Similarity Function Recommender Service Using Incremental User Knowledge Acquisition

Seung Hwan Ryu, Boualem Benatallah, Hye-Young Paik,
Yang Sok Kim, and Paul Compton

School of Computer Science & Engineering,
University of New South Wales, Sydney, NSW, 2051, Australia
{seungr,boualem,hpaik,yskim,compton}@cse.unsw.edu.au

Abstract. Similar entity search is the task of identifying entities that most closely resemble a given entity (e.g., a person, a document, or an image). Although many techniques for estimating similarity have been proposed in the past, little work has been done on the question of which of the presented techniques are most suitable for a given similarity analysis task. Knowing the right similarity function is important as the task is highly domain- and data-dependent. In this paper, we propose a recommender service that suggests which similarity functions (e.g., edit distance or jaccard similarity) should be used for measuring the similarity between two entities. We introduce the notion of "similarity function recommendation rule" that captures user knowledge about similarity functions and their usage contexts. We also present an incremental knowledge acquisition technique for building and maintaining a set of similarity function recommendation rules.

Keywords: Similarity Function, Recommendation, Entity Search, RDR.

1 Introduction

The community portals, such as DBLife, Wikipedia, are widely available for diverse domains, from scientific data management to end-user communities on the Web. In a community portal, data from multiple sources are integrated so that its members can search and query relevant community data. Community data typically contains different classes of entity instances, such as persons, documents, messages, and images, as well as relationships between them, such as *authorBy(person, document)*, *supervisedBy(person, person)*, and *earlyVersionOf(document, document)*. Each entity instance [1] is described by a set of attributes (e.g., person has name, title and address).

In this paper, we focus on similar entity search on such community data that is exposed as data services [7,8]. Unlike keyword based search, in similar entity search, entities are compared based on the similarity of entity attributes [2] [5,3,15]

[1] In this paper we use "entity instance" and "entity" interchangeably.
[2] We specify as attributes for short.

as well as entity relationships [3] [14,1], and a ranked list of entities is returned based on the degree of similarity.

A main challenge arises from the fact that entities may belong to different classes with potentially very different characteristics, and contain attributes of different data types. In such a situation, it is impractical to expect a *single* generic similarity function can work well for all attributes [3]. For example, in Figure 1(a), to measure the similarity between `q` and `m` from a discussion forum, we compare the attribute values individually: `Title` with `Title`, `Content` with `Content`, and `Size` with `Size`. When using one of the basic similarity functions (e.g., edit distance) [16,15] for all attributes and combining the scores, we obtain the final similarity score (0.55). However, the accuracy can be increased to 0.89 (Figure 1(b)) by choosing different similarity functions suited to each attribute. It is also possible to consider relationships with other entities if they exist, such as *repliedBy(message, person)*. The need for SES (Similar Entity Search) tasks is present in many application domains, such as product search, people search, document search, and data integration in business intelligence [3,20,15].

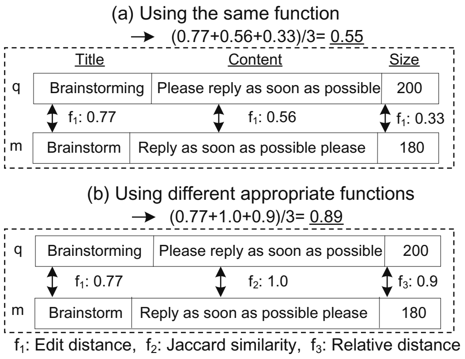

(a) Using the same function
→ (0.77+0.56+0.33)/3= 0.55
Title
Content
Size
q
Brainstorming
Please reply as soon as possible
200
f1: 0.77
f1: 0.56
f1: 0.33
m
Brainstorm
Reply as soon as possible please
180
(b) Using different appropriate functions
→ (0.77+1.0+0.9)/3= 0.89
f1: 0.77
f2: 1.0
f3: 0.9
f1: Edit distance, f2: Jaccard similarity, f3: Relative distance

Fig. 1. Computing similarity between two messages `q` and `m`

Existing approaches for similarity analysis can be roughly divided into two groups. The first group computes attribute similarities using methods such as jaccard similarity, edit distance, or cosine similarity [16,9,5,21]. The second group exploits the relationships among entities [14,1] or machine learning-based techniques [4,5,24,23,10] for estimating the similarity of entire entities. For example, supervised machine learning techniques [5,10,24] train a model using training data pre-labeled as "matching" or "not matching" and then apply the trained model to identify entities that refer to the same real-world entity. Some of the machine learning-based techniques use positive and negative examples to learn the best combination of basic similarity functions [4,24,23].

While existing techniques have made significant progress, they do not provide a satisfactory answer to the question of which of the presented techniques should

[3] We specify as relationships for short.

be used for a given SES task. Even a technique that shows good performance for some data sets can perform poorly on new and different ones [5,15]. In real-world application domains, as in community portals, we observe that, community users, especially advanced users like programmers, administrators, and domain experts, often have valuable knowledge useful for identifying similar entities - the knowledge about which combination of similarity functions is most appropriate in which usage contexts. For instance, `edit distance` function [21] works well for comparing short strings (e.g., person names or document titles). We believe that this information is beneficial in terms of reuse and knowledge sharing as community users would choose similar functions in similar contexts.

Unfortunately, this information is not effectively exploited in existing approaches when measuring similarity. In this paper, we provide a recommender service that suggests most appropriate similarity functions for a given SES task by utilizing the knowledge collected from community users. It should be noted that our approach is complementary to the machine-learning based techniques in the sense that it allows adaptive knowledge "learning" over time as the application contexts change. Examples of such context changes are: application domain changes, dataset changes, continuous or periodic updates of datasets, and so on. In this paper we only consider the similarity of entities that belong to a same class/category. In particular, we make the following contributions:

- We introduce the notion of *similarity function recommendation rules* (henceforth recommendation rules). The recommendation rules represent the information about which similarity functions are considered most appropriate in which usage contexts (Section 3).
- We propose *incremental knowledge acquisition techniques* to build and update a set of recommendation rules. The continuous updates of recommendation rules enable the proposed recommender service to make more fine-tune recommendation (Selection 4).
- We present an implementation of the *recommender service* and provide the experimental results that show the feasibility and effectiveness of our proposed approach (Section 5).

2 Preliminaries

In what follows, we first explain the data model for representing entities and their relationships. We then describe how to measure the similarity of entities and present the overall architecture of the proposed recommender service.

2.1 Community Data Graph

We use a graph-based model, named "Community Data Graph", to represent entities and their relationships. We model the community data as a set of entities $E = \{E_1, E_2, ..., E_n\}$ and a set of relationships $R = \{R_1, R_2, ..., R_n\}$, where each E_i/R_i is an entity or a relationship category. Each entity category E_i (e.g., *Person*) has a set of entity instances (e.g., *John* and *Alice*).

Each relationship R_i (e.g., *authorBy*) has a set of relationship instances (e.g., $authorBy_{John\_firstDraft.doc}$[4]). Each entity/relationship instance consists of a set of attributes $A = \{A_1, A_2, ..., A_m\}$ and is denoted as e_i or r_i.

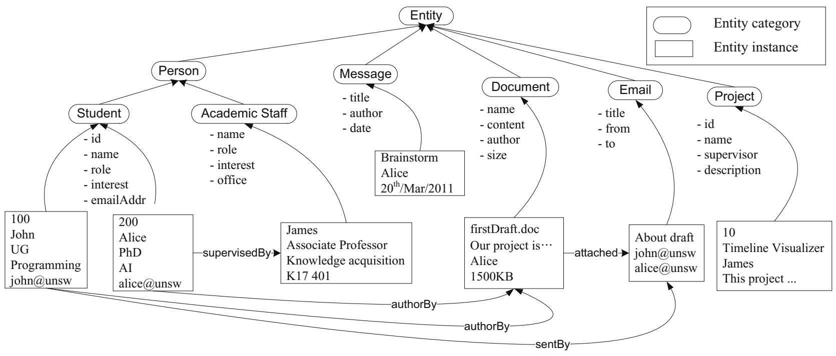


Fig. 2. Example excerpt of community data graph

Figure 2 shows a snapshot of a community data graph for an education community (called *courseWiki*)[5]. In courseWiki, the community users, such as supervisors, tutors, students, administrators, and even outside collaborators, could collaboratively work with each other and share their knowledge and experiences during the course. The courseWiki community data comes with a heterogeneous set of entities: project specifications (in Microsoft Word documents), wiki pages, emails exchanged, project reports (in PDF documents), and images/diagrams. In the graph, the nodes denote entity *categories/instances* and the edges denote *relationships* between them. For example, `John` and `Alice` are student instances of `Student` category. They can be also associated with a set of *attributes*, such as id `100` and email address `john@unsw`. The category `Student` denotes the collection of all students managed in the courseWiki community.

Definition 1. ***(Community Data graph)***
A community data graph is a direct labeled graph $G = < V, L_v, L_e, E >$, *where* V *is a set of nodes,* L_v *is a set of node labels,* L_e *is a set of edge labels and* $E \subseteq V^2 \times L_e$ *is a set of labeled edges. Each node represents an entity category or instance and each edge represents a relationship between two entities. Here, a node* $\in V$ *consists of a set of attributes* $\{A_1, A_2, ..., A_n\}$.

2.2 Measuring Entity Similarity

We now describe how to estimate the similarity between two entities:

Attribute-based Similarity: To measure the similarity between a query entity `q` and a same category of entity e_i, we compute the similarity between

[4] This can be specified as *authorBy* for short when there is no ambiguity.

[5] This is constructed from a project-based course "e-Enterprise Projects" in our school.

individual attributes and then produce the weighted similarity between the entities. In certain cases, a weight may be associated with each attribute, depending on its importance. Formally, we define the combined score *basic_sim* as follows:

- $basic\_sim(\mathsf{q}, e_i) = \sum_{k=1}^{N} \alpha_k f_k(\mathsf{q}.A_k, e_i.A_k)$

where f_k is the basic function being applied to a pair of attributes, α_k is its weight, and N is the number of basic functions.

Relationship-based Similarity: Apart from the attribute-based similarity, we exploit semantic relationships (called co-occurrence) between entities [14,1]. For example, two persons are likely to be similar (related), if they have co-occurring *authorBy* relationships. Like atomic attributes, relationships may have weights according to their importance (e.g., frequency of relationships). We adopt the weighted Jaccard distance to compute the co-occurrence coefficient between two entities. The weighted Jaccard distance is defined as:

- $co\_sim(\mathsf{q}, e_i) = \frac{\sum_{r \in A \cap B} w_r}{\sum_{r \in A \cup B} w_r}$

where A and B are the sets of relationships which the query entity `q` and the entity e_i have respectively, and w_r is a weight assigned to the relationship.

Composition-based Similarity: If entities have internal structures, such as XML schemas or process models, the entity similarity can be measured based on a complex process. Such a process is a directed graph that specifies the execution flow of several components [22]. The components could be a similarity estimator, a score combiner which computes a combined similarity value from results of other estimators, or a filter that identifies the most similar attribute pairs. We can integrate these measurement methods in our recommender service, if there is a need for finding correspondences between complex structures.

2.3 Overall Architecture

This subsection gives an overview of the recommender service architecture and describe components that support the concepts presented in our approach. The proposed architecture consists of the following three layers (see Figure 3).

Data service layer: To provide uniform and high-level data access to the data repository, we expose CoreDB [2] as a service by leveraging the data service technology [7,8]. CoreDB stores entities and their relationships extracted from community data sources, based on the entity-relationship model. For instance, `Person` entity table has attributes `name`, `role`, and `interest`, and stores all person entity instances. The data service layer also provides a set of CoreDB access open APIs [2], including basic CRUD operations, keyword search, rule creation, similarity functions recommendation, and so on.

Recommender service layer: This layer is composed of three main components: *function recommender*, *similarity computation* and *rule manager* components. The function recommender component takes as input `q` and

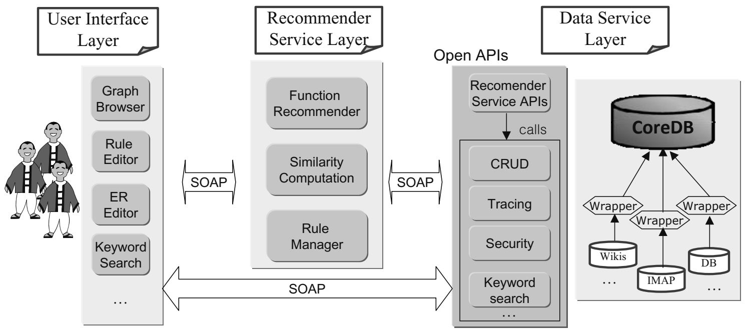


Fig. 3. Overall architecture

recommends as output a combination of similarity functions. This component accesses the recommendation rules managed in CoreDB via the open APIs. The similarity computation component asks users for a threshold (between 0.0 and 1.0) and computes the similarity scores using the recommended similarity functions. This component relies on multiple similarity functions that are represented as entities in CoreDB. The component returns and ranks the entities similar to `q` based on their final scores. The rule manager allows users to create and maintain recommendation rules. If a user is not satisfied with the returned result by the similarity computation component, she can create another recommendation rule using this rule manager. Then, she can immediately re-apply the newly added rule to get a different result.

User interface layer: At this layer, the community data accessible from data services is represented and visualized as a graph based on the mindmap metaphor [6] (the details will be described in Section 5.2). This layer is also responsible for providing various functionalities to enable users to intuitively browse and query the community data using the graph browser as well as to incrementally build recommendation rules using the rule editor.

3 Exploiting Community User Knowledge

In this section, we describe the notion of *recommendation rules* and their management. The rules represent user knowledge about similarity functions and their usage contexts. Then, we show how the recommender service makes recommendations on which functions to select, using the recommendation rules.

3.1 Recommendation Rule Representation Model

Community users, especially advanced users, often have their knowledge about the characteristics of individual similarity functions, such as usage purposes or function-specific parameters. For example, `edit distance` function [21] is expensive or less accurate for measuring the similarity between long strings (e.g., document or message contents). It is likely to be suitable for comparing short strings (e.g., document titles), capturing typographical errors or abbreviations.

Table 1. Examples of recommendation rules

RuleID	Usage Context	Function Combination	CS**
1	C*= "Message" $\wedge$ exist(title) $\wedge$ exist(author)	{(title, EditDistance), (author, Jaro)}	1
2	C= "Person" $\wedge$ exist(interest) $\wedge$ exist(role)	{(interest, Jaccard), (role, EditDistance)}	1
3	C= "Person" $\wedge$ hasRelationship(sentEmail)	{(sentEmail, co-occurrence)}	2
4	C= "Product" $\wedge$ exist(name) $\wedge$ price $\leq$ 1500	{(name, Jaccard), (price, RelativeDistance)}	1

* C stands for Category, ** Confidence Score (see Section 3.3)

As another example, `relative distance` [3] is good for comparing numerical values, like weight and price, and `Hamming distance` [15] is used mainly for numerical fixed values, like postcode and SSN.

Thus, the effective leverage of this kind of knowledge is important to improve the accuracy of SES. In addition, community users may know which attributes/relationships play an important role in identifying similar entities. As an example, if a task is to find similar persons for a given person, attribute `interest` might be more useful than attributes `id`. To capture such user knowledge, we propose recommendation rules that consist of two components: *usage context* and *function combination*.

Usage context. Briefly stated, a usage context refers to the constraints that `q` should satisfy before the recommender service suggests similarity functions. It consists of a conjunction of predicates, each of which is specified by a unary or binary comparison involving entity's categories, attributes or its relationships, and constants. For example, in Table 1, the usage context of RuleID `1` states that `q` should belong to a `Message` category and have two attributes `title` and `author`. Table 2 shows some of operations that are used for specifying such usage contexts.

Function combination. For each usage context, the recommendation rule is associated with a list of pairs (attribute/relationship, similarity function) that indicates which functions are most appropriate to which attributes/relationships. For instance, in Table 1, the function combination of RuleID `1` suggests that the `edit distance` function, good for short string comparison, should be used to compare `title` and the `Jaro` function, good for name similarity detection, should be used for `author`.

Table 2. Usage Context Operations

- *exist(a_k)* checks whether `q` has an attribute a_k.
- *valueOf(a_k)* returns the value of an an attribute a_k.
- *hasRelationship(r_k)* checks whether `q` has a relationship r_k
- *length(a_k)* returns the length of a_k attribute value.
- *contain(a_k, V)* checks whether attribute a_k contains the value V.
- *belongTo(a_k, C)* checks whether the value of a_k belongs to a semantic concept C.

Definition 2. ***(Recommendation Rule)***
Let C_e be a set of entity categories supported in the recommender service. Recommendation rule is of the form: $q \in E_i,\ P(q.A_1, ..., q.A_n) \rightarrow \sum_{k=1}^{N} f_k(q.A_k)$

where P is a conjunction of predicates on the entity category and attributes A_1, ..., A_k of q. Each predicate is either of the form $q.category = C_i \in C_e$ or unaryop ($q.attribute/relationship$) or $q.attribute$ op value where $op \in \{=, <, >, \leq, \geq, \neq, contain, belongTo\}$, $unaryop \in \{exist, valueOf, hasRelationship, length\}$.

3.2 Matching Recommendation Rules

When a user selects a certain entity as `q`, the recommender service identifies the potential combinations of functions being applicable to `q`. For this, the service provides an operation called `recommendFunctions()`, which takes as input `q` and produces as output a set of function recommendations that can be potentially applied to perform the corresponding SES task. The recommender service matches `q` against the usage contexts of a set of recommendation rules. The matching process relies on subsumption (containment) or equivalence between `q` and entity contexts. For example, given a query entity `q`: (category: `Student`, name: `John`, interest: `programming`, role: `undergraduate student`), the usage context of RuleID `2` is matched as the `Student` category is a sub-type of `Person` category (in Figure 2). The function combination of the matched recommendation rule is returned to the user. If no combination is found to be appropriate to `q`, the user might create a new recommendation rule with the help of rule editor.

3.3 Ranking Recommendation Rules

In our recommender service, each recommendation rule can be associated with a positive value, called *Confidence Score* (CS). The score indicates the level of credence in a corresponding recommendation rule. In fact, a CS reflects the user satisfaction level for the function combination in a recommendation rule. To obtain this score, we introduce a user feedback loop at the end of each SES task. After the user has applied the recommended function combination and examined the returned results by them, she can express the level of satisfaction with the recommendation rule (hence the function combination) by giving a score. The CS value for a recommendation rule is accumulated overtime, each positive feedback rating adding 1. The CS values are then used to show users a ranked list of the recommendation rules when there are more than one rule matching `q`. If no CS value is specified, we assume $cs = 1$. Also, completing the feedback loops is optional to users.

Definition 3. ***(Ranked recommendation rule)***
A ranked recommendation rule gives a confidence score $cs \geq 1$ to a recommendation rule definition: $q \in E_i, P(q.A_1, ..., q.A_n) \xrightarrow{cs} \sum_{k=1}^{N} f_k(q.A_k)$.

4 Incremental Knowledge Acquisition

In this section, we present how to incrementally obtain the recommendation rules from community users.

4.1 Knowledge Acquisition Method: Ripple Down Rule

To incrementally build and update recommendation rules, we adopt the knowledge acquisition method called Ripple Down Rule (RDR) [12] for several reasons: (i) RDR provides a *simple and easy* approach to knowledge acquisition and maintenance [18]; (ii) RDR works *incrementally*, in that users can start with an empty rule base and gradually add rules while processing new example cases.

In RDR, a rule has two components a condition and a conclusion: if [condition] then [conclusion]. Hence, the condition part of an RDR rule is mapped to the usage context in our recommendation rule, and the conclusion part to the function combination (with a confidence score). RDR organizes our recommendation rules as a tree structure. For example, Figure 4 shows an example rule tree, in which the rules are named *rule 0, rule 1, rule 2,* etc., according to their creation order. Rule 0 is the default root rule that is always fired for every query entity q. The rules underneath rule 0 are more specialized rules created by modifying rule conditions. The *rule inference* in RDR starts from the root node and traverses the tree, until there are no more children to evaluate. The conditions of nodes are examined as a depth-first traversal, which means the traversal result is the conclusion node whose condition is *lastly* satisfied.

4.2 Acquiring Knowledge through Different Rule Types

In what follows, we describe the incremental knowledge acquisition process using different rule types.

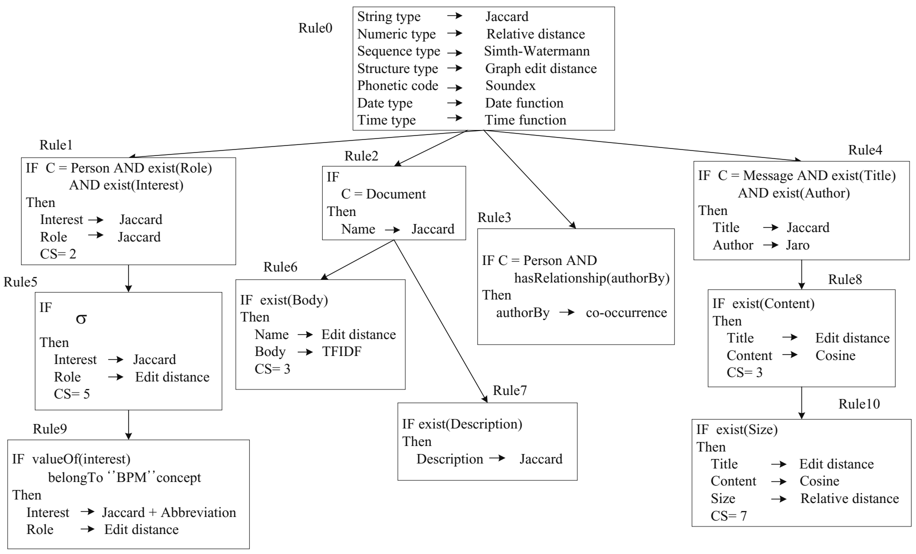

Rule0
String type → Jaccard
Numeric type → Relative distance
Sequence type → Simth-Watermann
Structure type → Graph edit distance
Phonetic code → Soundex
Date type → Date function
Time type → Time function
Rule1
IF C = Person AND exist(Role) AND exist(Interest) Then Interest → Jaccard Role → Jaccard CS= 2
Rule2
IF C = Document Then Name → Jaccard
Rule3
IF C = Person AND hasRelationship(authorBy) Then authorBy → co-occurrence
Rule4
IF C = Message AND exist(Title) AND exist(Author) Then Title → Jaccard Author → Jaro
Rule5
IF σ Then Interest → Jaccard Role → Edit distance CS= 5
Rule6
IF exist(Body) Then Name → Edit distance Body → TFIDF CS= 3
Rule7
IF exist(Description) Then Description → Jaccard
Rule8
IF exist(Content) Then Title → Edit distance Content → Cosine CS= 3
Rule9
IF valueOf(interest) belongTo ''BPM''concept Then Interest → Jaccard + Abbreviation Role → Edit distance
Rule10
IF exist(Size) Then Title → Edit distance Content → Cosine Size → Relative distance CS= 7

Fig. 4. Example RDR Tree: we omit duplicate conditions between parent and children rules for simplicity. If a confidence score (cs) is not available in the rule conclusion, *cs*= 1 is assumed. σ denotes the condition is the same as the parent one.

Attribute Type-Based Rule (Default Rule). The default rule contains no condition (i.e., it is always satisfied) and its conclusion consists of a list of pairs (attribute type, name of the suitable function). This rule only checks the types of attributes, meaning for example, all string-type attributes will be assigned the same string function (i.e., jaccard similarity function).

Example 1. In Figure 4, **Rule 0** specifies that `Jaccard` function is applied to string type attributes, `RelativeDistance` to numeric, `SimithWatermann` to attributes having sequence (e.g., biological sequence), and so on.

Key Attribute-based Rule. We note that there are some situations where choosing a few key attributes in an entity for comparison, rather than looking at all available attributes, may produce better results. For example, for `Person` entities, the attribute such as id may not be significant in determining similarity, but interest or role may provide a better clue. Using this rule, the user can identify any key attributes in an entity that she wants to compare. There are two possibilities in defining this rule:

Choosing key attributes only: the user identifies the attributes that play an important role for assessing the similarity without degrading the search accuracy. In this case, each attribute is paired with the default similarity function based on its type.

Example 2. **Rule 1:** $q \in E_i$, category= "Person" $\wedge$ exist(interest) $\wedge$ exist(role) $\rightarrow f_{Jaccard}(interest) \wedge f_{Jaccard}(role)$.

Choosing key attributes and functions: In this case, the user can choose key attributes as well as their similarity functions. Note that this may happen incrementally if the user determines later that a different function may produce better results. For example, in Rule 5, after looking at the results from Rule 1, she may realise that `edit distance` is better suited for `role` attribute, because the character-based function (`edit distance`) works better than the token-based function (`jaccard`) in detecting the similarity between strings including abbreviations, e.g., `Assoc. Professor` vs. `Associate Professor`.

Relationship-based Rule. Entity relationships can also contribute to analysing similarity. Out of all relationships linked with `q`, this rule allows the user to examine co-occurring ones only. Further, the user can specify the co-occurring relationships that are perceived more important. For example, the following example finds persons who have co-occurring relationship `authorBy`.

Example 3. **Rule 3:** $q \in E_i$, category= "Person" $\wedge$ hasRelationship(authorBy) $\rightarrow f_{co-occurrence}(authorBy)$

Lexical Relation-based Rule. This rule type takes into consideration the values of attributes where certain keywords/phrases or semantic information may play a role in determining similarity. For instance, consider two strings "BPM" and "Business Process Modelling and Management" of `interest` attribute in person entities. When normal string comparison function may fail to see the similarity, it is certainly desirable to be able to match the two as 'similar'.

To handle the semantic relationships between attribute values, this rule allows the users to specify lexical relations between words (e.g., synonym, hyponym) or abbreviation. We use synonym and abbreviation tables, including domain-specific terms. An example of the abbreviation table entry is: (*BPM*: Business Processes, Business Process Management, Business Process Modelling and Management)), which takes the format of (concept name: list of terms). For instance, Rule 9 states that if `q` has `interest` attribute and its value belongs to concept name "*BPM*", then, for `interest`, apply $f_{Jaccard}$ function in comparing syntactical differences and use `abbreviation` function in comparing semantic differences.

Example 4. **Rule 9:** $q \in E_i$, category= "Person" $\wedge$ exist(interest) $\wedge$ exist(role) $\wedge$ interest belongTo `BPM` $\rightarrow$ $(f_{Jaccard}(interest) \vee f_{Abbreviation}(interest)) \wedge f_{EditDistance}(role)$.

5 Implementation, Usage, and Evaluation

This section describes our prototype implementation, usage scenario and experiment results.

5.1 Implementation

The prototype has been implemented using Java, J2EE technologies, and some generic services from existing Web services environments to implement specific functionalities of the services proposed in our approach. We extract coureWiki community data from multiple data sources and store the data into CoreDB. For this, we have implemented a number of wrappers in which each wrapper has a particular purpose and pulls the data from its original location to populate the community data graph. For instance, a special wrapper would analyse email exchange logs and build relationships such as *sentEmail*, *repliedBy*. We expose CoreDB as data services [7,8] that allow uniform data access via open APIs [2]. We have developed the recommender service supporting our proposed approach, which consists of three components: the function recommender, similarity computation, and rule manager components (Section 2.3). Table 3 shows a set of operations that such components can invoke to perform their specific functionalities. We also present a graphical user interface (Section 5.2) that allows users to interact with the recommender and data services.

5.2 Usage Scenario of the Recommender Service

We propose the following scenario as an illustration. Figure 5 presents a screenshot of the graph browser. Initially, in the visualization area, an *entity*

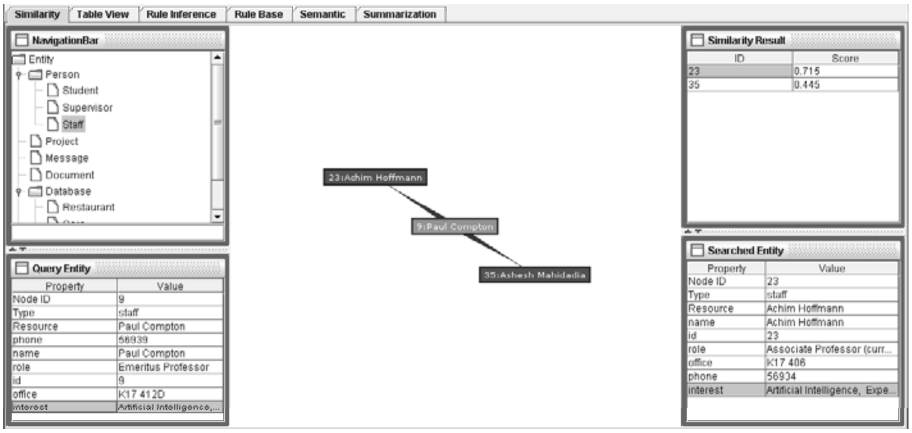
Table 3. The list of operations invoked by recommender service components

Function recommender/rule manager operations
- *recommendFunctions(q)* returns a list of similarity functions according to q' context.
- *createRule(c, d)* creates a rule with a condition c and a conclusion d.
- *refineRule(r, c, d)* refines a rule r with a condition c and a conclusion d.
- *rankRule(r)* increases the confidence score of a rule r by 1.
Similarity computation operations
- *computeSim(a_k,f_i)* computes similarity scores by applying function f_i to attribute a_k.
- *aggregateSim(q, e_i)* aggregates similarity values between individual attributes of q and e_i and computes a final score.

node (e.g., root entity) serving as a center node is displayed. The center node is directly connected with other nodes denoting entity categories or instances. A user can choose one of entity instances as q. The top left panel is used for navigating the graph according to the entity categories and the bottom left panel displays the details of the selected query entity (if any). After selecting q, she asks the recommender service for similarity functions. The service returns function combinations according to the recommendation rules that q satisfies. One recommendation rule example is:

– IF C= Person and exist(interest) and exist(role)
  THEN interest → Jaccard, role → Edit distance

Next, the user chooses one of function combinations, the similarity computation component calculates the similarity scores between q and the other entities, using the recommended functions, and then returns a list of similar entities. The right top panel shows the list of returned entities and the right bottom panel shows the details of a returned entity selected by the user. Here, the user can examine why the selected entity is similar to q.



Query Entity

Property	Value
Node ID	9
Type	staff
Resource	Paul Compton
phone	56939
name	Paul Compton
role	Emeritus Professor
id	9
office	K17 412D
interest	Artificial Intelligence,...

Similarity Result

ID	Score
23	0.715
35	0.445

Searched Entity

Property	Value
Node ID	23
Type	staff
Resource	Achim Hoffmann
name	Achim Hoffmann
id	23
role	Associate Professor (curr...
office	K17 406
phone	56934
interest	Artificial Intelligence, Expe...

Fig. 5. Usage scenario of recommender service: the service returns entities similar to q using recommended functions

5.3 Evaluation

We now present the evaluation results that show how the recommender service can be effectively utilized in the courseWiki domain.

Dataset: We uses the dataset from our project-based course for the courseWiki community. The dataset is a collection of different categories of entities: 210 persons, 684 messages, 942 documents, 580 issues and 22874 events. For the evaluation, we applied our system on three categories of entities (i.e., `Person`, `Message`, and `Document`).

Evaluation Metrics and Methodology: We measured the overall performance with accuracy. Accuracy is the number of correctly identified similar pairs divided by the total number of identified similar pairs. Whether returned entities are similar to `q` is dependent on users' subjective decision. Therefore, we decided to manually pre-classify the entities into different groups (i.e., within a group, the entities are considered similar). All entities in one group is pre-labelled with the same groupID. For example, for `Person` entities, we grouped them according to their project/assignment work groups. `Document` entities were grouped based on their revision history.

Starting with a default rule, we began the knowledge acquisition process by looking at the different categories of entities in chronological order (instance creation date). For example, for 210 people entity instances, the acquisition process is defined as follows, note that this process is repeated for every entity instance: (i) an entity instance is selected as `q`, (ii) rules are applied, (iii) we examine the result[6], if the result is satisfactory (i.e., the groupID of `q` is the same as that of the returned entity), the rule is untouched, if not, the existing rule is refined (e.g., changing the function). The above steps are repeated until all entity instances are considered as `q`.

Results: Figure 6 shows that overall, across all categories, the accuracy of our system improves overtime as the number of entity instances being processed is increased. This is because there are more (refined) rules created. Some other observations we made about different categories are: (i) for `Person` entities, the relationships played an important role in improving the accuracy. As shown in Figure 6, with only about 60 number of entity instances considered, the system already performed at accuracy 0.93 (in this case, the number of created rules is 4). (ii) for `Document` entities, knowing the right function to use for a certain attribute was a particularly important factor. For example, comparing `title` attribute worked better with character-based string match function.

Discussion: It should be noted that the number of rules created depends on how well the users know about the characteristics of the dataset and available functions. In addition, the RDR approach for knowledge acquisition enables domain experts or users to build rules rapidly since there is no need for understanding the knowledge base as a whole or its overall structure [13].

[6] In fact, we consider the entity returned with the highest similarity score.

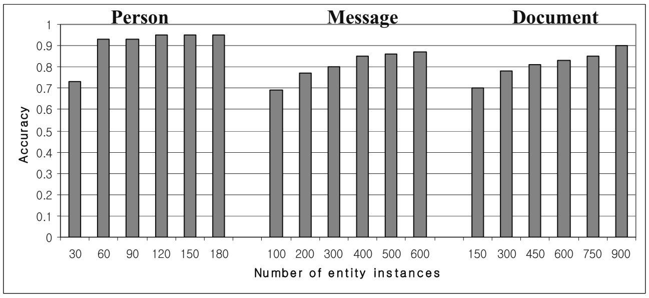


Fig. 6. Results of Evaluations

6 Related Work

Our recommender service is related to the efforts in basic similarity functions and record linkage.

Basic Similarity Functions. In this group of work, individual attributes are considered for similarity analysis. Many different basic functions for capturing similarity have been proposed in the last four decades [16,9,5,21]. For example, they are used for comparing strings (e.g., edit distance and its variations, jaccard similarity, and tf-idf based cosine functions), for numeric values (e.g., Hamming distance and relative distance), for phonetic encoding (e.g., Soundex and NYSIIS), for images (e.g., Earth Mover Distance), for assessing structural similarity (e.g., graph edit distance and similarity flooding), and so on. In contrast to those techniques, our work focuses on determining which similarity functions are most appropriate for a given similarity search task.

Record Linkage. The record linkage problem has been investigated in research communities under multiple names, such as duplicate record detection, record matching, and instance identification. The approaches can be broadly divided into three categories: supervised methods, unsupervised methods, rule-based methods. The supervised methods [5,24,11] train a model using training data pre-labeled as "match" or "no match" and later apply the model to identify records that refer to the same real-world object. The unsupervised methods [25] employ the Expectation Maximum (EM) algorithm to measure the importance of different elements in feature vectors. The rule-based approaches [19,17] enable domain experts to specify matching rules that define whether two records are the same or not. However, our work differs in that (i) we do not rely on the existence of training data. (ii) our recommender service helps users incrementally define recommendation rules and enables them to choose similarity functions suitable for the domain-specific similarity search task.

7 Conclusion and Future Work

In this paper, we presented a recommender service that suggests most appropriate similarity functions, which can be used when comparing two entities. Particularly, we introduced similarity function recommendation rules and their types. We also proposed an incremental knowledge acquisition process to build and manage the rules. In future work, we plan to investigate how to extend our approach to support large scale of SES tasks, such as identifying similar entities from millions of entities, using some high performance computing techniques.

References

1. Ananthakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In: VLDB, pp. 586–597 (2002)
2. Báez, M., Benatallah, B., Casati, F., Chhieng, V.M., Mussi, A., Satyaputra, Q.K.: Liquid Course Artifacts Software Platform. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 719–721. Springer, Heidelberg (2010)
3. Bilenko, M., Basu, S., Sahami, M.: Adaptive product normalization: Using online learning for record linkage in comparison shopping. In: ICDM, pp. 58–65 (2005)
4. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: KDD, pp. 39–48. ACM (2003)
5. Bilenko, M., Mooney, R.J., Cohen, W.W., Ravikumar, P.D., Fienberg, S.E.: Adaptive name matching in information integration. IEEE Intelligent Systems 18(5), 16–23 (2003)
6. Buzan, T., Buzan, B.: The mind map book. BBC Active (2006)
7. Carey, M.: Data delivery in a service-oriented world: the bea aqualogic data services platform. In: SIGMOD 2006, pp. 695–705 (2006)
8. Castro, P., Nori, A.: Astoria: A programming model for data on the web. In: ICDE, pp. 1556–1559 (2008)
9. Christen, P.: A comparison of personal name matching: Techniques and practical issues. In: ICDM Workshops, pp. 290–294 (2006)
10. Cochinwala, M., Kurien, V., Lalk, G., Shasha, D.: Efficient data reconciliation. Inf. Sci. 137, 1–15 (2001)
11. Cohen, W.W., Richman, J.: Learning to match and cluster large high-dimensional data sets for data integration. In: KDD, pp. 475–480 (2002)
12. Compton, P., Jansen, R.: A philosophical basis for knowledge acquisition. Knowl. Acquis. 2(3), 241–257 (1990)
13. Compton, P., Peters, L., Lavers, T., Kim, Y.S.: Experience with long-term knowledge acquisition. In: K-CAP, pp. 49–56 (2011)
14. Dong, X., Halevy, A.Y., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD Conference, pp. 85–96 (2005)
15. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng. 19(1), 1–16 (2007)
16. Hall, P.A.V., Dowling, G.R.: Approximate string matching. ACM Comput. Surv. 12, 381–402 (1980)
17. Hernández, M.A., Stolfo, S.J.: Real-world data is dirty: Data cleansing and the merge/purge problem. Data Min. Knowl. Discov. 2, 9–37 (1998)

18. Ho, V.H., Compton, P., Benatallah, B., Vayssière, J., Menzel, L., Vogler, H.: An incremental knowledge acquisition method for improving duplicate invoices detection. In: ICDE, pp. 1415–1418 (2009)
19. Lee, M.L., Ling, T.W., Low, W.L.: Intelliclean: a knowledge-based intelligent data cleaner. In: KDD, pp. 290–294 (2000)
20. Li, Q., Wu, Y.-F.B.: People search: Searching people sharing similar interests from the web. J. Am. Soc. Inf. Sci. Technol. 59(1), 111–125 (2008)
21. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. 48, 443–453 (1970)
22. Peukert, E., Eberius, J., Rahm, E.: Amc - a framework for modelling and comparing matching systems as matching processes. In: ICDE, pp. 1304–1307 (2011)
23. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: KDD, pp. 269–278 (2002)
24. Tejada, S., Knoblock, C.A., Minton, S.: Learning domain-independent string transformation weights for high accuracy object identification. In: KDD, pp. 350–359 (2002)
25. Winkler, W.E.: Using the em algorithm for weight computation in the fellegi-sunter model of record linkage. In: Survey Research Methods Section, American Statistical Association, pp. 667–671 (2000)