Information Extraction from Web Services: A Comparison of Tokenisation Algorithms

Alejandro Metke-Jimenez, Kerry Raymond and Ian MacColl

Faculty of Science and Technology, Queensland University of Technology, Brisbane, Australia

Abstract. Most web service discovery systems use keyword-based search algorithms and, although partially successful, sometimes fail to satisfy some users information needs. This has given rise to several semantics-based approaches that look to go beyond simple attribute matching and try to capture the semantics of services. However, the results reported in the literature vary and in many cases are worse than the results obtained by keyword-based systems. We believe the accuracy of the mechanisms used to extract tokens from the non-natural language sections of WSDL files directly affects the performance of these techniques, because some of them can be more sensitive to noise. In this paper three existing tokenization algorithms are evaluated and a new algorithm that outperforms all the algorithms found in the literature is introduced.

1 Introduction

Web services have become the de facto technology to enable distributed computing in modern platforms. Services directories, such as Programmable Web, list thousands of services that can be used to build complex software applications. Also, most modern backend software applications expose their functionality as web services in order to facilitate B2B interactions as well as integration with other software products deployed internally. Some applications, such as mashups, are built entirely by combining existing services.

With the ever increasing number of web services available both in the public Internet as well as in the private Intranets, web service discovery has become an important problem for software developers wanting to use services to build applications. The UDDI standard was initially proposed to address this problem by providing a centralized repository of service descriptions. However, some authors consider that the standard is too complex for end users who just want to publish their services [1] and also, several problems with the centralized architecture have been identified. Perhaps the most relevant of these is the fact that registration in a centralized repository is not mandatory and therefore all unregistered services will not be discoverable by potential clients [2]. These shortcomings of the UDDI standard have given rise to proposals of decentralized mechanisms that crawl the web in search of WSDL files and use only the information found in them.

In [2] the authors propose an extensive classification of different types of web service discovery approaches. We are interested in the approaches found in the *syntactic*

Metke-Jimenez A., Raymond K. and MacColl I..
Information Extraction from Web Services - A Comparison of Tokenisation Algorithms.
DOI: 10.5220/0003698000120023
In *Proceedings of the 2nd International Workshop on Software Knowledge* (SKY-2011), pages 12-23
ISBN: 978-989-8425-82-9

matching and *semantic matching* categories, since these techniques are decentralized and have proven to be effective in other search domains. Within these categories the approaches can be differentiated by the degree of user involvement required. Although several standards have been proposed to add semantic annotations to web services [3], all of these methods require both choosing an ontology and adding semantic annotations to the services. On the other hand, approaches based on information retrieval methods use the information already available in the service description files. We believe these latter approaches are likely to be more successful because currently there is little incentive for developers to manually annotate web services.

Several researchers have implemented service discovery tools that rely solely on the information contained in the web service description files. It is difficult to measure the relative performance of these implementations against each other, and the results found in the literature vary. In [3] the authors compared several approaches to web service discovery, including information retrieval based methods and semantics based methods, and found that the semantic based approach using Latent Semantic Analysis outperformed the information retrieval approach. However, in [1] the authors showed that when comparing the results of a system based on Latent Semantic Analysis with the results of a system based on the Vector Space Model, recall improved but precision fell.

In our own experiments we implemented a web service search engine based on the Vector Space Model [4] and a semantics based engine using Explicit Semantic Analysis. In the classic Vector Space Model approach, text documents (in this case, WSDL files) are represented as vectors in a term space. If a term occurs in a document then its value in that dimension of the vector is non-zero. The value is typically derived from the frequency of the term in the document and the commonness of the term in the whole document collection. One of the most common weighting formulas used is Term Frequency Inverse Document Frequency, although several variations of this formula exist. In our implementation a variation of TFIDF that uses sublinear scaling was used. The relevant documents for a query are calculated by representing the query as a vector in the same term space and then calculating the similarity between the query vectors and the vector representations of the documents in the collection. In this implementation cosine similarity, which compares the angle between the vectors, was used.

Explicit Semantic Analysis is a method introduced in [5–7] in which text in natural language form is represented in a concept space derived from articles found in Wikipedia. In this method each article in Wikipedia is treated as a concept in a general-purpose ontology and the text in the articles is used to determine the degree of relatedness between the concept and a text snippet. An inverted index is used to build a vector for each individual term. The inverted index keeps track of the articles in Wikipedia that contain each term and their weight (which is typically calculated using some variant of TFIDF). The resulting vector for the text snippet is the centroid vector of the vectors derived for each term. The similarity between two text snippets can then be calculated based on these ”concept vectors” by using standard vector-based similarity similarity metrics such as cosine similarity. The method is said to be explicit because the concepts are explicitly defined in Wikipedia in the form of articles.

Compared to the Vector Space Model, Explicit semantic analysis produces vectors

in a concept space rather than vectors in a term space. Also, depending on the document set, the vectors in ESA tend to be shorter (the length of a vector is the number of articles in Wikipedia) but less sparse than the term vectors used in VSM. ESA is considered a semantics-based method because the documents are represented in a concept space rather than just a term space.

We evaluated the performance of both engines using the same dataset kindly provided to us by the authors in [3]. Our initial results showed that the ESA implementation performed worse than the VSM approach[1]. Upon further investigation we found that one of the contributing factors for the poor results of the ESA implementation was the noise introduced by the incorrect tokenisation of some of the terms extracted from the non-natural language sections of the WSDL files.

Other researchers have identified the need to find better ways of dealing with the complexities of extracting usable information from the non-natural language text found in WSDL files [8]. However, to the best of our knowledge, there is no detailed information published regarding the accuracy of the tokenisation algorithms found in the literature. Therefore, we created a data set designed specifically to test the accuracy of different algorithms when tokenizing strings typically found in WSDL files. This paper addresses the research question of which tokenisation algorithm produces best results when dealing with non-natural language strings found in WSDL files.

The rest of this paper is structured as follows. Section 2 provides a review of related work. Section 3 talks about the impact of introducing noise in the tokenisation process. Section 4 describes the data set that is used to evaluate the tokenisation algorithms. Section 5 shows the evaluation results for three tokenisation algorithms. Section 6 introduces a new algorithm and Section 7 shows its evaluation results. Section 8 discusses future work. Finally, Section 9 summarises the paper's research contributions.

2 Background

Few of the WSDL tokenisation algorithms in the literature are explained in detail. One of the few exceptions can be found in [9]. The authors use a tokenisation algorithm that is similar to the Maximum Matching algorithms used in Chinese segmentation. The algorithm starts with the first character and tries to find the longest matching word in the dictionary that is completely contained in the string. If no word is found then the algorithm assumes that the single character is a token and moves on to the next character. If a word is found then the algorithm marks that substring as a token and moves on to the next character until there are no more characters in the string.

For example, assuming that we wanted to tokenise the string "downloadMP3Music", and that the words "download", "MP3", and "music" are in the dictionary, the algorithm would start looking for the longest word starting with a "d". It would first find the word "down" but then it would also find the longer word "download". Since no words longer than "download" are found in the dictionary, that substring is marked as a token and the algorithm starts over from the letter "M".

[1] The results are part of an ongoing project and are not shown here because they are out of the scope of this paper.

The authors argue that relying on naming conventions such as Camel Case is problematic because many strings do not follow the convention correctly and because certain words, such as eBay, make it difficult to comply with the conventions. Also, some strings found in WSDL files do not even follow a naming convention and are just a list of lower case or upper case characters. The authors provide a table with a few examples of the results obtained using the tokeniser but there is no information about the accuracy of the algorithm when applied to a large set of strings. Also, there is no reference to the dictionary that was used. This is important because the performance of this algorithm is directly related to the dictionary being used. In the example mentioned in the previous paragraph, if the term MP3 is not in the dictionary, the result of the tokenisation is going to be incorrect ("download" "MP" "3" "Music", assuming the abbreviation "MP" is in the dictionary). We will refer to this technique as **MMA**.

In [3], even though the authors do not explain the algorithm used to tokenise WSDL names explicitly, we were able to reverse engineer it based on their data sets. The tokeniser does the following:

1. Dashes and underscores are removed from the string.
2. The characters in the string are iterated over from left to right. The first character is flagged as the start of the current token.
3. When a dot or a space is found a new token is created using the characters from the start-of-token flag to the previous character. The dot or space character is discarded.
4. When an upper case letter is found a new token is created using the characters from the start-of-token flag to the previous character. The upper case letter is flagged as the start of the next token.
5. The process continues until there are no more characters in the string.

Using the same example we used to describe the previous tokenisation algorithm, the result of applying this algorithm would produce the following tokens: "download", "M", "P3", "Music". We will refer to this technique as **Simple**.

3 The Importance of Tokenisation

Tokenising strings in programming-language-type format is both challenging and important. Generating the incorrect tokens has a negative impact on the overall performance of a search engine, but the impact is different depending on the approach being used. For example, lets consider the string "BINNAME". This string was found in a WSDL file that describes a fraud detection service. The correct way of tokenising it is "BIN" and "NAME". However, some algorithms incorrectly split this string into the tokens "BINNA" and "ME". The token "BINNA" can be associated with the area next to the Lamington National Park in Australia. The term is uncommon and very discriminative. If an approach based on the Vector Space Model is used, this error will likely generate a false positive if someone is looking for a service related to Binna Burra (an online reservation system for the Binna Burra Lodge, perhaps). However, the error is likely to go unnoticed when the system is evaluated against a standard test collection, since the information needs included in these benchmarks tend to be generic and will almost certainly not include anything related to Binna Burra.

In the Explicit Semantic Analysis approach, documents (in this case web services) are represented in a concept space derived from Wikipedia articles. This is achieved by using standard information retrieval techniques to calculate the degree of relatedness between the document and every article in Wikipedia. If an incorrect token such as “BINNA” is introduced, the service is likely to be mapped closely to the concepts related to that token in Wikipedia (which are likely to be the articles related to Binna Burra). Depending on the technique being used to calculate the similarity between this concept vector and other concept vectors, such as the ones that represent queries, the negative impact on the overall results can be much more significant. For example, if only the top k concepts in the vectors are used in the comparison, a term such as “BINNA” can cause not only false positives but can also prevent the service from being ranked high in the results for a relevant query, thus affecting precision. This happens because the irrelevant concepts related to the term “BINNA” will displace the concepts related to the service’s real purpose[2].

4 Evaluation Data Set and Metrics

In order to evaluate the different tokenisation algorithms, a collection of strings was assembled from a collection of 576 WSDL files[3]. The values of the name attribute of the following tags in the WSDL namespace were used: service, port, binding, operation, message, and port. The values of the name attribute of the tags in the type declarations were also used. The tags used were: element, simpleType, complexType, and enumeration.

This extraction process produced a collection of 31126 strings. These strings were then manually tokenised. When more than one tokenisation alternative was found for a string, the one deemed most common was used. For example, the string “wikiwebservice.GetRecentChanges” was tokenised as “wiki”, “web”, “service”, “get”, “recent”, “changes”. Although another valid tokenisation could have been “wiki”, “webservice”, “get”, “recent”, because sometimes web services is written as a single word[4], we considered the first possibility as being the most common.

Three metrics were used to evaluate the tokenisation algorithms. The first one calculates the percentage of tokenisation results that are completely correct. For example, if the correct tokenisation of the string “BINNAME” is “bin” and “name”, then the result is considered correct only when the tokeniser produces these exact tokens. This metric is referred to as % Perfect Tokenisations.

The second metric is precision. In this context precision is defined as

$$precision = \frac{ct}{tp}, \tag{1}$$

[2] In this case the term “BINNA” is particularly problematic because it is very discriminative. The impact will depend on the incorrect tokens being generated.

[3] The original dataset included 785 WSDL files, but some of them were duplicates and were removed

[4] Wikipedia includes “webservice” as an alternative way to spell “web service”.

where ct is the number of correct tokens produced by the tokeniser and tp is the total number of tokens produced by the tokeniser.

The third metric is recall. In this context recall is defined as

$$recall = \frac{ct}{rt}, \tag{2}$$

where ct is the number of correct tokens produced by the tokeniser and tp is the total number of tokens in the ground truth (the total number of tokens that should have been returned).

When tokenising long multi-word strings, the results of the three metrics can be significantly different. For example, suppose the correct tokenisation of the string "wikiwebservice.GetRecentChanges" is "wiki", "web", "service", "get", "recent", and 'changes'". If an algorithm tokenises the string as "wiki", "webservice", "get", "recent", and "changes", the first metric just counts the result as being wrong. The second metric counts four correct tokens (the tokens "wiki", "get", "recent", and "changes") and five tokens produced by the tokeniser, resulting in a precision value of 80%. The third metric counts the same four tokens as correct and six tokens in the ground truth, resulting in a recall value of 66.66%. If this were the only string in the collection the percentage of perfect tokenisations would be 0%, the average precision would be 66.66%, and the average recall would be 80%.

5 Evaluation of Existing Tokenisation Techniques

Along with the two tokenisation techniques discussed in Section 2 a third technique that uses the Camel Case naming convention to decide how to tokenise strings was implemented. The technique does the following:

1. Dashes and underscores are removed from the string.
2. The characters in the string are iterated over from left to right. The first character is flagged as the start of the current token.
3. The second character is analysed to determine if the current token is in lower case (all characters are lower case), upper case (all characters are upper case), or camel case (the first character is upper case and the rest are lower case).
4. The token boundary is determined based on the type of word detected in the previous step. If the word is lower case or camel case then the word boundary is flagged when an upper case character is found. The detected token does not include the upper case character, which is flagged as the start of a new token.
5. If the word is upper case then the word boundary is flagged when a lower case character is found. In this case the upper case character right before the lower case character is not included in the token because it is assumed to be the first character of the next token.
6. Numbers are considered to be part of the current token.
7. When a dot or a space are found a new token is created using the characters from the start-of-token flag to the previous character. The dot or space character is discarded.
8. The process continues until there are no more characters in the string.

Using the same example once more, the result of applying this technique would produce the following tokens: “download”, “MP3”, “Music”. We will refer to this technique as **Naming Convention**.

The three techniques, Simple [3], Maximum Matching [9], and Naming Convention, were evaluated using the tokenisation data set. For the Maximum Matching technique, four dictionaries were used. The dictionaries were derived from WordNet, the entries in the English language Wiktionary[5], the titles of the English language Wikipedia, and the entire corpus of the English language Wikipedia. Table 1 shows the number of words in each of these dictionaries. The results of the evaluation are shown in Table 2.

Table 1. Sizes of the dictionaries.

Dictionary	Number of Words
WordNet	87539
Wikipedia Titles	1039508
Wiktionary Titles	1805285
Wikipedia Corpus	5549346

Table 2. Results of the evaluation of three tokenisation algorithms.

Tokeniser	Dictionary	% Perfect Tokenisations	% Precision	% Recall
Simple	NA	71.11%	81.59%	83.9%
Maximum Matching	WordNet	52.53%	67.11%	72.34%
Maximum Matching	Wiktionary	45.56%	61.62%	62.92%
Maximum Matching	Wikipedia Titles	41.55%	56.72%	57.11%
Maximum Matching	Wikipedia Corpus	24.53%	41.37%	39.73%
Naming Convention	NA	**88.15%**	**94.49%**	**93.49%**

The results show that the performance of the Maximum Matching tokenisation algorithm is heavily dependent on the dictionary being used. Also, using a dictionary containing more words doesn’t necessarily increase the algorithm’s performance. In fact, the worst performance was obtained when using the largest dictionary, in this case, the one derived from the entire Wikipedia corpus. The performance degrades because Wikipedia includes many non-words that are incorrectly identified by the algorithm as the right tokens. For example, when given the string ”APRSoapIn” the MMA algorithm using the WordNet-derived dictionary correctly tokenises it into ”apr”, ”soap”, and ”in”. However, the same algorithm produces the incorrect tokens ”aprs”, ”oapi”, and ”n” when using the Wikipedia-corpus-derived dictionary because the terms ”aprs” and ”oapi” are contained somewhere in Wikipedia (APRS is the acronym for Automatic Packet Reporting System and OAPI is the acronym for The African Intellectual Property Organization).

The best performing algorithm is the one based on naming conventions. This tells us that most non-natural language text within WSDL files is written following some naming convention and done correctly. However, around 16% of the text in the WSDL

[5] Technically the entries in Wiktionary are the titles of the wiki pages. The content in the body of the wiki pages was not used.

files does not follow a naming convention or does so incorrectly.These results were used as a motivation for the design of the WEASEL tokeniser.

6 The WEASEL Tokeniser

Finding the right tokenisation for a given string requires knowledge that can be obtained from the string itself or from some external source. The dictionary used in the Maximum Matching algorithm is an example of external information that can be used in the tokenisation process. The letter cases used in both the Simple and Naming Convention algorithms is an example of information that can be found in the string itself.

The results in Section 5 show that the best performing algorithms were the ones that used the letter case information to determine how to tokenise the strings. This suggests that this information should not be ignored. It would also seem that using a general purpose dictionary as a source of external information is not very useful in the quest for achieving the perfect tokenisation, considering that regardless of the dictionary used, the performance of the Maximum Matching algorithm was the worse. However, this can be attributed to the fact that the algorithm only explores one word combination possibility out of many. Also, a dictionary specifically tuned for a particular domain might improve the algorithm's performance.

Table 3 shows some examples of the strings that the Naming Convention tokeniser had trouble tokenising correctly. The incorrectly tokenised strings can be classified into the following categories:

1. Strings that contain numbers, where the tokens around the numbers are split incorrectly.
2. Strings that do not use any naming convention.
3. Strings that use some naming convention but use it incorrectly.

Table 3. Examples where the Naming Convention tokeniser had trouble tokenising correctly.

WSDL Name	Tokens	Expected Tokens
BINNAME	binname	bin, name
historicoptiondatawsdl	historicoptiondatawsdl	historic, option, data, wsdl
Census1850_GetSurname	census1850, get, surname	census, 1850, get, surname
AxesgraphType	axesgraph, type	axes, graph, type

Strings in the first category are difficult to tokenise correctly without using external information because the numbers are sometimes part of the tokens, in strings such as "findMP3Service", and sometimes are independent tokens, in strings such as "InterestRateSwaps10Month". Strings in the second category are also difficult to tokenize correctly because it is hard to differentiate between single-word strings written in lower camel case and multi-word strings written all in the same letter case. Finally, the strings in the third category are the most difficult to tokenise correctly because it is hard to identify the cases where a naming convention has been used incorrectly, especially when this information is being used as the primary mechanism to identify word boundaries.

The first version of the WEASEL tokeniser (referred to as **WEASEL v1** in the tables) attempts to improve the performance of the Naming Convention algorithm by

dealing with the first problem. By using a dictionary the algorithm tries to identify if a number found within a string belongs to the current token, the next token, or should be treated as a separate token. It also uses the commonness of the words, which can be easily calculated by counting word occurrences in a large general-purpose text (in this case the whole English language Wikipedia corpus). The algorithm does the following:

1. The string is tokenised using the Naming Convention algorithm.
2. When a number is detected in any token the number is separated from the token. For example, if the initial tokenisation of the string "InterestRateSwaps10Month" produces the tokens "Interest", "Rate", "Swaps10", and "Month", the token "Swaps10" is split into "Swaps" and "10".
3. The different combinations between the number and its surrounding tokens are looked up in the dictionary. In the example, these combinations would be "Swaps10" and "10Month".
4. If none of the combinations are found in the dictionary then the number is treated as a separate token. This is what happens in this example and the incorrect tokenisation is now fixed.
5. If only one of the combinations is found in the dictionary then that combination is considered the correct token.
6. If both combinations are found in the dictionary then the combination that is most common is chosen as the correct token.

The second version of the WEASEL tokeniser (referred to as **WEASEL v2** in the tables) attempts to solve the second problem by using an algorithm similar to the Maximum Matching algorithm to tokenise strings that do not use any naming convention. The algorithm is only used when a string is detected to be entirely upper case or entirely lower case and does the following:

1. Starting from the first character, the algorithm looks for all the words in the dictionary that start with that character and match the next characters in the string. For example, for the string "BINNAME", the algorithm would find the words "b", "bi", "bin", and "binna" (assuming these words are in the dictionary).
2. A score is assigned to each word based on its length and commonness. The simple formula $Score(t) = commonness(t) * length(t)^k$ is used, with $k = 2$. The rationale behind this formula is that it is more likely for a token to be the correct choice if it is longer and more common. The constant k is used to increase the relevance of the token's length against its commonness. The value of the constant was derived by experimenting with a subset of the ground truth that included only lower case strings.
3. For each word the algorithm is run recursively starting from the next character in the string, until no more characters are left. For example, for the token "b", the algorithm would run recursively using the rest of the string, in this case "inname". This produces a set with several possible tokenisations.
4. Each tokenisation possibility is given a score, which is simple the average of the scores for each token. The one with the highest score is returned.

The difference with the Maximum Matching algorithm is that this algorithm assembles several possible tokenisation alternatives, based on the words found in the dictionary, and also uses the commonness information to select the best possibility. The main drawback is its complexity, both in time and space. The algorithm is very inefficient, but in practice it works well for short strings. A simple optimization for strings longer than a certain K is to limit the recursive runs to the top two scoring words only. This reduces the amount of computation and memory significantly.

Although this algorithm performs well for multi-word strings that use no naming convention, it can introduce noise when used in a collection thats include single-letter words written in lower camel case, if the words are not part of the dictionary (words might be abbreviated or might even be in another language). In this case the algorithm will try to tokenize the words further and is likely to end up with several short words that are unrelated to the service.

Table 4. Results of the WEASEL tokeniser evaluation.

Tokeniser	Dictionary	% Perfect Tokenisations	% Precision	% Recall
Naming Convention	NA	88.15%	94.49%	93.49%
WEASEL v1	WordNet	94.17%	96.65%	96.71%
WEASEL v1	Wiktionary	94.31%	96.72%	96.75%
WEASEL v1	Wikipedia Titles	93.72%	96.55%	96.37%
WEASEL v1	Wikipedia Corpus	91.84%	95.89%	95.43%
WEASEL v2	WordNet	92.65%	95.23%	95.33%
WEASEL v2	Wiktionary	**94.34%**	**96.8%**	**96.84**%
WEASEL v2	Wikipedia Titles	93.93%	96.79%	96.61%
WEASEL v2	Wikipedia Corpus	92.02%	96.11%	95.66%

7 WEASEL Evaluation

Table 4 shows the results of evaluating the different versions of the WEASEL tokeniser using the same four dictionaries that were used with the Maximum Matching algorithm. The results of the Naming Convention algorithm, the best performing algorithm of the ones evaluated before, are also included.

The results show that version two using the dictionary derived from Wiktionary produced the best results. The numbers show a significant improvement over the performance achieved by the Naming Convention algorithm. Version one using the same dictionary comes in a close second. The performance of these two algorithms is very similar but version one performs slightly faster when dealing with long tokens that use no naming convention. The results also show that the performance of these algorithms is dependent on the external algorithm being used. The dictionary that produced the best results was the one derived from Wiktionary and the one that produced the worst results was the one derived from the whole WIkipedia corpus. These results are consistent with the ones observed in the evaluation of the MMA algorithms where these two dictionaries were also the best and worst respectively.

8 Future Work

Although the new tokenisation algorithm performs better than other algorithms available in the literature, there is still room for improvement. Another source of external information that was not used and might be useful in certain scenarios is context. Analysing tokens extracted from other sections of a WSDL file can help find common patterns that may be useful when dealing with a complex string. In fact, context is used a lot by humans when manually tokenising difficult strings.

Although the levels of noise generated by the new algorithms are much lower than the existing algorithms, it is important to inspect the types of tokens being generated incorrectly. Incorrect tokens such as “binna” are likely to have a significant impact on the performance of some semantics-based systems, but incorrect tokens such as “whoisgoingtobepresident” are not as problematic.

Finally, the new tokenisation algorithm will be used in the implementation of the web service engine based on Explicit Semantic Analysis. WEASEL v2 using Wiktionary produced the best results and will therefore be used instead of the original MMA implementation. We expect the performance to improve given the superior performance of the new algorithm.

9 Conclusions

In this paper, the performance of tokenization techniques used to extract information from programming-language-type text is identified as one of the key aspects that directly affects the performance of certain semantics-based service discovery approaches. A data set containing a large set of strings extracted from WSDL files is used to evaluate three existing tokenization algorithms. Finally a new algorithm that outperforms existing algorithms is introduced.

Acknowledgements

This research is supported in part by the CRC Smart Services, established and supported under the Australian Government Cooperative Research Centres Programme, and a Queensland University of Technology scholarship.

References

1. Wu, C., Chang, E.: Aligning with the web: an atom-based architecture for web services discovery. Service Oriented Computing and Applications 1 (2007) 97–116 10.1007/s11761-007-0008-x.
2. D’Mello, D., Ananthanarayana, V.: A review of dynamic web service description and discovery techniques. In: 2010 First International Conference on Integrated Intelligent Computing, IEEE (2010) 246–251
3. Bose, A.: Effective web service discovery using a combination of a semantic model and a data mining technique. Master’s thesis, Queensland University of Technology (2008)

4. Salton, G., Wong, A., Yang, C.: A vector space model for automatic indexing. Communications of the ACM 18 (1975) 613–620
5. Gabrilovich, E.: Feature generation for textual information retrieval using world knowledge. PhD thesis, Israel Institute of Technology (2006)
6. Gabrilovich, E., Markovitch, S.: Computing semantic relatedness using wikipedia-based explicit semantic analysis. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence. Volume 7., Morgan Kaufmann Publishers Inc. (2007) 1606–1611
7. Gabrilovich, E., Markovitch, S.: Wikipedia-based semantic interpretation for natural language processing. Journal of Artificial Intelligence Research 34 (2009) 443–498
8. J.Hou, J.Zhang, R.Nayak, A.Bose: Semantics-based web service discovery using information retrieval techniques. In: Pre-Proceedings of the Initiative for the Evaluation of XML Retrieval 2010, IR Publications (2010) 274 – 285
9. Wu, C., Chang, E., Aitken, A.: An empirical approach for semantic web services discovery. In: Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on, IEEE (2008) 412–421