Similarity Search for Web Services

Xin Dong Alon Halevy Jayant Madhavan Ema Nemes Jun Zhang

{lunadong, alon, jayant, enemes, junzhang}@cs.washington.edu
University of Washington, Seattle

Abstract

Web services are loosely coupled software components, published, located, and invoked across the web. The growing number of web services available within an organization and on the Web raises a new and challenging search problem: locating desired web services. Traditional keyword search is insufficient in this context: the specific types of queries users require are not captured, the very small text fragments in web services are unsuitable for keyword search, and the underlying structure and semantics of the web services are not exploited.

We describe the algorithms underlying the Woogle search engine for web services. Woogle supports similarity search for web services, such as finding similar web-service operations and finding operations that compose with a given one. We describe novel techniques to support these types of searches, and an experimental study on a collection of over 1500 web-service operations that shows the high recall and precision of our algorithms.

1 Introduction

Web services are loosely coupled software components, published, located, and invoked across the web. A web service comprises several operations (see examples in Figure 1). Each operation takes a SOAP package containing a list of input parameters, fulfills a certain task, and returns the result in an output SOAP package. Large enterprises are increasingly relying on web services as methodology for large-scale software development and sharing of services within an organization. If current trends continue, then in the future many applications will be built by piecing together web services published by third-party producers.

The growing number of web services available within an organization and on the Web raises a new and challenging search problem: locating desired web services. In fact, to address this problem, several simple search engines have recently sprung up [1, 2, 3, 4]. Currently, these engines provide only simple keyword search on web service descriptions.

As one considers search for web services in more detail, it becomes apparent that the keyword search paradigm is insufficient for two reasons. First, keywords do not capture the underlying semantics of web services. Current web service search engines return a particular service if its functionality description contains the keywords in the query; such search may miss results. For example, when searching zipcode, the web services whose descriptions contain term zip or postal code but not zipcode will not be returned.

Second, keywords do not suffice for accurately specifying users' information needs. Since a web-service operation is going to be used as part of an application, users would like to specify their search criteria more precisely than by keywords. Current web-service search engines often enable a user to explore the details of a particular web-service operation, and in some cases to try it out by entering an input value. Nevertheless, investigating a single web-service operation often requires several browsing steps. Once users drill down all the way and find the operation inappropriate for some reason, they want to be able to find *similar* operations to the ones just considered, as opposed to laboriously following parallel browsing patterns. Similarly, users may want to find operations that take similar inputs (respectively, outputs), or that can *compose* with the current operation being browsed.

To address the challenges involved in searching for web services, we built Woogle[1], a web-service search engine. In addition to simple keyword searches, Woogle supports similarity search for web services. A user can ask for web-service operations similar to a given one, those that take similar inputs (or

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[1]See http://www.cs.washington.edu/woogle

W_1: Web Service: GlobalWeather
  Operation: GetTemperature
    Input: Zip
    Output: Return
W_2: Web Service: WeatherFetcher
  Operation: GetWeather
    Input: PostCode
    Output: TemperatureF, WindChill, Humidity
W_3: Web Service: GetLocalTime
  Operation: LocalTimeByZipCode
    Input: Zipcode
    Output: LocalTimeByZipCodeResult
W_4: Web Service: PlaceLookup
  Operation1: CityStateToZipCode
    Input: City, State
    Output: ZipCode
  Operation2: ZipCodeToCityState
    Input: ZipCode
    Output: City, State

Figure 1: Several example web services (not including their textual descriptions). Note that each web service includes a set of operations, each with input and output parameters. For example, web services W_1 and W_2 provide weather information.

outputs), and those that compose with a given one. This paper describes the novel techniques we have developed to support these types of searches, and experimental evidence that shows the high accuracy of our algorithms. In particular, our contributions are the following:

1. We propose a basic set of search functionalities that an effective web-service search engine should support.
2. We describe algorithms for supporting similarity search. Our algorithms combine multiple sources of evidence in order to determine similarity between a pair of web-service operations. The key ingredient of our algorithm is a novel clustering algorithm that groups names of parameters of web-service operations into semantically meaningful concepts. These concepts are then leveraged to determine similarity of inputs (or outputs) of web-service operations.
3. We describe a detailed experimental evaluation on a set of over 1500 web-service operations. The evaluation shows that we can provide both high precision and recall for similarity search, and that our techniques substantially improve on naive keyword search.

The paper is organized as follows. Section 2 begins by placing our search problem in the context of the related work. Section 3 formally defines the similarity search problem for web services. Section 4 describes the algorithm for clustering parameter names, and Section 5 describes the similarity search algorithm. Section 6 describes our experimental evaluation. Section 7 discusses other types of search that Woogle supports, and Section 8 concludes.

2 Related Work

Finding similar web-service operations is closely related to three other matching problems: text document matching, schema matching, and software component matching.

Text document matching: Document matching and classification is a long-standing problem in information retrieval (IR). Most solutions to this problem (e.g. [10, 20, 27, 19]) are based on term frequency analysis. However, these approaches are insufficient in the web service context because text documentations for web-service operations are highly compact, and they ignore structure information that aids capturing the underlying semantics of the operations.

Schema matching: The database community has considered the problem of automatically matching schemas [24, 12, 13, 22]. The work in this area has developed several methods that try to capture clues about the semantics of the schemas, and suggest matches based on them. Such methods include linguistic analysis, structural analysis, the use of domain knowledge and previous matching experience. However, the search for *similar* web-service operations differs from schema matching in two significant ways. First, the granularity of the search is different: operation matching can be compared to finding a similar schema, while schema matching looks for similar components in two given schemas that are assumed to be related. Second, the operations in a web service are typically much more loosely related to each other than are tables in a schema, and each web service in isolation has much less information than a schema. Hence, we are unable to adapt techniques for schema matching to this context.

Software component matching: Software component matching is considered important for software reuse. [28] formally defines the problem by examining signature (data type) matching and specification (program behavior) matching. The techniques employed there require analysis of data types and post-conditions, which are not available for web services.

Some recent work (e.g., [9, 23]) has proposed annotating web services manually with additional semantic information, and then using these annotations to compose services [8, 26]. In our context, annotating the collection of web services is infeasible, and we rely on only the information provided in the WSDL file and the UDDI entry.

In [15] the authors studied the supervised classification and unsupervised clustering of web services.

Our work differs in that we are doing unsupervised matching at the operation level, rather than supervised classification at the entire web service level. Hence, we face the challenge of understanding operations in a web service from very limited amount of information.

3 Web Service Similarity Search

We begin by briefly describing the structure of web services, and then we motivate and define the search problem we address.

3.1 The Structure of Web Services

Each web service has an associated WSDL file describing its functionality and interface. A web service is typically (though not necessarily) published by registering its WSDL file and a brief description in UDDI business registries. Each web service consists of a set of operations. For each web service, we have access to the following information:

- **Name and text description**: A web service is described by a name, a text description in the WSDL file, and a description that is put in the UDDI registry.
- **Operation descriptions**: Each operation is described by a name and a text description in the WSDL file.
- **Input/Output descriptions**: Each input and output of an operation contains a set of parameters. For each parameter, the WSDL file describes the name, data type and arity (if the parameter is of array type). Parameters may be organized in a hierarchy by using complex types.

3.2 Searching for Web Services

To motivate similarity search for web services, consider the following typical scenario. Users begin a search for web services by entering keywords relevant to the search goal. They then start inspecting some of the returned web services. Since the result of the search is rather complex, the users need to drill down in several steps. They first decide which web service to explore in detail, and then consider which specific operations in that service to look at. Given a particular operation, they will look at each of its inputs and outputs, and if the engine provides a *try it* feature, they will try entering some value for the inputs.

At this point, the users may find that the web service is inappropriate for some reason, but *not* want to have to repeat the same process for each of other potentially relevant services. Hence, our goal is to provide a more direct method for searching, given that the users have already explored a web service in detail. Suppose they explored the operation GetTemperature in W_1. We identify the following important similarity search queries they may want to pose:

Similar operations: Find operations with similar functionalities. For example, the web-service operation GetWeather in W_2 is similar to the operation GetTemperature in W_1. Note that we are searching for specific *operations* that are similar, rather than similar web services. The latter type of search is typically too coarse for our needs. There is no formal definition for operation similarity, because, just like in other types of search, similarity depends on the specific goal in the user's mind. Intuitively, we consider operations to be similar if they take similar inputs, produce similar outputs, and the relationships between the inputs and outputs are similar.

Similar inputs/outputs: Find operations with similar inputs. As a motivating example for such a search, suppose our goal is to collect a variety of information about locations. While W_1 provides weather, operations LocalTimeByZipCode in W_3 and ZipCodeToCityState in W_4 provide other information about locations, and thereby may be of interest to the user.

Alternatively, we may want to search for operations with similar outputs, but different inputs. For example, we may be looking for temperature, but the operation we are considering takes zipcode as input, while we need one that takes city and state as input.

Composible operations: Find operations that can be composed with the current one. One of the key promises of building applications with web services is that one should be able to compose a set of given services to create ones that are specific to the application's needs. In our example, there are two opportunities for composition. In the first case, the output of the operation is similar to the input of the given operation, such as CityStateToZipCode in W_4. Composing CityStateToZipCode with GetWeather in W_1 offers another option for getting the weather when the zipcode is not known. In the second case, the output of the given operation may be similar to the input of another operation; e.g., one that transforms Centigrade and Fahrenheit and thereby produces results in the desired scale.

In this paper we focus on the following two problems, from which we can easily build up the above search capabilities.

Operation matching: *Given a web-service operation, return a list of similar operations.* □

Input/output matching: *Given the input (respectively, output) of a web-service operation, return a* *list of web-service operations with similar inputs (respectively, outputs).* □

We note that these two problems are also at the core of two other types of search that Woogle supports (See Section 7): *template search* and *composition search*. Template search goes beyond keyword search by specifying the functionality, input and output of a desired operation. Composition search returns not only single operations, but also compositions of operations that fulfill the user's need.

3.3 Overview of Our Approach

Similarity search for web services is challenging because neither the textual descriptions of web services and their operations nor the names of the input and output parameters completely convey the underlying semantics of the operation. Nevertheless, knowledge of the semantics is important to determining similarity between operation.

Broadly speaking, our algorithm combines multiple sources of evidences to determine similarity. In particular, it will consider similarity between the textual descriptions of the operations and of the entire web services, and similarity between the parameter names of the operations. The key ingredient of the algorithm is a technique that clusters parameter names in the collection of web services into *semantically meaningful* concepts. By comparing the concepts that input or output parameters belong to, we are able to achieve good similarity measures. Section 4 describes the clustering algorithm, and Section 5 describes how we combine the multiple sources of evidence.

4 Clustering Parameter Names

To effectively match inputs/outputs of web-service operations, it is crucial to get at their underlying semantics. However, this is hard for two reasons. First, parameter naming is dependent on the developers' whim. Parameter names tend to be highly varied given the use of synonyms, hypernyms, and different naming rules. They might even not be composed of proper English words—there may be misspellings, abbreviations, etc. Therefore, lexical references, such as *Wordnet* [5], are hard to apply. Second, inputs/outputs typically have few parameters, and the associated WSDL files rarely provide rich descriptions for parameters. Traditional IR techniques, such as TF/IDF [25] and LSI [11], rely on word frequencies to capture the underlying semantics and thus do not apply well.

A parameter name is typically a sequence of concatenated words (not necessarily proper English words), with the first letter of every word capitalized (*e.g.*, LocalTimeByZipCodeResult). Such words are referred to as *terms*. We exploit the co-occurrence of terms in web service inputs and outputs to cluster terms into meaningful concepts. As we shall see later, using these concepts, in addition to the original terms, greatly improves our ability to identify similar inputs/outputs and hence find similar web service operations.

Applying an off-the-shelf text clustering algorithm directly to our context does not perform well because the web service inputs/outputs are sparse. For example, whereas synonyms tend to occur in the same document in an IR application, they seldom occur in the same operation input/output; therefore, they will not get clustered. Our clustering algorithm is a refinement of *agglomerative* clustering. We begin by describing a particular kind of association rules that capture our notion of term co-occurrence and then describe the clustering algorithm.

4.1 Clustering Parameters by Association

We base our clustering on the following heuristic: *parameters tend to express the same concept if they occur together often*. This heuristic is validated by our experimental results. We use it to cluster parameters by exploiting their conditional probabilities of occurrence in inputs and outputs of web-service operations. Specifically, we are interested in *association rules* of the form:

$$t_1 \to t_2 \ (s, c)$$

In this rule, t_1 and t_2 are two terms. The *support*, s, is the probability that t_1 occurs in an input/output; *i.e.*, $s = P(t_1) = \frac{\|IO_{t_1}\|}{\|IO\|}$, where $||IO||$ is the total number of inputs and outputs of operations, and $||IO_{t_1}||$ is the number of inputs and outputs that contain t_1. The *confidence*, c, is the probability that t_2 occurs in an input or output, given that t_1 is known to occur in it; *i.e.*, $c = P(t_2|t_1) = \frac{\|IO_{t_1,t_2}\|}{\|IO_{t_1}\|}$, where $||IO_{t_1,t_2}||$ is the number of inputs and outputs that contain both t_1 and t_2. Note that the rule $t_1 \to t_2(s_{12}, c_{12})$ and the rule $t_2 \to t_1(s_{21}, c_{21})$ may have different support and confidence values. These rules can be efficiently computed using the A-Priori algorithm [7].

4.2 Criteria for Ideal Clustering

Ideally, parameter clustering results should have the following two features:

1. Frequent and rare parameters should be left unclustered; strongly connected parameters in-between are clustered into concepts. First, not clustering frequent parameters is consistent with the IR community's observation that such

technique leads to the best performance in automatic query expansion [16]. Second, leaving rare parameters unclustered avoids over-fitting.

2. The *cohesion* of a concept—the connections between parameters inside the concept—should be strong; the *correlation* between concepts—the connections between parameters in different concepts—should be weak.

Traditionally, cohesion is defined as the sum of squares of Euclidean distances from each point to the center of the cluster it belongs to; correlation is defined as the sum of squares of distances between cluster centers [14]. This definition does not apply well in our context because of "the curse of dimensionality": our feature sets are so large that a Euclidean distance measure is no longer meaningful. We hence quantify the cohesion and correlation of clusters based on our association rules.

We say that t_1 is *closely associated* to t_2 if the rule $t_1 \rightarrow t_2$ has a confidence greater than threshold t_c. The threshold t_c is chosen manually to be the value that best separates correlated and uncorrelated pairs of terms.

Given a cluster I, we define the *cohesion* of I as the percentage of closely associated term pairs over all term pairs. Formally,

$$coh_I = \frac{\| \{i, j \mid i, j \in I, i \neq j, i \rightarrow j(c > t_c)\} \|}{||I||(||I|| - 1)}$$

where $i \rightarrow j(c > t_c)$ is the association rule for term i and j. As a special case, the cohesion of a single-term cluster is 1.

Given clusters I and J, we define the *correlation* between I and J as the percentage of closely associated cross-cluster term pairs. Formally,

$$cor_{IJ} = \frac{C(I,J) + C(J,I)}{2 \| I \| \| J \|}$$

where $C(I,J) = \| \{i, j \mid i \in I, j \in J, i \rightarrow j(c > t_c)\} \|$.

To measure the overall quality of a clustering $\mathcal{C}$, we define the *cohesion/correlation score* as

$$score_{\mathcal{C}} = \frac{\frac{\sum_{I \in \mathcal{C}} coh_I}{\|\mathcal{C}\|}}{\frac{\sum_{I,J \in \mathcal{C}, I \neq J} cor_{IJ}}{\|\mathcal{C}\|(\|\mathcal{C}\|-1)/2}} = \frac{(||\mathcal{C}|| - 1)\sum_{I \in \mathcal{C}} coh_I}{2\sum_{I,J \in \mathcal{C}, I \neq J} cor_{IJ}}$$

The cohesion/correlation score captures the trade-off between having a high cohesion score and a low correlation score. Our goal is to obtain a high $score_{\mathcal{C}}$ that will indicate tight connections inside clusters and loose connections between clusters.

4.3 Clustering Algorithm

We can now describe our clustering algorithm as a series of refinements to the classical agglomerative clustering [18].

4.3.1 The basic agglomeration algorithm

Agglomerative clustering is a bottom-up version of hierarchical clustering. Each object is initialized to be a cluster of its own. In general, at each iteration the two most similar clusters are merged until no more clusters can be merged.

In our context, each term is initialized to be a cluster of its own; *i.e.*, there are as many clusters as terms. The algorithm proceeds in a greedy fashion. It sorts the association rules in descending order first by the confidence and then by the support. Infrequent rules with less than a minimum support t_s are discarded. At every step, the algorithm chooses the highest ranked rule that has not been considered previously. If the two terms in the rule belong to different clusters, the algorithm merges the clusters. Formally, the condition that triggers merging cluster I and J is

$$\exists i \in I, j \in J \ . \ i \rightarrow j(s > t_s, c > t_c)$$

where i and j are terms. The threshold t_s is chosen to control the clustering of terms that do not occur frequently. We note that in our experiments the results of operation and input/output matching are not sensitive on the values of t_s and t_c.

4.3.2 Increasing cluster cohesion

The basic agglomerative algorithm merges two clusters together when any two terms in the two clusters are closely associated. The merge condition is very loose and can easily result in low cohesion of clusters. To illustrate, suppose there is a concept for weather, containing temperature as a term, and a concept for address, containing zip as a term. If, when operations report temperature, they often report the area zipcode as well, then the confidence of rule temperature $\rightarrow$ zip is high. As a result, the basic algorithm will inappropriately combine the weather concept and the address concept.

The cohesion of a cluster is decided by the association of each pair of terms in the cluster. To ensure that we obtain clusters with high cohesion, we merge two clusters only if they satisfy a stricter condition, called *cohesion condition.*

Given a cluster C, a term is called a *kernel* term if it is closely associated with at least half[2] of the remaining terms in C. Our cohesion condition requires that *all the terms in the merged cluster be kernel terms.* Formally, we merge two clusters I and J only if they satisfy the cohesion condition:

$$\forall i \in I \cup J \ . \ \| \{j \mid j \in I \cup J, i \neq j, i \rightarrow j(c > t_c)\} \| \geq \frac{1}{2}(||I|| + ||J|| - 1)$$

[2]We tried different values for this fraction and found $\frac{1}{2}$ yielded the best results.

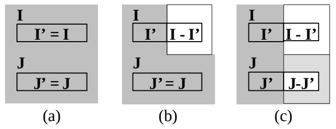


Figure 2: Splitting and merging clusters

4.3.3 Splitting and Merging

A greedy algorithm pursues local optimal solutions at each step, but usually cannot obtain the global optimal solution. In parameter clustering, an inappropriate clustering decision at an early stage may prevent subsequent appropriate clustering. Consider the case where there is a cluster for zipcode {zip, code}, formed because of the frequent occurrences of parameter ZipCode. Later we need to decide whether to merge this cluster with another cluster for address {state, city, street}. The term zip is closely associated with state, city and street, but code is not because it also occurs often in other parameters such as TeamCode and ProxyCode, which typically do not co-occur with state, city or street. Consequently, the two clusters cannot merge; the clustering result contrasts with the ideal one: {state, city, street, zip} and {code}.

The solution to this problem is to split already-formed clusters so as to obtain a better set of clusters with a higher cohesion/correlation score. Formally, given clusters I and J, we denote

$$\begin{aligned} I' = \{i \mid i \in I, ||\{j \mid j \in I \cup J, i \rightarrow j(c > t_c)\}|| \\ \geq \frac{1}{2}(||I|| + ||J|| - 1) \\ J' = \{j \mid j \in J, ||\{i \mid i \in I \cup J, j \rightarrow i(c > t_c)\}|| \\ \geq \frac{1}{2}(||I|| + ||J|| - 1) \end{aligned} \quad (1)$$

Intuitively, I' (respectively, J') denotes the set of terms in I that are closely associated with terms in the union of I and J. Our algorithm makes splitting decision depending on which of the four following cases occurs:

- If $I' = I, J' = J$, then I and J can be merged directly (see Figure 2(a)).
- If $I' \neq I, J' = J$, then merging I and J directly disobeys the cohesion condition. There are two options: one is to split I into I' and $I - I'$, and then merge I' with J (see Figure 2(b)); the other is not to split or merge. We decide in two steps: the first step checks whether the merged result in the first option satisfies the cohesion condition; if so, the second step computes the cohesion/correlation score for each option, and chooses the option with a higher score. The decision is similar for the case where $J' \neq J, I' = I$.
- If $I' \neq I, J' \neq J$, then again, merging I and J directly disobeys the cohesion condition. There are two options: one is to split I into I' and $I - I'$, split J into J' and $J - J'$, and then merge I' with J' (see Figure 2(c)); the other is not to split or merge. We choose an option in two steps: the first step checks whether in the first option, the merged result satisfies the cohesion condition; if so, the second step computes the cohesion/correlation score for each option, and chooses the option with a higher score.

After the above processing, the merged cluster necessarily satisfies the cohesion condition. However, the clusters that are split from the original clusters may not. To ensure cohesion, we further split such clusters: each time, we split the cluster into two, one containing all kernel terms, and the other containing the rest. We repeat splitting until eventually all result clusters satisfy the cohesion condition. Note that applying such splitting strategy on an arbitrary cluster may generate clusters of small size. Therefore, we do not merge two clusters directly (without applying the above judgment) and then split the merged cluster.

Remark 4.1. *Our splitting-and-clustering technique is different from the dynamic modeling in the Chameleon algorithm [17], which also first splits and then merges. We do splitting and clustering at each step of the greedy algorithm. The Chameleon algorithm first considers the whole set of parameters as a big cluster and splits it into relatively small sub-clusters, and then repeatedly merges these sub-clusters.* □

4.3.4 Removing noise

Even with splitting, the results may still have terms that do not express the same concept as other terms in its cluster. We call such terms *noise terms*. To illustrate how noise terms can be formed, we continue with the zipcode example. Suppose there is a cluster for address {city, state, street, zip, code}, where code is a noise term. The cluster is formed because the rules zip → city, zip → state, and zip → street all have very high confidence, *e.g.*, 90%; even if the rule code → zip has a lower confidence, *e.g.*, 50%, the rules code → city, code → state, and code → street can still have high confidence.

We use the following heuristic to detect noise terms. A term is considered to be noise if in half of its occurrences there are no other terms from the same concept. After one pass of the greedy algorithm (considering all association rules above a given threshold), we scan the resulting concepts to remove noise terms. Formally, for a term t, denote $||IO_t||$ as the number of inputs/outputs that contain t, and

```
procedure MergeParameters($\mathcal{T}, \mathcal{R}$) return ($\mathcal{C}$)
// $\mathcal{T}$ is the term set, $\mathcal{R}$ is the association rule set
// $\mathcal{C}$ is the result concept set
  for ($i = 1, n$) $C_i = \{t_i\}$; //initiate clusters
  sort $\mathcal{R}$ first by the descending order of confidence,
    then by the descending order of support value;
  for each ($r : t_1 \rightarrow t_2(s > t_s, c > t_c)$ in $\mathcal{R}$)
    if $t_1$ and $t_2$ are in different clusters $I$ and $J$
      Compute $I'$ and $J'$ according to formula (1);
      if ($I' = I \wedge J' = J$) merge $I$ and $J$;
      else if (splitting and merging satisfies the
        cohesion condition and has a higher $score_C$)
          split and merge;
          if ($I'' = I - I'$ and/or $J'' = J - J'$
            does not observe the cohesion condition)
            split $I''$ and/or $J''$ iteratively;
  scan inputs/outputs and remove noise terms;
  return result clusters;
```

Figure 3: Algorithm for parameter clustering

$||SIO_t||$ as the number of inputs/outputs that contain t but no other terms in the same concept of t. We remove t from the concept if $||SIO_t|| \geq \frac{1}{2}||IO_t||$.

4.3.5 Putting it all together

Figure 3 puts all the pieces together, and shows the details of a single pass of the clustering algorithm.

The above algorithm still has two problems. First, the cohesion condition is too strict for large clusters, so it may prevent closely associated large clusters to merge. Second, early inappropriate merging may prevent later appropriate merging. Although we do splitting, the terms taken off from the original clusters may have already missed the chance to merge with other closely associated terms. We solve the problems by running the clustering algorithm iteratively. After each pass, we replace each term with its corresponding concept, re-collect association rules, and then re-run the clustering algorithm. This process continues when no more clusters can be merged.

We illustrate with an example that the iteration of clustering does not sharply loosen the clustering condition. Consider the case where {zip} is not clustered with {temperature, windchill, humidity}, because zip is closely associated with only temperature, but not the other two. Another iteration of clustering will replace each occurrence of temperature, windchill and humidity with a single concept, say weather. The term zip will be closely associated with weather; however, the term weather is not necessarily closely associated with zip, because that requires zip to occur often when any of temperature, windchill, or humidity occurs. Thus, the iteration will (correctly) keep the two clusters.

4.4 Clustering Results

We now briefly outline the results of our clustering algorithm. Our dataset, which we will describe in detail in Section 6, contains 431 web services and 3148 inputs/outputs. There are a total of 1599 terms. The clustering algorithm converges after the seventh run. It clusters 943 terms into 182 concepts. The rest 656 terms, including 387 infrequent terms (each occurs in at most 3 inputs/outputs) and 54 frequent terms (each occurs in at least 30 of the inputs/outputs) are left unclustered. There are 59 *dense* clusters, each with at least 5 terms. Some of them correspond roughly to the concepts of address, contact, geology, maps, weather, finance, commerce, statistics, and baseball, etc. The overall cohesion is 0.96, correlation is 0.003, and average cohesion for the dense clusters is 0.76. This result observes the two features of an ideal clustering.

5 Finding Similar Operations

In this section we describe how to predict similarity of inputs/outputs sets and of web-service operations. We will determine similarity by combining multiple sources of evidence. The intuition behind our matching algorithm is that the similarity of a pair of inputs (or outputs) is related to the similarity of the parameter names, that of the concepts represented by the parameter names, and that of the operations they belong to. Note that parameter name similarity compares inputs/outputs on a fine-grained level, and concept similarity compares inputs/outputs on a coarse-grained level. The similarity between two web-service operations is related to the similarity of their descriptions, that of their inputs and outputs, and that of their host web services.

Input/output similarity: We identify the input i of a web-service operation op with a vector $i = (p_i, c_i, op)$, where p_i is the set of input parameter names, and c_i is the set of concepts associated with the parameter names (as determined by the clustering algorithm described in Section 4). While comparing a pair of inputs, we determine the similarity on each of the three components separately, and then combine them. We treat op's output o as a vector $o = (p_o, c_o, op)$, and process it analogously.

Web-service operation similarity: We identify a web-service operation op with a vector $op = (w, f, i, o)$, where w is the text description of the web service to which op belongs, f is the textual description of op, and i and o denote the input and output parameters. Here too, we determine similarity by combining the similarities of the individual components of the vector.

Observe that there is a recursive relationship between the similarity of inputs/outputs and the similarity of web-service operations. Intuitively, this relationship holds because each one depends on the other, and any decision on how to break this recursive relationship would be arbitrary. In Section 5.2 we show that with sufficient care for the choice of the combination weights, we can guarantee that the recursive computation converges.

5.1 Computing Individual Similarities

We now describe how we compute similarities for each one of the components of the vectors.

Input/output parameter name similarity: We consider the terms in an input/output as a bag of words and use the TF/IDF (*Term Frequency/Inverse Document Frequency*) measure [25] to compute the similarity of two such bags.

To improve our accuracy, we pre-process the terms as follows.

1. Perform word stemming and remove stopwords. Stemming improves recall by removing term suffixes and reducing all forms of a term to a single stemmed form. Stopword removal improves precision by eliminating words with little substantive meaning.
2. Group terms with close edit distance [21] and replace terms in a group with a normalized form. This step helps normalize misspelled and abbreviated terms.
3. Remove from the output bag the terms that refer to the inputs. For example, in the output parameter LocalTimeByZipCodeResult, the term By indicates that the following terms describe inputs; thus, terms Zip and Code can be removed.
4. Extract additional information from names of web-service operations. Most operations are named after the output (*e.g.*, GetWeather), and some include input information (*e.g.*, ZipCodeToCityState). We put such terms into the corresponding input/output bag.

Input/output concept similarity: To compute the similarity of the concepts represented by the inputs/outputs, we replace each term in the bag of words described above with its corresponding concept, and then use the TF/IDF measure. Note that the clustering algorithm is applied on the input/output terms *after* preprocessing.

Operation description similarity: To compute the similarity of operation descriptions, we consider the tokenized operation name and WSDL documentation as a bag of words, and use the TF/IDF measure. Furthermore, we supplement information by adding the terms in their inputs and outputs to the bag of words.

Web service description similarity: To compute the similarity of web service descriptions, we create a bag of words from the following: the tokenized web service name, WSDL documentation and UDDI description, the tokenized names of the operations in the web service, and their input and output terms. We again apply TF/IDF on the bag of words.

5.2 Combining Individual Similarities

We use a linear combination to combine the similarity of each component of the operation. Each type of similarity is assigned a weight that is dependent on its relevance to the overall similarity. Currently we set the weights manually based on our analysis of the results from different trials. Learning these weights based on direct or indirect user feedback is a subject of future work.

As noted earlier, there is a recursive dependency between the similarity of operations and that of inputs/outputs. We prove that computing the recursive similarities ultimately converges.

Proposition 1. *Computing operation similarity and input/output similarity converges.* □

Proof (Sketch): Let S_{op}, S_i and S_o be the similarity of operations, of inputs, and of outputs. Let w_i and w_o be the weights for input similarity and output similarity in computing operation similarity, and w_{op} be the weight for operation similarity in computing input/output similarity.

We start by assigning zero to the operation similarity, and based upon it compute input/output similarity and operation similarity iteratively. We can prove that if $z = w_{op}(w_{in} + w_{out}) < 1$, the computation converges and the results are:

$$\begin{aligned} S_{op}^{(\infty)} &= S_{op}^{(0)} \cdot \frac{1}{1-z} \\ S_i^{(\infty)} &= S_i^{(0)} + S_{op}^{(0)} \cdot \frac{w_{op}}{1-z} \\ S_o^{(\infty)} &= S_o^{(0)} + S_{op}^{(0)} \cdot \frac{w_{op}}{1-z} \end{aligned}$$

where $s_{op}^{(0)}, s_i^{(0)}$ and $s_o^{(0)}$ are the results of the first round, and $s_{op}^{(\infty)}, s_i^{(\infty)}$ and $s_o^{(\infty)}$ are the converged results. □

6 Experimental Evaluation

We now describe a set of experiments that validate the performance of our matching algorithms. Our goal is to show that we produce high precision and recall on similarity queries and to investigate the contribution of the different components of our method.

6.1 Experimental Setup

We implemented a web-service search engine, called Woogle, that has access to 790 web services from the main authoritative UDDI repositories. The coverage of Woogle is comparable to that of the other web-service search engines [1, 2, 3, 4]. We ran our experiments on the subset of web services whose associated WSDL files are accessible from the web, so we can extract information about their functionality descriptions, inputs and outputs. This set contains 431 web services, and 1574 operations in total.

Woogle performs parameter clustering, operation matching and input/output matching offline, and stores the results in a database. TF/IDF was implemented using the publicly available *Rainbow* [6] classification tool.

Our experiments compared our method, which we refer to as WOOGLE, with a couple of naive algorithms FUNC and COMB. The FUNC method matches operations by comparing only the words in the operation names and text documentation. The COMB method considers the words mentioned in the web service names, descriptions and parameter names as well; in contrast to WOOGLE, these words are all put into a single bag of words.

Performance Measure: We measured overall performance using *recall(r), precision(p), R-precision(p_r)* and *Top-k precision (p_k)*. Consider these measures for operation matching. Let *Rel* be the set of relevant operations, *Ret* be the set of returned operations, *Retrel* be the set of returned relevant operations, and $Retrel_k$ be the set of relevant operations in the top k returned operations. We define

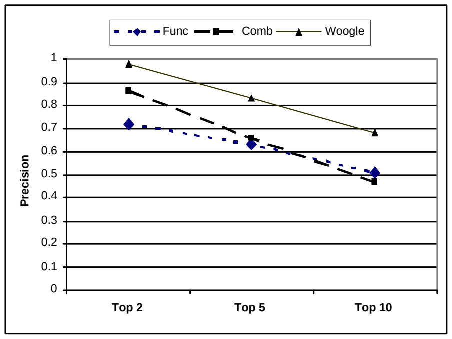
$$p = \frac{|Retrel|}{|Ret|}, \quad r = \frac{|Retrel|}{|Rel|}$$

$$p_k = \frac{|Retrel_k|}{k}, \quad p_r = p_{|Rel|} = \frac{|Retrel_{|Rel|}|}{|Rel|}$$

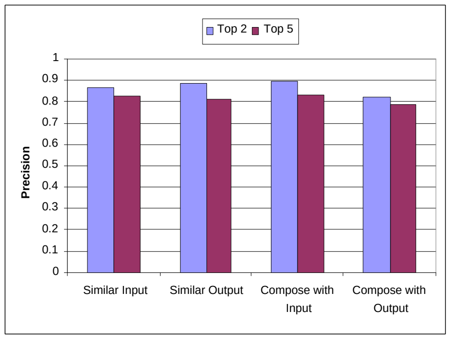
Among the above measures, p_r is considered to most precisely capture the precision and ranking quality of a system. We also plotted the *recall/precision curve (R-P curve)*. In an R-P curve figure, the X-axis represents recall, and the Y-axis represents precision. An ideal search engine has a horizontal curve with a high precision value; a bad search engine has a horizontal curve with a low precision value. The R-P curve is considered by the IR community as the most informative graph showing the effectiveness of a search engine.

6.2 Measuring Precision

Given a web service, Woogle generates five lists: similar operations, operations with similar inputs, operations with similar outputs, operations that compose with the output of the given operation, and operations that compose with the input of the given operation. We evaluated the precision of these returned lists, and report the average top-2, top-5 and top-10 precision.



(a)



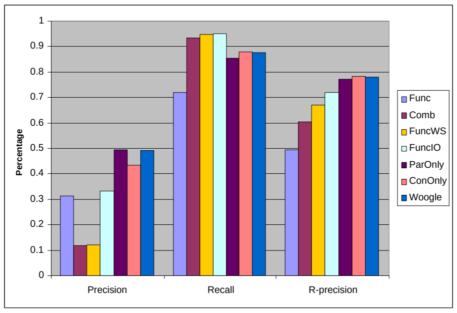
(b)

Figure 4: Top-k precision for Woogle similarity search.

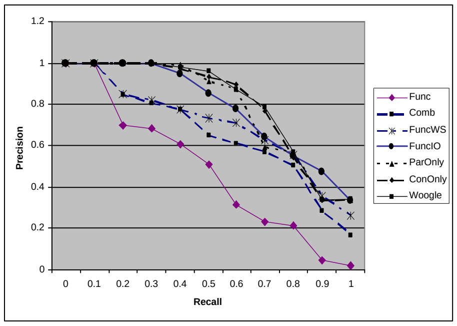
We selected a benchmark of 25 web-service operations for which we tried to obtain similar operations from our entire collection. When selecting these, we ensured that they are from a variety of domains and that they have different input/output sizes and description sizes. To ensure the top-10 precision is meaningful, we selected only operations for which WOOGLE and COMB both returned more than 10 relevant operations. (FUNC may return less than 10 relevant operations because typically it obtains result sets of very small size.)

Figure 4(a) shows the results for top-k precision on operation matching. The top-2, top-5, and top-10 precisions of WOOGLE are 98%, 83%, 68% respectively, higher than those of the two naive methods by 10 to 30 percentage points. This demonstrates that considering different sources of evidence, and considering them separately, will increase the precision. We also observe that COMB has a higher top-2 and top-5 precision than FUNC, but its top-10 precision is lower. This demonstrates that considering more evidence by simple combination does not greatly enhance performance.

Figure 4(b) shows the precision for the four other



(a)



(b)

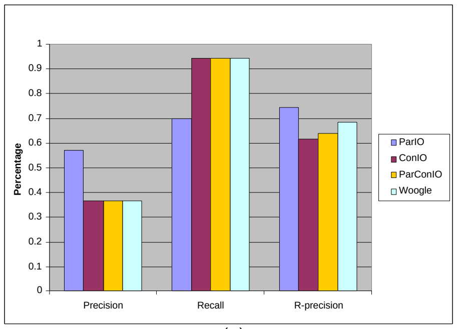
Figure 5: Performance for different operation matchers.

returned lists. Note that we only reported the top-2 and top-5 precision, as these lists are much smaller in size. From the 25-operation test set, we selected 20 where both input and output parameters are not empty, and the sizes of the returned lists are not too short. Figure 4(b) shows that for the majority of the four lists, the top-2 and top-5 precisions are between 80% and 90%.

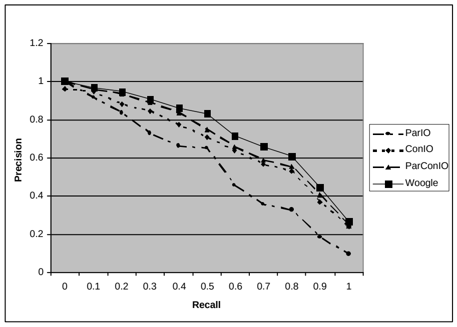
6.3 Measuring Recall

In order to measure recall of similarity search, we need to know the set of all operations that are relevant to a given operation in the collection. For this purpose, we created a benchmark of 8 operations from six different domains: weather(2), address(2), stock(1), sports(1), finance(1), and time(1) (weather and address are two major domains in the web service corpus). We chose operations with different popularity: four of them have more than 30 similar operations each, and the other four each have about 10 similar operations. Among the 8 operations, one has empty input, so we have 15 inputs/outputs in total. When choosing the operations, we ensured that their inputs/outputs convey different numbers of concepts, and the concepts involved vary in popularity.

For each of the 8 operations, we hand-labeled other operations in our collection as relevant or irrelevant. We began by inspecting a set of operations that had similar web service descriptions, or similar operation descriptions, or similar inputs or outputs.



(a)



(b)

Figure 6: Performance of different input/output matchers

From that list we chose the set of similar operations and labeled them as relevant. The rest are labeled as irrelevant. In a similar fashion, we label relevant inputs and outputs.

In this experiment we also wanted to test the contributions of the different components of Woogle. To do that, we also considered the following stripped-down variations of Woogle:

- FuncWS: consider only operation descriptions and web service descriptions;
- FuncIO: consider only operation descriptions, inputs and outputs;
- ParOnly: consider all of the four components, but compare inputs/outputs based on only parameter names;
- ConOnly: consider all of the four components, but compare inputs/outputs based on only the concepts they express.

Figure 5(a) plots the average precision, recall and R-precision on the eight operations in the benchmark for each of the above matchers and also for Func, Comb, and Woogle. Figure 5(b) plots the average R-P curves. We observe the following.

First, Woogle generally beats all other matchers. Its recall and R-precision are 88% and 78% respectively, much higher than those of the two naive methods. Second, considering evidences from different sources by simply putting them into a big bag of words (Comb) does not help much. This strategy only beats Func, which considers evidence from a single source. Even FuncWS, which discards all input and output information, has a better performance than Comb. Third, FuncIO performs better than FuncWS. It shows that in operation matching, the semantics of input and output provides stronger evidence than the web service description. This observation agrees with the intuition that operation similarity depends more on input and output similarity. Fourth, Woogle performs better than ParOnly, and also slightly better than ConOnly. ParOnly has a higher precision, but a lower recall; ConOnly has a higher recall, but a lower precision. By considering parameter matching (fine-grained matching) and concept matching (coarse-grained matching) together, Woogle obtains a recall as high as ConOnly, and a precision as high as ParOnly.

An interesting observation is that Woogle beats FuncIO in precision up till the point when the recall reaches 80%. Also, the recall of Woogle is 8 percentage points lower than that of FuncIO. This is not surprising because verbose textual descriptions of web services have two-fold effects: on the one hand, they provide additional evidence, which helps significantly in the top returned operations, where the input and output already provide strong evidence; on the other hand, they contain noise that dilutes the high-quality evidence, especially at the end of the returned list where real evidence is not very strong.

In our experiments, we also observe that compared with the benefits of our clustering technique and that of the structure-aware matching, tuning the parameters in a reasonable range and pre-processing the input/output terms improve the performance only slightly.

6.3.1 Input/output matching

We performed an additional experiment focusing on the performance of input/output matching. This experiment considered the following matchers:

- Woogle: matches inputs/outputs by considering parameter names, their corresponding concepts, and the operations they belong to.
- ParConIO: considers both parameter names and concepts, but not the operations.
- ConIO: considers only concepts.
- ParIO: considers only parameter names.

Figure 6(a) shows the average recall, precision and R-precision on the fifteen inputs/outputs in the benchmark for each of the above matchers. We also plotted the average R-P curves in Figure 6(b). We observe the following. Matching inputs/outputs by comparing the expressed concepts significantly improves the performance: the three concept-aware matchers obtain a recall 25 percentage points higher than that of ParIO. Based on concept comparison, the performance of input/output matching can be further improved by considering parameter name similarity and host operation similarity.

7 Searching with Woogle

Similarity search supplements keyword search for web services. Besides, its core techniques power other search methods in the Woogle search engine, namely, template search and composition search. These two methods go beyond keyword-search by directly exploring the semantics of web-service operations. Because of lack of space, we describe them only briefly.

Template search: The user can specify the functionality, input and output of the desired web-service operation, and Woogle returns a list of operations that fulfill the requirements. It is distinguished from the keyword search in that (1) it explores the underlying structure of operations; and (2) the parameters of the returned operations are relevant to the user's requirement, but do not necessarily contain the specific words that the user uses. For example, the user can ask for operations that take zipcode of an area and return its nine-day forecast by specifying input as zipcode, output as forecast, and description as the weather in the next nine days. The inputs of the returned operation can be named zip, zipcode, or postcode. The outputs can be forecast, weather, or even temperature, humidity at the end of the list of the returned operations.

Template search is implemented by considering a user-specified template as an operation and applying the similarity search algorithm. A key challenge is to perform the operation matching efficiently on-the-fly.

Composition search: Much of the promise of web services is the ability to build complex services by composition. Composition search in Woogle returns not only single operations, but also operation compositions that achieve the desired functionality. The composition can be of any length. For example, when an operation satisfying the above search requirement is not available, it will be valuable to return a composition of an operation with zipcode as input and city and state as output, and an operation with city and state as input and nine-day forecast as output.

Based on the machinery that we have already built for matching operation inputs and outputs, we can discover compositions automatically. The challenge lies in avoiding redundancy and loop in the composition. Another challenge is to discover the compositions efficiently on-the-fly.

8 Conclusions and Future Work

As the use of web services grows, the problem of searching for relevant services and operations will get more acute. We proposed a set of similarity search primitives for web service operations, and described algorithms for effectively implementing these searches. Our algorithm exploits the structure of the web services and employ a novel clustering mechanism that groups parameter names into meaningful concepts. We implemented our algorithms in Woogle, a web service search engine, and experimented on a set of over 1500 operations. The experimental results show that our techniques significantly improve the precision and recall compared with two naive methods, and perform well overall.

In future work, we plan to expand Woogle to include automatic web-service invocation; *i.e.*, after finding the potential operations, Woogle should be able to fill in the input parameters and invoke the operations automatically for the user. This search is particularly promising because it will, in the end, be able to answer questions such as "what is the weather of an area with zipcode 98195."

While this paper focuses exclusively on searches for web services, the search strategy we have developed applies to other important domains. As a prime example, if we model web forms as web service operations, a deep-web search can be performed by first searching appropriate web forms with a desired functionality, and then automatically filling in the inputs and displaying the results. As another example, applying template search and composition search to class libraries (considering each class as a web service, and each of its methods as a web-service operation) would be a valuable tool for software component reusing.

Acknowledgments

We would like to thank Pedro Domingo, Oren Etzioni and Zack Ives for many helpful discussions, and thank the reviewers of this paper for their insightful comments. This work was supported by NSF ITR grant IIS-0205635 and NSF CAREER grant IIS-9985114.

References

[1] Binding point. http://www.bindingpoint.com/.

[2] Grand central. http://www.grandcentral.com/directory/.

[3] Salcentral. http://www.salcentral.com/.

[4] Web service list. http://www.webservicelist.com/.

[5] Wordnet. http://www.cogsci.princeton.edu/ wn/.

[6] rainbow. http://www.cs.cmu.edu/ mccallum/bow, 2003.

[7] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining*, 1996.

[8] J. Cardoso. *Quality of Service and Semantic Composition of Workflows.* PhD thesis, University of Georgia, 2002.

[9] D.-S. Coalition. Daml-s: Web service description for the semantic web. In *ISWC*, 2002.

[10] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[11] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[12] H.-H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *Proc. of VLDB*, 2002.

[13] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: a machine learning approach. In *Proc. of SIGMOD*, 2001.

[14] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining.* The MIT Press, 2001.

[15] A. Hess and N. Kushmerick. Learning to attach semantic metadata to web services. In *ISWC*, 2003.

[16] K. S. Jones. Automatic keyword classification for information retrieval. *Archon Books*, 1971.

[17] G. Karypis, E. H. Han, and V. Kumar. Chameleon: A hierarchical clustering algorithm using dynamic modeling. *COMPUTER*, 32, 1999.

[18] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley & Sons, New York, 1990.

[19] L. S. Larkey. Automatic essay grading using text classification techniques. In *Proc. of ACM SIGIR*, 1998.

[20] L. S. Larkey and W. Croft. Combining classifiers in text categorization. In *Proc. of ACM SIGIR*, 1996.

[21] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Daklady*, 10:707–710, 1966.

[22] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm. In *Proc. of ICDE*, 2002.

[23] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proc. of International Semantic Web Conference(ISWC)*, 2002.

[24] E. Rahm and P. A. Bernstein. A survey on approaches to automatic schema matching. *VLDB Journal*, 10(4), 2001.

[25] G. Salton, editor. *The SMART Retrieval System—Experiments in Automatic Document Retrieval.* Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

[26] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition ofweb services using semantic descriptions. In *WSMAI-2003*, 2003.

[27] Y. Yang and J. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning*, 1997.

[28] A. M. Zaremski and J. M. Wing. Specification matching of software components. *TOSEM*, 6:333–369, 1997.