Using Relationships for Matching Textual Domain Models with Existing Code

Raghavan Komondoor*, Indrajit Bhattacharya†, Deepak D'Souza* and Sachin Kale‡
*CSA Department, Indian Institute of Science (IISc), Bangalore
Email: {raghavan,deepakd}@csa.iisc.ernet.in
†IBM Research India (part of this work was done when the author was with IISc). Email: indrajitb@in.ibm.com
‡Email: sachinpkale@gmail.com

***Abstract*—We address the task of mapping a given textual domain model (e.g., an industry-standard reference model) for a given domain (e.g., ERP), with the source code of an independently developed application in the same domain. This has applications in improving the understandability of an existing application, migrating it to a more flexible architecture, or integrating it with other related applications. We use the vector-space model to abstractly represent domain model elements as well as source-code artifacts. The key novelty in our approach is to leverage the relationships between source-code artifacts in a principled way to improve the mapping process. We describe experiments wherein we apply our approach to the task of matching two real, open-source applications to corresponding industry-standard domain models. We demonstrate the overall usefulness of our approach, as well as the role of our propagation techniques in improving the precision and recall of the mapping task.**

I. Introduction

A large number of organizations have built systems that have served them over a number of years or decades, and are hence considered "legacy" systems. Over time these systems become unwieldy and complex, and are no longer amenable to the continuous evolution that is required to meet changing business needs. They are monolithic, inflexible, and difficult to integrate with more recently developed systems (such as web-based clients). Also, the technology and languages that these systems use has aged, thus presenting an opportunity to make use of superior newer technology and skill sets. Yet, these systems are critical because they contain entity-relationship information as well as business rules (i.e., logic) that is key to the functioning of the organization.

Organizations follow various strategies [1] to escape this quandary, such as (a) migrate from a custom application to a packaged application, and (b) keep the custom application, but migrate its architecture from a monolithic architecture to a loosely-coupled, agile architecture, such as a rule-engine-based architecture, or service-oriented architecture. These *legacy transformation* projects all follow a common sequence of high-level activities [1], [2]: (1) Users of the current system (as well as proposed system) identify an abstract *target model* or architecture, which captures all strategic *requirements* from the system, and is not tied closely to the architecture and idiosyncrasies of the existing application(s). (2) The developers in conjunction with the users *validate* or *reconcile* the target model with the existing systems, and (3) The developers use the validated target model to construct (or reconstruct) the new system using the selected strategy.

The target model typically contains components such as a *data model*, a collection of *services*, as well as *workflows*. The validation step is required to flesh-out the constituent elements of the target model with *detailed* information that is present in the existing application(s), that is specific to the users of the organization, such as attributes required in the various entities, and specific *business rules* corresponding to each service. Typically, the output of the validation step is a set of *links* from target model elements to artifacts of the existing application(s) (e.g., source files, portions of source files, database tables).

Organizations with legacy existing systems are unlikely to already possess documentation that has the desired attributes of a target model as outlined above. However, fortunately, *standard domain models* are already being provided by vendors; e.g., models for the ERP (Enterprise Resource Planning) and CRM (Customer Relationship Management) domains by SAP [3], for insurance domain by IBM [4], and for the banking domain by BIAN [5] (an open standards body). Organizations prefer to use these as target models because (a) it saves the time required to create a target model, and (b) the pre-existing organizational knowledge of these standard models means that if an existing system is linked to such a model it automatically becomes more amenable to understanding and transformations. However, linking a domain model to an existing application that was developed independently of the model is a challenging activity. The difficulties arise due to various reasons, e.g., (a) large sizes of the domain model and the application, (b) lack of developers who have detailed knowledge of the application, (c) mismatches between the capabilities in the application and the concepts in the domain model, (d) mismatches in terminology even on concepts that are common between the application and the model, and (e) the application having a poor structure, resulting in code pertinent to a single domain concept or business rule being dispersed at various locations in the application's source code.

A. An IR-based approach for model to code matching

In this paper we propose an IR-based approach for inferring candidate matching artifacts in the source code for each element in the target model. In this work we consider *textual* models, which are still the most prevalent type of domain models available in practice. In our case studies we used the ERP and CRM domain models provided by SAP [3]. Each of these domain models contains different types of elements; the ones we focus on are descriptions of *services* (i.e., processing steps),

grouped into *collections* (each collection typically contains descriptions of services that pertain to a single data entity, e.g., purchase orders). We represent each collection in the model as well as each source-code artifact (i.e., code file, directory, table definition) using a *term frequency vector*, as is standard in IR. Our key observation, over those of previous researchers working on similar problems, is that various *relationships* exist among source-code artifacts, such as calling, accessing a common table, residing in a common directory, and that these can be leveraged in a principled way to increase the efficacy of matching. Therefore, we first propagate *features* between related source artifacts, then compute similarities between all pairs of source-code artifacts and collections, and then propagate the similarity values themselves between related source-code artifacts. This way we arrive at a "consensus" similarity vector for each source-code artifact in conjunction with its related artifacts. Finally, using a *cut-off* k for the number of source artifacts to be matched with each collection (or vice versa), and/or a threshold on the similarity values, we emit a matching between the source-code artifacts and the collections.

The specific contributions of this paper are:

- We propose and implement an approach to match the elements in a textual domain model with source-code artifacts.
- To our knowledge ours is the first approach to use *feature propagation* and *similarity propagation* over relationships in the source-code side to solve the model-to-code matching problem. Previously, these approaches have been used to answer *individual queries* on source code [8], or to label nodes in other kinds of graphs such as the web [9].
  Relationships may also be available within the domain model, e.g., flow order between the services. Leveraging these also in the matching process would be a topic of future work.
- We use the soft-tfidf similarity measure [10], [11] to account for synonymous but terminologically mismatched features. While soft-tfidf has been shown to be effective in handling inter-dependences between features for matching named entities, to the best of our knowledge it has not been applied to information retrieval tasks.
- For our case studies we consider two large, real-life domain models, in the ERP and CRM domains, as mentioned above. We also consider two *independently developed* real applications, namely, JAllInOne [6] (an ERP application) and SellWinCRM [7] (a CRM application), to be matched with the respective domain models. To our knowledge ours is the first paper to explore a solution to this practical problem. Previous results in the literature are about matching an *application-specific* document with the *same* application, which is typically a less challenging problem.
- We do an extensive manual study in which we match the two domain models with the two respective applications. The result is two "gold standard" matchings, that are likely to be useful to the community at large as benchmarks. This is the first manual study in matching of such scale to be reported in the literature, to our knowledge.
- We show that the output from our tool compares encouragingly with the "gold standard" results mentioned above. Moreover, we show that our propagation technique leads to significant improvements in matching accuracy (over no propagation). In our experiments, the average precision gain due to propagation for any given level of recall was 60% on the ERP case study, and 16% on the CRM case study.

The rest of this paper is structured as follows. We first formulate our problem in Section II. We then provide an overview of our solution with the main ideas in Section III. In Section IV we introduce standard IR techniques for computing similarity between documents using a vector space model. The main components of our solution are then described in the next two sections — the vector space model that we use in Section V, and improving the results by using propagation over relations in Section VI. We present our case studies and experimental results in Sections VII and VIII, discuss related work in Section IX, and finally conclude in Section X.

II. Problem Statement

In an instance of the problem in our setting, we are given a *domain model* and an *implementation*. The domain model is assumed to be a set of domain model elements, each of which is a piece of text. For compatibility with our case studies we refer to these domain elements as *collections*. We represent the textual *description* of each collection as a sequence of tokens from a finite vocabulary (or dictionary) V. In our representation, we will not distinguish between a collection and its description, and use for example c to refer to both the collection c and its textual description.

The implementation consists of a set $\mathcal{S}$ of *source code files*, organized in a set of directories $\mathcal{D}$, and a set $\mathcal{T}$ of database tables which are accessed by these source code files. We assume that we are given a relation $c \subseteq \mathcal{S} \times \mathcal{S}$ (with $(s, s') \in c$ representing the fact that source file s *calls* some method in source file s'; and similarly a relation $l \subseteq \mathcal{S} \times \mathcal{D}$ representing *location* (thus $(s, d) \in l$ represents the fact that source file s lies in directory d); and a relation $a \subseteq \mathcal{S} \times \mathcal{T}$ representing the table *access* relation between source files and tables.

The problem is to come up with a set $\mathcal{O}$ of "matches" between source files and collections: thus $\mathcal{O}$ is a set of pairs of the form (s, c) where s is a source file and c is a collection. A solution to this problem is an algorithm that takes an instance of the problem like the one above, and outputs a set of matches $\mathcal{O}$.

To evaluate a solution on an instance of our problem, one needs to have a *manual matching* that gives a "gold standard" against which we can compare the output of the algorithm. The manual matching of the instance of the problem given above will be represented as a set $\mathcal{M}$ of pairs in $\mathcal{S} \times \mathcal{C}$. Given a manual matching $\mathcal{M}$, we can measure the quality of the output $\mathcal{O}$ of a proposed algorithm for the problem, using the popular *precision* and *recall* measures [12]. Precision (P) measures

the fraction of correct pairs (that is those that are contained in the manual study) among the pairs returned by the algorithm: $P = \frac{|\mathcal{M} \cap \mathcal{O}|}{|\mathcal{O}|}$. Recall ($R$) measures the fraction of correct pairs returned by the algorithm among the total set of correct pairs: $R = \frac{|\mathcal{M} \cap \mathcal{O}|}{|\mathcal{M}|}$.

III. Solution Overview

In this section we present an overview of our proposed algorithm for the matching problem. Our approach is an information retrieval based one, in which we represent files and collections using the vector space model (VSM) [12], and compute similarities between them in this model.

Over and above the tokens contained in a source file s and a collection c, different kinds of relationships between source artifacts carry critical information for the similarity between s and c. For example, the source files "called" by s and database tables accessed by s, may contain tokens that are important for matching s and c. Similarly, it may be significant to know that a database table accessed by s and a source file called by s are both similar to collection c. We propose a framework that incorporates such "relational evidence" into similarity computations in two different ways: (a) propagating features and (b) propagating similarity scores between neighboring source-code over relationships.

Input: Collections $\mathcal{C}$, Source files $\mathcal{S}$, Directories $\mathcal{D}$, Tables $\mathcal{T}$, and relations c, a and l
Output: Set $\mathcal{O}$ of matches between Source Files and Collections
foreach *Document d in $\mathcal{C} \cup \mathcal{S} \cup \mathcal{D} \cup \mathcal{T}$* **do**
  Create document vector $\vec{V}(d)$ and add to Vector Space
end
Create Relationship Graphs $\mathcal{G}_r$, $r \in \{c, a, l\}$
Propagate Features Using $\mathcal{G}_r$, $r \in \{c, a, l\}$
Obtain similarities between individual features using Latent Semantic Analysis
foreach *Document d in $\mathcal{S} \cup \mathcal{D} \cup \mathcal{T}$* **do**
  Create Similarity Vector $\vec{S}(d)$ using soft-tfidf
end
Propagate Similarity Vectors Using $\mathcal{G}_r$, $r \in \{c, a, l\}$
// For an F2C based matching
// For a given a cut-off k and a similarity threshold U
$\mathcal{O} := \emptyset$;
foreach *source-file s in $\mathcal{S}$ and collection c in $\mathcal{C}$* **do**
  if *c is among the top-k collections similar to s in $\vec{S}(s)$, after filtering out c's with $\vec{S}(s)(c) < U$* **then**
    $\mathcal{O} := \mathcal{O} \cup \{(s, c)\}$;
  end
end

Algorithm 1: Overview of our matching algorithm

An overview of our approach is shown in Algorithm 1. We first create a feature vector $\vec{V}(d)$ for each collection, source file, database table and directory d. The feature vectors of the source-code artifacts are then then enriched using feature propagation between related artifacts. We then create an initial similarity vector $\vec{S}(d)$ over collections for each source file, database table and directory d. These similarities are enhanced using similarity propagation over the different relationships.

Finally, we use one of two *ranking modes*, called "F2C" (for "Files-to-Collections") and "C2F" (for "Collections-to-Files"), to produce the set of matches. In both cases we assume we are given a minimum similarity threshold U, and an upper bound k on the number of matches for a file (respectively collection). In the F2C mode we output for each source file s, the pairs (s, c) such that c is among the k most similar collections to s, and the similarity between s and c is above the threshold U. In the "C2F" mode, we look at the k most similar source files for a collection, and proceed similarly.

In the following sections we focus on computing the similarity between implementation and domain model elements, and describe these steps of the algorithm in greater detail.

IV. Vector Space Model and Similarity

The vector space model [12] is a popular way of representing text documents and computing the degree of similarity between them. In this section we briefly introduce this model and the techniques for computing similarity in this model.

In the vector space model, each document d is represented as a vector $\vec{V}(d)$ of non-negative real-valued weights corresponding to a vocabulary T of tokens (or "features"). For weighting each token in a document, we use the popular tf-idf scheme [12]. The tf-idf weight of token t in document d is given by $\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$ where $\text{tf}_{t,d}$ is the number of occurrences of token t in document d, and idf_t is the inverse document frequency. We use the following definition for idf: $\text{idf}_t = \log(N/\text{df}_t)$ where N is the total number of documents, and df_t is the number of documents that contain token t.

The standard approach for measuring similarity between two document vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$ is to consider their normalized dot-product, called cosine similarity:

$$\text{cosine-sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|}$$

where $|\vec{V}|$ denotes Euclidean length of the vector: $\sqrt{\sum_i \vec{V}_i^2}$.

Cosine similarity does not take into account similarity between *tokens* themselves, and may adversely affect the computed similarity between documents. Similarity between non-identical tokens may be syntactic or semantic. Syntactic similarity considers the lexicographic structures (spellings) of two tokens. For instance, the tokens "sales" and "selling", or "item" and "ITM" are syntactically similar. Various string similarity measures such as Edit Distance, Jaro, or Jaro-Winkler can be used to calculate the similarity $\text{string-sim}(s, t)$ between two different tokens s and t [10]. Semantic similarity on the other hand considers similarity in meaning between tokens that are not necessarily lexicographically similar (for example "item" and "material," or "tax" and "VAT"). For this one could use Latent Semantic Analysis (LSA) [13]. We first use LSA to factorize the original tf-idf matrix into a product of three matrices W,S, and D. We then define the LSA similarity of two tokens s and t to be the cosine similarity of the rows corresponding to s and t in W: $\text{LSA-sim}(s, t) = \text{cosine-sim}(W(s), W(t))$. We note that in the subsequent steps of our approach we continue to use the original tf-idf matrix for the matching task. The overall similarity of two tokens is taken to be the maximum of their syntactic and semantic similarities:

$$\text{sim}(s, t) = max(\text{string-sim}(s, t), \text{LSA-sim}(s, t)).$$

The soft-tfidf measure [10] extends cosine similarity to take into account the similarity between tokens. In addition to shared tokens in documents d_1 and d_2, soft-tfidf(d_1, d_2) also considers their weights for similar tokens, that have sim(s, t) above a threshold θ. We assume that sim(s, t) is set to 0 whenever its original value is less than θ. We define soft-tfidf(d_1, d_2) to be

$$\frac{\sum_{(t,t')\in T\times T} \vec{V}(d_1)[t]\vec{V}(d_2)[t']\,\text{sim}(t,t')}{|\vec{V}(d_1)||\vec{V}(d_2)|}.$$

It reduces to cosine similarity for $\theta = 1$.

Given a document corpus $\mathcal{C}$, and a query document q, $\vec{S}(q)$ denotes the similarity vector of q for corpus $\mathcal{C}$, consisting of similarities between q and each document in $\mathcal{C}$. Sorting $\vec{S}(q)$ in decreasing order of similarity values produces a ranked list $R(q)$ of the corpus documents for q.

V. VSM Model for our matching problem

We fix an instance of our problem as described in Section II, for the next couple of sections. We consider each artifact from the given domain model and implementation as a document, and make use of the vector space model to represent and measure similarities between them.

We first construct document vectors for all artifacts – namely source files, directories and database tables in the implementation, and collections from the domain model. The document vectors all are constructed from a *single universe* of features that is the union of features obtained from each document, as described below.

For each collection in the domain model we use all tokens from its description. The primary implementation artifacts are the source code files, which contain programming language constructs, variable names, method signatures, comments, etc. Depending on the application, we discard the non-informative aspects from each source code file s using domain knowledge, and consider only the remaining aspects as its features. This is discussed in more detail in Section VIII.

The other artifacts in the implementation are directories and database tables. For each directory, we create obtain its features by tokenizing its complete path. For each database table, the features used are its tokenized name and field names.

In the sequel, by "source documents" we will mean the documents corresponding to the source files, database tables, and directories.

VI. Using Relationships for Computing Similarity

If we measured similarity of a specific source code document s with a collection using the feature vectors generated so far, we will consider as evidence features intrinsic to s itself. However, quite often, significant evidence for our matching task can be found in the source artifacts that are 'related' to the source file s in different ways. For example, consider source file s_2 that contains a method invoked by s_1, or a file s_3 that invokes a method contained in file s_1. Clearly, the contents of s_2 and s_3 are relevant for understanding the semantics of s_1, and therefore for identifying its relevant collections. Similarly, the different field names of a database table accessed by s may also be relevant for determining similarity between s and the collections. In this section, we describe a framework for incorporating such relationships between files and other source artifacts into the matching algorithm.

Formally, we model such relationships as edges in a weighted directed graph, where nodes are documents and an edge in the graph represents a relationship between the incident nodes. The weight on each directed edge (u, v) will represent the "influence" of u on v.

We first define a framework for information flow over such a graph using local updates, and then describe how we use this for improving the similarity scores between source files and collections.

A. Information Propagation in Graphs

Consider a weighted directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represents the set of directed edges. We assume that the edges in $\mathcal{E}$ are *symmetric* (i.e. $(u, v) \in \mathcal{E}$ iff $(v, u) \in \mathcal{E}$). The weighting function $w : \mathcal{E} \to \mathbb{R}_{\geq 0}$, assigns a non-negative real value to each edge in $\mathcal{E}$.

For each node $v \in \mathcal{V}$, let D_v be a vector representing the 'information' at that node. Vector dimensions are the same for all nodes. To 'flow' the information associated with different nodes in the graph, we do the following. The information at each node v is locally updated using the information of its neighbors in the graph using:

$$D_v \leftarrow D_v + \sum_{u\in N(v)} w(u, v)D_u, \tag{1}$$

where $N(v)$ denotes the neighbors of v in $\mathcal{G}$. Observe that such updates may be applied repeatedly. In our framework, we perform a single local update.

The influence weights $w(u, v)$ may be defined in different ways and need not be symmetric [14]. We consider the following definitions:

1) $w(u, v) = 1/d(u)$, where $d(u)$ is the number of edges incident on node u in $\mathcal{G}$. As a result, a neighbor u that has many neighbors other than v has less influence on v.
2) $w(u, v) = 1/d(v)$. Here, a node v that has more neighbors is less influenced by any neighbors.
3) $w(u, v) = \text{sim}(u, v)$, where we use soft-tfidf similarity between D_u and D_v as sim(u, v). As a result, similar neighbors have a bigger influence on a node.

We often found it useful to threshold the influence weight using a given value T, to rule out the collective influence of many small weights. Thus, if a weight on an edge is less than T, we treat it to be 0.

In our problem, the multiple types of relations c (calling), a (accessing), and l (location) simultaneously influence the similarity between files and collections. We construct three different weighted directed graphs $\mathcal{G}_r = (\mathcal{V}, \mathcal{E}_r, w_r)$, for each relation $r \in \{c, a, l\}$, over the same set of nodes $\mathcal{V} = \mathcal{S} \cup \mathcal{D} \cup \mathcal{T}$. For the graph $\mathcal{G}_c$ we add an edge (in both directions) between the corresponding nodes in the graph. For example,

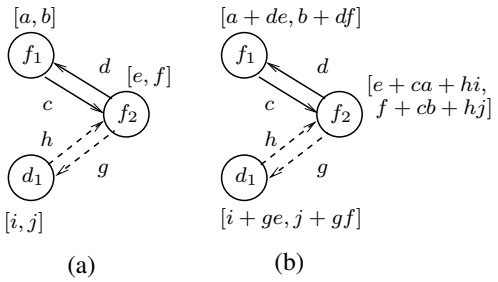

Fig. 1. Example to illustrate information propagation. Part (a) shows the superimposed graphs and initial vector values. Part (b) shows the vector values after propagation.

if one source file s calls a method in file s' (i.e. $(s, s') \in c$) then we add edges (s, s') and (s', s) in $\mathcal{E}_c$. Similarly, if a source file s accesses a table u or lies in directory d, we add the edges (s, u) and (u, s) in $\mathcal{E}_a$, and (s, d) and (d, s) in $\mathcal{E}_l$ respectively. We describe the weighting functions used for each of these graphs subsequently. We call $\mathcal{G}_c$ the *call graph*, $\mathcal{G}_a$ the *database access graph*, and $\mathcal{G}_l$ the *directory structure graph*.

The three graphs $\mathcal{G}_c$, $\mathcal{G}_a$ and $\mathcal{G}_l$ can be used simultaneously for information propagation over nodes in $\mathcal{V}$. We propagate along the different graphs separately, and then merge the updates at each node, as described in the two steps below:

$$D'_v(r) \leftarrow \sum_{u \in N_r(v)} w_r(u, v) D_u, \ r \in \{c, a, l\} \quad (2)$$

$$D_v \leftarrow D_v + \sum_{r \in \{c,a,l\}} \alpha_r D'_v(r) \quad (3)$$

where r ranges over the different relations, α_r is a constant representing the *propagation weight* or importance of the relation r for information propagation, and $N_r(v)$ denotes the neighbours of v in $\mathcal{G}_r$.

Fig. 1 illustrates the above technique for information propagation. Part (a) shows an example graph on 3 nodes corresponding to two files f_1 and f_2, and one directory d_1. We have one pair in the calling relation corresponding to the solid symmetric edges between f_1 and f_2, and one pair in the location relation corresponding to the dashed symmetric edges between f_2 and d_1. The figure shows the graphs $\mathcal{G}_c$ and $\mathcal{G}_l$ corresponding to these relations, superimposed on the same graph. The vectors D_v for each node v are shown next to the node. The weight for each directed edge is shown as a label on the edge. Part (b) shows the resulting vectors associated with each node after carrying out information propagation according to the steps (2) and (3). We have assumed $\alpha_c = \alpha_l = 1$.

B. Feature Propagation

We first use the relationships to enrich the document vector $\vec{V}(d)$ of a source document d (source files, directories and tables) using the vectors $\vec{V}(d')$ of related documents d'. For example, if a source file s invokes another source file s', we would like to use the features of s' as well for matching $\vec{V}(s)$ with collections. We call this *feature propagation*.

To propagate features, we use the document vector $\vec{V}(d)$ for each source document d as the local information D_d.

Different influence weights are used for different relation types. For the call graph and the database access graph, we use $w(u, v) = 1/d(u)$ where u is the neighboring node, or the sender of features. When a file s (or a database table t) is invoked (accessed) by many files s', its features have less influence on deciding the right collection for any of the invoking (accessing) files. In contrast, the influence is more when it is invoked (accessed) by few files. The pattern is different for the directory structure graph, since it forms a hierarchy. Here the features of directory nodes (parents) have larger influence on deciding the right collection for files (children) than the other way round. So, we use $w(u, v) = 1/d(v)$ where v is the receiving node for features.

We now use steps (2) and (3) to update the vector representations.

C. Computing Similarities

We now use the updated vector representation $\vec{V}(s)$ for each source document s to compute the collection similarity vector $\vec{S}(s)$ for s as follows:

$\vec{S}(s)[c]$ = soft-tfidf(s, c), for each collection c.

We note that the updated document vectors $\vec{V}(d)$ for directories and database tables do not directly impact the computation of similarity vectors for source files. However, the enriched vectors $\vec{V}(d)$ may lead to better collection similarity vectors $\vec{S}(d)$ for such documents. These can then be used to influence the similarity vectors $\vec{S}(s)$ of source files s that are related to d, again using the information propagation framework, as we describe next.

D. Similarity Propagation

Just like the tf-idf vectors, the collection similarity vector $\vec{S}(s)$ of a source file s can also be directly influenced by the similarity vector $\vec{S}(d)$ of a related source document d. We call this *similarity propagation*.

For this, we simply set D_d to be the similarity vector $\vec{S}(d)$ for each source document d in the propagation framework of Section VI-A above. We use the same influence weighting scheme for similarity propagation as for feature propagation in the case of the directory structure graph: $w(u, v) = 1/d(v)$, where v is the node receiving the similarity vectors. For the call graph and the database access graph, we have used $w(u, v) = \text{sim}(u, v)$, where neighbors having similar features are more likely to correspond to the same collection.

Finally we use steps (2) and (3) to update the similarity vectors.

VII. Case Studies

We now come to our experimental evaluation. In this section we describe the two case studies we chose and the manual matching exercise carried out on both of these. These manual matchings are then used in the next section as the gold standard for evaluating our proposed algorithm.

```
3. Employee Time
Document on working times of an internal or external
employee. ... documents absence times, break times, and
availability times.
Corresponding Collections:
  # Query Employee Time In.
    Group of operations to find Employee Time with relevant
    selection criteria using individual interfaces.
    Corresponding Services:
      $ Find Leave Employee Time for Employee by Employee
        Query to and response from Time and Labour
        Management to provide a list of Employee Absence
        Times for a specific Employee.
      $ Find Times for Employee Time Sheet by Elements
        Reads Employee Times for the Employee Time Sheet
        depending on the given elements.
  # Manage Employee Time In
    Group of operations for creating, changing, reading
    or deleting Employee Times or parts of it.
    Corresponding Services:
        $ Read Employee Time by Employee
          Reads an employee's Employee Times.
```

Fig. 2. An excerpt from the ERP domain model from SAP.

A. ERP case study

The domain model we used for this case study is an enterprise services model of the ERP domain from SAP [3]. From the HTML files provided in [3] we extracted a text document by unparsing the HTML files. This model contains 239 *process components*, each of which consists of several *groupings*. Each grouping comprises a description of the grouping along with a number of *collections* (totalling 630). In turn, each collection contains a description of the collection, and a number of *services* and their brief descriptions (totalling 2584). Fig. 2 shows an excerpt from this document for the grouping `Employee Time`, with two collections `Query Employee Time In` and `Manage Employee Time In`.

We treat each collection as a model-side document, so that the set $\mathcal{C}$ comprises only collections. In order not to lose the features in the groupings and services, we include the features from the grouping containing a collection c, and all services contained in c, in the features of c itself.

We chose an open-source Java-based ERP application called JAllInOne [6] as a potential implementation of the domain model. JAllInOne is a large real ERP application, with a lot of rich functionality to support the operations of medium-sized companies.

The source code of JAllInOne consists of 539 source files. We counted only the files which contain business-logic in them; these files are in the subdirectories named "`*/server/`", excluding the subdirectories "`commons/server/`", "`events/server/`", and "`sqltool/server/`". We ignored source files (i.e. `.java` files) that contained no business logic, but only "plumbing code" (for example code for establishing and closing sessions, or sockets). The 539 files are organized in approximately 130 "server" sub-directories, and access a total of 119 database tables.

Next we manually matched the JAllInOne source files with the SAP ERP domain model. Our manual study is an extension of an earlier one we had performed [15], in which we had mapped 10 of the collections in the ERP domain model to files in the JAllInOne application. For the present work we manually analyzed *each* of the 630 collections in the domain model, and matched it with all source files (if any) that appeared to implement the relevant functionality.

```
Manage Employee In
Query Employee In
employees/server/LoadEmployeeAction.java
subjects/server/PeopleBean.java

Manage Employee Time In
Query Employee Time In
scheduler/.../server/LoadScheduledEmployeesAction.java
scheduler/.../server/LoadEmployeeActivitiesAction.java

Manage Purchase Order In
Query Purchase Order In
Manage Document In
purchases/documents/server/UpdatePurchaseDocAction.java
purchases/documents/server/LoadPurchaseDocRowAction.java
```

Fig. 3. Some manually identified matches from the ERP case study.

The result of the manual matching is a matching with 869 (*file,collection*) pairs. In this matching, 48 of the 630 collections found matches, while 351 of the 539 files found matches. The remaining collections and files were found to not have appropriate matches on the other side. We have previously described [15] some of the principles that we use in our manual matching process.

We note that because the model and the code are independently developed, the manual matching is inherently a fuzzy, subjective process. Whenever in doubt we matched a file with all collections to which it could be relevant. Fig. 3 shows some of the matching pairs from our ERP manual study. For example the file `LoadEmployeeAction.java` was matched to both the `Manage Employee In` and `Query Employee In` service collections.

B. CRM case study

As a second case study we used a CRM enterprise services model also provided by SAP at [3], which is similar in structure to the ERP model described above. The CRM model has 162 service collections.

As a corresponding implementation we chose an open-source CRM application called SellWinCRM [7]. This is a medium-sized Java-based application for customer relationship management with functionalities specific to sales, marketing and service. SellwinCRM has 117 source files (we discounted 12 other files that contained only "plumbing" code).

We then manually matched the source files in SellWinCRM with the SAP CRM domain model. In the manual matching we carried out, there were 291 (*file*, *collection*) pairs, relating 61 source code files with 50 collections in the domain model.

The artifacts from our two case studies, including the domain models, source files, and manual matchings, are available from the home page of the first author of this paper.

VIII. Experimental Evaluation

We have implemented our approach in Java. We use Cohen's JaroWinkler [10] class for checking syntactic similarity, and the LSA package from Airhead Research [16]. We use the *doxygen* tool to extract an approximate call-graph from each application (in our applications virtual-methods are not used much). To find the database tables and fields referred to in each

source file we use a simple text-search technique, which, in our applications, turns out reasonably precise. In this section we describe our experiments on running our tool on the two case studies described in the previous section. We also discuss the precision and recall of our tool relative to the two respective "gold standard" matchings done there.

A. Experiments with ERP case study

We did our experimentation in two phases: (1) to identify appropriate choices for the basic settings of our approach, and (2) to explore the effects of propagation along relationship graphs.

Phase 1 – Basic settings: Since the use of relationship graphs and propagation is our chief technical contribution, our experiments overall are focused mainly on evaluating this idea. However, in order to run our tool at all, several basic settings need to be first made; we went about this task using a small amount of experimentation (wrt the gold standard), to arrive at what we considered were reasonable settings. In all our experiments in this phase we disabled propagation along all the graphs.

Our first experimental objective in this phase was to identify the kinds of features to extract and use from source code elements. Selecting features for directories and database tables is natural and straightforward, as was discussed in Section V. On the other hand, source files contain a variety of elements that can potentially be used as features, e.g., tokens in a file's name, tokens in its directory's pathname, the database table names and field names that occur in the file, tokens in the *header comment*[1] in the file, method names, and variable names. Our observation was that all the features named above *other than* variable names and method names were useful. If we used method and variable names the results actually became worse; this was because in these applications these names are typically library or framework artifacts (e.g., `executeCommand`, `getRequestName`), and are not based on domain concepts.

Our second objective was to identify the *ranking mode*, i.e., F2C or C2F (see Section III) that would give better results. Our observation was that on JAllInOne F2C gave vastly superior results than did C2F. Therefore in the rest of our JAllInOne experiments we used the F2C mode exclusively.

Our third objective was to identify appropriate parameter values for the syntactic and semantic similarity metrics that we use in the context of the soft-tfidf similarity measure (see Section V). We compared outputs from a few runs of the tool (using different values of these parameters) against the gold standard, and then decided on the following settings. For LSA, we reduced the number of feature dimensions (i.e., the wordspace) by 80%. In place of the single threshold θ on soft-tfidf similarity between two features (see Section IV), we use a threshold of 0.8 on the LSA similarity function and 0.97 on the string-sim similarity function. Finally, we set the *similarity threshold* U between documents (see Section III) to 0.15.

Phase 2 – The effect of propagation: Each of the three graphs $\mathcal{G}_r$, $r \in \{c, a, l\}$, that we use for propagation has

[1]A comment that typically appears near the top of each source file and briefly describes the file's contents. It is typically demarcated by tags that are application specific, which need to specified as input to our approach.

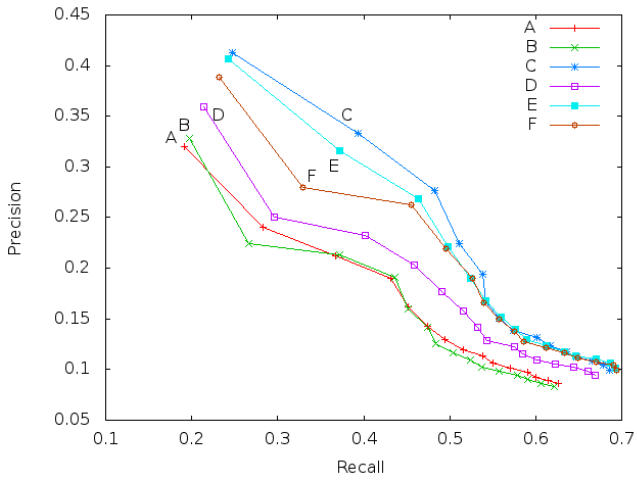

Fig. 4. Results of matching the ERP model with JAllInOne, for different configurations. Each point on a line is a successive value of k, starting from 1 at the left. (A) Baseline; Then, propagation along (B) Call graph; (C) Directory structure graph; (D) Database access graph; (E) Directory + database access; (F) Directory + call + database access graph

associated with it two parameters: its propagation weight α_i (see Section VI-A), and the threshold T_i on the influence weight $w_i(u, v)$ on its edges (introduced as parameter T in Section VI-A). Our eventual objective being to evaluate whether propagation per se is beneficial, any reasonable configuration of these parameters would make sense. We identified such a configuration by running the tool with a single relation enabled at a time. When relation i was enabled, we tried the values 0.0, 0.1, 0.2, and 0.3 for T_i, and values 0.5, 1.0, and 1.5 for α_i, thus resulting in twelve combinations (i.e., twelve runs of the tool). From among these twelve runs we identified the best run by manual inspection; our guiding principle was to choose a combination that resulted in the best possible recall, and in the case of ties, by looking at precision as a secondary factor. From the best run we picked up the corresponding values of α_i and T_i. In this way, the settings that we actually decided upon were as follows: α_c = 1.5, α_a = 0.5, and α_l = 1.5, T_c = 0.1, T_a = 0.1, T_l = 0.0. (For the parameter T_l we actually did not formally explore any values other than 0.0. This was because of our informal observation that 0.0 was almost always the best value for this.)

We now discuss, using Figure 4, the effect of propagation along relations in the source code. Each plot in the figure represents results from a *configuration*, which is a distinct subset of relations that are enabled for (feature as well as similarity) propagation. Each point on a plot represents precision and recall (wrt the JAllInOne gold standard) for a certain value of the cut-off k, with $k = 1$ being the leftmost point in each plot, $k = 2$ being the next point, and so on. For instance, consider Plot A, which is the baseline (i.e., no propagation at all); At $k = 1$ we get precision of 0.32 and recall of about 0.19; $k = 2$ yields precision of 0.24 and recall of about 0.28, and so on.

The Plots B through D show the results when we use each of the three relations in isolation, while Plots E and F show the results for selected combinations of relations.

The figure indicates that in the case of JAllInOne, configu-

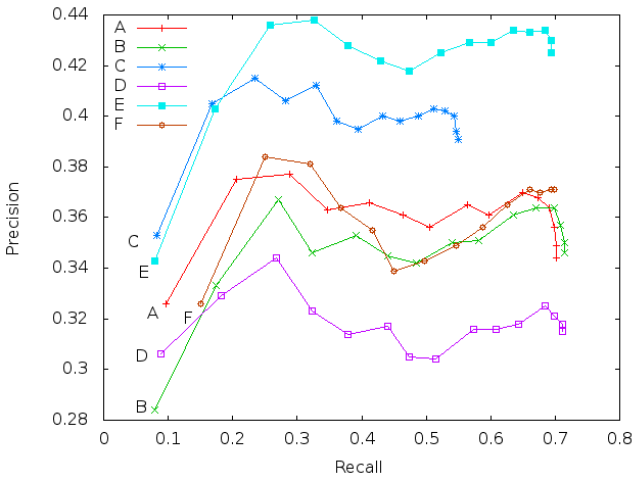

Fig. 5. Results of matching the SAP CRM model with SellWinCRM, for different configurations. Each point on a line is a successive value of k, starting from 1 at the left. (A) Baseline; Then, propagation along (B) Call graph; (C) Directory; (D) Database access graph; (E) Call graph and Directory; and (F) All three relations.

rations B and D, i.e., call graph alone, and database access alone, do not have a significant impact over the baseline. We investigated the reason for this. The implementation of most services in JAllInOne are heavily dependent on each other. They invoke each other, and also access each others tables as part of join operations. For instance, the files that implement sales order processing invoke the services that provide information about the customer that has placed the order, the services that provide information about the product being ordered, and so on (this is a simplified description; the actual inter-connections are more complex). Therefore, these relations do not play a significant role in the analysis of this application.

On the other hand, the JAllInOne source code is organized very well in directories. For most of the collections their implementing files happened to be in a single directory (there are about a 130 directories in this code structure). Therefore, propagation along the directory structure graph (Plot C) causes a big improvement to the results over the baseline.

Note that in Configuration C, which is the best configuration, the gain in precision by using propagation for a given value of recall ranges from about 52% (at recall of around 0.25) to about 90% (at recall of around 0.48). This demonstrates the excellent gain in precision that can be obtained using propagation.

B. Experiments with CRM case study

For our runs on SellwinCRM we used the exact same basic settings as we did for JAllInOne, except that we used the C2F ranking mode rather than the the F2C ranking mode (C2F gave much better results). We did re-tune the propagation parameters, in the same way as we did for JAllInOne, but using the SellwinCRM gold standard. We thus obtained the following parameter values: α_c = 0.5, α_a = 1.0, and α_l = 0.5, T_c = 0.0, T_a = 0.3, T_l = 0.0. We show the results of running our algorithm in six different configurations in Figure 5. The best results are obtained by propagation along the call graph and directory structures simultaneously (Plot E in the graph).

Our experience with SellwinCRM is different from our experience with JAllInOne in several of ways. Firstly, the (ideal) matching between the application and the corresponding domain model in this case turns out to be more fuzzy than in the case of JAllInOne. In fact, in the manual study, we had to match several of the source files with 7-10 collections each. Therefore, the recall of the tool increases with increasing values of k with almost no loss of precision. Secondly, even though the peak precision and recall are comparable with those obtained for JAllInOne, the gain due to propagation over the baseline is lower. The maximum value of this gain is around 20%, at a recall value of around 0.35. The primary reason for this effect is that the SellwinCRM source code is not as well organized in directories as in JAllInOne; in fact, there are only three directories, and many of the collections have files distributed over all three directories. Nevertheless, the gain due to propagation is significant. Finally, with JAllInOne, the F2C ranking mode does better, because very few of the collections in the domain model (only around 8%) are matched with any source file as per the gold-standard. Therefore, emitting k matching files for *each* collection in the domain model (which is what C2F would do) would reduce precision a lot. On other hand, in the case of SellwinCRM, around 32% of the collections in the domain model have matching files; moreover, many of the source files match a large number of collections, which means that recall would improve if we emitted k files for each collection. Therefore, the C2F ranking mode does better here.

C. More on parameter tuning

In practice, choosing appropriate values for the basic settings as well as the propagation parameters is important for good results. The tuning technique that we employed in our experiments above was basically to use the gold standards that we had earlier created, with our objective primarily being to demonstrate the potential value in using propagation. In practice, when gold standards usually don't exist, we feel that practitioners would need to manually match at least a small portion of the given application with the given domain model, thus creating a mini gold-standard. This can then be used for tuning the settings, which can then be used within the tool to obtain a full matching.

In order to test the above hypothesis we identified a small subset of the SellwinCRM gold standard, that consisted only of the source files and collections that pertained to two key domain concepts – customers, and "opportunities" (i.e., potential customers). This subset consisted of only 48 (*file, collection*) pairs (recall that the full gold standard consists of 291 such pairs). We then ran the tool, on the full source code and the full domain model, by enabling relations one at a time, just as we did during the tuning step described earlier; the only difference was that this time we computed the precision and recall numbers of each run wrt the subset gold standard (as opposed to the full gold standard). We then identified the best runs (as described earlier) using these numbers, and used these runs to identify the best propagation parameter values. To our surprise, these values turned out to be *identical* to the values that we had identified earlier using the full gold standard,

namely, α_c = 0.5, α_a = 1.0, and α_l = 0.5, T_c = 0.0, T_a = 0.3, T_l = 0.0. This is strong evidence that creating a small manual matching is an excellent way to seed the tool in order to obtain a good overall matching from it.

In the case of JAllInOne it turns out that with the best configuration, namely, propagation only along the directory structure, the quality of the results are not very sensitive to the value given to the propagation parameter α_l (recall that we had fixed T_l to be 0.0). Therefore, we did not perform a similar study as we described above for SellwinCRM.

D. Takeaways

We observe that propagation gives a significant boost to precision in both case studies. This has an important implication in the use cases mentioned in the introduction. For example in the JAllinOne case study, Configuration C gives slightly better recall at $k = 3$ than the baseline configuration (Plot A) does at $k = 6$. This results in 1617 fewer false-positive (*file, collection*) pairs for the user to wade through (each increase in the value of k by 1 causes 539 extra such pairs to be emitted).

We feel our tool is useful overall because it achieves good recall, which is very important in most applications. Our tool in some of its configurations gives recall of around 70%, with the corresponding precision being 10% and 42% (with the two applications, respectively). These levels of precision translate to about 9 (respectively 2.5) false positives per true positive, which is much more tolerable than the levels that might be expected from more naive tools such as grep. Furthermore, tools such as grep would likely not yield high recall, either, because they do not account for syntactic and semantic similarity between features.

We also note that the domain models we use have a high degree of redundancy within them; e.g., the two collections "Query Employee Time In" and "Manage Employee Time In', as shown in Fig. 2, have almost identical features. Therefore the gold standard's are over-approximated, as was discussed in Section VII.

This makes it difficult for any automated tool to achieve high precision and recall simultaneously.

Finally, we note that selecting the kinds of features to extract, the relationships to use for propagation, the values of parameters to the tool, the ranking modes F2C or C2F, etc., need to be done carefully. We considered three natural relations on the source code side, but note that there may be other relationships that could also prove to be useful. For all this tuning a user would typically have to manually match a small portion of the application with the given domain model, and tune the parameters of the algorithm using this experience. The algorithm can then be used on the whole application.

IX. Related Work

Previous work related to our paper falls into two categories – work specifically on matching documents or queries with source code, and work on information retrieval or classification in other contexts.

A. Matching documents with code

There is a significant body of work reported in the literature on matching or linking model documents with code artifacts. The closest approaches to ours among these are ones that use IR techniques: the approaches of Antoniol et al. [17], Marcus et al. [18], Zhao et al. [19], and Peng et al. [20]. They all use the vector space model and cosine similarity to find a ranking of model elements for each code element, or vice versa. Of these approaches, the first two do not use any kinds of relationships; our approach is conceptually very similar to these, except that we have add feature and similarity propagation as an extension.

The approach due to Zhao et al. uses an "extended" call-graph relationship among source code elements. However, they do not use feature propagation or similarity propagation. Rather, after initially labeling each source file with one or more model elements, they attempt to "extend" the labeling of each source file to other source files that are reachable from it in the call-graph, as long as this does not cause source files to receive labels from multiple other source files. As we observed in Section VIII, in our setting there are a large number of call-edges between functions that logically match with different model elements. Therefore, the approach of Zhao et al. is likely to frequently encounter situations where extension of labels is not possible at all. Our solution to this problem is to (a) assign each source file a *vector* of labels (weighted with similarity scores), and (b) judiciously propagate feature and similarity vectors only along immediate neighbors in a relationship, rather than along transitive paths.

The approach of Peng et al. [20] considers relationships *both* on the model side as well as on the code side. Their approach is to update the similarity score of a pair (f, p), where f is a model element and p is a source code element, by a factor that is tied to the *number* of neighbors of f that are already matched with neighbors of p. In contrast, our approach is to update the similarity score of (f, p) using the similarity score of (f', p), where f' is a neighbor of f.

A key difference between our setting and the settings of all the the approaches mentioned above is that we consider a domain model and an application that have been developed *independently* of each other. The previous researchers have all studied the problem of matching a document that was created specifically to describe an application with the same application. Our setting is significantly more complex, because of terminological as well as structural mismatches between the model and the code. In order to overcome these challenges we employ more sophisticated techniques to do the matching, namely, (a) the soft-tfidf measure to integrate different metrics of feature similarity, such as LSA similarity and syntactic similarity, and (b) feature and similarity propagation along relationships among source code elements.

Even though our setting is different and more complex than that of the previous approaches, our results compare reasonably with theirs. We specifically looked at the results reported by Marcus et al. on the LEDA application, and compared these with our results on JAllInOne. At low values of k their reported recall (resp. precision) is greater than ours by 160% (resp. 80%); however, at higher values of k this margin shrinks to about 60% (resp. 0%). Note that our applications and domain models are different from theirs, and *not co-developed* with

each other; whereas, they matched an application with its *own* documentation (created by the same developers). Considering this, we believe our precision is actually quite good. Our recall is lower than theirs, but one reason for this is that our gold-standard over-approximates the matches between the domain model and the application (as discussed in Section VIII-D).

The approach of McMillan et al. [8] addresses a related but different problem: that of identifying source code elements that match a given *query*. They use similarity propagation (but no feature propagation) along the call-graph relation. Their problem is different from ours, because our models can be thought of as a *set* of queries that need to be *simultaneously* answered, while avoiding (ideally) the assignment of any single source-code element to more than one model element. Our F2C ranking mode specifically targets this scenario; note their approach includes the analogue of our C2F mode, but not our F2C mode.

B. Information Retrieval

Here, we place our work in the context of existing approaches on information retrieval and machine learning outside the source-code matching domain.

The web information retrieval community has extensively studied networked documents. The page-rank family of algorithms (e.g., [21]) answers a query about a networked document by de-coupling the aspect of finding similarities between the query and the individual nodes in the network (i.e., the "online" part) from the aspect of propagating similarities via the network (i.e., the "offline" part). In principle our coupled approach is more powerful, although potentially not efficient enough for online use. Also, extending page-rank-style algorithms to work on source code, with multiple types of relations, requires non-obvious extensions.

The machine learning community has shown a lot of interest over the last decade in the problem of *classification* or *categorization* of nodes in networked documents. The stage for this direction of work was set by the work of Chakrabarti et al [22], who demonstrated the effectiveness of relaxation labeling, or propagating labels over a network, for this problem. These approaches use both feature propagation as well as similarity propagation. The approach of Baluja et al. [9] extended this problem to the problem of assigning multiple labels to each node, with each label weighted by a score. However, we are not aware of a systematic evaluation of these approaches for legacy code analysis, where the relations are of many different types, and semantically different from those of web or social networks.

X. Conclusions

In this paper, we have addressed the important problem of matching a textual model to existing code. Based on the vector space model, our solution comprehensively makes use of a wide range of evidences for this task — the textual descriptions in the model, the source files, directory structure and database tables in the implementation, and more importantly, the relations between the various entities in the implementation. We have evaluated this approach using two real domain models and two independently developed Java implementations, and showed that it leads to significant boost in both the precision and recall of the matching task.

Acknowledgments

This work was sponsored by Infosys Limited.

References

[1] W. M. Ulrich, *Legacy systems: transformation strategies*. Prentice Hall PTR, 2002.

[2] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley, "SOMA: A method for developing service-oriented solutions," *IBM Systems Journal*, vol. 47, no. 3, pp. 377–396, 2008.

[3] "Enterprise Services Workpl." http://esworkplace.sap.com/.

[4] J. Huschens and M. Rumpold-Preining, "IBM insurance application architecture (IAA): An overview of the insurance business architecture," in *Handbook on Architectures of Information Systems*. Springer, 2006, pp. 669–692.

[5] "Banking Industry Archictecture Netw." http://www.bian.org/.

[6] "JAllInOne," `http://jallinone.sourceforge.net`.

[7] "Sellwin," `http://sellwincrm.sourceforge.net`.

[8] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Int. Conf. on Software Engineering (ICSE)*, 2011, pp. 111–120.

[9] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *Proc. Int. Conf. on World Wide Web*, 2008, pp. 895–904.

[10] W. Cohen, P. Ravikumar, and S. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proceedings of IIWeb*, 2003.

[11] E. Moreau, F. Yvon, and O. Cappé, "Robust similarity measures for named entities matching," in *Proceedings of the 22nd International Conference on Computational Linguistics*, ser. COLING '08, 2008.

[12] C. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[13] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, pp. 391–407, 1990.

[14] D. Liben-nowell and J. Kleinberg, "The link-prediction problem for social networks," *J. American Society for Information Science and Technology*, 2007.

[15] H. S. Gupta, D. D'Souza, R. Komondoor, and G. M. Rama, "A case study in matching service descriptions to implementations in an existing system," in *Proc. 2010 Int. Conf. on Softw. Maint. (ICSM)*, 2010, pp. 1–10.

[16] D. Jurgens and K. Stevens, "The s-space package: an open source package for word space models," in *Proceedings of the ACL 2010 System Demonstrations*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 30–35.

[17] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 970–983, 2002.

[18] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. 25th Int. Conf. on Softw. Engg. (ICSE)*, 2003, pp. 125–135.

[19] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a static noninteractive approach to feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, pp. 195–226, 2006.

[20] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao, "Iterative context-aware feature location," in *Proc. Int. Conf. on Software Engineering – NIER Track*, 2011, pp. 900–903.

[21] M. Henzinger, "Link analysis in web information retrieval," *IEEE Data Engineering Bulletin*, vol. 23, pp. 3–8, 2000.

[22] S. Chakrabarti, B. Dom, and P. Indyk, "Enhanced hypertext categorization using hyperlinks," in *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1998, pp. 307–318.