9-2008

A Faceted Classification Based Approach to Search and Rank Web APIs

Karthik Gomadam
Wright State University - Main Campus

Ajith Harshana Ranabahu
Wright State University - Main Campus

Meenakshi Nagarajan
Wright State University - Main Campus

Amit P. Sheth
Wright State University - Main Campus, amit@sc.edu

Kunal Verma
Wright State University - Main Campus

Follow this and additional works at: <https://corescholar.libraries.wright.edu/knoesis>

Part of the [Bioinformatics Commons](#), [Communication Technology and New Media Commons](#), [Databases and Information Systems Commons](#), [OS and Networks Commons](#), and the [Science and Technology Studies Commons](#)

Repository Citation

Gomadam, K., Ranabahu, A. H., Nagarajan, M., Sheth, A. P., & Verma, K. (2008). A Faceted Classification Based Approach to Search and Rank Web APIs. *Proceedings of 6th IEEE International Conference on Web Services (ICWS)*, 177-184.
<https://corescholar.libraries.wright.edu/knoesis/329>

This Article is brought to you for free and open access by the The Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis) at CORE Scholar. It has been accepted for inclusion in Kno.e.sis Publications by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

A Faceted Classification Based Approach to Search and Rank Web APIs

Karthik Gomadam¹, Ajith Ranabahu¹, Meenakshi Nagarajan¹,
Amit. P. Sheth¹ and Kunal Verma²

¹kno.e.sis center, Wright State University, Dayton, OH

² Accenture Technology Labs, Palo Alto, CA

{karthik, ajith, meena, amit}@knoesis.org, kunal.verma@accenture.com

Abstract

Web application hybrids, popularly known as mashups, are created by integrating services on the Web using their APIs. Support for finding an API is currently provided by generic search engines or domain specific solutions such as ... Shortcomings of both these solutions in terms of and reliance on user tags make the task of identifying an API challenging. Since these APIs are described in HTML documents, it is essential to look beyond the boundaries of current approaches to Web service discovery that rely on formal descriptions. In this work, we present a faceted approach to searching and ranking Web APIs that takes into consideration attributes or facets of the APIs as found in their HTML descriptions. Our method adopts current research in document classification and faceted search and introduces the serviut score to rank APIs based on their utilization and popularity. We evaluate classification, search accuracy and ranking effectiveness using available APIs while contrasting our solution with existing ones.

1 Introduction

The rapid adoption of the REpresentational State Transfer, or REST, paradigm of services has resulted in a large number of usable services and made it convenient for organizations to expose their applications as Web services described in plain HTML documents. Developers and users occasionally also tag these APIs. Web applications created by combining two or more services are referred to as mashups. Broadly, mashup building involves two tasks: (1) finding suitable APIs, and (2) programatically combining the APIs. While technologies like Google Mashup Editor¹ and Yahoo! Pipes² have made it easier to combine APIs by abstracting some programming roadblocks, there

has been little work done to help users find suitable APIs. The rapid growth in the number of available APIs, coupled with the myriad of functionally similar services, complicates the task. For instance, the popular API directory, *ProgrammableWeb*³ lists more than 1500 different mapping APIs and Mashups. Current and previous research in Web service discovery and ranking have mostly been in the context of SOAP-based services. The fundamental difference in APIs for REST based services is the lack of a formal model or standard description of service capabilities. This makes use of conventional service publication and discovery approaches harder.

General purpose search engines such as Google are often used for finding Web APIs. However, these treat API pages like any others, using the same measure to index and rank a page that describes an API as is used for, say, a news item. As a result, the APIs for a search query are scattered all over the result set (which often contain few hundred thousand records), making it hard to find the right API. As an example, for the search query Map Service API in Google, popular services such as Live Maps from Microsoft and MapQuest maps do not appear in the first two pages. Web API directories such as ProgrammableWeb and WebMashup, offer a more domain-specific solution to the problem. They often adopt approaches that rely on user-given tags and descriptions for classifying and searching APIs. Given inconsistent tagging [2] and in some cases the lack of tags, this approach frequently yields subpar results. For example, a search for *image search*⁴ APIs in ProgrammableWeb, yields results that include APIs for *job search* and *travel search* among others. The shortcomings of both general purpose and domain specific solutions affect the quality of search results. In addition to this, using a ranking algorithm such as Pagerank, that rank API pages and other pages without distinction, one often finds it difficult to decide which API is better for the task at hand.

The main focus of this work is a simple and elegant ap-

¹<http://code.google.com/gme/>

²<http://pipes.yahoo.com/pipes/>

³<http://programmableweb.com>

⁴[http://www.ProgrammableWeb.com.com/apitag/?q=image search](http://www.ProgrammableWeb.com.com/apitag/?q=image+search)

proach to searching and ranking Web APIs. Since API descriptions are text documents, we adapt existing research in document classification for categorizing Web APIs based on their descriptions and user tags. Our categorization is not merely limited to the functionality of the API, but also includes other important attributes such as message format and protocol. Such a faceted classification, described in [6], makes it possible to classify an API into multiple categories based on its different attributes or facets. Our results indicate that this approach yields search results that are better than API tag centric approaches like that of ProgrammableWeb. Finally, we introduce *serviut* rank, a method for ranking APIs based on their utilization. Serviut rank of a API is influenced by the widely accepted notion that the popularity of a Web resource (traffic and re-use) is a reliable indicator of its quality. Serviut is based on the number of Mashups that use the APIs and the popularity of the Mashups in the community. Staying in tune with the *participation over publication* principle of Web 2.0 and ensure an evolving infrastructure, our methodology allows for community participation in adding new categories and APIs.

We demonstrate the effectiveness of our method using a working platform (*ApiHut.com*⁵) and compare and contrast our approach against conventional search engines (Google, Yahoo) and domain specific API directories for the task of an API search. It is in a limited invited user alpha phase with about 50 users. We propose to have a public beta release late this summer.

2 Overview

The core of our methodology is in classifying and indexing APIs based on terms in the API and available user tags. Indexing, search and ranking are built on top of well known document classification algorithms. Here we present a brief description of our method for classification, searching and ranking of APIs.

1. Defining Facets for Web API search: Since our method relies on the ability to classify APIs based on their facets, identifying and modeling the different facets for Web API search is the first step. We adopt the seven-step procedure for building a faceted classification, based on the work of [11] and [10]. The first step is to collect representative API samples to define the scope of the domain and the facets. This set of Web APIs was selected from a wide variety of domains, chosen for the richness of their description, which were manually inspected to isolate the concepts that they described. We found that all APIs described the functionality provided, the messaging formats supported, the protocols and the programming languages they support (known as programming language bindings). Using this informa-

tion, we create four facets for Web API search: 1) Functionality, 2) Messaging formats, 3) Programming language bindings and 4) Protocol. Further, each API also had information about the domain (mapping, image search). Using the principles for the citation order of facets and foci, described in [10], we organized the domains under the functionality facet. In addition to the representative set of Web APIs, we also derived inputs from the current domain based classification of APIs found at ProgrammableWeb, TechFreaks and TechMagazine.

The end product is a taxonomy, snapshot in Figure 1, that models 62 different domains, 11 messaging types, 2 protocols and 7 programming language bindings and is 4 levels deep. We also preserve information about the APIs that were used to define categories. The rest of the system uses this taxonomy and sample APIs in order to classify unseen APIs.

2. Classification of APIs using Facets : We use traditional term vector based approaches for classifying and indexing APIs. Each category (node in the taxonomy) has two initial set of term vectors created by considering a small set of representative APIs (manually classified in the categories) using bayesian techniques. One is a term vector built from terms spotted in an API and the other is built using user tags assigned to the APIs. Subsequently, when users add a new API, a term vector for the API is created by spotting entries that are in the API and in the category term vectors using a domain specific entity spotter. The API is then classified into relevant categories, based on the cosine similarity of the API and category term vectors. An API is classified into only those categories that pass a tuneable similarity threshold. Classification of APIs is discussed in detail in section 3.

3. Searching: Our system currently allows users to search on the following facets: 1) The functionality of the API, 2) Messaging formats, 3) Protocol, and 4) Programming language bindings. The functionality is a mandatory facet for every query, while the others are optional. We parse the search query and identify the services that match desired functionality using term vector similarity methods. Matched services in each category are grouped according to their facets before being passed on to a service ranking module.

4. Ranking: Services in each category are then ranked based on a utilization factor. We calculate a service utilization score or serviut score for each API that is used to rank APIs. The serviut score for an API is calculated by the number of Mashups that use a given API, the number of Mashups that are classified into the same functional categories as the API, the popularity of the Mashups based on user score and the Alexa traffic rank⁶. Computation of the serviut score and ranking of APIs is discussed in section 5. While services

⁵<http://apihut.com/>

⁶<http://alexa.com>

like ProgrammableWeb do offer a way to search for APIs, we believe that this work is one of the earliest to define a quantifiable metric for ranking APIs.

Here we discuss each of our core system components in more detail.

3 Indexing and Classifying APIs

Much like traditional classification procedures, we first create the weighted term vectors for the categories under each of the primary facets. A term vector for a facet is the union of all term vectors of APIs classified in categories grouped under the facets. Facet tag vectors are simply the union of tags that users have assigned to the APIs in categories grouped under the facets. For any new API that needs to be classified, we build a term vector for the API that consists of terms in the API that overlap with terms in facet term vectors and a tag vector that consists of tags assigned by the users. We use a term spotting technique that borrows from basic dictionary and edit-distance based spotting techniques [9]. Using term vectors of facets as dictionary entries, we use a variable window to spot an entity and their lexical variants (Levenshtein with a string similarity >0.9) in an API description. Finally, to decide which category an API is assigned to, we compute the vector cosine similarities between the API and category term and tag vectors. A tuneable threshold is used to pick the most relevant categories for classifying the API.

3.1 Creating Term and Tag Vectors

Typically, terms in a term vector have a weight assigned to them that is indicative of their discriminatory nature to a document or to a category. Variations of TF-IDF and the Naive Bayes method are the most commonly used term weights in document classification [7]. Here we explain how we assign weights to terms and tags in their vectors.

Weighted Term Vectors: A weighted term vector for a category is a collection of tuples where each tuple contains a term and a relevance factor. The relevance factor is a weight that is indicative of the association between the term and the category. The relevance of a term t_i to a category c_r is measured by computing the conditional probability $p(c_r|t_i)$. The conditional probability can be interpreted as the probability that an API containing the term t_i belongs to the category c_r . We start by finding term frequencies of different terms across the APIs. Let f_{t_i} be the frequency of a term t_i across all the APIs in a given category. We can estimate the probability that any API in this category will contain this term as

$$p(t_i|c_r) = \frac{f_{t_i}}{\sum_j f_{t_j}} \quad (1)$$

The probability of a category can be estimated as

$$p(c_r) = \frac{|A_r|}{\sum_j |A_j|} \quad (2)$$

where, $|A_r|$ is the number of APIs in c_r and $\sum_j |A_j|$ is the total number of APIs across all categories. Using equations 1 and 2 in Bayes theorem would yield $P(c_r|t_i)$. The weighted term vector ($WT(c_r)$) for a category c_r then is

$$WT(c_r) = \{(t_i, p(c_r|t_i))\} \quad (3)$$

The term vector for a primary facet is created by computing the union of the term vectors of the categories classified under the primary facet in the taxonomy. The weight of a term in the facet term vector is determined by the number of categories that are relevant to a term. A term that has fewer relevant categories has a higher weight than a term that has a large number of relevant categories. This is because, fewer categories indicate a stronger relevance.

$$TT = \{(t_i, w_i) : t_i \in WT(c_r) \text{ and } w_i = \frac{1}{|C_S(t_i)|}\} \quad (4)$$

where $C_S(t_i)$ is the set of relevant categories for a term t_i , defined as

$$C_S(t_i) = \{c_r : p(c_r|t_i) > 0\} \quad (5)$$

Weighted Tag Vectors: A weighted tag vector for a category is a collection of tuples where each tuple contains a tag and a relevance factor. The relevance factor is a weight that is indicative of the association between the tag and the category. Computing the relevance of a tag is similar to computing the weight of tags in a tag cloud. The relevance of a tag u_f to a domain c_r is computed as

$$R(u_f, c_r) = \frac{f_{u_f}}{\sum_g f_{u_g}}$$

The weighted tag vector ($WU(c_r)$) for a category c_r is defined as

$$WU(c_r) = \{(u_f, R(u_f, c_r))\} \quad (6)$$

The approach to creating a the facet tag vector is similar to the one described for creating facet term vectors. The tag vector of a facet is defined as,

$$TU = \{(u_i, w_i) : u_i \in WU(c_r) \text{ and } w_i = \frac{\sum R(u_i, c_r)}{m}\} \quad (7)$$

, where m is the total number of categories.

3.1.1 Bootstrapping Term Vectors

The initial term vectors for facets were created using the representative APIs from programmableweb that were manually classified into categories; see section 2.1. The APIs

Figure 1. Snapshot of the Web API Taxonomy

were chosen based on the richness of their description and their user popularity in programmableweb. The representative set consisted of 215 APIs across all categories. As in programmableweb, popular categories like Search and Mapping had more APIs than categories like Database and Weather. The method for creating the initial term vectors is determined by the number of distinct terms that can be used to describe a category. For the categories under the *messaging formats*, *programming language bindings* and *protocol* facets, the term vectors were created by manual inspection of the representative set of APIs. This was possible because the set of terms that can be used to describe them is rather sparse. Term vectors for the categories in the functionality facet were obtained from the initial set of APIs using Apache Lucene.

3.2 Classification

In this section, we discuss the classification of new APIs into categories defined in the taxonomy. To classify an API, we compute the similarity between the API and the categories in the taxonomy, using their weighted tag and term vectors.

3.2.1 Computing Similarity

To refresh, an API-Term vector (API_T) is the collection of the spotted terms in an API while an API-Tag vector (API_U) is created using user assigned tags for the API. To compute the similarity between an API and a category, we use the popular cosine similarity approach, although other techniques may well be applicable. We compute two similarity measures, one over term vectors of APIs and categories and other over tag vectors of the APIs and the categories.

$$\alpha_T(API, c_r) = \frac{WT(c_r) \cdot API_T}{|WT(c_r)| |API_T|} \quad (8)$$

$$\alpha_U(API, c_r) = \frac{WU(c_r) \cdot API_U}{|WU(c_r)| |API_U|}$$

Using the cosine similarity values, the overall similarity between an API and a category is calculated as the weighted sum of the similarity over terms and tags.

$$\alpha(API, c_r) = w_t \alpha_T(API, c_r) + w_u \alpha_U(API, c_r) \quad (9)$$

The similarity set of an API ($\psi(API)$) is the set of the similarity values between the API and all the categories. To eliminate the categories with weak similarity, we normalize using

$$\alpha^N(c_r) = \alpha(c_r) - (AVG(\psi(API)) - \sigma(\psi(API))) \quad (10)$$

where $AVG(\psi(API))$ is the average of the similarity values and $\sigma(\psi(API))$ is the standard deviation. The set of similar categories is then

$$sim\_cat(API) = \{c_r : \alpha^N(c_r) > 0\} \quad (11)$$

Example: We illustrate our method for classification with an example. Consider the categories of Mapping, Geo and Photo and a mapping API. The α_T and the α_U values are shown in the table below. Taking $w_t = 0.75$ and $w_u =$

Domain	Term (α_t)	Tag (α_u)
Mapping	0.7	0.8
Geo	0.4	0.6
photo	0.1	0.0

0.25 and using equation 9, we get $\alpha(API, Mapping) = 0.73$, $\alpha(API, Geo) = 0.45$ and $\alpha(API, Photo) = 0.075$. Using equation 10, we get $\alpha^N(Mapping) = 0.64$, $\alpha^N(Geo) = 0.36$ and $\alpha^N(Photo) = -0.01$. From equation 11, $sim\_cat(API) = \{Mapping, Geo\}$.

4 Searching

Here, we describe our method for a faceted search for Web APIs. In addition to providing a search based on the functionality, the flexible faceted search also allows users to optionally specify requirements related to the other facets. To allow the specification of faceted queries, we adopt a

Figure 2. Example of Search for *Image Search* API's that support XML or GData and use REST protocol

command line approach to search. *Image Search; MType: XML,GData; Protocol: REST* is an example of a search command to search for *image search* services that use the GData or XML messaging formats and the REST protocol. Each facet is identified by a facet operator, which if used has to be accompanied by a facet search value (called search facets). When the search command is executed, the search query (functional facet) and the search facets are identified by parsing the search command.

APIs for a given search query are identified by first identifying the categories that are relevant to the search query. To do this, we find the term that is the most similar lexical variant (Levenshtein with a string similarity >0.9) of the search query in the functional facet term vector. The terms for other search facets are identified in a similar manner. Using the lexical variants allows us to accommodate for typographical errors in the search command. Once the terms are identified for all facets, the categories belonging to the set of relevant categories for the term identified in the functional facet term vector are ranked in the descending order of their similarity. The set of relevant categories is defined in equation 5. APIs that are classified under each of the categories are selected and grouped. The APIs within each functional facet category are then grouped according to their fulfillment of the search facets. Serviut ranking, discussed in the next section, is used to rank the APIs according to their service utilization. Figure 2 illustrates an example execution for the *image search* command described above.

5 Serviut Rank

Once matched, APIs are ranked according to their relevance strengths. Here, we introduce **service utilization** (serviut) Rank, a method for rating APIs objectively, based on their utilization. In computing the serviut rank of an API, we adopt the widely accepted notion that traffic and re-use

Figure 3. Serviut Rank Example

are reliable indicators of the quality of a Web resource.

The serviut rank measure is inspired by the PageRank [3] approach for ranking Web pages. Central to the PageRank approach are the incoming links to a page and the PageRank of the source of the links. The Mashups that use a given API are analogous to the incoming links and their rating is analogous to the PageRank of the source of the links. More the number of highly rated Mashups that use a given API, higher is the serviut rank of the API.

To compute the serviut rank, we first compute the serviut score for each API. The serviut score of an API depends on the following five factors: 1) The set of mashups that use the API (M_a), 2) The set of mashups in the category (M_c), 3) User assigned popularity of the mashups in M_a ($P(M_a)$ and M_c ($P(M_c)$), 4) User assigned popularity of the mashups in M_c and 5) Popularity of the mashups based on Web traffic. The serviut score has two components. The first component, user popularity score, is the derived using the number of Mashups and their user assigned popularity. The second component, traffic popularity, is derived using the Alexa rankings.

To calculate the user popularity score, we consider the set of mashups, their user assigned popularity scores and

the number of user votes. For each Mashup that use a given API, we first calculate the normalized popularity score ($P_N(M_{ai})$) using equation 12.

$$P_N(M_{ai}) = (P(M_{ai}) - \sigma(P(M_c))) \quad (12)$$

where, $P(M_{ai})$ is the average user assigned popularity for this mashup and $\sigma(P(M_c))$ is the standard deviation of the user assigned popularity values for all mashups in this category. The user popularity score for an API is calculated using the normalized popularity scores of the mashups that use the API.

$$U_P(a) = \frac{V_{M_a}}{V_{M_c}} \sum_i P_N(M_{ai}) \quad (13)$$

where, $V(M_a)$ is the total number user votes for the mashups that use this API and $V(M_c)$ is the total number votes for all mashups in this category.

To calculate the relative traffic popularity of mashups, we first obtain the rank of all the mashups in a given category using Alexa Web service. Since the Alexa rank is calculated for the Web in general, and we are interested only in the traffic popularity of a mashup relative to other mashups in the same category, we first normalize the Alexa rank. The normalized traffic popularity of a mashup $M_{ai} \in M_a$ is given by

$$NT(M_{ai}) = \frac{T_H(M)}{T_R(M_{ai})} \quad (14)$$

where $T_R(M_{ai})$ is the traffic rank of the mashup M_{ai} and $T_H(M)$ is the highest traffic rank for any mashup in M .

Using the normalized traffic popularity defined above and the user popularity score defined in 13, we define the serviut score of an API as,

$$serviut(a) = w_t \frac{\sum_i NT(M_{ai})}{n} - w_u U_P(a) \quad (15)$$

Serviut rank is a ranking of the APIs based on their serviut scores.

Example We illustrate the serviut rank method with an example. Consider API1 and API2 illustrated in Figure 3. For the purposes of this example, we assume that both of them are the only APIs in a category c_r . From Figure 3 $M_c = \{Mashup1, Mashup2, \dots, Mashup7\}$, $M_{API1} = \{Mashup1, Mashup2, Mashup3, Mashup4\}$ and $M_{API2} = \{Mashup5, Mashup6, Mashup7\}$. The normalized popularity score, calculated using equation 12, is illustrated in Figure 4(a). The normalized traffic score, computed using 14, is illustrated in Figure 4(b).

Assuming the weight for the traffic rank to be 0.75 and user popularity to be 0.25 in equation 15, the serviut(API1)= 2.51. Similarly, serviut(API2)= 1.14. Using serviut rank, API1 would be ranked ahead of API2.

Mashup	$P_N(M_{ai})$	Mashup	$NT(M_{ai})$
Mashup1	3.66	Mashup1	0.216
Mashup2	4.10	Mashup2	0.335
Mashup3	3.62	Mashup3	0.055
Mashup4	2.64	Mashup4	0.211

(a)

(b)

Figure 4. (a) Normalized Popularity Scores for Mashups using API1; (b) Normalized Traffic Scores for Mashups using API1

Query	pWeb	ApiHut	Google
Query1	0.48	0.89	0.20
Query2	0.61	0.83	0.13
Query3	0.25	0.54	0.23
Query4	0.70	0.82	0.37

Table 1. Precision : Apihut, PWeb and Google

Even though Mashups creating using API2 attract higher Web traffic, the fewer number of user votes and the poorer user popularity attributed to the lower serviut score of API2. This example also illustrates the importance and significance of the social process in serviut ranking.

6 Evaluation

In this section we present the empirical evaluations of our method for classifying and ranking Web APIs. The data set for the evaluation was obtained by crawling the APIs in programmableWeb. The APIs were then classified using the method discussed in 3.2. The objective of our empirical evaluations is three fold: 1. Evaluate the accuracy of classification through a user study; 2. Evaluate the accuracy of our approach using conventional precision and recall measures; and 3. Evaluate the effectiveness of serviut rank using user evaluation. For the first and third experiments we use the widely accepted Cohen's Kappa statistical measure of inter-rater reliability [1].

Query	Precision	Recall
Query1	0.89	0.75
Query2	0.83	0.69
Query3	0.54	0.71
Query4	0.82	0.21

Table 2. Precision and Recall of ApiHut

6.1 Accuracy of Classification

To study the accuracy of classification, we presented fifteen users with eleven Web APIs and a set of six categories. Users were asked to rate the categories as *most similar*, *moderately similar* and *negligibly similar* for each API. Categories (in the similarity set of an API obtained by our classification method) were classified based on a threshold defined using the average of the similarity values and their standard deviation. Cohens measure was then used to calculate the level of agreement between ratings assigned by users and those calculated by our method. The agreement for an API is the average of the agreement between the user rating and the rating calculated by our method for that API. The overall agreement between the system and the set of users is the average agreement across all APIs. Using this measure, the average agreement between the system and the set of users was 0.627. Upon further inspection of the agreement score, we found that when the system classified a category as most-similar, 40% of the users agreed with the system. For moderately-similar classification, the agreement was 47%. However, nearly 87% of the users agreed with the system when a category was classified as negligibly-similar, thereby demonstrating the effectiveness of our normalization approach, defined in equation 10.

6.2 Precision and Recall

Our second experiment has two parts: 1) Comparing the precision of our system (ApiHut) with ProgrammableWeb and Google. 2) Measuring the precision and recall metrics of our system.

To compare the precision of the results returned by ApiHut, ProgrammableWeb and Google, we used the following queries: 1) *Map; Protocol: REST*, 2) *Video Search; messageType: XML*, 3) *Photo Editing; Protocol: REST* and 4) *Geocoding; messageType: XML*. Since Google is a general purpose search engine, it is not reasonable to expect it to support facets. Hence, we appended the functional facet (Map, Video Search, Photo Editing and Geocoding) of each query with *web service api* to create the queries for google. We used the advanced search feature of ProgrammableWeb that allows for searching based on additional parameters. However, the message format and protocol facets are collectively called protocol in ProgrammableWeb. This limits the search option to either a messaging format or a protocol. The results of the first part of the experiment are illustrated in table 1. In this experiment, we only considered the top 50 results returned by Google. A closer inspection of Google's result revealed that pages belonging same API's description occurred multiple times. For example, Google Maps API appears nearly 15 times in the 30 results, because of the Pagerank. This skew in results validates our claim that a

domain specific ranking approach is needed to rank Web APIs.

The second part of this experiment measures the precision and recall metrics. Since there is no way to determine the actual number of services that should be returned for ProgrammableWeb, we do not estimate its recall. To measure the recall of our system, users classified 100 services into 4 categories. The user classification was used as the gold standard. The results obtained by using the same set of queries described above were compared with the gold standard. The result of our experiment is illustrated in table 2. The average recall values were around 70%. The recall value for the geo-domain, however was very low (21%). Upon further analysis, it was found that a large number of APIs were either poorly described or the vocabulary was inconsistent, leading to poor quality of facet term vectors. One potential approach to alleviate this problem is by manual inspection and correction of the term vectors.

6.3 Effectiveness of Ranking

In this experiment we study the effectiveness of our ranking methodology and the adequacy of the serviut rank as an approach to rank API's. Since ranking is very personal and subjective, to study the effectiveness of the ranking methodology, users were asked to rank the results of seven search queries. The Cohen's kappa measure was then calculated between the ranks assigned by users and the results of the serviut rank. The average kappa score of 0.83 indicates a strong agreement in the ranks assigned by the users and the serviut rank.

To measure the adequacy of serviut rank as an approach to rank APIs, we asked 40 users to answer a short questionnaire. The questionnaire is available online at *ApiHut survey*⁷. The users were asked to respond to the following questions:

1. Is user popularity a sufficient measure for ranking an API?
2. Are the metrics used in serviut rank representative of the popularity of an API?
3. Which one of user popularity and traffic popularity is more indicative of the service utilization?
4. Is serviut rank, by itself a sufficient measure for ranking APIs?

Almost 93% of the users said that user popularity in itself cannot be used for ranking APIs. This vindicates our belief that while user participation is a very important factor

⁷<http://apihut.com/survey>

in ranking, it cannot be the only factor. 98% of the respondents agreed that the metrics used by serviut rank are representative of the popularity of an API. Asked to choose between user popularity and the traffic popularity metrics, all of the respondents said that the traffic popularity is a more important metric. To the last question, 40% of the felt that serviut rank was sufficient to rank APIs, while the rest said that other metrics such as facet fulfillment must be considered into ranking. This evaluation demonstrates that serviut rank as a measure is very useful in ranking APIs.

7 Related Work

The work presented in this paper is related to the research in the area of Web services discovery and ranking. Although there is a plethora of research available in the area of Web service discovery, much of the work in the area of Web services discovery has been in the context of SOAP based services and their approaches use a formal model for service description. Here we review a representative sample of the prior research in the area of Web services discovery and ranking. In [5] the UDDI specification is extended to accommodate the use of predictions about the behavior of Web services. The behavioral parameters include availability, reliability and completion time. An approach to importing the semantic Web into UDDI by mapping DAML-S service profiles to UDDI records is discussed in [4]. Another approach is to map semantic Web service descriptions in WSDL-S (now SAWSDL) to UDDI for service discovery, presented in [8]. Since these rely on a formal service description model, one has to invest considerable time in creating formal models for APIs to use them for searching RESTful APIs. The work presented in this paper, primarily addresses the problem of searching Web APIs and does not assume the existence of a formal service description model.

There have been however a handful of Web applications that facilitate the categorization and searching of Web APIs, of which ProgrammableWeb is the most popular. ProgrammableWeb allows users to add, tag, and describe APIs and mashups. ProgrammableWeb provides category-based API browsing and searching. Two other categorization worth mentioning here are Tech-News and TechFreaks. Both of them offer very good API categorization, but do not provide search or other capabilities provided by ProgrammableWeb. The applications mentioned above support limited faceted search and do not have a ranking mechanism. The work presented in this paper offers a more flexible faceted search while also ranking the APIs based on service utilization. Further, as demonstrated in section 6.2, the method presented in the paper performs considerably better than ProgrammableWeb.

8 Conclusions and Future Work

In this work, we motivated the need for alternate search mechanisms for Web APIs and presented one such simple but effective method. Central to our approach is a faceted classification of Web APIs. We build upon existing research in document classification for classifying APIs and propose a new method for ranking API search results based on their utilization. To the best of our knowledge, the *serviut rank* is one of the earliest attempts to define a domain specific method to rank Web APIs. Our evaluations demonstrate the effectiveness of our classification and serviut ranking method for searching and ranking web apis.

As the results of our search precision and recall measurement indicate, the richness of descriptions and the agreement of terms amongst APIs in a category have a telling effect on recall. We propose to investigate the use of microformat-based-annotation mechanisms such as SA-REST to address this limitation. The co-authors are active contributors to the IBM Sharable Code project and propose to integrate our prototype into the Sharable Code platform.

We expect to have a public beta of the search functionality of ApiHut by late summer and plan to release the rest of the features, including adding and tagging APIs and a developer centric social network by late fall.

References

- [1] J. Cohen. A coefficient of agreement for nominal scales. *educational and psychological measurement*. 18(20):3746, 1960.
- [2] G. Koutrika, F. A. Effendi, Z. Gyöngyi, P. Heymann, and H. Garcia-Molina. Combating spam in tagging systems. In *AIRWeb '07*, pages 57–64, 2007.
- [3] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998.
- [4] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Importing the Semantic Web in UDDI. In *WES*, pages 225–236, 2002.
- [5] S. Ran. A model for web services discovery with qos. *SIGecom Exch.*, 4(1):1–10, 2003.
- [6] S. Ranganathan. *Elements of library classification*. New York: Asia Publishing House, 1962.
- [7] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. 18(11):613–620, 1975.
- [8] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Discovery of web services in a federated registry environment. In *ICWS 2004*, pages 270–277. IEEE Computer Society, 2004.
- [9] S. Soderland. Learning to extract text-based information from the world wide web. In *KDD*, pages 251–254, 1997.
- [10] L. Spiteri. A simplified model for facet analysis: Ranganathan 101. *Canadian J of Info. and Library Science*, pages 1–30, 1998.
- [11] B. Vickery. *Faceted classification: a guide to construction and use of special schemes*. London: Aslib, 1960.