_2 \|A\| \times \sum_{w \in V} \|w\| + \|V\|$	$2\|V\| - 1$	$\|T\| \times E(P(w))$

Notation:
- V is the dictionary with each word entry $w \in V$
- $|V|$ denotes the number of word entries in V
- $|A|$ is the number of characters in the alphabet, on which each $w \in V$ is defined
- $|w|$ is the number of characters in w
- $|T|$ is the number of words in the tokenized text T
- $P(w)$ denotes the probability distribution of w in T
- $E(\ )$ is the entropy of $P(w)$ defined in Eq. (6.1)

Table 4
Description lengths calculated using Brent and Cartwright [36].

Tokenization scheme	Length of M		Length $D\|M$		Total length
	$3\|V\| + \log_2 \|A\| \times \sum \|w\|$ $A = \{26$ letters$\}$		$\|T\| \times E(P(w))$		
Convert from input **tooutput**	1 convert **2 tooutput**	149.61	1 3 5 **2**	13.08	
Input **tooutput**	3 from 4 data		5 **2**		162.69
Convert from data **tooutput**	5 input		1 3 4 **2**		
Convert from input to output	1 convert 2 to	152.61	1 3 5 2 6	16.11	
Input to output	3 from 4 data		5 2 6		168.72
Convert from data to output	5 input 6 output		1 3 4 2 6		
Convert **frominput** to output	1 convert 2 to	197.91	1 **7** 2 6	18.94	
Input to output	3 from 4 data		5 2 6		216.85
Convert from data to output	5 input 6 output **7 frominput**		1 3 4 2 6		

Brent and Cartwright [36] first utilized the MDL principle to segment phonetically-transcribed English utterances. To remove the potential bias introduced by the naive length measurement method used in Table 2, Brent and Cartwright [36] measured the description length by counting the bits of both M and $D|M$ as shown in Table 3.

Adding together terms from the third row of Table 3, and removing the constant (-1), we reformulate Eq. (1) as an objective function $F(V)$ with respect to dictionary V

$$\arg\min_V F(\bullet) = \arg\min_V \left[3\,|V| + \log_2 |A| \times \sum_{w \in V} |w| + |T| \times E(P(w)) \right] \tag{6}$$

subject to: Concatenations of $w \in V$ generate all NMTOKENs,

where the standard entropy function E of the distribution of w is defined as:

$$E(P(w)) = -\sum_{w' \in V} p(w = w') \times \log_2 p(w = w') = -\sum_{w' \in V} \frac{freq\,(w')}{|T|} \times \log_2 \frac{freq\,(w')}{|T|}. \tag{6.1}$$

In Eq. (6.1), we use the empirical frequency $freq\,(w')/\,|T|$ to estimate the probability $p(w = w')$. The Length of $D|M$ is the product between the expected word length under an optimal prefix-free code C (e.g. Huffman coding as shown in [36]) and the total number of words in T. It is C that determines the total number of identifiers in M is $2|V| - 1$.

Table 4 shows the description lengths calculated based on Eq. (6.1). Scheme 1 gives the shortest description length 162.69 compared to other two schemes at 168.72 and 216.85 respectively. In this example, both Table 2 and Table 4 suggest that scheme 1 is the optimal result.

To solve Eq. (6), Brent and Cartwright [36] designed a search algorithm that exhaustively tested every possible V in a batch mode, thus "reading in the entire input before segmenting any part of it" [36]. This search strategy aims to produce a globally optimized model V over all observations in one set-off. However, the problem is that this optimization algorithm is so computationally intensive that they were only able to run it on a tiny portion of the entire dataset.

5.2. Online MDL approach

To address the computational issue, we proposed a revised objective function that enables an iterative searching algorithm to incrementally learn the model one NMTOKEN at a time.

The Online MDL approach is depicted in Fig. 8. *Step* 1—A random sequence of WSDL NMTOKEN strings is **read** by the *DP* (dynamic programming) algorithm. *Step* 2—For each NMTOKEN string S, *DP* iteratively extracts a substring $S\,[0 \ldots i]$ $(i = 1$

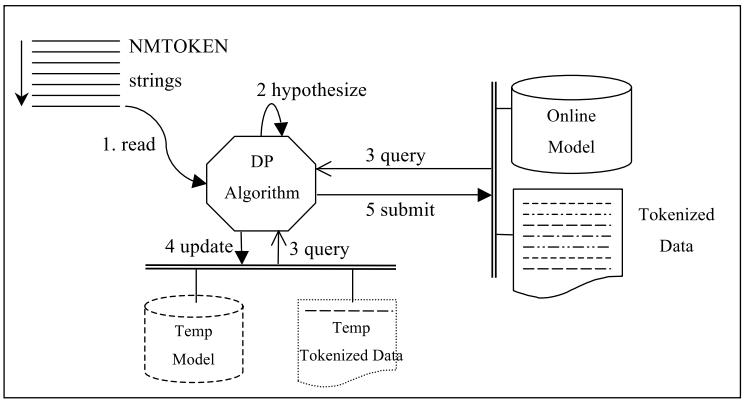

Fig. 8. Online MDL approach.

Table 5
Online MDL DP algorithm

```
10.   Input s: A NMTOKEN String
20.   Output tokens: String Array // each element is a token
30.
40.   size ← # of characters in s
50.   Initialize minLength[size], exLength[size], lastDelimit[size]
60.
70.   for (i ← 0; i < s.length(); i++)
80.     lastDelimit[i] ← −1
90.     minLength[i] ← F (s[ 0 to i ])
100.    for (j ← 0; j < i; j++)
110.      wd ← s[ j +1 to i ]
120.      length ← F ‘ (Farray[j], T[j], wd)
130.
140.    If (length < minLength[i])
150.      minLength[i] ← length
160.      lastDelimit[i] ← j
170.    //compute the cumulative form of F(V) for index i
180.    j ← lastDelimit[i]
190.    wd ← s[ j +1 to i ]
200.    list.add(wd)
210.    for (int k ← j; k >= 0; k = lastDelimit[k])
220.      prevWord ← s[lastDelimit[k] + 1 to k + 1]
230.      List.add(prevWord)
240.    Struct ← F0 (list)
250.    Farray[i] ← struct.F0
260.    T[i] ← struct.
270.  copyTokens (tokens, lastDelimit)
280.  submit(tokens)
```

…length(S)−1). For each S [0 …i], it finds an optimal tokenization **hypothesis** from a search space generated by iteratively inserting a white space at the position j as denoted in Eq. (7).

$$\boldsymbol{F}_{S_{[0\ldots i]}}(\bullet) \leftarrow \mathbf{min}\{\boldsymbol{F}_{S_{[0\ldots j]}}(\bullet) + \boldsymbol{F}_{S_{[j+1\ldots i]}}(\bullet)\} \tag{7}$$

where each j ($0 \leq j \leq i$) generates a tokenization hypothesis. *Step* 3—Each hypothesis in Eq. (7) is evaluated using Eq. (6). To compute $\boldsymbol{F}_{S_{[0\ldots j]}}(\bullet)$ and $\boldsymbol{F}_{S_{[j+1\ldots i]}}(\bullet)$ *DP* **queries** both the Online Model and the Temp Model and their associated (temp) tokenized data. *Step* 4—For each S [0 …i] *DP* **updates** the Temp Model using the optimal hypothesis. *Step* 5—DP **submits** the optimal Temp Model and Temp Tokenized Data to the Online Model and Tokenized Data when i = length (S) − 1, and repeats Step 1–5.

The complete DP algorithm is shown in Table 5. Each DP iteration (line 70–280 in Table 5) consists of several inner loops, each of which proposes a hypothetical model M_t and its associated temporary data D_t stored in the Temp Model and Temp Data respectively. Within each inner loop (line 100–270 in Table 5), DP uses the global knowledge from the Online Model to evaluate the length – Length ($D|M$) + Length (M) – of M_t in order to decide whether to keep the M_t from previous loop or update it with the current one. The temporary hypothetical model and data are set to empty at the end of each loop. When all iterations are completed, an M_t with the shortest total length is submitted to the Online Model, which then incrementally updates its knowledge for subsequent DP algorithm retrieval. Such a feedback-based updating mechanism allows the DP algorithm to continuously produce improved models for subsequent NMTOKEN strings.

In Eq. (7) the DP algorithm works by computing the optimal solution of the sub-problem $\boldsymbol{F}(S[0\ldots j])$ for only once, and reusing it for all subsequent iterations under different index i. This is possible only if $F(.)$ is cumulative. It is easy to verify that neither Eq. (6) nor Eq. (6.1) is cumulative and therefore the addition operation on two $F(.)$ functions is invalid. To address this issue, we redefine $F'(.)$ as in Eq. (8).

$$F'_{S[0\ldots i]}(.) \leftarrow \mathbf{min}\{Farray[j] + F_{\Delta}(S[j+1\ldots i], |T_j|)\}. \tag{8}$$

Farray[j] was already computed by the function $F_{0,S[0\ldots j]}(.)$ at iteration j Eq. (8.1). $|Tj|$ that denotes the number of words in the tokenized text Tj was also pre-computed at iteration j Eq. (8.1). Let $w_d = S[j+1\ldots i])$, we define $\boldsymbol{F}_{\Delta}(w_{delta}, |T_j|)$ as

$$\begin{cases} (freq(w_d)+1)\times \log_2[(|T_j|+1)/(freq(w_d)+1)] - freq(w_d)\times \log_2[(|T_j|+1)/freq(w_d)], & \text{if } w_d \in V \\ \log_2(|T_j|+1)+3+(\log_2|A|)\times \text{length}(w_d), & \text{if } w_d \notin V \end{cases} \tag{8.2}$$

where $freq(w_d)$ denotes the number of times w_d has appeared in the tokenized text T_j if w_d already has an entry in the dictionary V.

$F_0(.)$ is the cumulative form of $F(.)$. The major difference between $F_0(.)$ and $F(.)$ is the modified entropy function of $F_0(.)$ as shown in Eq. (8.1).

$$E(P(w)) = -\sum_{w'\in V} \frac{freq(w')}{|T_i|+1} \times \log_2 \frac{freq(w')}{|T_i|+1}, \tag{8.1}$$

where the denominators are $(|T_i|+1)$ instead of $|T_j|$ as in Eq. (6.1), $|T_i|$ represents the number of words in the tokenized text T_i at iteration i, and $freq(w')$ denotes the number of times that w' has appeared in T_i. $F_0(.)$ is called at Line 240 in Table 5 during each iteration i. The return value of $F_0(.)$ is a data structure with two items to fill respectively $Farray[i]$ and $T[i]$, which are subsequently used as parameters in Eq. (8).

To validate the correctness of Eq. (8) and 8.1, we compared the final description length result between Eq. (8) (8.1) and Eq. (6) (6.1). It showed that the differences are very minor (between 0 to 4.8E−4) due to floating point precision. This difference is seven orders of magnitude smaller than the value of description length, therefore we can safely ignore such minor difference and use cumulative functions Eq. (8) and (8.1) in lieu of Eq. (6) and (6.1) to tackle the scalability issue in Brent and Cartwright [36]. Our experiment suggested that Eq. (8) was at least one order of magnitude faster than Eq. (6). This is advantageous for processing large WSDL datasets.

To further improve the computational efficiency, we developed a parallel version of $F'(.)$ to compute the entropy function Eq. (8.1) on the multicore, shared-memory architecture. Given the cumulative nature of summation in Eq. (8.1), we used a simple data decomposition strategy—each core processes a mini-batch of words w's in the dictionary V, and we combine the results from all cores in the end to compute Eq. (8.1).

6. Transitional Probability (TP)

In investigating infants' language acquisition ability from continuous speech stream, Saffran et al. [37] proposed the notion of using Transitional Probability (TP) to discover word boundaries. The TP-based approach aims to minimize a different objective function based on the transition probability model. The transition probability (TP) is defined as the probability of observing character b immediately after observing a string a. Empirically, TP value is calculated as the conditional probability [37]:

$$transition\ probability(b|a) = Prob(ab)/Prob(a) \tag{9}$$

where a and b are strings with the number of characters) >= 1.

The TP specifies that a smaller transition probability between a and b with a 'surprisal' effect indicates a larger likelihood that a and b should have been separated. Vice versa, a larger TP increases the chance that a and b exist as a whole word. Therefore, a sequence should be tokenized into a set of words such that the average in-word TP is maximized or the average TP between words is minimized. The value of *Prob* (*ab*) and *Prob* (*a*) is computed using the empirical frequencies of *ab* and *a* (i.e. number of counts that they have appeared) in the entire untokenized texts denoted as **Freq** (*ab*) and **Freq** (*a*) respectively.

For example, the transitional probability between two words "input" and "to" is computed as: TP (*inputto*|*input*) = **Freq** (*inputto*) / **Freq** (*input*). Similarly, the average transitional probability within the word "input" is computed as (TP(*in* |*i*) + TP(*inp* |*in*) + TP (*inpu* |*inp*) + TP(*input* |*inpu*)) / 4. Table 6 shows both in-word TP and between-word TP calculated for the tokenization schemes used in Section 5. We see that scheme 1 achieves the best result in terms of in-word TP, whereas scheme 2 has the lowest between-word TP.

In our experiment, we attempted both in-word TP maximization and between-word minimization, and found that the minimization objective function performs slightly better as shown in Table 6. Although we will discuss the minimization objective function (between-word TP) in what follows, the overall DP algorithm Eq. (10) works regardless of which objective function is used. It is interesting though to see that simply combining the ranks of in-word TP and between-word TP methods

Table 6
In-word and between-word Transitional Probability for three schemes

Tokenization Scheme	In-word TP (to maximize)	Between-word TP (to minimize)
Convert from input **tooutput** Input **tooutput** Convert from data **tooutput**	0.778 (Rank 1)	0.407 (Rank 3)
Convert from input to output Input to output Convert from data to output	0.431 (Rank 2)	0.364 (Rank 1)
Convert **frominput** to output Input to output Convert from data to output	0.429 (Rank 3)	0.375 (Rank 2)

in Table 6 suggests that scheme 2 is the optimal method. Future work is needed to investigate how one can integrate these two types of optimization functions into a single equation with statistical reliability.

To implement this minimization optimization, we used a similar DP algorithm as shown in Table 5 but made two minor changes. First, we used a different $\boldsymbol{F}(.)$ defined as:

$$\boldsymbol{F}(.)_{S_{[0\ldots i]}} \leftarrow \mathbf{min}\ \{(\ \mathbf{F}_{\Delta\, S[j+1\ldots i]}(.) + Farray[j] * Larray[j])/(1 + Larray[j])\} \tag{10}$$

where the cumulative function $\mathbf{F}_{\Delta\, S[j+1\ldots i]}$ is defined as:

$$\mathbf{Freq}\ (S[lastDelimit[j] + 1, j].\mathbf{Concat}(S[j\ +1\ldots i]))/\ \mathbf{Freq}\ (S[lastDelimit[j] + 1, j]). \tag{10.1}$$

The second minor change is the added array *Larray*, each element *Larray*[*j*] stores the number of tokenized words for a (previously processed) sequence that ends with *S*[*j*] (i.e. during the *j*-th iteration). This value is used in Eq. (10) to derive the "average" transition probability for a sequence that ends with *S*[*i*] (i.e. during the *i*-th iteration) with its last split at *S*[*j*].

In addition to equation changes, we had to address two implementation issues. First, Eq. (9) may give a higher transitional probability to a very long sequence *a* followed by a character *b* than a short one (a') followed by *b*. This is because as longer sequences become increasingly rare, **Freq** (*ab*) will be infinitely close to **Freq** (*a*), resulting in a high probability of 1. This forces the minimization algorithm to combine *a* and *b* as long as *a* becomes long enough. To deal with this issue, we limit the length of the previous sequence to a fixed window size—*w*. For sequences that are over this size, we simply chop their preceding parts and only keep the last *w* characters. The second issue is the efficiency of computing **Freq** (*ab*) and **Freq** (*b*). These frequencies are based on untokenized texts that differ significantly from those used in Eq. (8.1)/Eq. (8.2), which are accumulated from tokenized words and hence are obtained in constant time.

To address this issue, we concatenated all NMTOKEN strings into a single string *T*, and built a suffix array index [21] to support sub-linear time frequency query. Computing **Freq** (*a*) on a suffix array index is transformed to a binary search with time complexity $O(\log n + p)$, where *n* is the length of the single string *T* and *p* is the length of *a*. The frequency is simply the number of matches in the binary search, and is obtained in constant time through the difference between two indices on the suffix array.

The TP-based approach uses the global knowledge, namely the single string *T*, to optimize an objective function. This is different from the MDL-based approach, which incrementally constructs the global knowledge while optimizing. Therefore, the TP-based approach might not be able to scale to large datasets with a massive number of NMTOKEN sequences. This is because it can be both memory and CPU intensive to construct and use a very large knowledge base from large datasets.

7. Prediction by partial matching (PPM)

Prediction by Partial Matching (PPM [38]) is a statistical method for model-based data compression. The PPM scheme allows a number of models to be dynamically obtained and continuously updated based solely on the preceding portion of the input stream that has just been processed by the encoder/decoder. The preceding portion is often defined as "context". PPM aims to generate a number of model "snapshots" for a number of context simultaneously at a particular timeslot.

To illustrate the use of PPM in our WSDL term tokenization context, we consider an example PPM model when processing the string "*input*to*output*" as shown in Table 7 (for illustration purpose, the character '*' represents a space).

Note that *A* is the alphabet from which these characters are drawn, $|A|$ is the size of the alphabet.

We first define the notion of context. A context is defined as a substring (i.e., a sequence of characters within a string). A context with order of *k* is a substring with length of *k* (*K*-gram). As shown in Table 7, a list of prediction probabilities is generated just after scanning the string "*input*to*output*". We used the PPMD method (see pg 86, section 4.3.2 in [39]) to compute the prediction probability *P*, which provides improved probability estimates than the PPM 'Method C' [40] and has been reported in [41] for Chinese word segmentation. Specifically, the prediction probability that a character has occurred *c* times in the preceding context with order of *k* ($k = 2, 1, 0, -1$) is calculated as:

Table 7
PPM model after processing the string *input*to*output*; c = count, p = prediction probability

Order 2					Order 1					Order 0				
Prediction			c	p	Prediction			c	p	Prediction			c	p
*o	→	u	1	1/2	*	→	t	1	1/4	→	*	2	1/10	
	→	esc	1	1/2		→	o	1	1/4	→	i	1	1/30	
*t	→	o	1	1/2		→	esc	2	1/2	→	n	1	1/30	
	→	esc	1	1/2	i	→	n	1	1/2	→	o	2	1/10	
in	→	p	1	1/2		→	esc	1	1/2	→	p	2	1/10	
	→	esc	1	1/2	n	→	p	1	1/2	→	t	4	7/30	
np	→	u	1	1/2		→	esc	1	1/2	→	u	3	1/6	
	→	esc	1	1/2	o	→	*	1	1/4	→	esc	7	7/30	
o*	→	o	1	1/2		→	u	1	1/4					
	→	esc	1	1/2		→	esc	2	1/2					
ou	→	t	1	1/2	p	→	u	2	3/4					
	→	esc	1	1/2		→	esc	1	1/4					
pu	→	t	2	3/4	t	→	*	1	1/6					
	→	esc	1	1/4		→	o	1	1/6					
t*	→	t	1	1/2		→	p	1	1/6	**Order -1**				
	→	esc	1	1/2		→	esc	3	1/2	Prediction			c	p
to	→	*	1	1/2	u	→	t	3	5/6	→	A	1	1/|A|	
	→	esc	1	1/2		→	esc	1	1/6					
tp	→	u	1	1/2										
	→	esc	1	1/2										
ut	→	*	1	1/4										
	→	p	1	1/4										
	→	esc	2	1/2										

$$p_c = \frac{c - 1/2}{n} = \frac{2c - 1}{2n},$$

where n is the number of times of this preceding context has appeared, and d is the number of different characters that have immediately followed it. The probability of an escape in the same context is formulated as:

$$p_{escape} = \frac{d/2}{n} = \frac{d}{2n}.$$

The probabilities in Table 7 constitute a complete PPM model that is essential to make probabilistic prediction for upcoming characters. For example, the probability of observing a space '*' immediately after "*input*to*output*" is predicted as 1/4, which is directly obtained from the Order 2 model in Table 7. Similarly, if the character '*t*' is observed, its prediction probability is a set (1/2, 1/2, 7/30). The first two probabilities in this set represent the escape probabilities for Order 2 and Order 1 respectively, the last one is the probability of observing '*t*' in Order 0. The reason for recording two escape probabilities is that neither '*utt*' (i.e., Order 2) nor '*tt*' (i.e., Order 1) has been observed in the context "*input*to*output*", so the PPM model has to fall back to Order 0.

The PPM-based Chinese word segmentation [41] used a Hidden Markov Model (HMM) combined with PPM models to yield the most probable segmentation schemes. The basic idea was as follows. Given a string a to be segmented with spaces, and suppose there exist m space insertion schemes ($S_1 \ldots S_m$), then S_k ($k = 1 \ldots m$) is optimal if and only if PPM (S_k) gives the best compression result, thus taking the minimal number of bits required to represent a. Namely, the better PPM can predict characters in some scheme S_k, the smaller number of bits required by PPM for this S_k. To discover the S_k with the largest probability, we create a lattice diagram to represent all possible segmentation schemes. Each node represents an Order-1 PPM model entry. For example, node "*i n*" represents the probability that character *n* appears right after *i*, and this probability can be directly obtained from Table 7 (i.e., =1/2). Formally, the edge and the node in the lattice represent the *transmission probability* and *emission probability* respectively. Note that to achieve the best PPM compression (i.e., the most probable segmentation) result, we need to create higher-order (e.g., 5) PPM lattice. We only use the order-1 lattice in Fig. 9 for presentation brevity.

A segmentation scheme is a path that consists of a vector of successive edges, the first of which starts from node *i* and the last of which ends with either node * *t* or node *u t*. In fact, Fig. 9 denotes $2^{12-1} = 2048$ lattice paths. Each path is associated with a path probability, which equals to the product between *transition probabilities* and *emission probabilities* along the path. Our goal is then to find the path $\boldsymbol{E}$ that has the highest such probability amongst all 2048 ones, $\boldsymbol{E}$ is considered as the most probable segmentation scheme. Thus, $\boldsymbol{E}^{max} = \mathbf{arg}\max p(\boldsymbol{E})$. Note that $\boldsymbol{E}^{max}$ is not necessarily equal to $\boldsymbol{E}^*$, in which each vector component (edge) e_i is selected such that $p(e_i) = \max[p(e_{i1}) \ldots p(e_{ik})]$, and k is the total number of edges between node i and node $i+1$. The reason is that a higher transition probability does not necessarily lead to a higher emission probability, and vice versa. A naive algorithm that enumerates the probabilities for all Es and select the one with the largest $p(\boldsymbol{E})$ has an unacceptable time complexity of $O(2^n)$.

Following [41], we employed the Viterbi algorithm [42] (see Table 8). When scanning characters from left to right in a string, at each character, we build K (e.g., $K = 2$ in Fig. 9) lattice nodes, and for each node, we will have K incoming paths

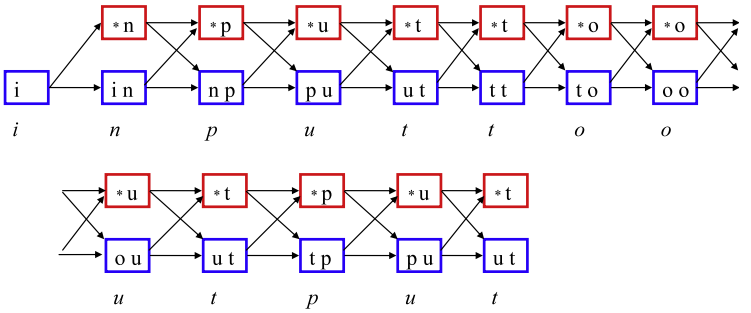

Fig. 9. The Order-1 PPM Lattice

Table 8
Viterbi algorithm

```
10.   Input name: String
20.   Input order: Integer //context length
30.   Output tokens: String Array // each element is a token
40.
50.   k ← Fibonacci(order)
60.   initialize pr-path[1. . . k]: Pr-Path Array //k best paths and its joint probability
70.   for each character in name
80.     for each node in character //k nodes for each character
90.       for each path that reaches this node //at most k such paths
100.        calculate the path's probability pr using PPM models
110.        record pr-path pair in a set P
120.      in the set P, find the pair pr-path_max with the maximum pr_max
130.      record pr-path_max in pr-path[1..k]
140.  let path_m ← max(pr-path[i]) // (i = 1..k)
150.  for each node in path_m
160.    tokens[i] ← node.label
170.  return tokens
```

Table 9
Path probabilities under four segmentation schemes

	input to output	*inputto output*	*in put to output*	*input to out put*
Pr	2.8391E-11	9.81192E-12	2.8391E−13	8.51729E-12
Log (pr)	−10.54681983	−11.00824609	−12.54681983	−11.06969857

reaching that particular node simply because the preceding character also has such K nodes. It appears that we have to deal with K^2 paths for each character. However, out of K paths reaching each node, we only need to keep the one that has the highest probability. This way, when we finish this character, we record K paths (instead of K^2), each of which corresponds to a lattice node for that character. After completing the last character, we are left with K paths, and the one has the highest probability is the output of the Viterbi algorithm.

The time complexity of the Viterbi algorithm is $O(n \times k^2)$ given that the Fibonacci number can be computed using a constant time. This time complexity, although non-linear, is still much more efficient than the naive algorithm with a time complexity $O(2^n)$ if n is large. Suppose we are to segment a string inputtooutput, the Viterbi algorithm will generate a path with the highest joint probability (Pr = 2.8391E-11) using the PPM models in Table 7. Table 9 shows the joint probabilities under four different segmentation schemes. We see that the "*input to output*" segmentation scheme has achieved the largest probability, thus forming the most probable sequence.

In addition, the PPM algorithm (line 100, Table 8) requires pre-built PPM models such as those shown in Table 7. To do this, we collected 51 articles from Wikipedia for deriving the PPM models. Each article has a URL pattern like: "http://en.wikipedia.org/wiki/$theme", where $theme = comedy_film, mass_media, bookselling, credit_card, book_review, automobile, dvd_player, mp3_player, novel, bank, coffee, hotel, etc. The selection of these themes was based on the seven domains associated with the evaluation data set.

8. Evaluation

As mentioned at the beginning, we used two datasets to evaluate the three statistical methods and two baseline methods for IR-style Web services discovery. The first dataset (denoted as SAWSDL) is the standard dataset sawsdl-tc1 provided

in [15]. This dataset contains 26 queries and 894 Web services described in SAWSDL using WSDL1.1, which were semi-automatically transformed from SAWSDL specifications in the owls-tc 2.2 test collection [15]. For each query also given is the relevant set, against which the service discovery performance can be quantitatively evaluated. The themes of these services include seven domains (*education*, *medical care*, *food*, *travel*, *communication*, *economy*, and *weapon*). Each Web service only has one *PortType*, thus a single operation per interface. The second dataset (denoted as Assam) is the categorized WSDL collection provided in [16]. Each WSDL document in Assam was hierarchically classified by humans and can be seen as similar to those within the same category. We further examined the dataset and manually removed those files that are semantically different from others within the same category. As a result, the second dataset contains 23 queries and 294 Web services described in WSDL1.1. These services mainly include finance, accounting, and IT-related development tools. All test queries used the TF–IDF retrieval technique (Fig. 3) on an inverted index as exemplified in Fig. 2.

8.1. Metrics

We use IR measures to evaluate the service discovery performance: *Precision–Recall*, *F1 value*, *Fallout value*, and *Mean Average Precision* (MAP). *Precision* is defined as "*the fraction of the retrieved documents which is relevant*" [17], i.e.

$$Precision = \frac{|R_a|}{|A|}$$

where R_a is the set of relevant Web services retrieved and A is the set of all Web services retrieved. *Recall* is defined as "*the faction of the relevant documents which has been retrieved*" [17], i.e.

$$Recall = \frac{|R_a|}{|R|}$$

where R is the set of all relevant Web services in the dataset.

The *F1 value* [19] integrates *Precision* and *Recall* into a single, composite harmonic mean, which is defined as:

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

The harmonic mean balances *recall* and *precision* by assigning an equal weight to them, and mitigates the impact of large outliers and intensifies the impact of small ones.

The *Fallout value* [43] is defined as the faction of irrelevant documents which has been retrieved (i.e., a contrary measure of *Recall*), i.e.

$$Fallout = \frac{|A| - |R_a|}{N - |R|}$$

where N is the total number of Web services in the dataset. Indeed, *Fallout value* measures of how fast precision drops as recall is increased.

The *Average Precision* (*AP*) measure produces a single-valued rating of a matchmaker for a single query result ranking. The AP is defined as the sum of the *precision* up to each relevant WSDL document that has been retrieved divided by the total number of relevant WSDL documents:

$$AP = \frac{\sum_{i=1..|A|} precision(i) \times isrel(i)}{|R|}, \qquad precision(i) = \frac{\sum_{k=1..i} isrel(k)}{i}$$

where $isrel(i)$ is 1 if the ith WSDL document in the retrieved result list is relevant, and 0 if otherwise. The Average Precision combines precision, relevance ranking, and overall recall, and is invulnerable to varying sizes of returned rankings. Therefore, it is an ideal measure of the service discovery performance.

8.2. Result

The *Precision–Recall* curve is generated by examining an incrementally changing ranked list of query result A. As A gradually increases Ra also changes, so do *Precision* and *Recall* metrics while R always remains the same. Starting from the top ranked Web service w, we check whether w is relevant, and calculates the correspondent *Precision* and *Recall*. After appropriate interpolation and normalization, the formal *precision–recall curve* records the *precision* value at each (predefined) *recall* level (e.g., 0.1, 0.2, . . ., 1.0).

An aggregated precision–recall curve was used to measure the true performance of a service retrieval method in a 'statistical' sense across all the queries. We used the *Average Aggregated Precision* (AAP) formula introduced in [17]:

$$\overline{P}(r) = \sum_{i=1}^{N_q} \frac{\overline{P_i}(r)}{N_q}$$

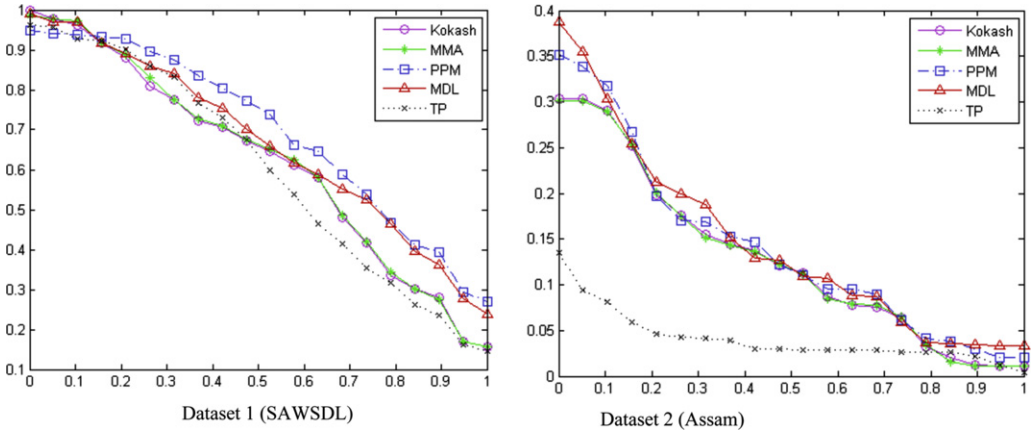

Fig. 10. Precision/recall curve.

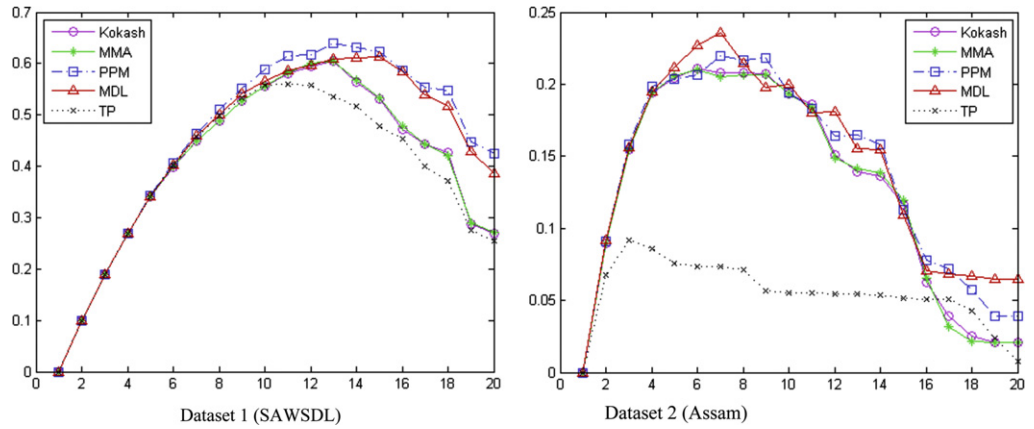

Fig. 11. F1–lambda curve.

where $\overline{P}(r)$ represents the average aggregated precision at the recall level r, N_q is the number of queries, and $\overline{P_i}(r)$ is the precision at recall level r for the ith query.

The aggregated *precision–recall curve* is plotted in Fig. 10. We observe that in Dataset 1, compared to Kokash and MMA, PPM and MDL curves have been moved up and out to the right, which means that both recall and precision are higher at every point along the MDL and PPM curves. The same observation applies to Dataset 2. Given the same indexing, retrieval, and ranking techniques employed, it is evident that MDL and PPM tokenization methods have achieved better result that leads to a superior precision–recall curve. For both datasets, the TP method performs poorer than the two baselines.

The aggregated *F1–Lambda* curve is plotted in Fig. 11, which computes F1 values for different *recall* levels ($0 \leq \lambda \leq 20$). Its score is maximized when the values of recall and precision are equal or close; otherwise, the smaller of recall and precision dominates the value of F1. We observe from Fig. 11 that for Dataset 1, PPM and MDL-based F1–Lambda curves show improved retrieval performance in that in each every level of recall, the F1 of the PPM-/MDL based method is equal to or greater than that of the Kokash/MMA-based. The superiority of PPM/MDL over Kokash/MMA is clear in terms of the F1 curve. Also clear is the inferiority of TP compared to the two baselines methods in terms of the F1 curve.

The *Fallout–Recall* curve is plotted in Fig. 12. For each recall level on the X-axis, we record the Fallout value on the Y-axis. The slope (tangent) at a particular point (recall) on the *Fallout–Recall* curve denotes the probability that a Web service in set A (i.e. all Web services retrieved) at a particular recall level is irrelevant (see [44] for a proof).

The *Fallout–Recall* curve in Dataset 1 indicates that, when *recall* is low (thus when *precision* is high), MDL and PPM exhibit a lower probability (compared to Kokash and MMA) that WSDL documents in A is irrelevant. This suggests that the relevance ranking scores in MDL and PPM are more consistent than those in Kokash and MMA since they ensure that a higher ranked WSDL document is less likely to be irrelevant. Given that all methods used exactly the same relevance TF–IDF ranking function, we believe that MDL and PPM method produced better tokenization result, which led to a more consistent *Fallout–Recall* curve. The *Fallout–Recall* curve in Dataset 2 shows the similar result for PPM but not for MDL: PPM performs constantly better than all other methods, MDL performs slightly poorer at recall range [0.2, 0.5], but performs better than Kokash and MMA when recall falls between [0.5, 0.7]. It becomes worse again after recall at 0.8.

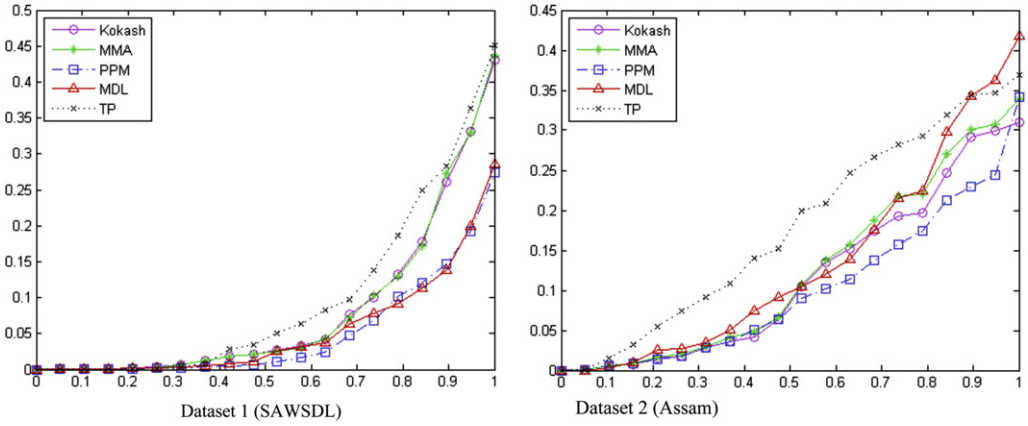

Fig. 12. Fallout–recall curve.

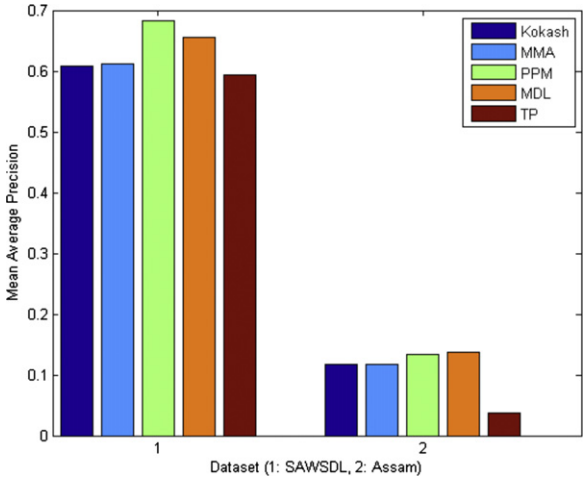

Fig. 13. Mean average precision (MMA vs. PPM).

The Mean Average Precision (MAP) bar chart is shown in Fig. 13. MAP values were obtained by taking the arithmetic mean of the 26 and 23 Average Precisions (AP) on Dataset 1 and 2 respectively over the above five methods. For both datasets, PPM and MDL methods are clearly superior than the two baseline methods and the TP method.

We report the Friedman test [45] result in Table 10. For Dataset 1, there exists a statistically significant difference between MDL and Kokash/MMA. In other words, it is beyond reasonable doubt that MDL has achieved a superior service discovery result than Kokash/MMA in terms of the MAP value. Since PPM performs even better than MDL, we can make the same conclusion for PPM. For Dataset 2, MDL does not exhibit statistically significant difference than other methods in the MAP metrics although MDL has achieved the highest MAP. In statistical terms, such superiority might have been attributed to random factors. However, other important metrics as shown in Precision/Recall Curve (Fig. 10), F1–Lambda Curve (Fig. 11), Fallout–Recall Curve (Fig. 12) all consistently point to the fact that MDL/PPM has achieved better tokenization results that led to better retrieval performance.

8.3. Discussion

The previous sub-sections suggest that, for each one of the evaluation metrics (precision–recall curve, F1–Lambda curve, recall–fallout curve, and MAP), PPM and MDL have shown a significant enhancement in retrieval performance compared to Kokash/MMA. Unlike MMA or PPM that relies on an English dictionary or human heuristics, PPM/MDL use rigorous statistical models and dynamic programming algorithms. Moreover, PPM/MDL can be used for tokenizing WSDL documents written in any languages with domain-specific phrases or jargons as it automatically learns a set of models from either the training set or the untokenized sequence per se. This is in contrast with an MMA or Kokash-based method that may depend on some language-specific heuristics obtained through human efforts.

MMA also has its own advantage over PPM. Unlike PPM that requires a training process, using MMA is rather straightforward provided a dictionary in that language is available. Furthermore, the retrieval performance of MMA could be improved by using a more updated and comprehensive dictionary. MMA is also easier to manipulate by injecting

Table 10
Friedman tests on MAP

Dataset 1 (SAWSDL)				
MDL			*p*-Value	Statistically significant?
MAP: 0.6555	Kokash	0.6077	0.01537	Yes
	MMA	0.6115	0.01537	Yes
	PPM	0.6834	0.70293	No
	TP	0.5934	0.00063	Yes
Dataset 2 (Assam)				
MDL			*p*-Value	Statistically significant?
MAP: 0.1371	Kokash	0.1178	0.60411	No
	MMA	0.1174	0.60411	No
	PPM	0.1328	0.81446	No
	TP	0.0370	0.00229	Yes

Table 11
Comparison of five methods.

	Mean Average Precision[*]	*Dictionary*	*Training*	*Hardcoded heuristics*
Kokash	0.3628	No	No	Yes (all rules)
MMA	0.3645	Yes	No	No
MDL	0.3963	No	No	No
TP	0.3152	No	No	TP for a single word
PPM	0.4081	No	Yes	for Markov order number

[*] MAP is the average of two datasets obtained from Table 10.

new heuristics rules to deal with ad-hoc situations. In terms of efficiency, MMA has a desirable linear time complexity proportional to the length of the string to be tokenized. However, since the tokenization process is carried out during the indexing time, this does not improve the run-time service discovery efficiency.

It appears that MDL has both virtues—simplicity and effectiveness. On the one hand, MDL can start to tokenize WSDL terms without any external resources or training processes. It is very simple to use and configure. On the other hand, MDL produces better tokenization results on both datasets, and significantly outperformed baseline methods on Dataset 1. The major drawback with MDL also lies in its online nature. Since no global knowledge is obtained, the first few WSDL NMTOKEN strings streaming through the algorithm tend to have poorer tokenization results than subsequent ones. In an extreme case, the first WSDL nmtoken will not be tokenized whatsoever due to the empty dictionary V. For both datasets, TP generates poorer results than all other methods. In particular for Dataset 2, TP has shown severely degraded retrieval performance compared to others. This suggests that distribution cues, proven useful for simulating infants' language acquisition in cognitive science, may not apply well to tokenization of WSDL NMTOKEN strings.

We compare the five methods in Table 11 against four criteria including: *MAP* (how effective is the method for retrieval performance?), *Dictionary* (does the method need any external resources such as a dictionary, thesaurus, etc.?), *Training* (does the method need an explicit training process to build the knowledge?), and *Hardcoded heuristics* (does the method use hardcoded rules based on prior human experience or heuristics?). In terms of *MAP*, PPM and MDL are on top of other three and TP is left well behind. Only MMA needs external dictionary, which limit its potential usage and performance. PPM needs a dedicated training process to build the probabilistic model, which can be time and resource consuming. This is in contrast with MDL, which conducts unsupervised learning using online data. The Kokash method consists of three hardcoded rules based on string patterns in English. TP used somewhat heuristics to determine the transitional probability for a single word since TP would normally require two words to compute. In PPM, we set the order of Markov model as 5 based on previous studies reported in the literature. MDL and MMA do not use any hardcoded rules, which reduce the random, variation effect caused by human experience and heuristics.

9. Conclusion and future work

The IR-style Web services discovery represents an important approach that leverages proven techniques well developed in the field of information retrieval. Given the little textual information contained in the Web service description, many researchers exploited the XML-based WSDL syntax in order to extract more valuable metadata that can be added to traditional IR indexes. However, a fundamental issue associated with this approach is the WSDL term tokenization. Failing to address this issue will severely affect the service discovery performance given the mismatch between WSDL terms and service discovery requests. In this paper, we used three WSDL term tokenization methods—MDL, TP, and PPM. We discussed the overall IR-style Web services discovery approach that makes use of these three methods. We presented a detailed

quantitative evaluation that compares the service discovery performance between these three methods and two baseline methods. Our finding suggested that PPM and MDL have shown a quantitative enhancement in retrieval performance compared to MMA and Kokash, and is a preferred term tokenization approach for IR-style Web services discovery. To the best of our knowledge, no formal investigation has systematically addressed the issue of WSDL term tokenization like our work in this paper. For future work, we plan to apply PPM and MDL to source code written in various programming languages. This is because most programming languages (e.g. C/C++/C#, Java, etc.), similar to WSDL NMTOKENs, do not permit white spaces between characters constituting identifiers (names) of variables, functions / methods, classes / interfaces, modules / packages, etc. To facilitate software reuse and composition, a key prerequisite is to correctly tokenize these identifiers—so that they can be discovered by developers who search from large software libraries and APIs using separated query words from a particular natural language. Another interesting direction is to tokenize URLs as they do not have in-between white spaces (e.g. http://abcnews.go.com/) and often consist of all-small-case characters (case insensitive). This may lead to further applications such as anti-spam and search engine systems, which substantially exploit automated URL understanding.

References

[1] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1., 2001, 18/05/2007. Available: http://www.w3.org/TR/wsdl.
[2] N. Srinivasan, M. Paolucci, K. Sycara, Adding OWL-S to UDDI, implementation and throughput, in: First International Workshop on Semantic Web Services and Web Process Composition, San Diego, USA, 2004.
[3] M. Kifer, R.e. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, D. Fensel, A logical framework for web service discovery, in: ISWC 2004 Workshop on Semantic Web Services, 2004.
[4] J.M. Rodriguez, M. Crasso, A. Zunino, M. Campo, Improving Web Service descriptions for effective service discovery, Science of Computer Programming 75 (2010) 1001–1021.
[5] T.S. Somasundaram, R.A. Balachandar, V. Swaminathan, A. Kumar, V. Paramasivan, Semantic description and discovery of grid services using WSDL-S and QoS based matchmaking algorithm, in: International Conference on Advanced Computing and Communications, 2006.
[6] S. Oh, D. Lee, WSBen: a web services discovery and composition benchmark toolkit, in: IGI-Global (Ed.), Web Technologies: Concepts, Methodologies, Tools, and Applications (4 Volume), 2010.
[7] J. Zobel, A. Moffat, Inverted files for text search engines, ACM Computing Surveys 38 (2006) 1–55.
[8] Y. Wang, E. Stroulia, Flexible interface matching for web-service discovery, in: Fourth International Conference on Web Information Systems Engineering, 2003.
[9] C. Platzer, S. Dustdar, A vector space search engine for Web services, in: Third IEEE European Conference on Web Services, Sweden, 2005.
[10] A. Sajjanhar, J. Hou, Y. Zhang, Algorithm for web services matching, in: APWeb, 2004, pp. 665–670.
[11] N. Kokash, A comparison of web service interface similarity measures, University of Trento, 2006.
[12] N. Kokash, W.-J.v.d. Heuvel, D.A. Vincenzo, Leveraging web services discovery with customizable hybrid matching, Technical Report, University of Trento, vol. DIT-06-042, 2006.
[13] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, Extensible Markup Language (XML) 1.0 (Second Edition), http://www.w3.org/TR/2000/WD-xml-2e-20000814#NT-NameChar, 2000.
[14] M.B. Blake, M.F. Nowlan, Taming Web Services from the wild, Internet Computing, IEEE 12 (2008) 62–69.
[15] M. Klusch, P. Kapahnke, Semantic web service selection with SAWSDL-MX, in: The 7th International Semantic Web Conference, 2008, p. 3.
[16] http://www.andreas-hess.info/projects/annotator/ws2003.html.
[17] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison Wesley, 1999.
[18] R. Baeza-Yates, N. Fuhr, Y.S. Maarek (Eds.), Proceedings of the SIGIR Workshop on XML and Information Retrieval, 2002.
[19] C. Van Rijsbergen, Information Retrieval, Buttersworth, London, 1979.
[20] E. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM (JACM) 23 (1976) 262–272.
[21] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, 1990, pp. 319–327.
[22] J.M. Ponte, W.B. Croft, A language modeling approach to information retrieval, in: 21st annual international ACM SIGIR conference on Research and development in information retrieval, 1998, pp. 275–281.
[23] M. Crasso, A. Zunino, M. Campo, A Survey of approaches to web service discovery in service-oriented architectures, Journal of Database Management (JDM) 22 (2011) 102–132.
[24] S.E. Robertson, S. Walker, S. Jones, M.M. Hancock-Beaulieu, M. Gatford, Okapi at TREC-3, in: Third Text REtrieval Conference, 1994, pp. 109–126.
[25] K.-H. Lee, M. Lee, Y.-Y. Hwang, K.-C. Lee, A framework for XML web services retrieval with ranking, in: International Conference on Multimedia and Ubiquitous Engineering, 2007.
[26] C. Fellbaum (Ed.), WordNet: An Electronic Lexical Database, The MIT Press, 1998.
[27] S. Deerwester, S. Dumais, G.W. Furnas, T.K. Landauer, R. Harshamn, Indexing by Latent Semantic Analysis, Journal of the American Society for Information Science 41 (1990) 391–407.
[28] M. Crasso, A. Zunino, M. Campo, Easy web service discovery: A query-by-example approach, Science of Computer Programming 71 (2008) 144–164.
[29] J.-S. Chang, K.-Y. Su, An unsupervised iterative method for chinese new lexicon extraction, International Journal of Computational Linguistics, Chinese Language Processing (1997).
[30] L. Tao, Elektronische Tokenisierung fuer das Chinesische. Master thesis, Mater thesis, Uni-muenchen., 2001.
[31] E. Al-Masri, Q. Mahmoud, Investigating web services on the world wide web, in: World Wide Web Conference, Beijing, China, 2008.
[32] J. Rissanen, Modeling by shortest data description, Automatica 14 (1978) 465–471.
[33] J. Rissanen, Stochastic Complexity in Statistical Inquiry, World Scientific, Singapore, 1989.
[34] C. Shannon, A mathematical theory of communication, Bell System Technical Journal 27 (1948).
[35] L. Kraft, A device for quantizing, grouping, and coding amplitude-modulated pulses, Master's Thesis, 1949.
[36] M. Brent, T. Cartwright, Distributional regularity and phonotactic constraints are useful for segmentation, Cognition 61 (1996) 93–125.
[37] J. Saffran, E. Newport, R. Aslin, Word segmentation: the role of distributional cues, Journal of Memory and Language 35 (1996) 606–621.
[38] J.G. Cleary, W.J. Teahan, Unbounded length contexts for PPM, The Computer Journal 40 (1997).
[39] P.G. Howard, Ph.D. Thesis—The design and analysis of efficient lossless data compression systems, dept. of Computer Science, Brown University, 1993.
[40] I.H. Witten, T.C. Bell, The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression, IEEE Transactions on Information Theory 37 (1991) 1085–1094.
[41] W. Teahan, Y. Wen, R. McNab, I. Witten, A compression-based algorithm for Chinese word segmentation, Computational Linguistics 26 (2000) 375–393.
[42] A.J. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, IEEE Transactions on Information Theory IT-13 (1967) 260–267.
[43] L. Egghe, The measures precision, recall, fallout and miss as a function of the number of retrieved documents and their mutual interrelations, Information Processing and Management 44 (2008) 856–876.
[44] S. Robertson, On score distributions and relevance, Lecture Notes in Computer Science 4425 (2007) 40.
[45] R.L. Iman, J.M. Davenport, Approximations of the critical region of the friedman statistic, Communications in Statistics A9 (1980).