Journal of Web Engineering, Vol. 15, No. 1&2 (2016) 110–129

SEMANTIC SIMILARITY BASED CONTEXT-AWARE WEB SERVICE DISCOVERY USING NLP TECHNIQUES

SOWMYA KAMATH S
Department of Information Technology, National Institute of Technology Karnataka Surathkal, Mangalore, Karnataka 575025 INDIA
sowmyakamath@nitk.ac.in

ANANTHANARAYANA V.S.
Department of Information Technology, National Institute of Technology Karnataka Surathkal, Mangalore, Karnataka 575025 INDIA
anvs@nitk.ac.in

Received March 25, 2015
Revised October 9, 2015

Due to the high availability and also the distributed nature of published web services on the Web, efficient discovery and retrieval of relevant services that meet user requirements can be a challenging task. In this paper, we present a semantics based web service retrieval framework that uses natural language processing techniques to extract a service's functional information. The extracted information is used to compute the similarity between any given service pair, for generating additional metadata for each service and for classifying the services based on their functional similarity. The framework also adds natural language querying capabilities for supporting exact and approximate matching of relevant services to a given user query. We present experimental results that show that the semantic analysis & automatic tagging effectively captured the inherent functional details of a service and also the similarity between different services. Also, a significant improvement in precision and recall was observed during Web service retrieval when compared to simple keyword matching search, using the natural language querying interface provided by the proposed framework.

Keywords: Web service discovery; Semantic similarity; Automatic tagging; Service classification; Natural language processing

Communicated by: M. Gaedke & O. Diaz

1. Introduction

Service orientation has played a very important role in the development and deployment of distributed systems over the Web. As the standardized approach for implementing service oriented systems on the Web platform, Web services have gained immense popularity in multi-domain enterprise system integration. Given the tremendous potential that full service orientation brings to enterprises, the number of published services available on the Web has increased greatly in recent years [1]. Hence, there is a need for automated approaches for improving tasks like service discovery and retrieval for seamless integration. However, discovering these published services in a distributed environment is still a challenging task due to their large numbers, the heterogeneous sources on the Web from which they are available

and also the availability of mostly syntactic service descriptions.

At present, most Web search engines mainly rely on matching keywords from the user query with those in the indexed webpages and returning a ranked list based on their own proprietary algorithms. However, when applied to web services, keyword matching will result in poor precision mainly because the documentation is not as content-rich as that of a normal webpage. If the exact word is not present in the service's natural language description, then it would simply not be included in the generated results. Hence, potentially relevant services may never be in the result set, while others with the same keywords used in different contexts may feature on it, thus affecting precision. This was also the problem that was faced in the UDDI[a] Universal Business Registry, as web service search result generation was primarily by keyword matching. This problem is compounded by the fact that, in general, a service's natural language documentation provided by service providers is often insufficient. It is usually less than 20 words in length, quite generic and sometimes even non-existent [2, 3, 4].

As services are openly accessible on the open Web, adapted conventional information retrieval (IR) methods with additional semantics can be used to find published services and discover relevant ones for given requirements. Popular search engines now employ emerging new technology in the field of machine learning, natural language processing (NLP) and semantics to provide better and more accurate results. Hence, these techniques can also be used to enhance the efficacy of service discovery on the Web. Based on these concepts, we intend to develop a framework for facilitating web service discovery based on semantics and NLP techniques. In this paper, we present a framework for web service discovery by semantically analyzing real world web service descriptions to capture their functional information. Using the extracted functional semantics, services are tagged and classified. The framework also supports natural language querying for exact and approximate matching of relevant services for a given user request. The main contributions of the work presented in this paper are -

1. Similarity analysis of real world web services by using their functional information.
2. Automatic metadata generation using the extracted functional semantics of a service.
3. Similarity based classification of tagged services, and generation of metadata for each class using the metadata of services belonging to that particular class.
4. Semantic analysis of the user query to capture user context to improve the matchmaking and ranking process for effectively meeting user requirements.

The paper is organized as follows. Section 2 discusses existing work in the area of web service search and discovery. In section 3, we discuss the proposed system, its components and processes in brief. Section 4 presents a detailed description of the processes of feature extraction, service similarity computation and automatic tagging of services. In section 5, we describe the process of web service classification using the computed similarity and the tags generated for each service. In section 6, the process of semantic analysis of user query and service discovery using the semantically processed service dataset and query is discussed. The experimental evaluation of the proposed methodology and the results obtained during web service retrieval is presented in section 7, followed by conclusion & future work and references.

[a]Universal Descriptions, Discovery and Integration, http://uddi.xml.org/

2. Related work

Many researchers have tried to tackle the problem of web service discovery and the lack of semantics in service descriptions by proposing techniques for semi-automated annotation of service interfaces. Patil et al [5] developed a framework called METEOR-S Semantic Web Service Annotation Framework (MWSAF) for semi-automated annotation of web service interfaces. Verma et al [6] used SAWSDL[b] service descriptions for semantics supported service discovery. Several researchers focused on the problem of mapping WSDL to OWL-S[c] to semantically enhance and describe the functionalities, capabilities and profile information of a service [7, 8]. Keller et al [9] used WSMO[d] to semantically define a service's goals, preconditions and effects to organize available services, for improving the service discovery process.

However, the main issue with these approaches is that it is an extremely difficult task to annotate all available service descriptions semi-automatically, especially since the number of already published services available on the Web is large. Semi-automatic annotation of service interfaces is time consuming even with the few tools that are available currently, as all of them require manual intervention. It is clearly impractical to expect all service providers to make available explicitly defined semantic descriptions for their published descriptions. Hence, any service discovery method must be able to overcome the lack of semantics despite having to deal with syntactic service descriptions.

In view of this problem, several researchers proposed various techniques to mitigate these issues in service discovery. Dong et al [10] used WSDL operation similarity & other parameters for clustering a given service dataset and experimentally showed that a clustering algorithm to semantically cluster services resulted in a significant increase in precision and recall. They also developed a specialized web service search engine called *Woogle* based on this technique, which is not online anymore. Other efforts at building a service discovery engine, *Seekda!* [11] and *Service-finder* [12] employed clustering of various functional parameters of services for relevant service discovery and also similar service discovery. The system also allowed Web 2.0 style user interaction by allowing users to tag services with their own tags. This system reportedly contained the details of more than 15,000 services, but both *Seekda!* and *Service-finder* are currently unavailable.

Recently, researchers have focused on using the inherent meaning in the natural language terms in a service description for tasks like discovery, clustering and service matchmaking. Nayak et al [13] used semantic processing of services using OWL-S and WordNet for enhancing service retrieval efficiency and reported good results. Sajjanhar et al [14] applied Latent Semantic Indexing (LSI) on the natural language documentation of a web Service descriptions for understanding the functional capabilities of a service. Chan et al [15] and Hao et al [16] applied the concepts of vector space model based indexing for extracting feature vectors from each service. Paliwal et al [17] proposed a method for semantics based discovery that first clusters service published in an UDDI using an ontology hierarchy. They also enhance the service request using ontologies to achieve semantics based matchmaking between relevant services and the service request. Others applied Probabilistic LSI [18] on WSDL documents in

[b] Semantic Annotations for WSDL and XML Schema `http://www.w3.org/TR/sawsdl/`
[c] Web Ontology Language for Services `http://www.w3.org/Submission/OWL-S/`
[d] Web Service Modeling Ontology, `www.wsmo.org`

order to extract the inherent semantics about the service's functional description for clustering and fast retrieval [19, 20]. Fang et al [21] used handmade (manual) tags and tag enriching methods to cluster web services and showed that their system achieved good results. Elgazzar et al [22] used Normalized Google Distance [23] for computing the relative similarity between various terms extracted from a dataset of 400 services and applied clustering to the services as a preliminary step to web service discovery.

In the proposed approach, we focus on using the inherent functional semantics of published real world web services for finding relevant services for a given query during service retrieval. The extracted information is enhanced using semantics based approaches and NLP techniques to generate additional metadata for each service in the form of tags. Services are then classified into domain specific classes to reduce the search space, after which generated classes are also tagged. The proposed system also incorporates a natural language querying interface to capture user requirements and context for effective web service discovery.

3. Proposed System

Figure 1 depicts the system architecture of the proposed framework. The main processes of the proposed system are discussed briefly below.

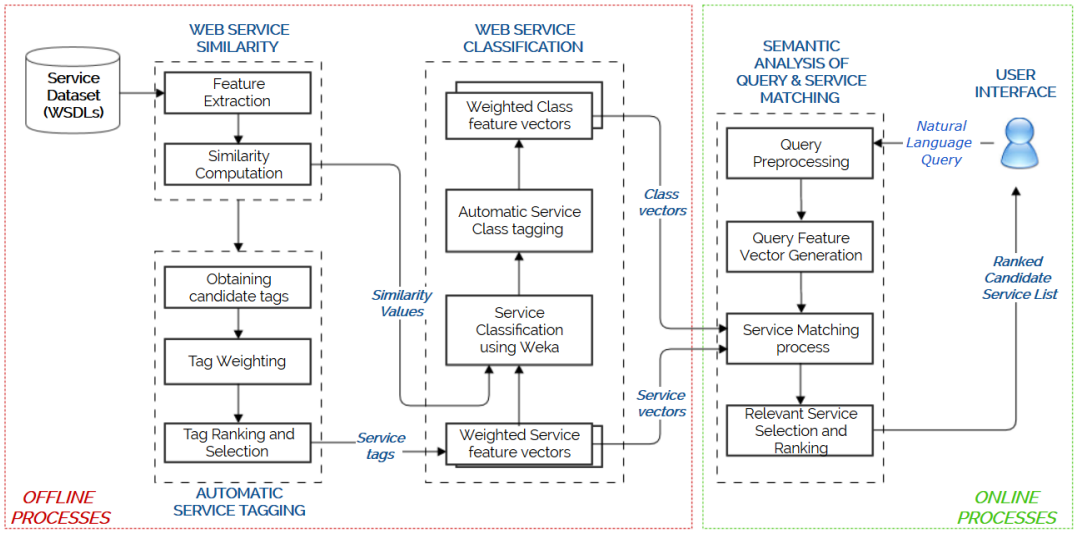

Fig. 1. Proposed system architecture

The methodology and processes can be divided into the *offline processes* and an *online processes*

1. Offline Processes

   - *Web Service Similarity and Automatic Service Tagging:* This process deals with the extraction of functional information from a service's WSDL document. Each service document in the dataset is first parsed and the element names are processed to generate natural language tokens, based on which the functional seman-

tics is extracted and used in the form of tags for each service. It consists of three sub-processes, namely - feature extraction, similarity computation and automatic service tagging. We describe each one of these processes in detail in section 4.

- *Web Service Classification:* After computing the similarity between each service pair and also generating tags for each service using its functional semantics, the next process deals with categorizing the services using machine learning techniques. After performing service classification, we also automatically generate tags for each class, by taking into account the tags of its member services, which is discussed in further detail in section 5.

2. Online Processes

   - *Semantic Analysis of Query & Service Matching:* The natural language query submitted by the user is pre-processed using NLP techniques to capture user context. During this process, the services, classes and the query that are represented in vector space are used for similarity based exact and approximate service matching. Section 6 presents a detailed view of the service discovery related processes.

4. Web Service Similarity & Automatic Service Tagging

4.1. *Feature Extraction*

During this phase, each WSDL is further processed in order to extract its functional information. For every service in the repository, we parse the WSDL documents and extract the various elements like service name, documentation, operation names, input/output messages and types. For each of the features extracted from the service's WSDL, we first split these natural language names into tokens. These element names are mostly a combination of natural language words, as most service designers follow standard programming conventions and good naming practices like camel-casing, pascal-casing & underscores to ensure maintainability while naming such service elements. For example, one operation of a service that is used to calculate scholarship based on the academic degree is named as *getScholarship*. This phrase cannot be considered as one token as it is not a valid English word. To ensure that the name of each service element is properly captured, we follow certain rules -

Case 1: Service element names are split into term tokens by considering *capital letters* as the start of a new token and *special characters* as token separators.

Case 2: Contiguous capital letters are considered as a single token as these mostly represent acronyms (e.g. SMS, SSN and ISBN).

Case 3: If no capital letters or special characters mark the beginning of a new word in the element name, proper splitting positions cannot be found. In this case, these element names need to be processed specially to identify tokens correctly, by using algorithm 1.

In the first two cases, splitting positions are easy to determine and terms can be obtained. Once all the term tokens are extracted, the stop words (like *'get', 'by'* etc., that do not contribute to the semantics of a service) are filtered out and a term vector is obtained

for each feature. For example, the name of the service *Academic-degreeScholarship* is a feature, and will result in the term tokens *'academic', 'degree'* and *'scholarship'*. Similarly, the service's operations are also considered as a feature, so after splitting, an operation named *'getAcademic-degreeType'*, will result in *'get', 'academic', 'degree'* and *'type'*. Finally, words like *'get', 'type'* etc are discarded. We used the NLTK stopword list for English. In addition to this, words used commonly in web services domain, referred to as *function words* (for e.g. *'service', 'input', 'output', 'request', 'response', 'SOAP', 'parameter'* etc) are also filtered as they contribute very little to the context of a service. A separate list of function words is maintained, for lookup during function word removal process. After this, the term vector is obtained for each feature of each service in the dataset.

As for the third case, we observed that some real world services in the dataset considered had quite unkempt element names. For example, consider a service named *citycountrySkilledoccupation_Service* that does not follow standard naming conventions. It provides operations such as *'get_skilledoccupation'*, which if considered as a single token will result in incorrect similarity values, that can adversely affect the accuracy of classification. To correctly generate term tokens for this particular name, we need to split it into *'city', 'country', 'skilled', 'occupation'* and *'service'* tokens. For extracting tokens automatically from such bad names, we used a dynamic programming approach to select the splitting positions by exploiting relative word frequencies in English as per Zipf's law [24].

Algorithm 1 Term token generation from badly named WSDL elements

Input:
List of badly named WSDL elements
D = custom dictionary consisting of words successfully extracted from the service dataset (sorted by their relative frequencies).

Output: *term-list*

```
1:  term-list = φ
2:  for (each badly named element in the list) do
3:      string str = element-name
4:      start-index = 0
5:      end-index = 0
6:      for i = (str.length − 1) till i=0 do
7:          end-index = i                    ▷ Start from the end of the element-name string
8:          term = splice(str, start-index, end-index)    ▷ check if spliced text exists in D
9:          Check in dictionary D for occurrence of term
10:         if found = true then
11:             Add term to term-list
12:             start-index = i              ▷ Reset current position to end of extracted term
13:             i = (str.length − 1)         ▷ Find current length of element-name string
14:         end if
15:         Decrement i by 1                 ▷ Start checking for next valid term
16:     end for
17: end for
```

Algorithm 1 shows the approach used for determining correct splitting positions for badly names WSDL elements. Zipf's law states that - *the probability of encountering the r^{th} most*

common word in the English language is given roughly by $P(r) = \frac{0.1}{r}$ *for* r *up to approximately 1000.* Zipf also noted that the law does not hold good for less frequent words due to the divergence of the harmonic series. Hence, assuming the relative word frequencies follow Zipf's law, one can obtain the probability of a word by $\frac{1}{r \cdot log\ N}$ where r is the rank of the word in a dictionary of N words sorted by their relative word frequencies. We use this concept in case of bad naming conventions such as in the example discussed above.

We used a custom dictionary modeled specifically for our service dataset, containing words successfully extracted from the services in the dataset. The dictionary consists of the words extracted successfully from well-named service features (during case 1 and 2), which is then used for inferring correct splitting positions for case 3. Assuming that all words in the corpus are independently distributed, the words as sorted as per their relative frequency of occurrence. Then, a word with rank r in the list of words will have a probability of occurrence as given by Zipf's Law. Using this approach, the splitting positions could be determined in linear time, as the badly named service elements are small sized strings. Since, the proper domain i.e., a Web service dataset itself was modeled, this method achieved nearly 94% accuracy. However, if a generic corpus of English language words was used, then the results of this approach would not be as effective due to ineffective domain modeling.

4.2. *Similarity Computation*

Once all the terms are extracted from each service's features, the similarity between every service pair is to be computed. To compute the similarity $sim(t_i, t_j)$ between any two terms t_i and t_j, we use WordNet [25]. WordNet is an online lexical database that can be used to compute semantic relatedness between two concepts. WS4J [26] is the Java implementation of WordNet::Similarity [27] for calculating synset similarity based on the path distance between two words in the WordNet taxonomy.

The maximum similarity as given by WS4J between the pair wise combinations of synsets of the two considered terms, extracted from each service is taken as the similarity value between them. Terms from the service name, documentation, input/output messages and operations are processed separately and the similarity between each token pair is recursively computed. Finally, the similarity between 2 term vectors f_{s_1} of service s_1 and f_{s_2} of service s_2 of a feature f is given by equation (1).

$$sim_f(s_1, s_2) = \frac{\sum_{i=1}^{i=|f_{s_1}|\cdot|f_{s_2}|} \left\{\alpha \cdot (1-\alpha)^{i-1} \times sim^i(t_1, t_2)\right\}}{\sum_{i=1}^{i=|f_{s_1}|\cdot|f_{s_2}|} \{\alpha \cdot (1-\alpha)^{i-1}\}} \tag{1}$$

where term $t_1 \in f_{s_1}, t_2 \in f_{s_2}$ and α is a constant less than 1. $|f_{s_1}|$ and $|f_{s_2}|$ represents the number of terms in f_{s_1} and f_{s_2} respectively. $sim^i(t_1, t_2)$ is the similarity corresponding to the i^{th} index in an ordered list of decreasing values from the set $S = \{sim(t_p, t_q) | t_p \in f_{s_1}, t_q \in f_{s_2}\}$. The main idea was to stress on similar pairs of tokens in the similarity computations.

We illustrate the need for an emphasis on similar pairs of tokens in the similarity computations, instead of taking the average value of S as was the case in [21] and [22] with an example. Consider three term vectors, T_1=[*'book', 'price'*], T_2=[*'horror', 'book', 'recommended', 'price'*] and T_3=[*'car', 'price'*] of services s_1,s_2 and s_3 respectively. Intuitively, T_1, T_2 are more similar in comparison to T_1, T_3. However, when similarities were computed by Fang et al's [21]

approach, it was found that $sim_T(s_1, s_2) = 0.42 < sim_T(s_1, s_3) = 0.55$. Similarities computed by equation (1) gave correct results i.e., $sim_T(s_1, s_2) = 0.83 > sim_T(s_1, s_3) = 0.74$. Also, over a large set of similarity values computed, the values got by equation (1) were more distributed in the range of 0 to 1.

The similarity between any two service descriptions s_1 and s_2 can be computed by -

$$\begin{aligned} sim_{wsdl}(s_1, s_2) = vw_1 \cdot sim_{servicename}(s_1, s_2) \ + vw_2 \cdot sim_{documentation}(s_1, s_2) + \\ vw_3 \cdot sim_{operations}(s_1, s_2) \ + vw_4 \cdot sim_{messages}(s_1, s_2) + vw_5 \cdot sim_{types}(s_1, s_2) \end{aligned} \tag{2}$$

where vw_1, vw_2, vw_3, vw_4 and vw_5 are variable weights whose summation should be equal to 1. $sim_{service}(s_1, s_2)$ and $sim_{documentation}(s_1, s_2)$ are calculated by equation 1. $sim_{operation}(s_1, s_2)$ is calculated by equation 3.

$$sim_{operation}(s_1, s_2) = \frac{\sum_{o_i \in O_{s_1} \cup O_{s_2}, o_i \notin O_{s_j}, j \in \{1,2\}} sim_{oo}(o_i, O_{s_j})}{|O_{s_1}| + |O_{s_2}|} \tag{3}$$

where, similarity between o_i and operation set of s_j, O_{s_j}, ($sim_{oo}(o_i, O_{s_j})$) is given by -

$$\begin{aligned} sim_{oo}(o_i, O_{s_j}) = Max_{o_k \in O_{s_j}} \big(0.4 \cdot sim_{o\ name}(o_i, o_k) + 0.3 \cdot sim_{o\ input}(o_i, o_k) \\ + \ 0.3 \cdot sim_{o\ output}(o_i, o_k)\big) \end{aligned} \tag{4}$$

where $sim_{o\ name}(o_i, o_k)$ is calculated using (1). Similarity $sim_{o\ i/o}(o_i, o_k)$ between input/output messages with operations o_i, o_j is computed as the average of $sim_{i/o\ name}(o_i, o_j)$ (calculated by eqn (1)) and $sim_{i/o\ type}(o_i, o_j)$ is given by equation 5.

$$sim_{o\ i/o\ type}(o_i, o_j) = \frac{2 \cdot |type_{o_i} \ \cap \ type_{o_j}|}{|type_{o_i}| + |type_{o_j}|} \tag{5}$$

where $type_{o_j}$ is the set of data types of input or output used in operation o_j.

4.3. *Automatic Service Tagging*

4.3.1. *Obtaining tag candidates*

In order to generate tags for service description, it is essential to ensure that tags denote the functional information of a service to the maximum extent. As such, this information lies in terms extracted from the service's WSDL, which can be used as potential tag candidates. We used the libraries provided by Python's NLTK to discover noun phrases from the term vectors obtained from the WSDL features. Noun phrases are found in a service's natural language description and in element names (e.g. operations, input/output messages etc) that indicate the service's purpose. For example, the natural language documentation for the *ZipCodeLookup* service, given as *"Returns latitude and longitude for a given US city zipcode"* contains the noun phrases *'latitude', 'longitude', 'US city zipcode'*. Similarly, from one of its operations *'getZipcodeforCityState'*, the noun phrases *'zipcode', 'city'* and *'state'* were extracted. These together with all the other noun phrases discovered from the WSDL forms the potential tag candidate set.

4.3.2. *Tag Weighting*

The tag candidate set of a service may contain several potential tags and all of these cannot be used as a service's tags. Since all the features in the generated vector cannot have equal importance for a particular service, a term weighting or a suitable feature selection technique should be applied. In order to rank the tag candidates and choose the top few as the tags for a service, we used an integrated approach to generate a ranking for the tags. We used a term weighting method called Tf-idf (term frequency-inverse document frequency) [28] to compute the importance of a particular tag candidate to a given service when compared to its importance in other services in the dataset. Tf-idf values are computed as follows.

$$weight_{frequency} = (0.5 + \frac{tf}{tf_{max}}) \cdot log(0.5 + \frac{D}{df}) \tag{6}$$

where tf is the term frequency of a tag candidate t in the given WSDL document d; tf_{max} is the highest frequency of any term in the WSDL document d, df is the frequency of occurrence of the term t in other WSDL documents and D is the total number of WSDL documents in the dataset. Using the computed values for each tag candidate, a TF-IDF matrix is built to compute the relative importance of each term word in the service document.

After the list of potential tags is obtained, the Tf-idf matrix is calculated and both the taglist and the matrix are stored in `.csv` files. In order to calculate the semantic relatedness matrix of terms obtained in the previous step, we used Natural Language Toolkit library of Python (NLTK) [29]. for obtaining synsets of a term as given by WordNet. For each pair of tags, a list of associated words is constructed. Semantic similarity is calculated between each pair of terms in the lists and a weighted average is produced. The computed *semantic relatedness matrix* is also stored in a `.csv` file. In order to combine the syntactic score (Tf-idf score) and semantic relatedness score, we perform matrix multiplication on the two matrices and the result thus obtained is a vector for each service where, a value for a tag is a combination of syntactic and semantic importance of that tag in the service.

4.3.3. *Tag Ranking and Selection*

The Tf-idf matrix will have a very high dimensionality as it captures the relative importance of all tag candidates generated for a particular dataset, with respect to each service. However, very few tags will be relevant for a given service, hence it is important to identify these terms only. In order to obtain a feature vector made up of the most significant features, we used Singular Value Decomposition (SVD) [30], a popular feature reduction method. SVD is a Linear Algebra matrix decomposition technique, which is used to obtain factorization of a complex matrix. SVD helps in factorizing the Tf-idf matrix to identify that set of features which are most important for a particular service. We used Rapidminer Studio [31] for Tf-idf matrix generation for the tag candidate set obtained after pre-processing and also for the SVD process.

5. Web Service Classification

5.1. *Weighted Service Feature Vectors*

SVD was very effective in reducing the initial tag set and in identifying the tag candidates that

represented the functionality of a given service to the maximum extent. In our experiments, we initially obtained more than 4785 tag terms. Applying SVD with a variance limit 0.95 (the allowed range is between 0 and 1), resulted in more than 700 attributes. With further experiments, it was found that a variance value of 0.4 gave the most optimal results and 10 attributes were selected after the SVD process. Then, the tags were ranked as per their relative importance. Based on their rank, the top 5 weighted terms were taken as the tag set and also the feature vector of each service. After obtaining the tag set, we expanded the set of tags by using cognitive synsets obtained from WordNet, in order to capture synonyms of the words that were used as tags for each service.

5.2. *Service Classification using Weka*

The main processes in this module are applying various machine learning techniques to the weighted service tag sets generated during the previous stage to categorize services into relevant categories and finally generate class tags from the identified classes. After reducing the dimensionality and obtaining an optimal weighted feature set for each service in the dataset, we used various classifiers available in the Weka Data Mining Suite [32] for classification of the services into domain specific categories. We used six different classifiers available in Weka, the details of which are given below:

1. *Naäve-Bayes*: Naäve-Bayes is a simple probabilistic classifier based on Bayes' theorem that assumes strong independence between the considered features [33]. Here, the assumption is that each feature has an independent contribution in the probability. For example, consider that an animal can be called as a dog if it has four legs and it barks. In Naäve Bayes, there is no correlation between the *number of legs* and *barking* features, which can often lead to false results.

2. *Decision tree*: Decision tree classifier uses a decision tree for building a predictive model [34]. Here, numerous test questions and conditions are modeled as a tree structure, where each internal node and the root of the tree denote a testing attribute while each leaf corresponds to a class label. When some input variables are given, the class label can be predicted by traversing the tree such that the values of the given input variables represent the path from the root to leaf.

3. *Random forests*: Random forests is an ensemble classifier which involves growing many classification trees [35]. A service is classified by assigning its input vector to each of the trees. Each tree assigns a class label to the service. This process is called as voting as each tree votes for a class label. The forest then chooses the class label that obtained the highest number of votes.

4. *Bagging*: Bagging is another ensemble classifier that applies base classifiers on random subsets of the original dataset [36]. The subsets are made by drawing random samples and replacing them at later stages of the classification. Each base classifier gives its individual predictions and these are then aggregated either by averaging or voting.

5. *Regression*: Regression is a probabilistic statistical model which uses the relationship between the categorical dependent variable (which needs to be predicted) and various other independent variables [37].

6. *Artificial Neural Network (ANN)*: ANNs [38], also called Multilayer Perceptrons, are inspired by biological nervous systems (i.e., the brain), and are modeled as a network comprising of many highly interconnected processing elements called *neurons*, organized in layers. Neurons work together in order to approximate a specific transformation function. An ANN uses a learning process during which the weights of the inputs in each neuron are updated, by a training algorithm, such as *back-propagation*, based on previous iterations, to reduce the value of an error function.

We used Weka's regression model in the work described in this paper. The classification accuracy was determined using a 10-fold cross validation [39] method. Cross validation is a method of accuracy estimation of a model when applied to a dataset and is very useful in case of limited data availability. In such a situation, if enough data is held for training to construct a good model then testing would not yield reliable results. The most common cross validation technique is k-fold cross validation, where, the dataset is divided into k parts with a homogeneous distribution of classes. Out of the k parts, one is retained for testing while $(k-1)$ parts are used for training. This process is performed k times with different testing partitions each time. The final accuracy is calculated by taking the average of k accuracy scores for each fold. The results of the classification and accuracy achieved with various classifiers is provided in subsection 7.2.

5.3. *Automatic Service Class Tagging and Class Feature Vector Generation*

After classification, we used the most optimal results as the service categorization. The ANN classifier achieved the highest accuracy in classifying the service data with reduced feature set (obtained after SVD). The classes as given by the ANN Classifier and the details of the services belonging to each class are noted. The process of generating tags for these classes is then carried out, based on each class's member services. In order to generate class tags, initially, we considered the tags of all services within a given class as potential tag candidates to represent the class. These tags are then ranked based on two factors as below:

- how well a tag g describes a given service s (represented as $sim_{tag}(g, s)$). This is given by the $weight_{total}$ for tag g in service s.
- how well a service s whose tag is being considered describes its class c (represented as $csim_{service}(s, c)$). This is given by eq. 7, where $|c|$ is the number of services in class c.

$$csim_{service}(s, c) = \sum_{i=1, s_i \neq s}^{i=|c|} \frac{sim_{wsdl}(s, s_i)}{|c| - 1} \tag{7}$$

$$sim_{classTag}(g, c) = \sum_{g \in \tau(s), s \in c} \left\{ sim_{tag}(g, s) \times csim_{service}(s, c) \right\} \tag{8}$$

The similarity $sim_{classTag}(g, c)$ between a class tag g & its class c is given by eq. 8, where $\tau(s)$ is the set of tags for service s. Based on the resultant ranking, we consider the top-5 tags as class tags. In addition to this, we also enhance the final tag set with synsets provided by WordNet for each tag phrase. These together form the feature vector of each class, which will be utilized during the query to service matching process.

6. Semantic Analysis of Query & Service Matching

Figure 2 depicts the process of web service discovery on the tagged dataset, with a semantically enhanced service request. These are the online phase of the framework, and the main processes are semantic analysis of query and service matching (as depicted in Fig. 1).

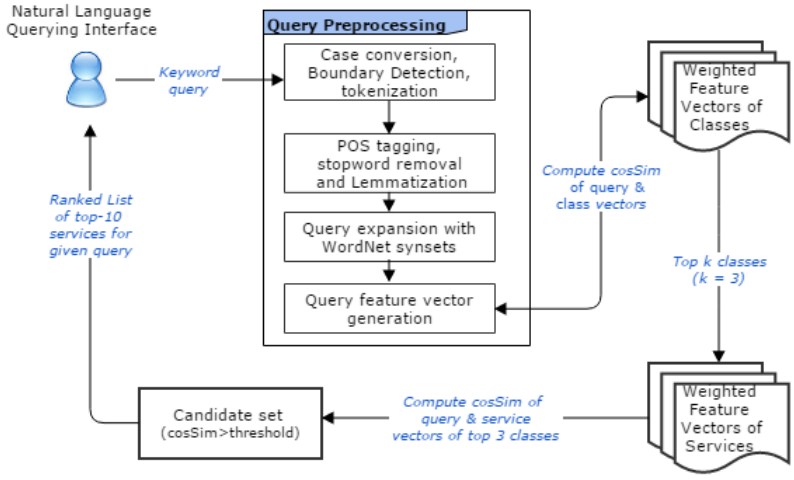

Fig. 2. Web Service Discovery process

6.1. *Query Preprocessing*

In order to understand user's context and only display relevant results, it is necessary to analyze the meaning and the structure of the query text entered by the user. In the proposed framework, we first parse and process the input query phrase and perform tasks like special character removal, word boundary detection etc. The query analysis process is composed of several preprocessing tasks as below -

1. *Case Conversion, Boundary Detection & Tokenization:* Natural language query is taken as a string (a single word or a phrase). All words are converted to lowercase and input is tidied to retain only non-special characters. Boundary detection is used to get individual words by considering a single space as the boundary between words (as is the case for English language), after which the identified query words are tokenized. A sample query *"find weather by city"* results in *'find', 'weather', 'by'* and *'city'* tokens.

2. *POS tagging, Stopword Removal & Lemmatization:* Part of Speech (POS) tagging is performed for the query tokens. For the previous example, this would result in {`find/VB weather/NN by/IN city/NN`}. We used the Stanford POS Tagger [40] for parts-of-speech tagging, which follows the Pennstate Treebank tagset [41]. Stop words, if any, are filtered out. Lemmatization is performed for the query in order to take care of the different forms of a natural language word, so they can be analyzed as a single item. Examples are words like *'booking'*, *'booked'* etc which are different forms of the word *'book'*. However, the word *'book'* may be used as a verb (e.g. *"book tickets"* or *"ticket booking"*) or as a noun (e.g. *"fiction book"*). Since POS tagging is performed on the

original query, the word will be appropriately tagged as noun or verb or adjective, based on which the lemmatizer performs suffix stripping. This ensures that different variants of a term can be reduced to only one. This also reduces the vector size, as the unique terms that make up the document are reduced. We used the Wordnet Lemmatizer available in Python's NLTK for performing lemmatization.

3. *Query Expansion:* WordNet synsets of each final query term are retrieved to capture the synonyms and morphological variants.
4. *Query feature vector generation:* The query terms and the synsets together form the feature vector $\vec{q_t}$ that represents the enhanced query.

6.2. *Query feature vector generation*

For retrieving relevant services for a given user query, we represented the services, the service classes and the query in the Vector Space Model (VSM), using their respective feature vectors. As all are represented as vectors of weighted index terms in VSM, vector operations like dot product can be performed. Thus, the process of web service discovery can be handled by matching similar documents, that are 'close' in the vector space. In VSM, services, classes and queries are represented as vectors as shown below -

$$\text{Class } c_j = (w_{1,j}, w_{2,j}, \ldots, w_{n,j})$$
$$\text{Service } s_k = (w_{1,k}, w_{2,k}, \ldots, w_{n,k})$$
$$\text{Query } q_t = (w_{1,t}, w_{2,t}, \ldots, w_{n,t})$$

Each term $t_{x,y}$ corresponds to the weight of a word w_i in that document (either class, service or query). Since, each service is pre-processed and its functional information is expressed as *top 5 weighted tags*, these form the service's feature vector. Similarly, since each class has *class tags* associated that capture the collective functionalities of its member services, we can represent a class also in the VSM using its tag vector. The query also is represented in the same way, i.e., the terms extracted after pre-processing the submitted user query form the *query vector*. We use these three vectors for relevant web service retrieval for a given query.

6.3. *Service Matching Process*

The first step in generating search results consisting of relevant services, is to compute cosine similarity of the query vector with the class vectors. The Cosine Similarity measure is used compute the level of matching between each class/service and the query. As the similarity between two vectors increases, the angle θ approaches zero, due to which cosine similarity value is closer to one. Similarly, as the dissimilarity between a class/service and query vector increases, the θ between their vector representations increases, due to which cosine similarity value is nearer to zero, thus indicating that the class/service is irrelevant to the query. In this way, the computed cosine similarity values can be used to generate a ranked list of classes/services for a given query. Cosine similarity values between the query and a class/service is computed using the equation (9) and (10).

$$\cos\theta_{c_j,q_t} = \frac{\vec{c_j} \cdot \vec{q_t}}{\|\vec{c_j}\|\|\vec{q_t}\|} = \frac{\sum_{i=1}^{n}(w_{i,j} \ . \ w_{i,q_t})}{\sqrt{\sum_{i=1}^{n} w_{i,j}^2} \cdot \sqrt{\sum_{i=1}^{n} w_{i,q_t}^2}} \tag{9}$$

$$\cos\theta_{s_k,q_t} = \frac{\vec{s_k}\cdot\vec{q_t}}{\|\vec{s_k}\|\|\vec{q_t}\|} = \frac{\sum_{i=1}^{n}(w_{i,k}\ .\ w_{i,q_t})}{\sqrt{\sum_{i=1}^{n} w_{i,k}^2}\cdot\sqrt{\sum_{i=1}^{n} w_{i,q_t}^2}} \tag{10}$$

where, c_j represents a given service class, s_k represents a given service, q_t represents the terms in the query and $w_{x,y}$ represents relative weight of a term with respect to entities x and y.

6.4. *Relevant Service Selection and Ranking*

Based on the computed cosine similarity values, a list of potentially relevant classes is generated and classes are ranked by their cosine similarity values. The first three classes from this ranked list are considered and the cosine similarity values between query vector and the member services of the three classes is computed. Again, a ranked list is generated based on the computed service *cos* θ values, which gives us the relative relevancy of services for the submitted query. This is displayed to the user in order of relevance. Algorithm 2 depicts the service discovery process.

Algorithm 2 Service selection and ranking process

Input:
Feature vector of the Query Q
Weighted feature vectors of generated classes and services.
Output: Candidate set of most relevant services for Q

string *Candidate-set* = ϕ
Compute cosine similarity value for feature vectors of query and classes.
Rank classes by their *cosine-similarity* value
for each service in the top-3 candidate classes in *ranked-list* **do**
    Compute cosine similarity with query vector
    return *cosine-similarity* value of service
end for
if $cosineSimilarity(queryVector, service) > Threshold$ **then**
    Add service to *Candidate-set*
end if
Rank services in *Candidate-set* by cosine similarity value.
Return *Candidate-set*

7. Experimental Evaluation

In this section, we present the experimental evaluation of the proposed approaches for functional semantics extraction, metadata generation, service classification and web service retrieval. First, we evaluated the quality of service tagging based on the extracted functional semantics using the approach discussed in section 4. After the processes of tag weighting and ranking, it was seen that meaningful tags were generated for each service (discussed further in subsection 7.1). Next, we evaluated the accuracy of service classification using the various classifiers. The impact of dimensionality reduction on the classification results was analyzed. After classification, class tags were generated automatically and the results of these processes are described in subsection 7.2. Finally, we measured the performance of web service retrieval using the basic keyword matching and with semantically analyzed query, and observed the effect on precision and recall (details in subsection 7.3).

7.1. *Quality of Service Tagging*

As stated earlier, the Zipf's Law based approach for determining splitting positions for token generation from badly named service elements achieved more than 94% accuracy. To evaluate the efficiency and quality of tagging services and classes, we conducted several experiments. Since tags are natural language text, we used manual inspection techniques for a sample random subset. The service tag quality was checked manually for a set of randomly selected services to determine relevancy of tags generated for each service. Table 1 lists the tags generated for one such sample service *calculateDistanceInMiles* and the corresponding tag weights. As seen from Table 1, the system generated meaningful tags that effectively represented the functionalities of individual web services.

Table 1. Tags generated and associated tag-weight for the service *calculateDistanceInMiles*

Tags	**Weight**
miles	2.48886180264
calculate	2.32660625689
distance	1.5484550065
geographic	1.14360118508
latitude	0.715121673171
longitude	0.715121673171

7.2. *Service Classification Accuracy*

For the experimental evaluation of the proposed system, we used the OWL-S TC 4 dataset[e], a publicly available dataset providing WSDL descriptions of real world web services. It was seen during similarity calculation that, best results were obtained for $vw_1 = 0.3$, $vw_2 = 0.25$, $vw_3 = vw_4 = vw_5 = 0.15$ and $\alpha = 0.4$. Once the feature vectors for each service was generated, SVD was performed and the reduced feature set was obtained for all the services in the dataset. Next, we used six different classifiers available in Weka and the accuracy was determined using 10-fold cross validation [39] method.

The Weka classifiers used for experimental evaluation of service classification accuracy using the feature vectors generated before SVD were - Bagging, Näive-Bayes, Decision Tree, Random Forest and Classification by Regression (CbR). An additional classifier, the Multilayer Perceptron (ANN) was also used after dimensionality reduction. It was seen that Näive Bayes, Decision tree and Bagging achieved almost similar accuracy, in the range of 70-72%, while Regression achieved an accuracy of around 77%. Since regression is based on assignment of weights to features, it can be inferred that following a similar approach before classification may result in higher accuracy. For Random Forests classifier, the observed accuracy is the highest and it is around 79%. It is an ensemble approach and works with a randomly selected set of features in each iteration. So, it can be inferred that some subsets of features tended to have higher accuracy than other subsets. If a feature selection method is applied on the vectors, it might achieve better accuracy. The accuracy obtained for service classification using each classifier for the dataset before and after dimensionality reduction is shown in Figure 3.

[e]http://projects.semwebcentral.org/projects/owls-tc/

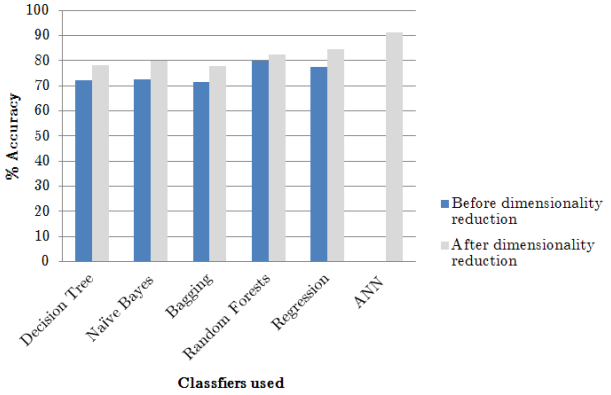

Fig. 3. Service Classification Accuracy

After applying dimensionality reduction, the accuracy of each of the classifiers improved by at least 10% in the case of all classifiers. We applied Multilayer Perceptron (ANN) as well in the second case, i.e. after dimension reduction. ANN takes a lot of time for training over a dataset with a large number of attributes, so it was not applied in the first case, when the dimensionality of the features was very high. In the case of the reduced feature set, ANN was very efficient and we achieved an accuracy of 91.14%. We used this class information to generate tags for each of the classes, as per the strategy described in subsection 5.1. Table 2 shows the tags generated for a few classes obtained after ANN classification with dimensionality reduction. It can be seen that the term weighting applied to the tags of the member services of each class resulted in good quality class tags.

Table 2. Some service classes and their corresponding tags

Domain	Number of Services		Class Tags
	As per dataset	After classification	
Economy	395	372	price, account, purchase, credit, tax, money.
Education	286	249	book, academic, author, fund, degree, price.
Travel	197	174	city, hotel, country, weather, destination, location.
Medical Care	73	71	medical, hospital, insurance, care, patient, doctor.
Geography	60	57	location, address, distance, zipcode, map, area, city.

7.3. *Web Service Retrieval*

Since the proposed system is modeled as per traditional IR methods, the most appropriate metrics to evaluate the performance are *precision* and *recall*. In this context, precision can

be defined as the fraction of retrieved services that are relevant for the given query. Recall is the fraction of all relevant services successfully retrieved for the given query.

$$Precision = \frac{|relevant\ services \cap retrieved\ services|}{|retrieved\ services|} \tag{11}$$

$$Recall = \frac{|relevant\ services \cap retrieved\ services|}{|relevant\ services|} \tag{12}$$

In order to evaluate the service discovery process, we conducted some experiments using different lengths of queries, i.e. by varying the number of keywords in the input query. Some query categories and sample queries are given in table 3. Queries of length 1 to 10 terms were submitted to the system. Since each query is pre-processed, an average of 3.4 keywords were obtained per query. Enhancing the query by including each keyword's WordNet synsets in the query vector added an average of 8.1 more search terms per query. Due to this, each query resulted in an average of 11.5 keywords to be submitted to the service retrieval module. Using these, we tried different queries on the system to determine retrieval performance.

Table 3. Examples of Queries used for performance evaluation

Query Characteristics	Example Query
Generalized query	"search books"
Particular query	"price of business class flight tickets"
Proper nouns	"weather in Seattle"

Calculating recall for each query when the number of underlying documents is diverse is somewhat problematic as the returned documents may be small when compared to the size of the dataset itself. Hence, the recall numbers would always be small. So we used a measure called relative recall [42]. In computing *relative recall*, we assume that the query which returned the most number of documents in the set of queries executed on the system is the one which achieved 100% recall. Therefore, the recall of all queries should be measured against this value and hence the concept of relative recall. The calculated precision-relative recall values are given in Table 4 and shown graphically in Figure 4.

Table 4. Observed precision & relative recall for *Keyword Query* and *Semantic Query*.

Query Length	For Keyword Query		For Semantic Query	
	Precision (%)	Relative Recall(%)	Precision (%)	Relative Recall(%)
1	82.73	6.31	81.19	27.90
2	79.17	19.98	85.38	58.41
3	74.60	24.75	91.63	69.42
4	69.97	28.40	89.29	78.40
6	65.21	34.71	83.14	85.51
8	59.44	39.48	77.69	90.37
10	53.12	43.65	73.38	100

As seen from Table 4, for the *Keyword Query approach*, it can be seen that precision is reasonably high at 82.73% when a single keyword is used, but recall is the lowest at 6.31%.

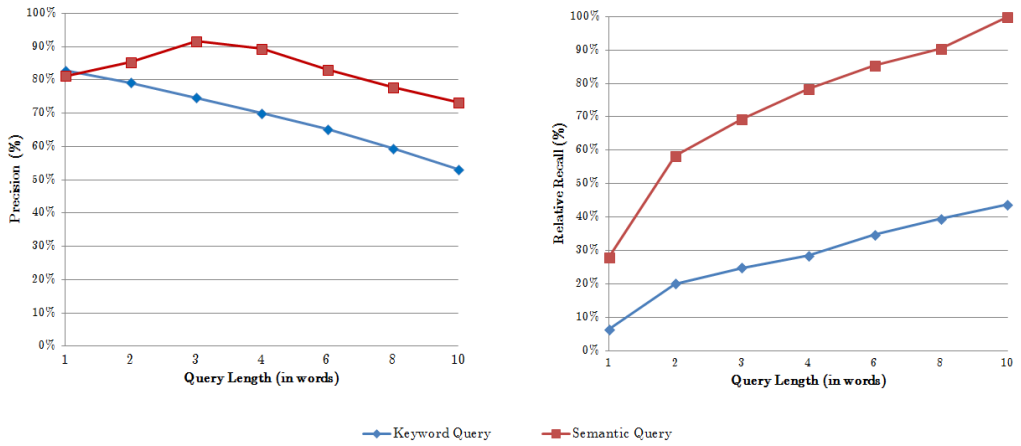

Fig. 4. Observed values of *precision* and *relative recall* for varied query lengths for *Keyword Query* and *Semantic Query* Approaches

This is because the system literally matches the single keyword to the tags of the classes formed and returns only those services having this tag. However, the precision is not 100% as sometimes services in other classes representing related domains may also have the same tag. Also, it can be observed that as the number of terms in the input query increases, the recall values improve to a small extent as more services would match the terms in the query. However, precision falls as the number of relevant services retrieved will continuously reduce.

The relative recall was 100% for a query length of 10 words, for the *Semantic Query approach*, because this essentially retrieved about 659 services which was the maximum number of services for the considered dataset. Hence, as per the definition of *relative recall*, we consider this as the query which resulted in 100% recall. However, due to this high recall, the precision observed was only about 73%. The highest precision of 91.63% was observed when the query length was 3, while the relative recall for this case was about 69.42%. This is because, in order to successfully tag certain terms in the query as verb or noun-phrase etc, the analysis of its nearest neighbor terms is required. For example, when the length of the query is 1, a term like *'book'* can be either a verb or a noun-phrase. Therefore, results will differ and precision will suffer if the user meant it as either word. However, if we consider the query *"book author"* which is a query of length 2, the context of the user is much clearer and so the precision improves. We observed the best precision when the query length was between 2 - 4, as the relationships between the words can be correctly captured and the most relevant services can be retrieved. Overall, the semantic analysis of the query resulted in approximately 16% increase in precision and 40% increase in recall, when compared to the *Keyword Query* approach.

8. Conclusion and Future Work

In this paper, a semantics based web service retrieval framework employing NLP techniques for the extraction of services' functional information was proposed. We used this extracted information for the computation of similarity between service pairs and for the generation of service tags. The similarity and tags were used in the service classification process using Weka,

after which generated classes are also tagged by taking the tags of all member services into consideration. A classification accuracy of 91% was obtained with the ANN classifier and class tags were generated for each class based on the tags of their member services. In addition, the framework also provided a natural language interface for capturing user requirements and context. Relevant services were retrieved based on the cosine similarity between the query vector and the class/service vectors. Experimental results showed that a query size of 2-4 terms resulted in the highest precision, as the relationships between the user context could be correctly captured and hence, the most relevant services could be retrieved. Also, a significant improvement of 16% increase in precision and more than 40% increase in recall was observed during service discovery when the *Semantic Query* approach was used.

As part of future work, we intend to also support complex querying, that will be required for composite service discovery. The process of composite service discovery requires the identification of simple services that can be chained together in a sequence, in the right order, to perform a given complex task. Such automatically identified composite service templates will be helpful for application designers. In summary, the aim is to develop a comprehensive web service discovery framework that provides an context-aware querying interface for both simple and composite service discovery.

References

1. M. Klusch, "Service discovery," *in Encyclopedia of Social Networks and Mining (ESNAM), R. Alhajj and J. Rokne, Eds. Springer*.
2. J. Fan and S. Kambhampati, "A snapshot of public web services," *ACM SIGMOD Record*, vol. 34, no. 1, pp. 24–32, 2005.
3. E. Al-Masri and Q. H. Mahmoud, "A broker for universal access to web services," in *Communication Networks and Services Research Conference, CNSR'09. Seventh Annual*, IEEE.
4. E. Al-Masri and Q. H. Mahmoud, "Investigating web services on the world wide web," in *17th Intl. Conf. on World Wide Web*, pp. 795–804, ACM, 2008.
5. A. Patil, S. Oundhakar, A. Sheth, and K. Verma, "Meteor-s web service annotation framework," in *13th international conference on World Wide Web*, pp. 553–562, ACM, 2004.
6. K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller, "Meteor-s WSDI: A scalable P2P infrastructure of registries for semantic publication and discovery of web services," *Information Technology and Management*, vol. 6, no. 1, 2005.
7. D. Martin, M. Paolucci, S. McIlraith, M. Burstein, *et al.*, "Bringing semantics to web services: The OWL-S approach," in *Semantic Web Services and Web Process Composition*, Springer, 2005.
8. N. Srinivasan, M. Paolucci, and K. Sycara, "An efficient algorithm for OWL-S based semantic search in UDDI," in *Semantic Web Services and Web Process Composition*, Springer, 2005.
9. U. Keller, R. Lara, A. Polleres, I. Toma, M. Kifer, and D. Fensel, "WSMO web service discovery," *WSML Working Draft D*, vol. 5, 2004.
10. X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for web services," in *30th Intl. Conf. on Very large data bases-Volume 30*, pp. 372–383, VLDB Endowment, 2004.
11. N. Steinmetz *et al.*, "Web service search on large scale," in *Service-Oriented Computing*, pp. 437–444, Springer, 2009.
12. S. Brockmans *et al.*, "Service-finder: First steps toward the realization of web service discovery at web scale," *Camogli (Genova), Italy June 25th, 2009 Co-located with SEBD*, p. 73, 2009.
13. R. Nayak and B. Lee, "Web service discovery with additional semantics and clustering," in *Web Intelligence, IEEE/WIC/ACM Intl. Conf. on*, 2007.
14. A. Sajjanhar, J. Hou, and Y. Zhang, "Algorithm for web services matching," in *Advanced Web Technologies and Applications*, pp. 665–670, Springer, 2004.

15. N. Chan, W. Gaaloul, and S. Tata, "A web service recommender system using vector space model and latent semantic indexing," in *Advanced Information Networking and Applications (AINA), 2011 IEEE International Conference on*, pp. 602–609, IEEE, 2011.
16. Y. Hao, Y. Zhang, and J. Cao, "Web services discovery and rank: An information retrieval approach," *Future generation computer systems*, vol. 26, no. 8, pp. 1053–1062, 2010.
17. A. V. Paliwal, B. Shafiq, J. Vaidya, H. Xiong, and N. Adam, "Semantics-based automated service discovery," *Services Computing, IEEE Transactions on*, vol. 5, no. 2, pp. 260–275, 2012.
18. T. Hofmann, "Probabilistic latent semantic indexing," in *22nd ACM SIGIR Intl. Conf. on Research and development in IR*, pp. 50–57, 1999.
19. C. Wu *et al.*, "An empirical approach for semantic web services discovery," in *Software Engineering, 2008. 19th Australian Conference on*, pp. 412–421, IEEE.
20. J. Ma, Y. Zhang, and J. He, "Web services discovery based on latent semantic approach," in *Web Services, 2008. IEEE Intl. Conf. on*, pp. 740–747, IEEE.
21. L. Fang, L. Wang, M. Li, J. Zhao, Y. Zou, and L. Shao, "Towards automatic tagging for web services," in *Web Services (ICWS), IEEE 19th International Conference on*, IEEE, 2012.
22. K. Elgazzar, A. E. Hassan, and P. Martin, "Clustering WSDL documents to bootstrap the discovery of web services," in *Web Services (ICWS), IEEE International Conference on*, IEEE, 2010.
23. R. L. Cilibrasi and P. M. Vitanyi, "The google similarity distance," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 3, pp. 370–383, 2007.
24. M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.
25. G. A. Miller *et al.*, "Introduction to wordnet: An on-line lexical database*," *Intl. Journal of lexicography*, vol. 3, no. 4, pp. 235–244, 1990.
26. H. Shima, "Ws4j-wordnet similarity for java," 2013. Available from `https://code.google.com/p/ws4j/`.
27. T. Pedersen, S. Patwardhan, and J. Michelizzi, "Wordnet:: Similarity: measuring the relatedness of concepts," in *Demonstration papers at HLT-NAACL 2004*, pp. 38–41, 2004.
28. J. M. Ponte and W. B. Croft, "A language modeling approach to information retrieval," in *21st ACM SIGIR Intl. Conf. on Research and development in IR*, pp. 275–281, 1998.
29. S. Bird, "NLTK: the natural language toolkit," in *COLING/ACL on Interactive presentation sessions*, pp. 69–72, Association for Computational Linguistics, 2006.
30. V. Klema and A. J. Laub, "The singular value decomposition: Its computation and some applications," *Automatic Control, IEEE Transactions on*, vol. 25, no. 2, pp. 164–176, 1980.
31. G. Ertek, "Text mining with rapidminer," *RapidMiner: Data Mining Use Cases and Business Analytics Applications*, p. 241, 2013.
32. G. Holmes, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Second Australian and New Zealand Conf. on*, pp. 357–361, IEEE, 1994.
33. K. P. Murphy, "Naive bayes classifiers," *University of British Columbia*, 2006.
34. P. H. Swain and H. Hauska, "The decision tree classifier: Design and potential," *Geoscience Electronics, IEEE Transactions on*, vol. 15, no. 3, pp. 142–147, 1977.
35. L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
36. M. Skurichina and R. P. Duin, "Bagging, boosting and the random subspace method for linear classifiers," *Pattern Analysis & Applications*, vol. 5, no. 2, pp. 121–135, 2002.
37. D. W. Hosmer and S. Lemeshow, "Introduction to the logistic regression model," *Applied Logistic Regression, Second Edition*, pp. 1–30, 2000.
38. C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.
39. R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, pp. 1137–1145, 1995.
40. K. Toutanova, "Stanford log-linear part-of-speech tagger," 2000.
41. M. Marcus, A. Taylor, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.
42. M. Frické, "Measuring recall," *Journal of Information Science*, vol. 24, no. 6, pp. 409–417, 1998.