Discovering Information Explaining API Types Using Text Classification

Gayane Petrosyan, Martin P. Robillard, and Renato De Mori
School of Computer Science
McGill University
Montréal, QC, Canada
Email: {gayane.petrosyan@mail,martin@cs, rdemori@cs}.mcgill.ca

***Abstract*—Many software development tasks require developers to quickly learn a subset of an Application Programming Interface (API). API learning resources are crucial for helping developers learn an API, but the knowledge relevant to a particular topic of interest may easily be scattered across different documents, which makes finding the necessary information more challenging. This paper proposes an approach to discovering tutorial sections that explain a given API type. At the core of our approach, we classify fragmented tutorial sections using supervised text classification based on linguistic and structural features. Experiments conducted on five tutorials show that our approach is able to discover sections explaining an API type with precision between 0.69 and 0.87 (depending on the tutorial) when trained and tested on the same tutorial. When trained and tested across tutorials, we obtained a precision between 0.74 and 0.94 and lower recall values.**

I. Introduction

Many software development tasks require developers to quickly learn the subset of an Application Programming Interface (API) related to the task. Information that may be necessary to complete a task can include explanation of relevant domain concepts, functional descriptions of API elements, directives and patterns for using the API correctly, etc. [1]. A natural way to learn about an API is to peruse its learning resources. These typically include reference documentation, but also other types of documentation such as *tutorials*. API tutorials are learning resources that combine both descriptive text and code examples. Tutorials are usually organized as a sequence of API usage examples, where different sections address different programming tasks.

Tutorials are especially well-suited for developers unfamiliar with an API as they often teach readers about popular API features used in a basic context. During real-life development and maintenance tasks, API usage scenarios will not necessarily align with predefined learning tasks, but may instead involve program understanding activities anchored around specific API types (e.g., "How are instances of these types created and assembled?", "What is the behavior that these types provide together and how is it distributed over the types?" [2, p. 441]). In addition to being located in reference documentation explicitly associated with API types, conceptual knowledge about type usage is also scattered across different documents, which makes finding the necessary information more challenging.

A number of issues make searching API tutorials for type usage challenging. First, because API tutorials are task-based, titles usually describe the task (e.g. "Error Handling", "Input and Output") without referencing the types within. Second, when they heavily rely on the general tutorial context, the titles of tutorial sections are often not very informative (e.g., "Next", "Example", "Overview", "General Case").

Another alternative is to use search engines in the hopes of finding type usage information in tutorials or other resources [3]. In the specific case of searching for API type usage information, there are two major causes of inefficiency. First, API types are often mentioned in a tutorial section without being the topic of the section (e.g., "`SessionFactory` is null"), creating a large amount of noise for searchers interested in learning about how to use the type [4]. For example, in the tutorial for the JodaTime API (see Section II), 18 sections have a reference to the type `DateTime`, but only eight of these sections actually explain how to use the type. A second, and related, problem is that search engines only return documents; if the document includes many references to a type of interest, it is necessary to visit every reference and assess each individually for relevance.

In this work we investigate the use of text classification to discover tutorial sections *explaining* how to use a given API type. Among many features we could analyze, we focused our experimentation on the use of textual information. Leveraging textual information in the software engineering domain requires coupling features related to the use of *code words* with features obtained from natural language processing techniques. We consider that a tutorial section *explains* an API type if it would help a reader unfamiliar with the corresponding API to decide when or how to use the API type to complete a programming task. We consider API types (classes and interfaces) as the best level of granularity for finding usage information because a single section of an API tutorial usually describes a solution for a programming task by using a set of methods.

The problem we tackle is not simply one of information retrieval (IR). Even if, in a first step, queries expressed in terms of API element names retrieve tutorial sections that contain the query word, in a second step the retrieved sections have to be classified as relevant (i.e., containing an explanation) or irrelevant. The difficulty of this step is to design features

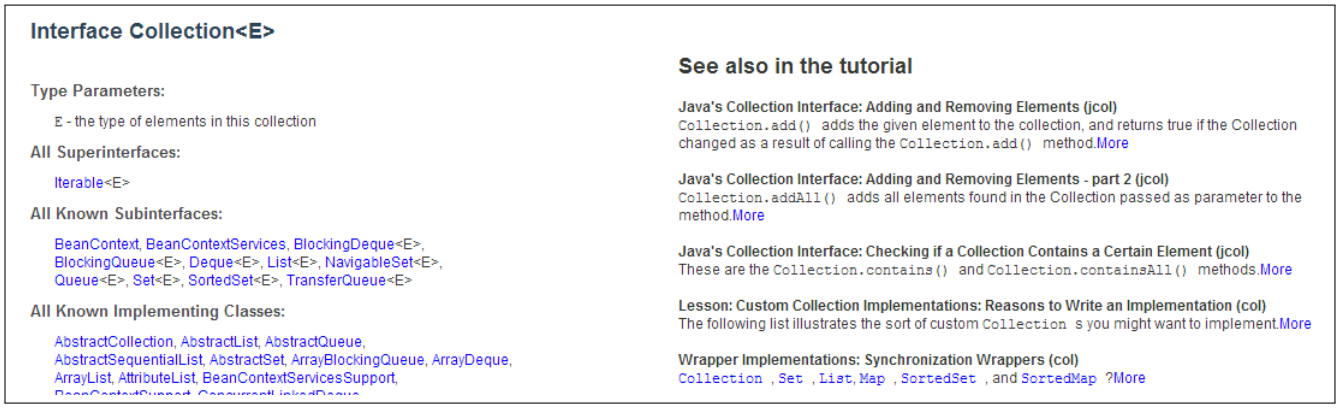

Fig. 1. API Reference documentation annotated with links to tutorial sections.

TABLE I
EXPERIMENTAL CORPUS

API	Tutorial URL (verified 28 August 2014)	Alias	Length (words)
JodaTime API	http://joda-time.sourceforge.net/userguide.html	JodaTime	4659
apache.commons.math library	http://commons.apache.org/proper/commons-math/userguide/	Math Library	28 971
Java Collections Framework	http://docs.oracle.com/javase/tutorial/collections/implementations	Col. Official	23 583
Java Collections Framework	http://tutorials.jenkov.com/java-collections/index.html	Col. Jenkov	12 915
Smack API	http://www.igniterealtime.org/builds/smack/docs/latest/documentation/	Smack	19 075

that are sufficiently refined to robustly capture the essence of *explanation* of API elements.

As part of this work, we developed a coding guide to allow human annotators to reliably label tutorial sections as explaining or not explaining an API type and produced an experimental training and evaluation corpus consisting of five manually-labeled tutorials. This corpus is our **first contribution**, described in Sections II and V.[1] We then studied the structure of these API tutorials and derived a procedure to fragment tutorials into roughly equivalent units of information (our **second contribution**, Section III), and designed a set of features that we hypothesized captured important clues as to what aspects of a tutorial section are indicative of explanations of an API type. The features we designed involve the use of natural language analysis techniques to capture the essence of descriptive language for an API type. These features, and the text transformation operations necessary to extract them, form our **third and most important contribution** and are described in Section IV. Finally, we applied a MaxEnt classifier [5] to our features and corpus to study different aspects of our proposed technique, and compared the results with a baseline obtained though a standard information-retrieval based approach. The results of this multi-faceted empirical evaluation is our **fourth contribution**.

Experimental results on our corpus showed that we can classify tutorial sections as explaining a given API type or not with a precision between 0.74 and 0.94 (depending on the tutorial) when training on four tutorials and testing on a fifth one. In similar conditions, we obtain a recall between 0.48 and 0.76. Finally, we verified that the text classifier-based approach consistently obtained better precision than a standard implementation of vector-space information retrieval that analyzed the similarity between a tutorial section and a type's Javadoc comments.

As a demonstration application of our traceability technique, we inject references to discovered tutorial sections directly into the reference documentation of Java API types, together with a by-line extracted from the corresponding section. Figure 1 illustrates the resulting outcome for interface `Collection` of the Java Core Library. The right side panel has been added by applying our approach. The panel shows links to *specific sections* of two different tutorials that explain how to use the interface. Finding these resources using a search engine requires a non-trivial amount of manual effort.

II. EXPERIMENTAL CORPUS

Studying how to discover tutorial sections relevant to API types requires a corpus of tutorials. We selected five tutorials covering four different Java APIs. We selected the API tutorials based on several criteria. First, since part of the solution will require access to source code, we chose Java APIs that are open source. Our solution requires people to look at the tutorial and manually label different parts of the tutorial as relevant or not to an API type. For this reason we selected APIs with common application domains so that annotators could label the data without special training. Tutorials should also exhibit adequate use of the English language. Finally, we wanted tutorials that are diverse in size, format and origin.

Table I lists the tutorials we selected for this research. The JodaTime tutorial was mainly used for development. The Math Library tutorial was used for testing during the development period, and the other three tutorials were used only for testing purposes.

[1] http://cs.mcgill.ca/~swevo/icse2015/

TABLE II
RECODOC RESULTS FOR THE EXPERIMENTAL CORPUS. # CLTS SHOWS THE NUMBER OF CODE-LIKE TERMS DETECTED IN EACH TUTORIAL. # LINKS SHOWS THE NUMBER OF THESE CLTS THAT WERE LINKED TO A SPECIFIC TYPE OF THE TUTORIAL'S TARGET API.

Tutorial	# CLTs	# links
JodaTime	136	72
Math Library	901	418
Col. Official	711	574
Col. Jenkov	556	409
Smack	574	273

III. PREPROCESSING

Discovering relevant API sections in tutorials requires two main preprocessing steps: 1) finding references to API types in tutorials, and 2) fragmenting the tutorials.

A. Finding References to API Types

Finding API types mentioned in a tutorial is not a trivial task because references to API elements in natural language do not usually include the complete, correct, and unambiguous identifier of the element. For example, the phrase "to create a List..." may refer to `java.util.List`, `java.awt.List`, or any other existing class named `List` in other libraries. To find the exact API types referenced in a tutorial, we used Recodoc [6], which identifies API types in two phases. First, it finds all tokens that *might* be API elements. Such tokens are called *code-like terms* (CLTs). Second, it disambiguates code-like terms by analyzing the surrounding text and explicitly linking them to the exact API type or type member they refer to, if applicable. This problem is especially challenging when resolving method names, such as `add`, which are heavily overloaded. Because our strategy involves replacing references to type members with references to their declaring type, we must also disambiguate members. For example, in a phrase such as "use the `add` method to...", we would need to detect that, for example, the text references `java.util.List.add(...)` to know that the text indirectly contains a reference to `java.util.List`. Recodoc was showed to link API elements from API tutorials and mailing lists with 96% precision and 96% recall [6].

We applied Recodoc separately to each tutorial to link code-like terms in the text to the corresponding API elements (e.g., to link terms in the Math Library tutorial to elements in the org.apache.commons.math packages). Table II reports on the outcome of the process. For example, Recodoc found 901 code-like terms in the Math Library tutorial, out of which 418 were linked to the apache.commons.math API. Out of these 418 cases, 35 code-like terms could not be automatically disambiguated by Recodoc. We manually reviewed and disambiguated these cases based on the Recodoc recommendations.

B. Fragmenting Tutorials

To help users find information about API types efficiently, it is necessary to point them to sufficiently short *fragments* containing the relevant information (as opposed to referring to the tutorial in general, or to very long sections). A basic strategy could be to simply consider terminal sections (sections without subsections) to be these fragments. Unfortunately, this idea does not generalize well because different tutorials have different section lengths, and some sections can be very long.

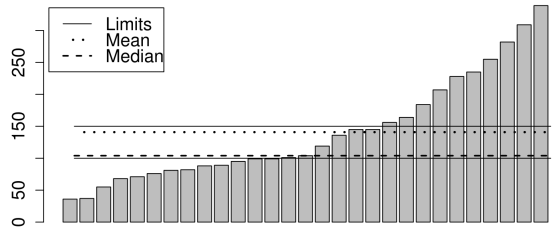

Fig. 2. Section lengths (in number of words) for the JodaTime tutorial

We addressed this challenge by designing a procedure that attempts to split long tutorial sections so that each resulting fragment is between a target minimum and maximum length, while preserving the cohesiveness of the content. For determining the target minimum and maximum lengths for fragments, we used the JodaTime tutorial as a benchmark because, in this particular case, the length of sections was small and relatively homogeneous.

Figure 2 shows section lengths for the JodaTime tutorial. Each bar corresponds to a section and the height corresponds to the number of words in a section. As can be seen from the figure, the average length is between 100 to 150 words. Based on this observation, we selected 100 words as a (target) min length and 150 as a (target) max length for tutorial fragments. In the following, the parameters min and max should be understood as "target min" and "target max".

To fragment a tutorial, we split the tutorial into structurally or conceptually atomic units, and progressively reassemble these units until the target length is obtained. The main steps of our fragmentation algorithm are presented below:

1) Split the tutorial based on the table of contents or HTML header tags. Exclude the section titles from further processing.
2) For all sections extracted in Step 1, recursively split all HTML elements if they are longer than $max - min$ (e.g. longer than 50 words). We try to split HTML elements so they are smaller than $max - min$ so that it will later be easier to keep fragments within the range $[min, max]$. We do not split an HTML element even if it is longer than $max - min$ if it does not contain children, or if it is an HTML `p`, `ul`, `table`, `dd`, or `dt` tag.
3) Merge `dt` elements with the following `dd` elements, and merge `ul` elements with the previous paragraph (for continuity).
4) Iteratively merge sibling HTML elements until the length of the resulting element exceeds the min length, and add any subsequent sibling with length 0. Here adding even a single element might increase the length of a fragment to exceed the max length. However, as

TABLE III
FRAGMENTATION RESULTS

Tutorial	# Sec.	# Seg.	Length (words) Mean	Median	Linked	Pairs
JodaTime	33	33	140	104	29	72
Math Library	77	158	203	201	102	251
Col. Official	56	73	172	163	57	233
Col. Jenkov	70	79	141	132	69	150
Smack	60	65	229	212	46	86

elements were split to be smaller than $max-min$ except for a few exceptions, the section length will generally be shorter than the max length.

5) If a section has been fragmented in Steps 2, 3, or 4 then the title of the subsection is formed by concatenating the title of the closest enclosing section with a sequential index.

We fragmented all tutorials except the JodaTime tutorial using the algorithm described above (JodaTime was our baseline for fragment length boundaries); Table III presents the results. For example, the Math API tutorial was originally divided into 77 sections. Those 77 sections were subsequently fragmented into 158 sections with an average length of 203 words. This is longer than the target length (150 words) but our approach is heuristic and prioritizes the structural coherence of a section fragment above its length. For this tutorial, Recodoc identified references to elements in the Math API in 102 out of the 158 sections fragments. Overall, there were 251 ⟨API type, section fragment⟩ pairs. For all tutorials, we annotated all ⟨API type, section fragment⟩ pairs except for the Math Library, where time constraints limited us to 102 of the 251 available pairs.

In this paper, we consistently use the term "section" for readability. However, the term should be understood as "section fragment" whenever the results of the approach are concerned.

IV. RELEVANCE CLASSIFICATION

In this section we describe the text transformation operations we implemented to support the extraction of classification features (Section IV-A) and present our design of classifier features (Section IV-B).

A. Text Transformation Operations

Most text analysis approaches require a pipeline of processing steps to extract features from text. Here we summarize the main steps of our pipeline, putting the emphasis on the problems and solutions specific to the task of analyzing API tutorials. A complete description of the related algorithms can be found in a separate report [7].

To extract sub-paragraph features, we use the Stanford Parser [8] to **split paragraphs into sentences**. This parser relies on the detection of sentence-ending characters (i.e., ., !, or ?); however, in HTML files, many logical text units are not terminated by a symbol (e.g., list items), or are split by structural elements (e.g., code snippets). To overcome these problems, we **collapse code snippets** to treat them as words, and then **automatically inject additional punctuation**.

We collapse all code snippets into a special keyword that holds a unique identifier referring to the original snippet. When we later add punctuation, the collapsing function checks if the code snippet is followed by a new sentence. We say that a code snippet is followed by a new sentence if the next symbol is in uppercase or is an opening HTML tag. If so, a period is added at the end of the code snippet marking the end of the sentence. If the preceding sentence did not end before the code snippet (e.g., there was no period preceding the code snippet), then the code snippet is folded into that sentence. For example, Figure 3 shows a section of the JodaTime tutorial in HTML before collapsing code snippets.

```
1 ... In datetime maths you could say: </p>
2 <div class="source">
3       <pre>   instant + duration = instant </pre>
4 </div>
5 <p>Currently, there is only one implementation of....
```

Fig. 3. Tutorial text before code snippet collapse

When the code collapsing algorithm is applied, the HTML markup is removed and this excerpt gets transformed into the result presented in Figure 4. Note that in the modified version, a period is added after the code snippet, to indicate the end of the current sentence and the beginning of the new one.

```
1 ... In datetime maths you could say: CODEID=7.
2 Currently, there is only one implementation of...
```

Fig. 4. Tutorial text after code snippet collapse

Collapsing code snippets not only makes sentence detection more accurate, it also creates a possibility to exploit the relationships between sentence words and code snippets. After being collapsed, a code snippet becomes just another word in the sentence and as any other word in the sentence, it has relationships with sentence words. As API types are also part of the sentence, we can identify the relationship between an API type and the code snippet. If a relationship does not exist, at least we can identify whether the API type appears in the same sentence with a code snippet. This is important information because usually important API types of the code snippet are mentioned before it.

When extracting the text from HTML, if the content of an element such as `h1`, `h2`, `dt`, `dd`, `p`, or `tr` ends without punctuation, then we add a period at the end. In the case of items such as `li` or `td`, we add a comma for every item except the last one, after which we add a period.

Most of the textual features we extract require the **tagging of words with their part-of-speech (POS) tags** (e.g., adjective vs. noun). For this purpose, we use the Stanford Parser. Unfortunately, this off-the-shelf tool was trained on a corpus of Wall Street Journal articles and makes many tagging errors when applied to API documents.

To overcome the mistagging of technical concepts, we re-implemented a **multi-word term detection algorithm** [9]. This method outputs multi-word phrases with corresponding confidence scores. With the use of this algorithm, we extracted more than 30 000 multi-word phrases from the official Java

Tutorials[2] and selected all the phrases with a confidence score above an arbitrarily-determined threshold of 10. This procedure resulted in 1037 multi-word concepts associated with API terminology.

We produced the list automatically for reproducibility, which implies that it contains some noise: some of the selected multi-word concepts are not real concepts, but reoccurring word combinations (e.g. new connection, current JDK, etc.). We did not remove such cases to prevent subjective bias, but we observed that spurious multi-word phrases do not have a negative effect on the classification procedure. In any case, for some phrases it is questionable whether they represent a concept or not (e.g., "last element" or "default layout").

After obtaining the list of concepts, occurrences of these concepts were concatenated so that from here on they would be treated as one token (e.g., "defaultLayout"), and we forced the POS tagger to tag the phrases as a noun.

During multi-word concept extraction we observed that usually an API type is used in the sentence as part of a noun phrase. For example, in the sentence from the JodaTime tutorial: "Within Joda-Time an instant is represented by the ReadableInstant interface", `ReadableInstant` is an API type and the noun phrase it is part of is "the ReadableInstant interface". According to the Stanford Parser, "interface" is the main noun and `ReadableInstant` is a descriptive word for it. This means that in the above example, "interface" is the object which "is represented by ...". In practice, we found it necessary to map relationships to API types and not their qualifiers. In this example, we want to associate instant with `ReadableInstant`, as opposed to "interface". We thus **replace all noun phrases which contain an API type with the API type itself**. For example, in our previous case the sentence would become (words in brackets removed): "Within Joda-Time an instant is represented by [the] `ReadableInstant` [interface]".

B. Feature Design

In designing features, we took into account both linguistic and structural properties of the text. We can distinguish between five groups of features. The first group of features comprises general features expressed as a real value (as opposed to boolean features). Tutorial-, section-, and sentence-level features are first-order logical predicates whose truth is evaluated by taking into account information about the entire tutorial, a single section, or a single sentence, respectively. Finally, dependency-based features are real-valued features that take into account dependencies between words in the text.

Table IV summarizes our entire set of features. Below, we clarify only the features that are not self-evident from the table. A complete description of each feature can be found in a separate report [7]. All features describe a tuple $\langle s, t\rangle$ where s is a section (fragment) and t is an API type mentioned in the section.

[2]http://docs.oracle.com/javase/tutorial/

TABLE IV
SUMMARY OF FEATURES

Feature	Description
	Real-valued features
freq	How strongly an API type is associated to a tutorial section.
wordNum	Frequency with which an API type or part of it is mentioned as simple words, not as code words
substituteNum	Frequency with which an API type is a substitute for its methods and fields
	Tutorial-level features
inParentTitle	TRUE for $\langle s, t\rangle$ if s is a subsection of another section S and t is present in the title of S
isOnlyOne	TRUE for $\langle s, t\rangle$ if s is the only section of the tutorial which mentions t
	Section-level features
inCode	TRUE for $\langle s, t\rangle$ if s contains a code snippet and the code snippet contains t
notInCode	TRUE for $\langle s, t\rangle$ if s contains a code snippet and the code snippet does not contain t
moreThanOnce	TRUE for $\langle s, t\rangle$ if t is mentioned in s as a code term more than once
once	TRUE for $\langle s, t\rangle$ if t is mentioned in s as a code term only once
inTitle	TRUE for $\langle s, t\rangle$ if t is present in the title of s
inFirstSent	TRUE for $\langle s, t\rangle$ if the first sentence of s contains t as a simple word or as a code word
	Sentence-level features
isExample	TRUE for $\langle s, t\rangle$ if any sentence of s mentions t as an example
isInParentheses	TRUE for $\langle s, t\rangle$ if any sentence of s mentions t in a phrase surrounded with parentheses
withCode	TRUE for $\langle s, t\rangle$ if any sentence of s mentioning t contains a code snippet
importantSentence	TRUE for $\langle s, t\rangle$ if any sentence of s mentioning t is considers as "`important`"
modal	TRUE for $\langle s, t\rangle$ if in any sentence of s mentioning t, a verb applied to t has modal verb
negation	TRUE for $\langle s, t\rangle$ if in any sentence of s mentioning t, negation is applied to the verb connected to t
inEnum	TRUE for $\langle s, t\rangle$ if in any sentence of s, t is enumerated with more than one other API types, or connected to other API type with "or".
	Dependency-based features
depScore	A relevance score based on the dependencies of t.
relScore	A relevance score based on the relation types of dependencies of t.

freq: We use the following formula to calculate *freq*:

$$\text{freq}(s,t) = \frac{tf_{s,t}}{df_t} \quad (1)$$

where $tf_{s,t}$ is the number of occurrences of the API type t in the section s and df_t is the number of sections containing the API type t.

wordNum: Some of the API types or part of API type names can be used in the text of the section as simple words and not as a code term. For calculating the *wordNum* feature, we first extract the lemmas of all words in the section using the Stanford toolkit [8]. Then we calculate a weight:

$$w(s,t) = \sum_{lw \in LW(s)} AreAlike(t, lw) \quad (2)$$

where $LW(s)$ is the set of lemmas of all words in the section s. $AreAlike$ is a function which returns 1 if lw matches fully the name of the API type t and 0.5 if lw matches t partially. For detecting a partial match, t is split by CamelCase and each part is compared with lw.

As Nigam et al [5] observed, unscaled frequencies of the terms negatively affected the accuracy of the MaxEnt classifier. During the initial experiments we observed the same behaviour, and for that reason we decided to normalize the *wordNum* feature. As there is no theoretical upper limit for w, we used an approximate upper limit. We chose 5 to be the upper limit and calculated the final value for the feature as follows:

$$wordNum(s,t) = \begin{cases} 1, & \text{if } w(s,t) \geq 5 \\ \frac{w(s,t)}{5}, & \text{Otherwise} \end{cases} \quad (3)$$

substituteNum: This feature is the ratio of references to a type t where t is a substitute for one of its declared elements, over all mentions of t including substitutions.

isExample: This feature is a predicate evaluated to TRUE if the name of the API type is preceded, in the same sentence, by one of the following phrases: "such as", "for example", or "for instance".

isInParentheses: When an API type is mentioned in parentheses, it also can be an example and carries a secondary function for the text. The only exception we considered is when the text in parentheses starts with the word "note", in which case it usually carries important information.

withCode: As mentioned in Section IV-A, after collapsing code snippets, they can become part of the sentences. This usually happens when an API type is one of the main components of the code snippet. Based on this observation, we introduced the *withCode* feature, which is *TRUE* if an API type appears in the same sentence as the code snippet.

importantSentence: A sentence is considered important if it is in the imperative mood or if it starts with instructive words (to, when, by, try, note, in order). We consider a sentence to be in the imperative mood if any verb of the sentence does not have a subject or its subject is "you".

Dependency-based Features: These features are based on the typed dependencies of the code-like term. Typed dependencies are the triples consisting of (governor-relation-dependent words), as (cat-subject-eat), where *relation* is the linguistic relation type connecting words such as subject, object, adjective, etc. Certain dependencies might be indicative of the relevance of a section to a type or not. For example, when a type is the subject of a verb, then it is likely that it is the main focus of the sentence. We decided to exploit this information for improving the classification results. API tutorials are particularly suitable for exploiting the regularities of text because they contain repetitive grammatical structures. For example, three sections in the JodaTime tutorial start with the phrase ("A <concept> in Joda Time represents...").

To leverage typed dependencies for our classification, first we had to identify positive and negative dependencies and create a database of such dependencies with corresponding weights. We used the Java tutorials to create the database. We extracted 1785 dependencies in which either the governor or the dependent was a code-like term (CLT). Afterward, the first author manually annotated each dependency as positive, negative or not useful. She labeled a typed dependency as positive or negative if it contributed to the relevance or non-relevance of the section for explaining the code-like term. From 1785 extracted typed dependencies, she classified 725 as positive, 445 as negative, and 615 as not useful. Those 615 triples classified as not useful were ignored and were not used in the following steps. The useful instances overall mapped to 246 distinct typed dependencies and 39 distinct relations.

TABLE V
A FEW EXAMPLES OF DEPENDENCIES.

Gov	Rel	Dep	#Total	#Pos	Z-score	Norm
use	dobjMDneg[3]	clt	5	0	-2.85	0.32
clt	nsubj[4]	specify	11	11	2.6	0.93
catch	dobjMD[5]	clt	2	0	-1.81	0.44
define	prepIN[6]	clt	12	0	-4.42	0.19
use	dobj[7]	clt	88	61	1.42	0.79

For each type of dependency, we computed a weight based on the annotation results. The intuition for the weight calculation was that the more often a dependency appears as positive, the larger the weight should be. However, both the ratio of positive instances and the absolute popularity of the dependency should also have a role. Taking this intuition into account, we use the *Z-score* to calculate the weight for each dependency.

Table V provides a sample of dependencies and relevant values indicating how strongly they show evidence for the relevance of an API element. As a justification for using the Z-score, the third and fourth lines contain dependencies in which all instances were marked as negative, so their percentage of negative cases is identical; however the Z-score takes into

[3]object of a verb, where the verb is preceded by a modal verb and the negation word "not"

[4]subject of a verb

[5]object of a verb, where verb is preceded by a modal verb

[6]prepositional phrase with preposition IN

[7]object of a verb

account the number of occurrences and thus distinguishes these two cases, giving more weight to the third example, which is six times more frequent than the fourth one.

To use the Z-score as a feature for classification, we calculated the normalized version of all scores. The total score for the section was calculated by taking all dependencies which contain an API type and averaging their score.

V. Annotating The Experimental Corpus

The investigation of a supervised classification approach for discovering tutorial sections explaining API types requires the creation of a set of labeled data items to train the classifier. In this research, a data item is a tuple $\langle s, t \rangle$ consisting of a section s and an API type t. Annotating (or labeling) the tutorials for this research requires a subjective judgment of whether the section s explains type t or not. To ensure a high level of rigour in our annotation process, we constructed a detailed annotation guide [7, Appendix], developed a dedicated annotation tool, and asked two human annotators to label each data item $\langle s, t \rangle$. In case of disagreement, the final label for each data item was obtained by consensus of the two annotators.

The annotation tool we developed displays each section with HTML formatting, sequentially highlights all API types found in the section, and provides functionality to save and resume the current state of the annotation task. It displays the current progress and provides access to a concise form of the annotation guide.

The annotation process for each portion of data is an *annotation session*. During an annotation session, two participants (the annotators) independently annotate the same data and then meet to reconcile disagreements. At the beginning of each annotation session, the annotators were introduced to the annotation guide and annotation tool. Each session began with 10 warm-up examples so that the annotators could get used to the task. The amount of work was divided so that it would not take more than one hour to complete an annotation task. Shortly after completing the task, the two annotators discussed their disagreements to reach a common decision for each data item. The kappa inter-annotator agreement score for each session is presented in Table VI. The annotators were two of the authors, a post-doctoral fellow, a Ph.D. student, a Master's student and an intern of the same lab, and two Master's students from two different groups at McGill University. As the table shows, some of the sessions led to poor initial agreement. However, we accepted the results because post-session discussions revealed that most of the disagreements were due to the different levels of conservatism from the annotators rather than sloppy annotation work. A particular case was the first annotation session for the official Collections tutorial ($\kappa = 0.29$). Given the utility nature of the related API, it is far from obvious to estimate whether a section explains a type for certain scenarios. This observation can be confirmed by looking at the tutorial.

Table VII summarizes the annotation results. The second column contains the total number of tutorial sections and API type pairs preceded by the number of relevant pairs. The third column contains the number of distinct API types preceded by the number of relevant distinct API types. For example, Col. Official, annotated in two sessions, has $107 + 113 = 220$ pairs from which only 56 (25%) were annotated as explaining the type in question. However, the third column shows that of the 58 distinct API types mentioned in the tutorial, 31 had at least one matching explanatory section. This means that the tutorial covered 53% of the mentioned API types with information judged to be explaining the type.

TABLE VI
Kappa Agreement Scores

Tutorial	**# pairs**	***Kappa***
JodaTime	68	0.31
Math library	98	0.51
Col. Official	107	0.29
	113	0.61
Col. Jenkov	150	0.57
Smack	86	0.63

TABLE VII
Tutorial Statistics

Tutorial	(Section,Type)-(Relevant/Total)	Unique API Types (Relevant/Total)
JodaTime	30/68	21/36
Math Library	54/98	45/74
Col. Official	56/220	31/58
Col. Jenkov	42/150	21/28
Smack	56/86	29/40

Finally, we note that experiments conducted with this corpus have the following **threats to validity:** different people will differ in their judgment of whether a section explains an API type, and tutorials can vary greatly in length, style, and quality. Different results can be expected for tutorials with characteristics markedly different from those of the tutorials in our sample.

VI. Classification Experiments

A. Basic Classification Results

For evaluating the performance of the system for each tutorial, we performed leave-one-out cross validation (LOOCV). For each section-type pair, we trained the classifier on the rest of the corresponding tutorial and tested on the held-out pair, using the oracle to determine if our classification yielded a true positive (TP), true negative (TN), false positive (FP), or false negative (FN). We then computed the aggregated precision (P=#TP/(#TP+#FP), recall (R=#TP/(#TP + #FN), and F1 measure (2PR/(P+R)) for all the section-type pairs in a given tutorial. Table VIII shows the results.

We investigated the classification details for the Math Library and Col. Official tutorials, which were the hardest ones to classify. For the Math Library tutorial, out of 31 incorrect classifications (false positives + false negatives), 18 cases had very few features present. In the case of Col. Official, the cases with few applicable features were only 5 out of 35 incorrectly classified cases. However, the Col. Official differs from other

TABLE VIII
LEAVE-ONE-OUT CROSS-VALIDATION RESULTS

Tutorial	Precision	Recall	F1
JodaTime	0.81	0.73	0.77
Math Library	0.69	0.74	0.71
Col. Official	0.71	0.62	0.67
Col. Jenkov	0.84	0.76	0.80
Smack	0.87	0.80	0.83

tutorials because of its large number of negative cases. In this case, more positive evidence is needed to classify the data into the positive category, which explains the low recall value for this tutorial. Math Library and Col. Official are also the tutorials with the biggest average section lengths, which creates some challenges related to sentence-level features. For example, because more sentence-level features will fire in a larger section fragment, it may become more important to devise advanced strategies to combine them.

We also analyzed the results per API types because this information gives us a notion of the *coverage* of a system using our classification. In other words, for how many different API types can we discover relevant information? Table IX presents coverage information per API type. The second column (# Types) shows the number of distinct API types mentioned in each tutorial. The third column (Min One) shows the distinct number of types that had at least one relevant section according to the annotation. The fourth column (Rec) indicates the number of types for which at least one section was annotated as relevant and at least one relevant section was recommended. The next column (FP) shows the distinct number of API types which did not have any relevant sections but nevertheless had (falsely) recommended sections. Overall 73% of API types that had at least one relevant section would have at least one valid link to a tutorial section. The last two columns show information about the average number of linked sections per type including or excluding API types with no linked section.

TABLE IX
COVERAGE PROVIDED BY CLASSIFICATION PER API ELEMENT

Tutorial	# Types	Min One	Rec	FP	Avg(+0s)	Avg
Joda	36	21	17	3	0.78	1.47
Math	74	45	30	17	0.78	1.23
Col. Official	58	31	22	7	0.79	1.70
Col. Jenkov	28	21	16	5	1.36	1.73
Smack	40	29	22	5	1.55	1.94

B. Results for Different Sets of Features

Features for the classifier can be conceptually divided into groups as shown in Table IV. We explored the effect of each group of features on the classification results. For this purpose we considered dependency and relation-based features to be in separate groups.

Table X presents the results of leave-one-out cross-validation for each tutorial by different sets of features. The first six rows are the classification results for feature groups considered individually. Afterwards, features are added group by group, in decreasing order of detail.

According to these results, one of the weakest groups of features is the group of tutorial-level features. This can be simply explained by the small number of features in the group. However, real-valued features combined with tutorial level features already shows improvement for the majority of tutorials.

Col. Official is the most difficult to classify also for separate groups of features. The same relation can be observed for the Smack tutorial, which has the overall highest performance, and accordingly, separate groups of features have the highest performance among other tutorials. In other words, there is no dominant or weak feature group. If a tutorial is well-served by our choice of features then all features work well, and vice versa. It is also worth mentioning that adding features based on their level of detail usually improves the performance. The only level of features which causes problems, for example for JodaTime, is sentence-level features.

Another interesting observation is the effect of the dependency and relation features. Including dependency features, in the case of JodaTime and Math Library, improves recall and pulls down the precision. In the case of the other three tutorials, both scores go down. In contrast, the addition of the relation feature always improves or does not change precision and recall. Surprisingly, the combination of these two always improves or does not change the performance. For example, for Col. Official the addition of the dependency feature brings down both precision and recall. However, the dependency feature combined with the relation feature improves performance, compared to using the feature sets without dependency.

C. Generalizing Across Tutorials

We investigated to what degree a classifier would generalize to an unseen tutorial. For this purpose we trained a classifier on four tutorials and tested on the fifth one. Table XI presents the results, where each line corresponds to the case in which a tutorial was used as testing and the other tutorials were used for training. As usual, we calculated precision, recall and F1 score using the accumulated false positives, true positives, false negatives, and true negatives for all API tutorial section-type pairs from a given tutorial.

TABLE XI
CROSS TUTORIAL RESULTS

Test Tutorial	Precision	Recall	F1
JodaTime	0.94	0.57	0.71
Math Library	0.87	0.48	0.62
Col. Official	0.74	0.76	0.75
Col. Jenkov	0.80	0.68	0.73
Smack	0.87	0.64	0.74

Here recall is overall lower compared with the LOOCV results. However, for the JodaTime, Math library, and Col. Official tutorials the precision actually improves and for Smack it stays constant. One of the main reasons for the observed

TABLE X
CLASSIFICATION RESULTS FOR DIFFERENT SET OF FEATURES. RELATION-BASED (R), DEPENDENCY-BASED (D), SENTENCE-LEVEL (ST), SECTION-LEVEL (SC), TUTORIAL-LEVEL (T), AND REAL-VALUED (RV).

Tutorial	JodaTime			Math Library			Col. Official			Col. Jenkov			Smack		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
R	0.70	0.70	**0.70**	0.65	0.78	**0.71**	0.72	0.32	**0.44**	0.65	0.48	**0.55**	0.70	0.96	**0.81**
D	0.67	0.67	**0.67**	0.67	0.69	**0.68**	0.40	0.07	**0.12**	0.68	0.40	**0.51**	0.70	0.96	**0.81**
ST	0.76	0.63	**0.69**	0.76	0.54	**0.63**	0.55	0.20	**0.29**	0.61	0.83	**0.71**	0.69	0.91	**0.78**
SC	0.66	0.63	**0.64**	0.65	0.81	**0.72**	0.62	0.43	**0.51**	0.56	0.45	**0.50**	0.91	0.77	**0.83**
T	0.72	0.43	**0.54**	0.61	0.89	**0.72**	0.33	0.04	**0.06**	0.62	0.31	**0.41**	0.70	0.96	**0.81**
RV	0.77	0.77	**0.77**	0.58	0.78	**0.67**	0.50	0.21	**0.30**	0.79	0.52	**0.63**	0.68	0.91	**0.78**
RV,T	0.84	0.87	**0.85**	0.56	0.65	**0.60**	0.62	0.23	**0.34**	0.84	0.62	**0.71**	0.73	0.73	**0.73**
RV,T,SC	0.82	0.77	**0.79**	0.62	0.74	**0.68**	0.66	0.48	**0.56**	0.83	0.81	**0.82**	0.85	0.79	**0.81**
RV,T,SC,ST	0.81	0.70	**0.75**	0.70	0.69	**0.69**	0.70	0.55	**0.62**	0.85	0.79	**0.81**	0.87	0.80	**0.83**
RV,T,SC,ST,D	0.81	0.70	**0.75**	0.68	0.74	**0.71**	0.68	0.54	**0.60**	0.82	0.79	**0.80**	0.86	0.79	**0.82**
RV,T,SC,ST,R	0.81	0.73	**0.77**	0.73	0.76	**0.75**	0.68	0.61	**0.64**	0.84	0.76	**0.80**	0.85	0.79	**0.81**
All	0.81	0.73	**0.77**	0.69	0.74	**0.71**	0.71	0.62	**0.67**	0.84	0.76	**0.80**	0.87	0.80	**0.83**

changes was the difference in positive and negative examples ratios between training and testing sets. For example, for JodaTime the number of positive examples is almost the same as the number of negative examples, but in the training set for JodaTime the percentage of positive examples is around 35%. As a result the negative class has a slightly higher weight. Therefore, fewer sections are classified as relevant, which lowers the recall, but improves the precision. One possible reason for the observed changes can also be the different training set sizes for each of the tutorial, but this hypothesis needs further study.

D. Comparison with Information Retrieval

We use a MaxEnt classifier for determining relevant sections of an API tutorial for a particular API type. A MaxEnt classifier requires training data and our assumption was that the cost of training the classifier could be justified by better precision over a classical unsupervised approach. As a basic test of this assumption, we conducted an experiment to see whether it was possible to achieve similar results by using information retrieval (IR). Intuitively, the more common words exists between a section and an API type description (e.g. JavaDoc), the more similar the section is to the API type description, and thus, the more focused is the section on the API type.

For this experiment we considered API tutorial sections and complete Javadoc blocks as documents in a corpus. For each API type-section pair, we calculated the similarity between the section text (lemmas excluding code snippets) and the API type documentation using the cosine similarity metric with tf-idf term weighting [10, p.111].

We consider an API type relevant if the similarity value is higher than a certain threshold. The algorithm is based on threshold and is not top-n-based because the number of relevant API types per section has a large variance and can even be 0. For each tutorial, we calculate a threshold according to the following procedure: For each API type t_i, the top n_i most similar sections are retrieved, where n_i is the number of relevant sections for t_i according to the annotation results. The lowest similarity value of the retrieved sections for all API types is averaged for a tutorial and considered the threshold. Based on the calculated threshold, sections were selected for each API type. Here again we calculated precision, recall and F1 score as described in Section VI-A. The results are presented in Table XII and contrasted with the results obtained with text classification across tutorials (see Section VI-C).

TABLE XII
MAXENT VS. COSSIM

Tutorial	MaxEnt			CosSim		
	P	R	F1	P	R	F1
JodaTime	0.94	0.57	0.71	0.73	0.73	0.73
Math	0.87	0.48	0.62	0.67	0.65	0.66
Col. Official	0.74	0.76	0.75	0.30	0.94	0.45
Col. Jenkov	0.80	0.68	0.73	0.33	0.88	0.48
Smack	0.87	0.64	0.74	0.74	0.52	0.61

As can be seen, the cosine similarity technique is not as precise, especially in the cases where there are many irrelevant examples, such as in the Col. Official and Col. Jenkov tutorials. However, generally the performance of cosine similarity approaches that of MaxEnt if we consider recall an important factor. Its major limitation is however that it relies on the presence of high-quality Javadocs to serve as an anchor to discover relevant sections. In contrast, its advantage is that it does not require training. In the future, it may be worth investigating the potential of this approach as part of a compound solution.

VII. RELATED WORK

This work is multi-disciplinary as it relies on results in natural language processing (NLP), information retrieval (IR), text classification, and software archive mining. In the space available, we discuss the main influences on the work and closely related projects on traceability and text classification in software engineering.

Text Processing

By trying to determine if a section explains an API type, we are in fact asking the question of whether a given API type is an important topic for the section. In this way the problem we

address is a bit similar to the task of text summarization. In particular, *extractive* text summarization techniques attempt to form a summary by extracting important sentences or phrases from the text [11]. Positional features (e.g. position of a sentence or paragraph) have long been an important feature for selecting part of the text to summarize [10], [11]. Similar to positional features in text summarization, we used the location of the API types as features for our classifier. Although text summarization has not yet been employed for traceability purposes, it has been used to support a variety of software engineering tasks, such as summarizing bug reports [12].

In the last decade numerous applications of text processing have been proposed in software engineering. We illustrate the richness of the field by discussing projects representative of different sub-problem spaces.

For example, NLP techniques have been used to support the discovery of new requirements by mining the text of online forums [13], or to support quality control by identifying missing objects and actions in requirements documents [14]. NLP also supports techniques to discover undocumented software specifications in natural language artifacts such as documentation [15]–[18] or comments in source files [19]. Text-processing techniques such as topic modeling are also useful for helping developers sift through software development information, for example by semi-automatically building summaries of forum posts in the form of "Frequently Asked Question" (FAQ) documents [20]. Finally, text classification has also been used in a software engineering context, for example to automatically tag forum posts [21]. This last work used a Naive Bayes classifier applied to word features for multinomial classification. Our work advances the state of the art in classification of software documents by investigating more semantic features and the relations between code and normal language words in sentences.

The potential of text processing techniques to support software engineering activities is now well recognized, and the growing body of work in this area means that many lessons learned in one project can carry over other projects. In our work we have employed and extended many useful domain adaptation ideas from previous work, for the example to assist with part-of-speech tagging [22].

Software Traceability and Feature Location

Our new approach for discovering relevant section integrates an existing body of work on *software traceability*. The problem of software traceability is to link conceptually related software elements (e.g., requirements with source code) when such links are not explicitly provided.

Many researchers have experimented with the use of information retrieval techniques to recover the links between source code elements and free-text documents. Early attempts include the work of Antoniol et al., who applied two information retrieval techniques, the probabilistic model and the vector space model, to find the page in a reference manual that were related to a class in a target system [23], and the work of Marcus and Maletic, who experimented with latent semantic indexing (LSI) for similar purposes [24]. That approach was later refined by Poshyvanyk and Marcus, who added Formal Concept Analysis to cluster the results obtained through LSI [25]. More generally, finding tutorial sections shares some techniques with work on the problem of locating features in source code, a challenge that has been investigated by many researchers [26].

Another common problem in software traceability is linking source code elements with mailing list message or bug reports that refer to them (e.g., [27], [28]). Bacchelli et al. compared both the vector space- and LSI-based information retrieval techniques with a simple pattern-matching approach [29]. They concluded that, for the purpose of linking emails with type-level source code entities, the lightweight approach was consistently superior. The problem with LSI and its successors is that indexing is based on latent semantic features that are automatically learned without any control on their meaning and potential for expressing relevance. Recently, Tathagata et al. showed that enhancing source code document models with related documentation can improve the precision of requirements-to-source traceability [30].

RecoDoc [6], ACE [4], and Baker [31] are three recent traceability techniques designed to precisely resolve code-like terms found in various types of natural language documents into the specific API elements they refer to. In our approach we used RecoDoc as a means to obtain the list of API elements mentioned in a section.

The various API linking techniques mentioned above are discrete in the sense that they link a section to an element if any mention of the element is present in the document. In contrast, with this work we are tackling the more approximate question of determining, when an element is present, whether it is really the focus of the document. This problem had been attempted by Rigby and Robillard for forum posts with only limited success [4]. This paper shows that much better results can be obtained for tutorials with a richer set of features and the use of text classification technology.

VIII. Conclusion

We proposed a technique for discovering API tutorial sections that help explain API types. Experiments conducted on five tutorials for Java APIs showed that it is possible to get meaningful results with just a small amount of training data: when a classifier was trained on four out of five experimental tutorials and tested on the fifth, precision varied between 0.74 and 0.94, and recall between 0.48 and 0.76. The results show good generalization and encourage further investigation of text classification for traceability between software elements and various types of software documents.

Acknowledgment

The authors are grateful to the annotators, and to Bradley Cossette and Annie Ying for comments on the paper. R. De Mori is co-affiliated with the University of Avignon, France. This work was funded by NSERC.

REFERENCES

[1] W. Maalej and M. P. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, September 2013.

[2] J. Sillito, G. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, July 2008.

[3] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[4] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*, 2013, pp. 832–841.

[5] K. Nigam, J. Lafferty, and A. McCallum, "Using maximum entropy for text classification," in *IJCAI-99 workshop on machine learning for information filtering*, vol. 1, 1999, pp. 61–67.

[6] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*, 2012, pp. 47–57.

[7] G. Petrosyan, "Discovering information relevant to API elements using text classification," Master's thesis, McGill University, 2013.

[8] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014, pp. 55–60.

[9] K. T. Frantzi, S. Ananiadou, and J.-i. Tsujii, "The C-value/NC-value Method of Automatic Recognition for Multi-Word Terms," in *Proceedings of the 2nd European Conference on Research and Advanced Technology for Digital Libraries*, 1998, pp. 585–604.

[10] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[11] D. Das and A. F. T. Martins, "A survey on automatic text summarization," Literature Survey for the Language and Statistics II course at Carnegie Mellon University, Tech. Rep., 2007.

[12] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 505–514.

[13] N. Hariri, C. Castro-Herrera, J. Cleland-Huang, and B. Mobasher, "Chapter 17: Recommendation systems in requirements discovery," in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer, 2014.

[14] L. Kof, "Scenarios: Identifying missing objects and actions by means of computational linguistics," in *Proceedings of the IEEE Requirements Engineering Conference*, 2007, pp. 121–130.

[15] K. Arnout and B. Meyer, "Uncovering hidden contracts: The .net example," *Computer*, vol. 36, no. 11, pp. 48–55, nov 2003.

[16] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*, June 2012, pp. 815–825.

[17] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated Extraction of Security Policies from Natural-Language Software Documents," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2012.

[18] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Proceedings of 24th IEEE/ACM Conference on Automated Software Engineering*, 2009, pp. 307–318.

[19] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 145–158.

[20] S. Henß, M. Monperrus, and M. Mezini, "Semi-automatically extracting FAQs to improve accessibility of software development knowledge," in *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering*, 2012, pp. 793–803.

[21] D. Hou and L. Mo, "Content categorization of API discussions," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 60–69.

[22] S. Thummalapenta, S. Sinha, D. Mukherjee, and S. Chandra, "Automating test automation," IBM Research Division, Tech. Rep. RI11014, 2011.

[23] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions of Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[24] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering*, 2003, pp. 125–135.

[25] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Proceedings of the 15th IEEE International Conference on Program Comprehension*, 2007, pp. 37–48.

[26] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 25, no. 1, pp. 53–95, 2013.

[27] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[28] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Proceedings of the 20th IEEE International Conference on Program Comprehension*, 2012, pp. 63–72.

[29] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 2010, pp. 375–384.

[30] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk, "Enhancing software traceability by automatically expanding corpora with relevant documentation," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 320–329.

[31] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering*, 2014, pp. 643–652.