Inferring Likely Mappings Between APIs using Text Mining

Rahul Pandita[†], Raoul Jetley[∓], Sithu D Surarsan[∓], and Laurie Williams[†]
[†]North Carolina State University, Raleigh, NC, USA
[∓]ABB Corporate Research, Bangalore, KA, India
rpandit@ncsu.edu, abc@xyz.com, sudarsan.sd@in.abb.com, williams@csc.ncsu.edu

ABSTRACT
Mapping across different platform or language Application Programming Interfaces (APIs) is highly desirable. Among other reasons, mappings facilitate machine-based migration of an application from one API to another. Such translation is particularly relevant in current context as developers are increasingly migrating applications to diverse platforms or languages to attract and retain users across such platforms or languages. Given a typical platform or language expose a large number of API methods for developers to reuse, manually writing these mappings is prohibitively resource-intensive and may be error prone. Recently, researchers proposed to automate the process by mining API mappings from existing codebases. However, these approaches rely on existence of manually ported (or at least functionally similar) code across source and target API's. To address the shortcoming, in this paper we propose TMAP: **t**ext **m**ining based approach to infer **API** mappings from the textual description present in API documents. To evaluate our approach, we apply TMAP to infer API mappings across: 1) `Java` and `C# API`, 2) `J2ME` and `Android` API. We next compare the inferred mappings with state-of-the-art code mining based approaches. Our results indicate that TMAP is effective in inferring API mappings with the more than XX accuracy.

Categories and Subject Descriptors
H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms
Theory

Keywords
ACM proceedings, LaTeX, text tagging

1. INTRODUCTION
Software is ubiquitous and lately people are increasingly interacting with applications that run on variety of software platforms on their consumer devices on daily basis. For instance, a typical user may view emails on his/her android or iOS or windowsPhone handheld device such as a mobile phone or a tablet while on the go. The user then can switch to a desktop environment such as Windows or Linux or OSX in an office like setting through either a web browser or a standalone application such as Microsoft Outlook or Mozilla Thunderbird.

To retain users across different platforms, developers are increasingly releasing different versions of their applications. For example, a mobile software developers often release their applications for all the popular mobile platforms such as Android, iOS, and Windows. In context of desktop software, many well-known projects such as JUnit, Hibernate provide multiple versions in different languages, in an attempt to attract developer community to use these libraries across different languages.

However, manually migrating a software from one platform (/language) to another can be time consuming and may be error prone. Existing language migration tools such as Java2CSTranslator [9] partially alleviate the problem, as they require a programmer to manually describe how Application Programming Interfaces (APIs) of a source platform(/language) maps to API's of the target platform (/language). Given a typical platform (/language) expose a large number of API methods for developers to reuse, manually writing these mappings is prohibitively resource intensive and may result in manual error.

The goal of this research is to support software developers in migrating an application from one API to another by automatically inferring method mappings across API using text mining on the natural language API method descriptions

In the literature, there exist approaches that address the problem of finding mapping between software API's using static [26] and dynamic [5] analysis. Recently Nguyen et al. [15] also proposed to apply statistical language translation techniques to achieve language mapping by mining large open source software repositories. However, aforementioned approaches rely on the presence of manually ported (or at least functionally similar) software across source and target API's. Furthermore, accuracy of static analysis and mining based approaches [15, 26] is dependent upon the quality of the code under analysis. Likewise, accuracy of dynamic approaches like [5] is function of the quality and completeness of test case to execute the comprehensive API behavior dynamically.

To address the shortcomings of existing program-analysis based approaches, we propose to use the natural language API method descriptions to infer method mappings across APIs. Our key intu-

ition is: *"since the documentation is targeted towards developers, there may be an overlap in the language used to describe similar concepts that can be leveraged."* In general, API documentation provides developers with useful information about class/interface hierarchies within the software. Additionally, API documents also provides information about how to use a particular method within a class by means of method descriptions. Method descriptions typically describe specifications in terms of the expectations of the method arguments and functionality of method in general.

This paper presents TMAP: An approach that leverages the natural language method descriptions to find the likely mapping. In particular, TMAP proposes to create a vector space model [13, 20] of the target API method descriptions. TMAP then queries the vector space model of target API using automatically generated queries from the source API method descriptions. TMAP automates the query generation in source API using the concepts from text mining such as emphasizing (or de-emphasizing) certain keywords over others and querying multiple facets (such as class description, package names etc...).

We pose the following research question : *How accurately can the similarity in the language of API method descriptions be leveraged to infer API Mappings?* To answer our question, we apply TMAP to infer API mappings across: 1) `Java` and `C# API`; 2) `J2ME` and `Android` API. We next compare the inferred mappings with state-of-the-art code mining based approaches [5, 15]. Our results indicate that TMAP is effective in inferring API mappings with the more than XX accuracy.

In summary, TMAP leverages natural language description of API's to infer likely mapping thus facilitating cross API migration of applications. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of the library. Our paper makes the following major contributions:

- A technique that effectively infers mapping across source and target API.
- A prototype implementation of our approach based on extending the Apache Lucene [12]. An open source implementation of our prototype can be found at our website [1].
- An evaluation of our approach on J2ME to Android API and Java to C# API

The rest of the paper is organized as follows. Section 2 presents an real world examples that motivate our approach. Section 3 discusses related work. Section 4 presents the background on Text Mining. Section 5 presents our approach. Section 6 presents evaluation of our approach. Section 7 presents a brief discussion and future work. Finally, Section 8 concludes.

2. EXAMPLES

We next present an example to motivate our work and list the considerations for applying text mining techniques on API documents. This example is from real Java Platform, Micro Edition (JME) formerly known as J2ME and Android API. Both J2ME and Android use Java as the language of implementation and targeted towards hand-held devices. However, our approach is independent of language of implementation and thus can be used for any similar source and target API.

[1] `http://xx.com/`

`public void drawString(String str,int x,int y,int anchor)`
Draws the specified String using the current font and color. The x,y position is the position of the anchor point. See anchor points.
Parameters
str - the String to be drawn
x - the x coordinate of the anchor point
y - the y coordinate of the anchor point
anchor - the anchor point for positioning the text
Throws
NullPointerException - if str is null
IllegalArgumentException - if anchor is not a legal value
See Also
`drawChars(char[], int, int, int, int, int)`

Figure 1: drawString method of Graphics class in J2ME

`public void drawText (String text, float x, float y, Paint paint)`
Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted based on the Align setting in the paint.
Parameters
text - The text to be drawn
x - The x-coordinate of the origin of the text being drawn
y - The y-coordinate of the origin of the text being drawn
paint - The paint used for the text (e.g. color, size, style)

Figure 2: drawText method of Canvas class in Android

Figure 1 shows the API method description of `drawString` method from J2ME API. Figure 2 shows the API method description of the Android API equivalent method `drawText`. Notice the overlap in the language of these two methods. From a human perspective its easier to conclude that the two methods offer similar functionality after reading the descriptions. However, there are lot of intermediate steps to reach that conclusion. For instance, Android API has more than 23,000 public methods. Manually going through each method description is impractical. Provided we are given a keyword based search framework (a very basic text mining infrastructure) for querying the Android API method descriptions, the next step is: *what does the query look like?*.

Table 1 lists some of the queries we tried on an index of Android API method descriptions to find a mapping for the J2ME method `drawString` shown in Figure 1. Column "Query" describes the keywords we used to query the index. Column "Hits" lists the number of matches found, and Column "Top-10" lists the rank of the first relevant method in top ten results. For instance, when we queried for the class name "Graphics" in Android API we did not get any hits. We also did not get any hits for when we queried for method name using keywords "drawString".

When we queried for the method signatures we got a 23,547 hits (almost all methods in Android API), mostly because of the confounding effects of keywords such as "public". Most of the methods signatures will have the keyword "public". Although the ranking mechanism did push methods with most keyword matches first, we did not find a relevant method in top 10. Likewise the method

Table 1: Query Results

#	Query	Hits	Top-10
1	Class Name: "Graphics"	0	-
2	Method Name: "drawString"	0	-
3	Method Signature	23547	-
4	Method Description: (Complete)	16820	
5	Method Description: (summary sentences)	94230	-
6	Combination	1479	**3**

descriptions as a whole or in parts also did not yield better results. In this example a combination of various attributes such as method name split in camel case notation "draw String", keywords from both class and method description resulted in the Android API equivalent method `drawText` shown in Figure 2 in the top ten results.

The examples demonstrates difficulties faced by simple keyword-based searches and text mining in general to find likely method mappings. In general, there are following difficulties in applying text mining approaches to API documents :

1. *Confounding effects*. Certain method names have an confounding effects. For instance `toString()`, `get()`, `set()` are too generic and tend to have similar descriptions across different method definitions. These generic methods often cause interference with the output of text mining based approaches. The problem is to automatically identify such methods to de-emphasize their importance in the query.

2. *Weights*. Not all terms in a method descriptions are equally important keywords. For instance, the term "zip" in the sentence "opens a zip file" is more important than the terms "opens" and "files" as the term empathizes on the specific type of file. The problem is to automatically identify the importance of a term.

3. *Structure*. API documents are not flat contiguous text blobs. They often have well defined structure, that is often shared by method descriptions. Ignoring the structure may cause ineffective queries negatively affecting the results. This is addressed by querying individual elements (such as class description, class names, method names, method descriptions) and then aggregate the results. The problem is to effectively aggregate the results.

3. RELATED WORK

Language migration has been an active area of research [7, 14, 23, 24], with myriad techniques that have been proposed over time to achieve automation. However, most of these approaches focus on syntactical and structural differences across languages. For instance, Deursen et al. [23] proposed an approach to automatically infer objects in legacy code to effectively deal with differences between object-oriented and procedural languages.

However, El-Ramly et al. [3]'s experience points out that most of these approaches support only a subset of API's for migration. Another recently published survey by Robillard et al. [19] provides a detailed overview of techniques dealing with mining API mappings.Among other works described in [19], Mining API Mapping (MAM) [26] is most directly related to our work.

MAM mines API mapping relations across different languages for language migration, however there is a significant difference between MAM and TMAP. MAM takes into input a software S written in source language and manually ported version of S' written in target language. MAM then applies a technique called "method alignment" that pairs the methods with similar functionality across s and S'. These methods are then statically analyzed to detect mappings between source and target language API. While MAM relies on existence of software that has been ported manually from a source to target API, our approach is independent of such requirement. In contrast, our approach relies on text mining of source and target API document to achieve the same. Furthermore, we demonstrate that our approach also infers equally good mappings if not better **pending evaluation**.

Gokhle et al.'s [5] approach Rosetta addresses the limitations of MAM to infer method mappings. In particular, Rosetta relaxes the constraint of having software that has been manually ported from a source to target API. Instead, they use functionally similar software in source and target API. For instance, they use two different 'tic-tac-toe' game applications in J2ME and Android API not necessarily manually ported. They then manually execute these applications under identical (or near identical) inputs and collect execution traces. Finally they analyze the collected execution traces to infer method mappings. In contrast our approach is independent of both rthe requirements: 1) to have functionally similar applications, 2) to manually execute such applications using similar inputs.

Recently Nguyen et al. [15, 16] proposed to apply statistical machine translation based techniques to achieve language migration. Their approach builds upon a previous result [8] that demonstrates the effectiveness of using a n-gram model to predict the next token in software source file given a large corpus of software source files to learn from. Similar to MAM [26] approach they need a software S written in source language and manually ported version of S' written in target language. They then consider source code as a sequence of lexical tokens and apply a phrase-based statistical machine translation model on the lexemes of those tokens. In contrast our approach is independent of such requirement(presence of S and S').

Furthermore, from an infrastructure perspective, TMAP is independent of the programming language or API under consideration. In contrast program analysis based approaches like [5, 15, 26] may need significant efforts for adding support to additional API's.

Zheng at al. [25] mine search results of web search engines such as Google to recommend related APIs of different libraries. In particular, they propose heuristics to formulate keywords using the name of the source API, and the name of target API to query web search engine. For instance, to search for an equivalent class in C# for the `HashMap` class in Java, a user may manually enter "HashMap C#" in a web search engine. The results are computed one by one and candidates are ranked by relevance, mainly according to their frequency of the appearance of keywords in the query. This work provides only preliminary results and queries proposed are of a coarse grain. Furthermore, the results are susceptible to influence by the outcome of the web search engines. In contrast, our approach is independent of the web search engine. Furthermore, the queries used in our approach are more sophisticated than the heuristics proposed by their approach.

Information retrieval techniques [1, 6, 10, 18] are also being in-

creasingly used in Code Search. We next describe some relevant approaches. Sniff [1] annotates the source code with API document descriptions. Sniff then performs additional type analysis on the source code to rank relevant code snippets. Exemplar [6] uses the text in API documents to construct a set of keywords associated with an API call. Exemplar then uses the keywords list to perform query expansion to achieve code search. However, these approaches are targeted towards code search in a single API. In contrast TMAP works on inferring mappings across multiple APIs.

Text analysis [2, 11, 17, 27, 28] of API documentation is increasing being used to infer interesting properties. We next describe most relevant approaches.

API property inference Zhong et al. [27] employ NLP and ML techniques to infer resource specifications from API documents. They define resource specifications as the rules governing the usage of resources such as File. Recently Treude et al. [22] leverage rule-based NLP approaches to infer action (programming task) oriented properties from an API document. They describe tasks as specific programming actions that have been described in the documentation. First, TMAP uses text mining stead of sophisticated NLP techniques used in these approaches. Second, TMAP infers API mapping relations across different API for language migration, whereas the previous approaches mine API properties of a single API to detect defects or to assist programming. Finally, the performance of the preceding ML-based approaches is dependent on the quality of the training sets used for ML. In contrast, our approach is independent of such training set and thus can be easily extended to target respective problems addressed by these approaches.

4. BACKGROUND

Text mining is a broad research area including but not limited to the techniques facilitating retrieval/manipulation of useful information from a large corpus to text. It is claimed that more than 80% of business intelligence is captured in unstructured data **need citation**. A large part of this is in documents containing narrative text, such as reports, presentations, web data, design documents, etc. Text mining is used to unearth the knowledge buried in these documents through a combination of extraction and analytical methods.

As opposed to traditional data mining, text mining analyzes free-form text spread across a large number of documents (rather than data localized and maintained within a database). In order to analyze this data, text mining uses concepts from traditional data analytics, natural language processing, and data modeling.

We next introduce the concepts from text mining that have been used in the presented approach.

Indexing [4, 13]: Indexing is the process of extracting text data from source documents and storing them in well-defined indexes. Indexing process constitutes of following steps.

A document is divided into constituent unit, known as tokens, based on a well-defined criterion. A token is usually an individually identifiable unit of the document (such as a word) and relates to the individual terms stored in the index.Once the tokens within each document are identified, they are added to the index, with the corresponding link to the document and associated term frequencies. An optional pre-processing step further assists with indexing: such as removing stop-words, grouping similar words together.

Among various indexing strategies, use of inverted file indexing [4] is well suited for large document collections. In its simplest form inverted file index maintains the mapping of terms and their location in a text collection. For instance a text document can be thought of as a collection of m words. It is made up of a sequence of n unique words such that $n <= m$. The number n is usually far less than m as most of the words are repeated while forming a document. For instance the word "the" is repeated many times in this paper. The set of unique words within an index forms the "Term List" v of the index. If a pointer (say numeric location) is associated with each word in v to the location of that word in text document, the resultant data structure is a form of inverted file index. As the document collection grows, the number of documents matching a word in the index becomes sparser.

The index is oftentimes annotated with the information of the frequency of occurrence of each term in the document. This representation of a document as a vector of frequencies of terms is referred to as *vector space model* [4, 20].

TF-IDF [13]: Term -frequency inverse-document-frequency (`tf-idf`) is a numerical statistic that is intended to capture the importance of a term to a document in a corpus. Often used as a weighting metric in information retrieval and text mining, the `tf-idf` weight increases proportionally to frequency of the occurrence of a term in a document, however the weight is also offset by the frequency of occurrence the term in the corpus.

Cosine Similarity [20]: In mathematics Cosine similarity is a numerical statistic to measure the similarity between two vectors. Cosine similarity is defined as a dot product of magnitude of two vectors. In context of text mining, the Cosine similarity is used to capture the similarity between two documents represented as term frequency vectors.

5. APPROACH

We next present our approach for inferring likely mappings of API methods across APIs. Figure 3 provides an overview of TMAP approach. TMAP consists of three major components: 1) an Indexer; 2) Query Builder; and 3) Searcher components.

The Indexer accepts the API documents of the target API and creates indexes (a vector space model) of these documents by extracting intermediate contents from the method descriptions. The Query Builder accepts the API documents of the source API and creates queries to be executed on the indexes (a vector space model). Finally, Searcher component execute the queries on the indexes and generates an ordered set of the API methods from target API documents as mapping results to be presented to the developers for confirmation. We next describe each component in detail.

5.1 Indexer

This component accepts the API method descriptions of the target API and creates indexes (a vector space model) of these documents. In particular, Indexer extracts the following fields from the API method descriptions:

- **F1) Type Name**: The name of enclosing class/interface of the method. For the method description of `drawString` shown in Figure 1, indexer extracts the type Name as "Graphics".

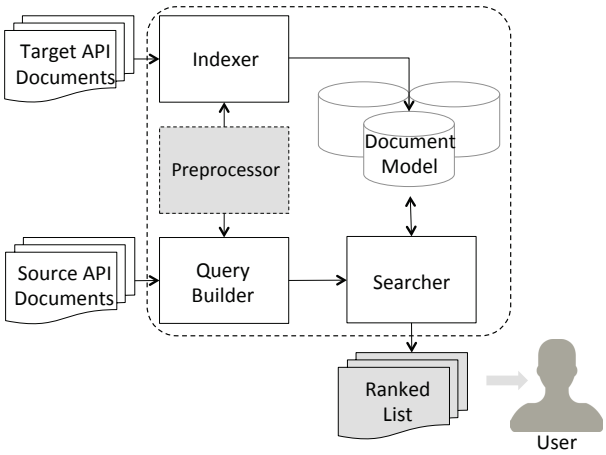

Figure 3: Overview of TMAP approach

- **F2) Package Name**: The package name of the enclosing type. For the method description shown in Figure 1, indexer extracts the package name as "javax.microedition.lcdui".
- **F3) Method Name**: The name of the method. For the method description shown in Figure 1, indexer extracts the method name as "drawString".
- **F4) Method Modifiers**: The access modifiers of the method. For the method description shown in Figure 1, indexer extracts the method modifiers as "public".
- **F5) Method Return Type**: The return type of the method. For the method description shown in Figure 1, indexer extracts the return type as "void".
- **F6) Exceptions Thrown**: Name of the exceptions thrown by the method. For the method description shown in Figure 1, indexer extracts the exception names as "NullPointerException" and "IllegalArgumentException".
- **F7) Parameters**: The name and type information of parameters. For the method description shown in Figure 1, indexer extracts the parameter name and type information as "String str","int x","int y", and "int anchor".
- **F8) Class Description**: The description of enclosing type. Class description is not shown in Figure 1 for the space considerations.
- **F9) Method Description**: The description of the method. For the method description shown in Figure 1, indexer extracts the method description as the all the text except method deceleration in Line 1.

This step is required to extract the desired descriptive text from the API method descriptions. Additionally, this step allows TMAP to deal with the *structure* issue as presented in Section 2. Different API documents may have different styles of presenting information to developers. Such stylistic differences may also include the difference in the level of the details presented to developers. TMAP relies only on basic fields that are generally available for API methods across different presentation styles.

After extracting desired information, extracted text is further preprocessed. The preprocessing steps are required to make the text amenable to text mining techniques that are used in the subsequent phases of the TMAP approach. In particular, TMAP performs the following basic prepossessing steps:

- **P1) Presentation Elements**: A typical API method description is often interleaved with presentation elements for better readability. For instance, `JavaDoc` provides a list of identifiers such as `@Code` and `@link`. These identifiers are automatically translated into presentation markup such as links and fonts. Although such elements are part of the description text, these elements often cause noise in the text mining techniques to compute similarity. Therefore, this preprocessing step cleans the method descriptions to remove such elements. We use a static list of presentation elements to achieve cleaning in this step with relatively high accuracy.
- **P2) Split Package Notation**: In method descriptions, the "." character is used as a separator character for package names like "javax.microedition.lcdui". We use regular expressions to split the package name into constituent words to facilitate search on individual words in the package name. For example, "javax.microedition.lcdui" is broken into "javax microedition lcdui".
- **P3) Split CamelCase Notation**: API method descriptions are often interleaved with programming identifiers such as class names and method names. Oftentimes these identifiers use CamelCase notation. The CamelCase notation is the *de facto* mechanism used by programmers for combining phrases into a single word, such that each word in the phrase begins with a capital letter. Previous research [11] demonstrated the benefit of splitting the CameCase word into its constituent phrase for automated code completion. TMAP splits such identifiers into constituent phrase to better facilitate searching. TMAP leverages the well-formed structure of CamelCase notations to encode a regular expression to achieve splitting with relatively high accuracy. For example, "drawString" is broken into "draw String".
- **P4) Lowercase**: This step involves converting the text description to lower case. The step is performed to normalize the text for making the keyword match case intensive, further increasing the range of queries.
- **P5) Stemming**: This steps transforms the words in the description to their base form. Stemming is very effective in extending the range of keyword based queries to match various operational forms of the words. For instance, "has", "have", and "had" are are reduced to the stem "ha".

After preprocessing, TMAP next creates indexes of the API method descriptions. An index is collection of documents where each document is made up of values organized into well defined fields. TMAP considers every method description as an individual document and uses the following major fields (as previously described) : 1) combination of package and class name; 2) class description; 3) method signature; 4) method name; and 5) method description. The values are of these fields is text after preprocessing. TMAP uses a vector space model representation of the documents for each field. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors

Type Name: `java util iter`
Type Description: `iter over collect iter take enumer java collect framework`
Method Name: `ha next`
Method Description: `return true iter ha more element other word return true next would return element rather than throw except`

Figure 4: Query

java.util Interface Iterator

hasNext

`boolean hasNext()`

Returns `true` if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

Returns:

`true` if the iteration has more elements

Figure 5: hasNext method of Iterator

of identifiers and their frequency of occurrence. In case of TMAP, each word is considered as a term except the stop words such as "a", "the", and "and".

5.2 Query Builder

This component accepts the API method descriptions of the source API and creates queries for method descriptions. These queries are executed on the target API index to retrieve an ordered list relevant API methods. In particular Query Builder uses the same preprocessing steps followed by Indexer (listed in Section 5.1). After extracted desired descriptive text from the API method descriptions, this component systematically creates search queries to search for different fields in Indexes. Keywords for searching in "Type Name", "Type Description", "Method Name", and "Method Description" fields are derived from their equivalents extracted descriptive text.

For instance consider the method description shown in Figure 5 and equivalent query in shown in Figure 4. The Keywords for "Type Name" are derived from preprocessing "java.util.Iterator", that results in "`java util iter`". Notice the package notation is split into individual words and "Iterator" is further transformed to lower case and its stem "iter". Likewise, keywords for field "Method Name" is derived by preprocesing "hasNext", which is first split into "has Next" and then transformed using stemming into "`ha next`".

For generating keywords to query the "Type Description" field TMAP considers following heuristic: *Heuristic H1 : the first paragraph or the first five sentences of the type description (whichever is shorter) provides reasonable keywords for searching equivalent class in target API.*

Likewise for generating the keywords to query the "Method Description" field, we consider the following heuristic: *Heuristic H2 : the first paragraph or the first two sentences of the method description (whichever is shorter) provides reasonable keywords for searching equivalent method in target API.*

TMAP uses these heuristics to improve the performance of searching infrastructure that tends to be inversely proportional to the complexity and length of the query. Furthermore, our insight is that the document writers tend to describe the general overview of class and method description in the first few sentences followed by implementation and design specific details later. **Need citation from the summarization**

Weigths for terms. As mentioned in Section 2 that all terms in a description are not equally importance keywords. TMAP further enhances the query by quantifying the importance of a term in description and then translating that importance into a weight of that keyword in the query. In particular, we propose to use tf-idf [13] as a means to quantify importance of a term. For each term in the method description TMAP calculates the number of times the term occurs in that method description $freq_{mtd}$. TMAP then calculates the number of documents in the corpus that contains the term $freq_{doc}$. TMAP then calculates the tf-idf sore of the term as : $(1 + log(freq_{mtd}) * (1 + log(total_{mtd}/freq_{doc}))$. The tf-idf sore of the top-k term as used as the weights for the terms occurring in the query.

For the API method description shown in Figure 5, TMAP calculates "iter", "element", and "more" as most important terms with tf-idf scores of 9, 8.2, and 7.9 respectively. We augment the query shown in Figure4 with the computed weights for the keywords respectively.

5.3 Searcher

The searcher components accepts the query from Query Builder component and queries the index generated by Indexer component. The results are then ranked and presented to end user for review. In searcher is realized as follows. First all the documents that match the keywords and clauses in a query are returned. Then, the returned documents are ranked using the cosine similarity [20] of the terms in query and the terms in returned documents. In mathematics Cosine similarity is a numerical statistic to measure the similarity between two vectors. In information theory [13], cosine similarity is the standard statistic to rank relevant documents.

5.4 Implementation

We implemented a prototype version the TMAP approach. We first manually download the HTML version of API documents of libraries. We then implemented a parser for extracting the requisite text from these documents using Jsoup [2], which is a java library for working with HTML documents. In particular our prototype implementation parses: 1) Oracles Javadoc style; 2) Android style documentation; and 3) Microsoft's MSDN documentation.

We next implemented the indexing, query building, and searching infrastructure using Lucene [12]. Lucene is a high-performance, full-featured text search engine library written entirely in Java.. Our prototype implementation and evaluation subjects are publicly available on the project website.

6. EVALUATION

[2] `http://jsoup.org/`

Table 2: Comparison with SNIFF

Query	Relevant Mtd rank	
	SNIFF	TMAP
get active editor window from eclipse workbench	1	1
parse a java source and create ast	1	1
connect to a database using jdbc	1	1
display directory dialog from viewer in eclipse	1	1
read a line of text from a file	1	1
return an audio clip from url	1	1
execute SQL query	2	1
current selection from eclipse workbench	1	1

We conducted an evaluation to assess the effectiveness of our approach. In our evaluation, we address three main research questions:

- **RQ1**: What are the precision and recall of our approach in identifying API mappings?
- **RQ2**: How do the mappings inferred by our approach compare with the human written mappings?
- **RQ3**: What is the effectiveness of using free form queries using our approach?

6.1 Subjects

We used the API documents of the following library pairs as subjects for our evaluation.

J2ME-Android. Java Platform, Micro Edition, or Java ME, is a Java platform designed for embedded systems (mobile devices are one kind of such systems). Target devices range from industrial controls to mobile phones (especially feature phones) and set-top boxes. Java ME was formerly known as Java 2 Platform, Micro Edition (J2ME).

Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers.

Java -C#

Effectiveness in free form queries:

Discussion with Authors

None of the mapping methods point to AlertDIalog Class in Android that seems to be the class implementing alert functionality. Furthermore, consider the first-mapping

```
Alert.setString Paint.setAlpha; CompundButton.setChecke
```

None of the mapped methods set text in the alert. How to interpret the mapping in such cases?

Amruta's Response :

First, a note of clarification. As explained in the paper, we restricted ourselves to sequences of length up to 2 when inferring mappings

Table 3: Comparison with Rosetta

S. No.	Method	TMAP Rank
1	Alert.setTimeout	
2	Alert.setType	
3	Canvas.getHeight	
4	Canvas.getWidth	
5	Canvas.repaint	
6	Canvas.serviceRepaints	
7	Canvas.setFullScreenMode	
8	Command.getCommandType	
9	Command.getLabel	
10	Display.getCurrent	
11	Display.getDisplay	
12	Display.setCurrent	
13	Displayable.addCommand	
14	Displayable.getHeight	
15	Displayable.getWidth	
16	Displayable.removeCommand	
17	Font.charWidth	
18	Font.getHeight	
19	Font.stringWidth	
20	Form.addCommand	
21	Form.setCommandListener	
22	game.GameCanvas.getHeight	
23	game.GameCanvas.getWidth	
24	game.GameCanvas.repaint	
25	game.GameCanvas.serviceRepaints	
26	game.GameCanvas.setFullScreenMode	
27	game.Layer.getHeight	
28	game.Layer.getWidth	
29	game.Layer.setPostion	
30	game.Sprite.paint	
31	game.Sprite.setFrame	
32	game.Sprite.setPosition	
33	game.Sprite.setRefPixelPosition	
34	Graphics.clipRect	
35	Graphics.drawArc	
36	Graphics.drawChar	
37	Graphics.drawImage	
38	Graphics.drawLine	
39	Graphics.drawRect	
40	Graphics.drawString	
41	Graphics.fillArc	
42	Graphics.fillRect	
43	Graphics.fillRoundRect	
44	Graphics.fillTriangle	
45	Graphics.getClipHeight	
46	Graphics.getClipWidth	
47	Graphics.getClipX	
48	Graphics.getClipY	
49	Graphics.setClip	
50	Graphics.setColor	
51	Graphics.setFont	
52	Image.createImage	
53	Image.getGraphics	
54	Image.getHeight	
55	Image.getWidth	

(i.e., A->p;q, or A->p). So this might mean that p-q only implements some of the functionality of A on the target platform, and we still count this as a valid mapping.

With that said, let me answer your questions: Let's consider the mapping: "Alert.setString -> Paint.setAlpha; CompundButton.setChecked". In many of our traces, we observed that setting a string first involves setting the attributes of the paint (with is used to draw the text), followed by a call to setText method, which led us to believe that the sequence "Paint.setAlpha; CompundButton.setChecked" could be a likely mapping, if at least in part. The same reasoning applies to mappings of the other two methods in your list.

Technically, you're right that "Alert" related methods would be implemented using methods of the "AlertDialog" class. However, "Alert" is used to display messages, and we considered Android methods/sequences that could also be used to display messages as valid mappings. These methods/sequences may not strictly be considered "AlertDialogs" in the Android sense, but can still be part of rendering a text message on the screen.

7. DISCUSSION AND FUTURE WORK

We next describe the limitations and future work of TMAP, followed by discussions on threats to validity.

A key limitation of the presented work is reliance on the human developer to confirm or refute mappings. In future, we plan to extend the TMAP infrastructure to achieve an end-to-end automation. Particularly, we plan on using the program analysis techniques such as type-analysis proposed in existing approaches [15, 27].

Sometimes the functionality achieved by a method call in a source API, is achieved by a sequence of method calls in the destination API and vice versa. Although TMAP may return individual methods as relevant, TMAP does not provide explicit sequences of method calls as relevant suggestions. In future, we plan to extend the current text mining infrastructure to provide method sequences as relevant suggestions when applicable. In particular, we plan to leverage the NLP techniques such as specification inference [17] to identify method sequences.

From an implementation perspective, TMAP does not take into account the API fields. This limits TMAPs ability in reporting mappings involving API fields. For instance, the functionality achieved by a method call in a source API may be achieved by accessing a API field in destination API. However, this is a limitation of the current implementation and in future iterations of TMAP we plan to include API fields in the indexes.

Finally, TMAP operates under the assumption of the availability of the API documents. Thus TMAP is not applicable in situations where API documents are low quality or are unavailable altogether. In future, we plan to extend TMAP infrastructure to workaround such situations by integrating with existing source code mining based approaches. Specifically we plan to leverage techniques like code summarisation [21] and IR based approaches like Exoa [10].

7.1 Threats to Validity

The primary threat to external validity is the representativeness of our experimental subjects to the real world software. To address this threat we chose real world API pairs: 1) J2ME and Android APIs are two java based platforms to develop mobile applications; 2) Java and C# APIs are the top object-oriented programming language APIs used for generic application development. The threat can be further minimized by evaluating TMAP on more APIs from different domains.

Our chief threat to internal validity is the accuracy of TMAP in identifying API mappings. To minimize this threat we compared the mappings inferred by TMAP with the mappings provided by previous approaches. We thank Gokhale et al. [5] for sharing with us the J2ME and Android API mappings inferred by their approach. We also thank Nguyen et al. [15] for making their mappings publicly available. Furthermore, first two did manually identify some of the mappings that could not be compared to previous work. Thus, human errors may affect our results. To minimized the effect, first two authors randomly sampled and inspected each others work.

8. CONCLUSION

API mapping across different platforms/languages mappings facilitate machine-based migration of an application from one API to another. Thus automated inference of such mappings is highly desirable. In this paper, we presented TMAP: a lightweight text-mining based approach to infer API mappings. TMAP compliments existing mapping inference techniques by leveraging natural language descriptions in API documents instead of relying on existence of manually ported (or at least functionally similar) code across source and target API's. We demonstrated the effectiveness of TMAP by comparing the inferred mappings with state-of-the-art code mining based approaches. Our results indicate that TMAP is effective in inferring API mappings (from over human-annotated API sentences) with the more than XX accuracy.

9. ACKNOWLEDGMENTS

This work is supported by xx. Any opinions expressed in this report are those of the author(s) and do not necessarily reflect the views of XX. Finally, we thank the Realsearch research group for providing helpful feedback on this work.

10. REFERENCES

[1] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[2] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

[3] M. El-Ramly, R. Eltayeb, and H. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proc. IEEE CSA*, pages 1037–1045, 2006.

[4] W. Frakes. Introduction to information storage and retrieval systems. *Space*, 14:10, 1992.

[5] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.

[6] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 475–484. IEEE, 2010.

[7] A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.

[8] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE),*

Table 4: Comparison with Tien Nyugen

S.No.	Source Class	Target Class	TMAP Ranking
1	java.util.concurrent.atomic.AtomicBoolean		
2	java.util.jar.JarInputStream		
3	java.util.jar.Attributes		
4	java.util.jar.Manifest		
5	java.util.LinkedHashMap	System.Collections.Hashtable	
6	java.util.AbstractCollection	System.Collections.ICollection	
7	java.util.Date	System.DateTime	
8	java.util.HashSet	ILOG.J2CsMapping.Collections.HashedSet	
9	java.util.SortedSet	ILOG.J2CsMapping.Collections.SortedSet	
10	java.util.Set	ILOG.J2CsMapping.Collections.ISet	
11	java.util.TreeSet	ILOG.J2CsMapping.Collections.SortedSet	
12	java.util.BitSet	ILOG.J2CsMapping.Collections.BitSet	
13	java.util.Hashtable	System.Collections.Hashtable	
14	java.util.TreeMap	System.Collections.SortedList	
15	java.util.SortedMap	System.Collections.SortedList	
16	java.util.Iterator	ILOG.J2CsMapping.Collections.IIterator	
17	java.util.IdentityHashMap	System.Collections.Hashtable	
18	java.util.Vector	System.Collections.ArrayList	
19	java.util.WeakHashMap	System.Collections.Hashtable	
20	java.util.StringTokenizer	ILOG.J2CsMapping.Util.StringTokenizer	
21	java.util.EventObject	ILOG.J2CsMapping.Util.EventObject	
22	java.util.Locale	System.Globalization.CultureInfo	
23	java.util.Map.Entry	System.Collections.DictionaryEntry	
24	java.util.UUID	System.Guid	
25	java.util.EventListener	ILOG.J2CsMapping.Util.IEventListener	
26	java.util.AbstractList	System.Collections.ArrayList	
27	java.util.TimerTask	ILOG.J2CsMapping.Util.IlTimerTask	
28	java.util.Timer	ILOG.J2CsMapping.Util.IlTimer	
29	java.util.EmptyStackException	System.Exception	
30	java.util.Collections	ILOG.J2CsMapping.Collections.Collections	
31	java.util.ListIterator	ILOG.J2CsMapping.Collections.IListIterator	
32	java.util.HashMap	System.Collections.Hashtable	
33	java.util.Calendar	ILOG.J2CsMapping.Util.Calendar	
34	java.util.ResourceBundle	System.Resources.ResourceManager	
35	java.util.Properties	ILOG.J2CsMapping.Util.Properties	
36	java.util.Stack	System.Collections.Stack	
37	java.util.MissingResourceException	System.Resources.MissingManifestResourceException	
38	java.util.Comparator	System.Collections.IComparer	
39	java.util.ArrayList	System.Collections.ArrayList	
40	java.util.Collection	System.Collections.ICollection	
41	java.util.Map	System.Collections.IDictionary	
42	java.util.ConcurrentModificationException	System.InvalidOperationException	
43	java.util.Arrays	System.Array	
44	java.util.Currency		
45	java.util.NoSuchElementException	System.InvalidOperationException	
46	java.util.Enumeration	ILOG.J2CsMapping.Collections.IIterator	
47	java.util.LinkedList	ILOG.J2CsMapping.Collections.LinkedList	
48	java.util.GregorianCalendar	ILOG.J2CsMapping.Util.GregorianCalendar	
49	java.util.Random	System.Random	
50	java.util.PropertyResourceBundle		
51	java.util.List	System.Collections.IList	
52	java.util.regex.Pattern	ILOG.J2CsMapping.Text.Pattern	
53	java.util.regex.Matcher	ILOG.J2CsMapping.Text.Matcher	

2012 34th International Conference on, pages 837–847. IEEE, 2012.

[9] Java 2 CSharp Translator for Eclipse. `http://sourceforge.net/projects/j2cstranslator/`.

[10] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *AAAI*, 2010.

[11] G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.

[12] Apache Lucene Core. `http://lucene.apache.org/core/`.

[13] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[14] M. Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.

[15] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.

[16] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547. ACM, 2014.

[17] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, 2012.

[18] S. P. Reiss. Semantics-based code search. In *Proc. 31st ICSE*, pages 243–253, 2009.

[19] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Trans. on Software Engineering*, 39(5):613–637, 2013.

[20] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[21] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proc. of 19th ICPC*, pages 71–80, 2011.

[22] C. Treude, M. Robillard, and B. Dagenais. Extracting development tasks to navigate software documentation.

[23] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.

[24] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207âĂŞ–1228, 1988.

[25] W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.

[26] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[27] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *Proc. 24th ASE*, pages 307–318, 2009.

[28] H. Zhou, F. Chen, and H. Yang. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSIC*, pages 225 –234, 2008.