ANNE: Improving Source Code Search using Entity Retrieval Approach

Venkatesh Vinayakarao
IIIT-Delhi, India
venkateshv@iiitd.ac.in

Anita Sarma
Oregon State University, USA
anita.sarma@oregonstate.edu

Rahul Purandare
IIIT-Delhi, India
purandare@iiitd.ac.in

Shuktika Jain
IIIT-Delhi, India
shuktika12163@iiitd.ac.in

Saumya Jain
IIIT-Delhi, India
saumya12089@iiitd.ac.in

ABSTRACT

Code search with natural language terms performs poorly because programming concepts do not always lexically match their syntactic forms. For example, in Java, the programming concept *array* does not match with its syntactic representation of [ ]. Code search engines can assist developers more effectively over natural language queries if such mappings existed for a variety of programming languages. In this work, we present a programming language agnostic technique to discover such mappings between syntactic forms and natural language terms representing programming concepts. We use the questions and answers in Stack Overflow to create this mapping. We implement our approach in a tool called ANNE. To evaluate its effectiveness, we conduct a user study in an academic setting in which teaching assistants use ANNE to search for code snippets in student submissions. With the use of ANNE, we find that the participants are 29% quicker with no significant drop in correctness and completeness.

Keywords

Code Search, Natural Language Processing, Information Retrieval, Assignment Grading

1. INTRODUCTION

Search has become indispensable in the modern programming context [26], where navigating through thousands of lines of code is infeasible. Developers search for code fragments or keywords to locate or navigate to a particular program concept, when they debug, write code, look for code fragments to reuse, or try to understand an API usage.

Current IDE search features (e.g., Eclipse search) or code search tools (e.g., Sourcerer [5], Portfolio [24]) work by indexing string tokens in the code as keywords. Therefore, searching for a particular programming concept requires an understanding of the syntactic equivalents of that concept. For example, if a developer wants to find whether her C code contains a function that uses an *integer array*, she may formulate a query based on the construct "`int files[]`". However, such a query will miss the occurrences of *functions* that have "`int *files`" in their definitions.

Currently, queries that include natural language (NL) give poor results [13]. Table 1 shows that this problem exists in established web-scale code search engines as well. This is because there is no mapping between the programming concepts and their associated syntactic forms in code.

We extract these mappings from developer discussions in Stack Overflow (SO) [3]. Programming concepts are identical in principle to *named entities* [8]. They are also named and may have multiple surface forms. First, to discover them, we infer the NL terms that refer to these entities by using Parts of Speech (PoS) tagging and pattern matching of these sequences. Then we extract the associated syntactic forms to create an entity knowledge base. Finally, each line of a given source code is annotated with their associated NL terms using the knowledge base, which then allows for regular keyword-based searches on these terms. The process of creating the knowledge base is largely agnostic to programming languages.

To the best of our knowledge, this is the first attempt to discover entities that appear in different forms in text and source code. As a proof of concept, we have implemented our approach in a tool named ANNE[1]: ANNotation Engine, which includes entity knowledge bases for C and Java.

We evaluated the usefulness of the approach through a user study, in the context of Teaching Assistants (TA) providing feedback on submissions to class assignments. We recruited 16 participants, who were currently or had been a TA for introductory CS courses. We used a within-subject study design, where participants in the Control condition used regular code, and those in the Experimental condition used code annotated with the programming concepts (as identified by ANNE). There were two tasks, one an assignment in C and the other in Java. We found correctness scores (precision) of participants to be equivalent across the treatment groups. This is likely because participants were TAs and had experience grading this type of submissions. The time to search was reduced by 29% without compromising on correctness and completeness.

Our key contribution is a programming language agnostic technique to map lines of source code with relevant programming concepts, so as to support code search engines for NL queries. This allows the users to query on programming concepts using NL terms, and need not recall the exact syntactic terms or patterns.

[1]http://tools.pag.iiitd.edu.in:8092/anne/index.html

WSDM '17, February 6–10, 2017, Cambridge, United Kingdom.
© 2017 ACM. ISBN 978-1-4503-4675-7/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3018661.3018691

Table 1: P@10 of existing code search engines for NL queries containing programming concepts.

Query	Krugle [1]	openHUB [2]
declare array	0.1	0
concatenate two arrays	0.2	0
check if a string is a numeric type	0	0
assign to first element in an array	0	0

2. MOTIVATION

In this section, we present our formative study to motivate the work, followed by a discussion on potential applications, and a specific use case for this line of work.

2.1 Formative Study

As formative work, we surveyed to understand whether, and under what conditions developers use natural language terms in their search queries, and whether they face any difficulties when querying for a programming concept by using its syntactic form. We surveyed 25 developers working in leading software development organizations. 18 of these developers had 3 years or less experience in programming, while others had experience ranging from 3 to more than 10 years in industry. We consider the former group as novices.

The survey questions and a summary of responses are listed in Table 2. More details are shared on Anne website. Our results indicate that novice developers, developers who are starting on a new project, or investigating code that they had not worked on for some time face difficulties in finding the right "code" in their project. 44% participants faced situations where the code did not match the NL term they would have used to search for that functionality. 72% participants felt that it is difficult to search for specific code syntax in current IDEs. One participant responded: "*I'd search for the patterns manually using a simple find operation.*" Another said, "*I might search for [different] keywords like 'mid' or 'pivot' when searching for a particular sort algorithm*". Unless the developers can find the correct query words, they will need to manually examine the code.

Code search still happens through keyword search. One participant noted "*[I] search the source code files using keywords/ Mnemonics in the hope that the developer might have used meaningful keywords in the code. Example: for the quicksort example, I might [search] for keywords like mid or pivot*". 60% felt it to be "important" or "very important" to support NL queries. For example, a participant mentioned: "*... [search function in an IDE] was highly useful, but it had no concept of natural language based result, which in my opinion would have proved to be even more useful and would have led to faster search results*".

Therefore, we conclude that programmers in the industry will benefit from a programming language agnostic technique that maps source code lines with its associated programming concepts, which can be used to support search engines that need NL querying.

2.2 Applications

In general, wherever there are NL queries involved in code search, our approach would be useful. In addition to web-scale code search, NL-based search over source code occurs in several software engineering contexts too, such as coding [26], maintenance [25], summarization [30, 23] and program comprehension [22]. To evaluate our work, we have used one scenario from academic code search related to programming assignments. Code search for programming assignments is another real problem and TAs find our tool very useful.

Table 2: Formative study on 25 real industry developers indicates that this research will be useful.

Survey Questions and Responses
(1) *Sometimes when reading a piece of code, the code snippet feels familiar but you may not know how it is popularly called. For example, in a very simplistic case you may be looking at a Quick Sort implementation. After reading the code, you may understand that the code is sorting integers, yet, you may not know that this algorithm is called "quick sort". Have you experienced this?*
Yes: 44% (11); No: 56% (14);
(2) *Sometimes it is difficult to search for a specific code snippet in a project by using existing IDEs. For example, you may want to search how a particular element is initialized in the code. As another example, you may want to search for multiple substring computations and return statement with increment to find the Levenshtein distance implementation. Have you ever experienced this?*
Yes: 72% (18); No: 28% (7);
(3) *Let us assume a situation where a developer wants to search for a quick sort implementation. The implementation is not available under the name "quicksort" and hence the developer wants to find all code snippets where "increment", and "mid-point computation" occur. To do so, the developer opens an IDE and creates a natural language query, "increment, mid-point computation". The IDE then automatically understands that, for Java, increment is "`++`" or "`+ 1`", and mid-point is "`/2`". It finds all methods where such constructs exist.*
(3.1) *How important do you consider this functionality? (>=3 on a scale of 5)*
Important or Neutral: 92% (23); Not Important: 8% (2);
(3.2) *How often would you need to use this functionality? (>=3 on a scale of 5)*
Often: 80% (20); Not so often: 20% (5);

2.3 Problem Overview

Here, we list one use case and explain how our approach solves the problem.

Problem. Sarah is a TA for an introductory Java programming class that has 100 students. She has to evaluate a programming assignment on *enum* which specifically requires students to use *parameterized enumeration*. To assess correctness, she has to look for the syntactic pattern of *parameterized enums* in all the 100 submissions. A keyword search on *enum* alone is insufficient as it will match all *enum* declarations (e.g., `enum { Mercury, Venus, ...}`). Note that *parametrized enum* takes the following form, `enum { Mercury(9,12), ...}`. Therefore, Sarah would either have to use regex in her query (such as `enum.*\{.*(.*);`) or search for occurrences of *enum*, and manually check each assignment to find the submissions that did not use *parameterized enum* properly. Even for experienced TAs, writing regex can be a challenging and error-prone task, especially when

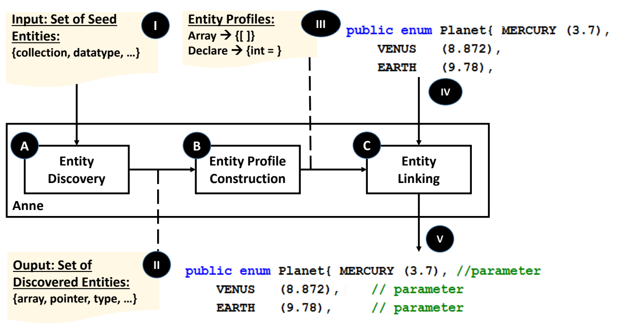

Input: Set of Seed Entities:
{collection, datatype, ...}
I
Entity Profiles:
Array → {[ ]}
Declare → {int = }
III
public enum Planet{ MERCURY (3.7),
VENUS (8.872),
EARTH (9.78),
IV
A
Entity Discovery
B
Entity Profile Construction
C
Entity Linking
Anne
V
Ouput: Set of Discovered Entities:
{array, pointer, type, ...}
II
public enum Planet{ MERCURY (3.7), //parameter
VENUS (8.872), // parameter
EARTH (9.78), // parameter

Figure 1: Anne annotates input code snippets, line by line with natural language terms. These annotations help keyword based search engines to address NL queries.

they have to discriminate between entities such as *type casting* and *parameters*. ANNE takes NL input as *parameterized enum* and lists all occurrences and thus will help Sarah in this task.

Solution. Figure 1 provides an overview of our solution to this problem. Our tool, ANNE, mines SO posts to identify terms in SO questions that pertain to programming concepts, such as *enum* by using Parts of Speech (PoS) tagging. We refer to these programming concepts as *named entities*. More specifically, in Step A, when given a seed entity (e.g., *array*), ANNE identifies other entities (such as *enum, arraylist, collection,* etc.) through PoS tagging.

In step B, ANNE analyzes the SO posts to mine associations between the entities and their syntactic patterns. In our scenario, ANNE will create associations between the programming concept *enum, parameter* with its syntactic pattern: `enum { ( , ) }`. A result of this step, is a knowledge base of named entities and syntactical elements most associated with each entity.

Finally, in step C, given an input source code, ANNE links the relevant programming concepts to each line of code. In our scenario, ANNE tags the code lines from input source code, `enum {Mercury (9,12), Venus (10,13),...}`, with *parameter* as inline comment. As a result of this step, Sarah can now use keyword based search engines to locate the code using NL terms, such as *parameter*.

3. DEFINITIONS

We define the terms that we use in the rest of the paper as follows.

DEFINITION 1. *Let $c = (\tau_1, \ldots, \tau_n)$ be a code snippet where each τ_i is a syntactic token. Any subsequence p of c is a* ***Syntactic Pattern*** *occurring in c.*

DEFINITION 2. ***Named Entities*** *in source code is a binary relation $E \subseteq P \times T$ such that $\forall e_i = (p_i, t_i) \in E$, t_i is a NL phrase used by developers repeatedly to identify an associated syntactic pattern p_i.*

We aim to map a given source code to its collection of entities. Hence, we first need to discover these entities. A bug description, a code comment, SO title, or any NL description of source code q, can be modeled as a sequence of terms or NL phrases $(t_1, \ldots, t_m)$ that points to a set of code snippets $\{c_1, \ldots, c_n\}$. Each code snippet c_i is a collection of lines $(l_{i_1}, \ldots, l_{i_o})$. From a large collection of such (q, c) pairs, our task is to find (p, t) pairs, where p is a syntactic pattern, and t is a term sequence from the vocabulary of q.

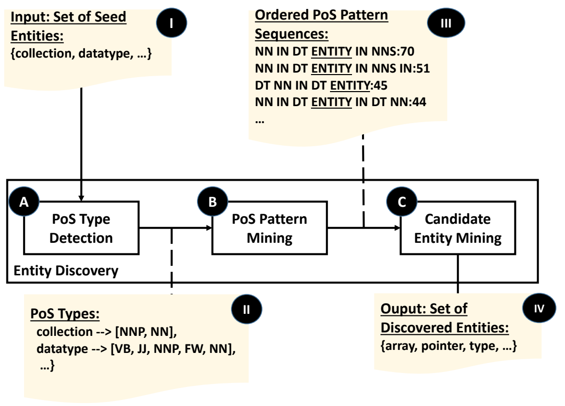

Input: Set of Seed Entities:
{collection, datatype, ...}
I
Ordered PoS Pattern Sequences:
NN IN DT ENTITY IN NNS:70
NN IN DT ENTITY IN NNS IN:51
DT NN IN DT ENTITY:45
NN IN DT ENTITY IN DT NN:44
...
III
A
PoS Type Detection
B
PoS Pattern Mining
C
Candidate Entity Mining
Entity Discovery
PoS Types:
collection --> [NNP, NN],
datatype --> [VB, JJ, NNP, FW, NN],
...}
II
Ouput: Set of Discovered Entities:
{array, pointer, type, ...}
IV

Figure 2: Entity discovery subsystem works on NL text using parts-of-speech (PoS) approach. We use seed entities to discover more entities that fit into the same grammatical sequence.

DEFINITION 3. ***Entity Discovery*** *is the process of extracting candidate term sequences that represent entities in source code. In other words, we find the set of term sequences $T = t_1, \ldots, t_n$, such that each item t_i in it has at least one named entity (p_j, t_i) associated with it. We extract them from the available query-code pairs (q, c).*

DEFINITION 4. *The* ***Entity Profile*** *is a mapping $\psi : T \rightarrow 2^P$ that takes term t as an input, and returns a set P' of syntactic patterns, $\{p_1, \ldots, p_k\}$ associated with t.*

DEFINITION 5. ***Entity Linking*** *in source code is the process of associating a named entity e to a unit u of source code. In other words, it is a mapping $\Delta : C \rightarrow E$, where C is the set of source code units. In this paper, we use a line of code as a unit.*

Note that a line of source code can contain several syntactic patterns, and each syntactic pattern may be associated with several distinct named entities. Hence, the entity linking process may associate several entities to a line of code.

4. APPROACH

Our objective is to automatically tag lines of source code with their associated named entities. This is accomplished through three major steps: a) Entity Discovery, b) Entity Profile Construction, and c) Entity Linking.

4.1 Entity Discovery

We leverage PoS tagging to discover entities from SO titles. The intuition is that developers use similar sentence structures when they ask questions about a programming concept. For example, some of the SO titles are in the form of: *How to declare an* ***array*** *in Java?* and *How to declare a* ***list*** *in Java?* We exploit this similarity in sentence structures to extract entities. Figure 2 gives the workflow for this step.

Table 3: Patterns and frequencies for *conditional* in Java snippets found in SO.

Uni-gram Pattern	Normalized Frequency	n-gram Pattern	Normalized Frequency
`(`	1.00	`if ( ) {`	1.00
...		`( )`	0.50
`if`	0.65	`= ( ( ) )`	0.25
...		`( new ( ) {`	0.25
`while`	0.10	...	
`string`	0.06	...	

We need to identify the entities relevant to a programming language. We select the seed entities from a popular tutorial site, Tutorialspoint[2]. Tutorialspoint groups topics related to the programming language into a short list. For example, Java had 39 topics and C had 29 topics. Some of these topics were phrases such as *type casting* which we mapped to a single word, *cast* in this case, with the help of a language expert. We use this list of processed topics as the seed entities (Figure 2 (I)).

For each seed entity, we extract and group the SO titles that contain that entity. Next, for each title in a group, we annotate its words with their corresponding PoS types (noun, verbs, etc.,) by using an off-the-shelf PoS tagger (Stanford Log-linear PoS Tagger [31]). We extract all PoS sequences for every seed entity as shown in Figure 2 (II).

We use the position of the seed entity and its sequence to discover other entities. This is in accordance with research that has used PoS sequences to understand sentence structures [7]. Therefore, we identify the PoS sequence of each seed entity in all the SO titles in which they appear. For each seed entity, we rank the PoS sequences as per their frequency of occurrences (Figure 2(III)). For example, for *array*, the most frequent pattern was NN IN DT ENTITY IN NNS where ENTITY is the placeholder for *array*. As an example, the SO title, "How to determine type of object/NN in/IN an/DT array/ENTITY of/IN objects/NNS" has this frequent pattern. Same pattern appears in another title, "Get an array of int/NN from/IN a/DT string/ENTITY of/IN numbers/NNS". So we gather that both *array* and *string* have the same PoS sequence. Thus we discover more entities. To tune for precision, we use the top-most frequently occurring PoS sequence (Figure 2(C)). The output of this step is a list of discovered entity names (Figure 2(IV)).

4.2 Entity Profile Construction

Our goal in this step is to link the discovered entities with their syntactic patterns, to create a profile for each entity.

We leverage the fact that source code has repetitive syntactic patterns [15]. For instance, an *array declaration* has a syntactic structure composed of a few tokens, that are repeated across multiple source code snippets. Tu et al. [32] find that source code exhibits redundancies even in local context i.e., in short snippet of code being edited by a developer. They also show that frequency based n-gram patterns can be used to extract these redundancies. Further, Gabel and Su [10] find that the syntactic redundancy peaks at the line level. We leverage all these observations in finding repeating syntactic forms for entities at line-level using n-grams.

We intend to discover these patterns ($p_i \in P'$) in source code that are associated with specific entities (t) (e.g., *array* and *conditional*). These pattern lengths ($|p_i|$) can vary. For example, an *array* has $|[\ ]| = 2$, but a *conditional* has $|if\ (\ )\ \{| = 4$. Let $SO_{\mathcal{L}}$ be the set of all SO posts tagged with a specific programming language $\mathcal{L}$ and containing at least one code snippet per post. We need to identify the most appropriate n-grams that represent a specific entity from $SO_{\mathcal{L}}$. We use the TF-IDF [21, 28] over n-grams to identify the syntactic patterns that are most associated with a given entity in $SO_{\mathcal{L}}$ (Table 3). To compute term frequency $tf(t, g)$ of an n-gram g, we use the $SO_{\mathcal{L}}$ posts containing the entity name in title. For IDF computation, we use all $SO_{\mathcal{L}}$ posts. Since SO post titles and code snippets are short in nature, we ignore the length normalization. Thus, we use the TF-IDF weight $= tf(t, g).log\frac{|D|}{df(g)}$ where $|D|$ is the total number of posts in $SO_{\mathcal{L}}$ and $df(g)$ is the number of such posts containing the n-gram, g. Table 6 shows the results of these steps for a few entities.

Controlling n-gram explosion. We are interested in keywords related to programming concepts and not user defined terms (variable or identifier names). Hence, we collect and tokenize all code snippets from $SO_{\mathcal{L}}$, and rank distinct tokens (τ_i) by frequency ($tf(\tau_i)$). Tokens that are programming concepts will be ranked higher as opposed to user-defined terms, since fewer snippets would have overlap between usages of user defined terms in SO posts. So, we construct a list, $\phi(k) = \{(\tau_1, tf(\tau_1)), ..., (\tau_k, tf(\tau_k))\}$, of top-k tokens with highest frequency. The value of k needs to be large enough to contain all keywords of the programming language. We define a filter $F(\phi(k)) : c_i \rightarrow c_{if}$ which uses $\phi(k)$ to transform every line of code snippet c_i into a line c_{if} with only the top-k uni-grams. This reduces the total n-grams for each line of code, making the TF-IDF computation over n-grams tractable. Table 3 lists the n-gram associations that we mined using this approach for one entity, *conditional*.

In summary, in this step, for each entity that we discover, we identify its associated syntactic pattern, which we call the *entity profile*. The collection of these entities and their profiles serves as our entity profile knowledge base.

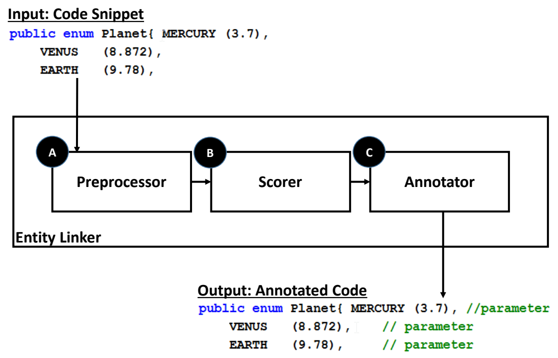
4.3 Entity Linking

Our goal in this step is to annotate every line in a given input source code snippet with entity names of those entities that appear in that line. We use the entity profile knowledge base for this purpose. Figure 3 gives an overview of this step.

We apply the same transformation $F(\phi(k))$ as in Section 4.2 to every line of input code (Figure 3(A)) to remove user defined terms, so that we get reduced number of tokens that pertain to programming concepts. We start with each term being a uni-gram and continue with cumulative aggregation into bi-grams, tri-grams, and so on, until all the n-grams are covered.

However, not all of these n-grams represent entities. For example `else ==` is not an entity. Therefore, we find the n-grams that actually represent entities by using the entity profile knowledge base. That is, we match the syntactic-patterns of an entity with that of the source code to deter-

[2]We used the categories from http://www.tutorialspoint.com



```
public enum Planet{ MERCURY (3.7),
    VENUS   (8.872),
    EARTH   (9.78),
```

```
public enum Planet{ MERCURY (3.7), //parameter
    VENUS   (8.872),     // parameter
    EARTH   (9.78),      // parameter
```

Figure 3: Entity linker subsystem works line by line on the input code, to find matching entity profiles. Entity names whose profiles match are stamped across the line, as shown in the example.

mine if (and which) n-grams from the source code reflect surface forms of entities in source code.

Performing this matching is non-trivial, since n-grams are of different lengths. Therefore, one pattern can be subsumed within another. For example, the best syntactic match for *parameter* is the bi-gram `( )`, whereas the best match for a *conditional* is the four-gram `if (==)`. However, the bi-gram `( )` for *parameter* is subsumed by the four-gram `if (==)` for *conditional*. Therefore, if a line of code contains a *conditional* then both n-grams will match, where marking that line of code with both *parameter* and *conditional* is clearly wrong. However, when we consider the code statement, `if (isTrue(...))`, both *parameter* and *conditional* entities exist. Consider the example shown in Figure 4(B)(line 6). ANNE stamps *parameter* along with a *loop* for this reason. We call this the subsumption problem.

To alleviate the subsumption problem, we use a scoring (Figure 3(B)) function. It creates a metric signifying how well an entity matches the line of code. In the first example, the longer n-gram is a better match (*conditional* vs. *parameter*). However, shorter n-grams are also of interest, as we see in the second example. Therefore, instead of making the entity assignment a binary decision, we rank the n-grams using the scoring function (Equation 1).

$$Score = \delta w_u + (1 - \delta) w_n \tag{1}$$

We use weights for the uni-gram and n-grams based on their TF-IDF values. Table 3 shows the normalized weights for an entity. Since we need to balance between the uni-gram weights (w_u) and n-gram weights (w_n), we empirically identify the distribution factor δ to be 0.6 as this gives us the best results.

Once we determine an entity for a line of code, the annotator tags the line with that entity as an in-line comment (Figure 3(C)). This allows regular keyword-based searches to search on the entities. Note, a line of code can have multiple relevant entities. However, too many entities per line can reduce the quality of the search, as well as cause readability issues if the end user wishes to look at the entities. In the assignments that we use for our user study, we did not find a line of code with more than four entities, therefore, we use that as our threshold. We leave finding the optimal number of entities per line of code as future work.

Table 4: No. of entities discovered is related to the length of PoS patterns considered in our approach. Longer patterns produce fewer entities that exhibit higher level of similarity to seed entity.

#PoSTerms	#Entities	#PoSTerms	#Entities
5	10k	7	12k
6	22k	8	6k

5. EVALUATION

Our evaluation goal is twofold: a) *How well does Anne link entity names to lines of code snippets?* and b) *How useful are these annotations*? However, due to the multi-stage nature of the entity linking process, we divide the first goal into two sub-goals and address them in this section. We address the second question by conducting a user study, which we present in Section 6.

5.1 Entity Discovery

The first question we need to answer to fulfill our goal is: *How well can we automatically identify entities that represent programming terms from SO titles?*

We automatically discover entities based on the position of the seed entity in the PoS sequence in SO titles. Therefore, it is possible that some of the terms that we identify as entities are incorrect. The discovery of entities depends on the length of the SO titles and the PoS sequence lengths.

Of the 0.9 million posts in our dataset, 639K were questions in Java, whose answers also contained source code snippets. The median number of terms contained in these titles was 7. Similarly, we had 139K titles in C, associated with answers containing source code snippets; the median for number of terms in titles was 6.

Next we perform a sensitivity analysis using the PoS sequences around the median values and the number of entities generated. More specifically, we evaluate PoS sequence lengths of 5, 6, 7, and 8 in our dataset. Our findings are presented in Table 4.

For each sets of entities discovered, we calculated the precision of results. True positives were manually evaluated by the first two authors who were experienced in Java and C. They verified that: (1) the entity name appeared in a Java [29] or a C [18] textbook as a term related to a programming concept or a programmatic structure, and (2) the discovered entity had a syntactic pattern. The authors individually identified the true positives and compared their results. Any differences were discussed until they both agreed about a term. If there was disagreement that could not be resolved, then that term was dropped from the list.

From a random sample of SO titles, two experts manually extract the first 30 entities. The first 25 entities that they agreed on (i.e., the intersection of their results) is used to build a *goldset*. The goldset creation process is shown in Figure 5. Stemming gives the root of the words and thus helps in precision. Classifier separates titles with seeds from the rest. Mixer adds noise in required proportion. Our goldset consists of all SO posts containing these 25 entities (162K posts) and thrice (a 1:3 split) that much of noise (i.e., other posts that do not contain any of these 25 entities). To compute recall, we run ANNE with five of these 25 entities as seeds. Table 5 gives the F-measure and count of entities

(A) Enum Task

```
public enum Planet{//enum,parameter
        MERCURY (3.7),    //parameter
        VENUS   (8.872),// parameter
        EARTH   (9.78), // parameter
        MARS    (3.7),  // parameter
        JUPITER (24.79),// parameter
        SATURN  (10.44),// parameter
        URANUS  (8.87), // parameter
        NEPTUNE (11.15);// parameter
        final double surfaceGravity;

        Planet (double
            surfaceGravity){// parameter
                this.surfaceGravity =
                 surfaceGravity;
        }
        ...
```

(B) IncDec Task

```
int main() {
    int N,A;
    scanf("%d%d",&N,&A);// parameter
    int arr[N];             // array
    int i,left,right,flag=0,sum;
    for (i=0;i<N;i++) {       // loop,parameter
      scanf("%d", &arr[i]);}//array,parameter
    left=0;
    right=N-1; //decrement
    while (left!=right){ // parameter
      sum=arr[left]+arr[right];// array
      if (sum<A)          // parameter,
       increment,decrement
        left++;          // increment
      else if (sum>A) // parameter
        right--;         // decrement
        ...
```

Figure 4: Tagged versions of the tasks (A) Enum and (B) IncDec that were provided to the participants in the user study.

Table 5: Performance of Anne Entity Discovery module. Experiments were carried out on a gold set with 1:1 noise and 1:3 noise. F_1 indicates the F_1-score and |E| stands for the number of entities discovered.

P_f	Java 1:1		Java 1:3		C 1:1		C 1:3	
	F_1	|E|	F_1	|E|	F_1	|E|	F_1	|E|
2	0.76	21K	0.14	35K	0.66	12K	0.71	20K
3	0.94	5K	0.87	9K	0.74	10K	0.64	17K
4	0.82	7K	0.87	12K	0.81	6K	0.71	11K
5	0.82	5K	0.76	7K	0.87	3K	0.86	5K
6	0.94	2K	0.91	3K	0.89	1K	0.91	2K
7	0.81	696	0.78	1K	0.86	751	0.86	1K
8	0.81	524	0.78	807	0.89	376	0.89	606
9	0.51	99	0.47	136	0.77	253	0.71	402

discovered. We report these values for both 1:1 and 1:3 splits. Notice that with increase in noise, the F-measure drops. We observe recall of 0.91 for both Java and C, at a proximity of 6.

Next, we use *array* as the seed entity for evaluation for both Java and C on the entire SO corpus. We discovered 20 additional entities for Java, and 18 for C, which resulted in a precision of 0.78 for Java when considering a PoS sequence of 7 terms. For C, it gave a precision of 0.77 when considering a PoS sequence of 6 terms.

Closest to our work are the Named Entity Recognizers (NER). Stanford NER [9] is a popular implementation of linear chain Conditional Random Field (CRF) sequence models. Our approach is much simpler heuristic-based approach which does not need training data. Yet, in principle, this can be modeled as a 2-class classification problem. We trained it with 10K tokens with POS tags where each SO title is a document. Trained NER models that we built are shared on ANNE website. We get near-zero precision and recall of 0.2 on our goldset with these models. The objective for this work is to showcase that entities can be detected and are useful for search. Hence, we do not focus on improving the training data or finding features for the classifier.

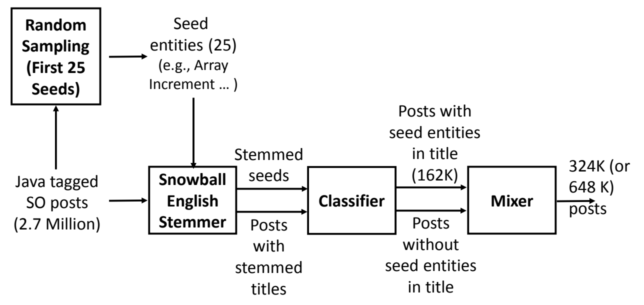

Random Sampling (First 25 Seeds)
Seed entities (25) (e.g., Array Increment …)
Java tagged SO posts (2.7 Million)
Snowball English Stemmer
Stemmed seeds
Posts with stemmed titles
Classifier
Posts with seed entities in title (162K)
Posts without seed entities in title
Mixer
324K (or 648 K) posts

Figure 5: The goldset for evaluation is created from SO posts by mixing posts that contain seed entities in the title with those that do not have them.

5.2 Entity Profile Construction

The evaluation question that we ask here is: *How well can we map entities to their syntactic patterns?*

For each entity, we calculate the precision of the syntactic patterns (a n-gram sequence) extracted in the entity profile construction stage. That is, we evaluate our pattern recommendation for an entity. To do this, we can analyze the top-1, top-2,..., top-n pattern recommendations for each entity. Note, when we consider top-k recommendations, the order in which a pattern appears does not matter, since "all" these k-patterns are linked to the entity.

We analyzed top-1 to top-8 patterns, and found that the best precision is at top-4. This is likely, because if we have too few recommended patterns, then we miss entities. However, if there are too many recommended patterns, it adds noise to the process. Therefore, we assess our recommendation by computing precision@4 (p@4) [21].

The precision of the entity profile knowledge base depends on the richness and the volume of our data. In SO titles for C, the entity *array* appeared for more than 14K times. Because of this, ANNE gets perfect precision (Table 6). However, although some entities, such as *pointers* had more than 10K occurrences, they had many associated patterns: `struct *, int *`. This leads to lower preci-

Table 6: Manually computed precision@4 and the top pattern discovered for some of the entities. We use top four patterns while annotating source code.

Entity	p@4	Pattern	Entity	p@4	Pattern
array	1.00	[ ]	*conditional*	0.75	if ( ) {
decrement	0.75	- -	*increment*	0.75	+ +
loop	0.50	for (; ;)	*parameter*	0.75	( )
pointer	0.50	int *	*variable*	0.75	int

Table 7: Two factor design that counterbalances the treatment and the task.

	Enum	IncDec
Tagged	Group 1	Group 2
Untagged	Group 2	Group 1

sion. We evaluate the patterns for the eight entities that we found in the user study tasks. Table 6 gives the p@4 for these entities, and shows the top pattern. The average precision for these entities is 0.72. We also compute mean reciprocal rank (MRR). MRR is computed as: $MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$ where N is the number of entities, and $rank_i$ is the rank of first relevant pattern for the i^{th} entity. MRR across Java and C for the entity profiles turns out to be **0.71**. ANNE loses on longer patterns and gains on shorter patterns primarily because of subsumption.

6. USER STUDY

We evaluate the usefulness of ANNE through a user study. We recruited 16 participants who were currently a TA or had been one for programming courses. All participants had similar background and programming language skills. They were given two real programming assignments to grade from classes at a lead university. One assignment was from the class taught using C, and the other assignment was from a class taught using Java.

Each submission for these assignments was annotated with the associated named entities by using the entity profile knowledge base, which was created by using the September 2015 SO dump. We implemented a simple search tool (downloadable from ANNE website) to serve as a testbed to evaluate the usefulness of ANNE in a controlled environment.

6.1 Study Design

We selected the tasks for the study by first analyzing all the assignments from the two classes. We focused on assignments given earlier in the semester as these were likely to be easy to evaluate. We needed the tasks in our study to be within 20 minutes, so as to allow the study to be completed in an hour. We performed a pilot study with three graduate students to identify the tasks to be used for the study. For the pilot, we randomly identified six assignments (3 from each class) and their student submissions. Based on our pilot studies, we selected the following two assignments, since the pilot participants could easily understand the code of these two assignments, and took about 15-20 minutes to complete the task.

The first assignment (referred to as *Enum*) expected students to use parameterized enumerators when calculating

Table 8: Descriptive statistics of number of incorrect assignments found by participants.

	Enum (27 Incorrect)		IncDec (15 Incorrect)	
	Mean	Median	Mean	Median
Tagged	24.63	24.00	13.88	14.00
Untagged	22.38	25.50	12.00	13.00

Table 9: Terms used to calculate correctness and completeness scores for a submission S.

Term	Description
True Positive (tp)	S correctly classified as incorrect.
False Positive (fp)	S wrongly classified as incorrect.
False Negative (fn)	S wrongly classified as correct.
True Negative (tn)	S correctly classified as correct.

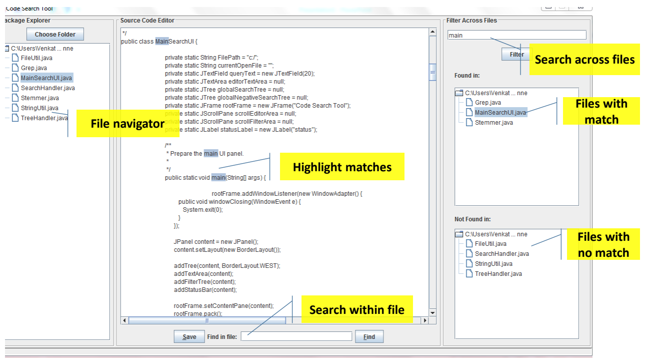
the weight of a person on different planets. The second assignment (referred to as *IncDec*) asked students to operate over a sorted list, while ensuring that their algorithm had a time complexity of O(n). The former assignment was in Java and had 73 student submissions; the latter was in C and included 96 submissions. Figure 4 provides snippets of a student submission for both tasks. Participants had to evaluate the correctness of each student submission and stamp their feedback on the incorrect ones. The Enum and IncDec tasks had 27 and 15 incorrect submissions, respectively.

We followed a two-factor, within-subject study design. We created untagged and tagged versions for each set of submissions, where the former was used as the control condition, and the latter as the experimental condition. We counterbalanced the order in which participants were placed in a treatment group, as well as the task-order that was associated with a specific treatment (Table 7). That is, eight participants had to evaluate assignment submissions that were untagged as their first task, while the other eight participants evaluated the tagged submissions as their first task. Similarly, half the times Enum appeared as a tagged version, and as untagged for the rest.

Participants first filled out background information, and were provided a tutorial of the tool (10 min). They were then asked to evaluate a sample assignment (on pointer usage) to gain a hands-on understanding of the tool and the evaluation that they had to perform (10-15 min).

They were given instruction sheets that explained the different features of ANNE (see Figure 6 for a snapshot of UI). This ensured that they spent their time on the task and not on learning the tool. They were also provided with instructions on how to evaluate submissions, which included the problem statement of the assignment, an explanation of the expected answer, and the feedback that needed to be stamped on the submission. We used the instructions that were provided to the TAs of the (actual) classes to create these materials.

Once they were comfortable with the tool, the experiment started. The time for each task was fixed at 20 minutes. We conducted an exit interview, where we asked whether they would use ANNE for the next class that they TA for. Resources used in this study including video recordings of the study are available on the ANNE website.



```
C:\Users\Venkat ... nne
  FileUtil.java
  Grep.java
  MainSearchUI.java
  SearchHandler.java
  Stemmer.java
  StringUtil.java
  TreeHandler.java
```

```
*/
public class MainSearchUI {

        private static String FilePath = "c:/";
        private static String currentOpenFile = "";
        private static JTextField queryText = new JTextField(20);
        private static JTextArea editorTextArea = null;
        private static JTree globalSearchTree = null;
        private static JTree globalNegativeSearchTree = null;
        private static JFrame rootFrame = new JFrame("Code Search Tool");
        private static JScrollPane scrollEditorArea = null;
        private static JScrollPane scrollFilterArea = null;
        private static JLabel statusLabel = new JLabel("status");

        /**
         * Prepare the main UI panel.
         *
         */
        public static void main(String[] args) {

                rootFrame.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            System.exit(0);
          }
        });

        JPanel content = new JPanel();
        content.setLayout(new BorderLayout());

        addTree(content, BorderLayout.WEST);
        addTextArea(content);
        addFilterTree(content);
        addStatusBar(content);

        rootFrame.setContentPane(content);
        rootFrame.pack();
```

```
main

Found in:
C:\Users\Venkat ... nne
  Grep.java
  MainSearchUI.java
  Stemmer.java

Not Found in:
C:\Users\Venkat ... nne
  FileUtil.java
  SearchHandler.java
  StringUtil.java
  TreeHandler.java
```

Figure 6: Code search tool used for giving feedback to student assignments. This tool allows us to toggle tagging on and off for evaluation.

Table 10: Correctness/Completeness metrics of participants with std. deviation in parentheses.

	Correctness		Completeness	
	Tagged	Untagged	Tagged	Untagged
Enum	1.00(0.00)	1.00(0.00)	0.91(0.25)	0.83(0.06)
IncDec	0.98(2.79)	0.98(2.59)	0.93(0.20)	0.80(0.09)

6.2 Results

Subjects in Experimental condition (S_{tag}) heavily used NL terms in their queries. For example, for the IncDec task, one participant (P17), by using a single search query "increment decrement", was able to identify all the incorrect submissions. Some participants (e.g., P13) created more queries: "increment; decrement; increment decrement; feedback" to get the same results. In contrast, subjects in the Control condition (S_{raw}) made more sophisticated queries, many of which were not successful. For example, P6 tried many queries: "left;++;++ --;binary;while <;for", and was only able to find 13 out of the 15 incorrect submissions.

Table 8 provides the mean and median number of the "incorrect submissions" that participants found. Note that Enum had 27 incorrect submissions out of the 73 total submissions, and IncDec had 15 incorrect submissions out of the 96 total submissions.

We evaluate the quality of the participants' work by calculating the completeness and correctness metrics, for each task (Enum vs. IncDec) and treatment (tagged vs. untagged). Table 9 lists the terms used for these metrics. The correctness metric is calculated as the number of correct classifications divided by the total number of classifications (tp / (tp + fp)). The completeness metric is calculated as the number of true positives divided by the total number of true positives and false negatives (tp / (tp + fn)).

Table 10 provides the correctness and completeness metrics along with standard deviations. We observe that both S_{tag} and S_{raw} obtained very similar correctness scores. This likely occurred because after identifying the submissions, participants evaluated the code before stamping their feedback. Since they were previously TAs and the assignment was relatively simple, their evaluations were accurate. However, for the completeness metric we see that S_{tag} perform better than S_{raw}. This means that more incorrect submissions were missed by S_{raw}. The experience and individual differences play a larger role when participants use a keyword-based search, explaining the higher variance in S_{tag}.

Table 11: Time taken to complete (in minutes) assessment for tagged and untagged versions.

	Enum			IncDec		
	Mean	Median	SD	Mean	Median	SD
Tagged	8.71	7.90	2.61	11.50	13.88	4.66
Untagged	12.90	14.78	3.43	12.98	12.80	4.12

Next, we analyze the time to complete the task. We found that a majority of participants in both treatments followed a two-stage process. As a first pass, participants used the search feature to locate those submissions that *did not* contain the terms in which they were interested. Then as a second pass, they manually investigated the submissions to double check their work[3] if they had time.

Here we report on times to complete the task of the first phase alone, since this best compares the two search processes. Table 11 reports the time (in min) to complete the task and the standard deviation. We observe that S_{tag} was faster. There is a bigger time difference for the Enum task as compared to the IncDec task. This is likely because evaluating the IncDec task was more complex, since participants analyzed the algorithm to determine its complexity. In the Experimental treatment, participants more quickly obtained the set of incorrect submissions, therefore, they may have spent less time in evaluating the other correct submissions.

Next we test for statistical difference between the two treatments for the completeness and time metrics. We perform Shapiro-Wilk test of normality (at $p < 0.05$) and find that both time and completeness are normally distributed. Therefore, we use two-way ANOVA to account for any interaction effects between task and type. For completeness, we find no statistical significance at $p < 0.05$ level; F(1,29)=3.15, p=0.08. S_{tag} had higher completeness metrics. There was no interaction between the treatment (tagged vs. untagged) and the task; F(1,29)=0.018; p=0.89. When using Cohen's d, we get an effect size of 0.64 (medium). So, we gather that tagging does not negatively affect completeness significantly.

When considering time, we see a significant difference ($p < 0.05$) between the two treatment conditions; F(1,29)=4.50, p=0.04. S_{tag} took less time to complete tasks. There was no interaction between the treatment and task; F(1,29)=1.15; p=0.29. We get a Cohen's d value of 0.75 (medium).

In summary, participants are able to complete the tasks in much shorter time (29% less) without compromising significantly on correctness and completeness. The post-task interviews show that tagged search is useful.

7. LIMITATIONS AND THREATS

Utility of our approach can be increased by discovering entities with multiple terms in their name, and also by detecting patterns across multiple lines of code. In principle, our approach can still be used where we treat every snippet as one single line and apply the same technique as we did for uni-gram entity names. As the length of the line

[3]We report correctness and completeness metrics after they finished their tasks and, therefore, include both passes.

increases, number of n-grams in the lines increase and this causes computational overhead. Hence, we look forward to work on more efficient models to support this research.

Handling the subsumption problem where shorter patterns appear inside longer patterns is very hard to address in a language agnostic manner. Although our scoring function alleviates this issue, this can still impact correctness and hence needs attention in future work.

Users may find it unintuitive to formulate queries with terms that do not appear in code. In our case, we tagged the code with terms and thus we circumvented the issue.

We have limited the implementation and evaluation of our work to Java and C programming languages. Yet, other languages, especially, markup languages and functional languages may put forth different challenges. While our technique is statistical by nature and leverages Information Retrieval techniques, the implementation uses language dependent techniques for parsing, and to clean up the snippets.

8. RELATED WORK

To the best of our knowledge, this is the first work to tag code using an Entity Linking based approach.

Code Search. Existence of several code search engines such as Krugle [1], *codease.com* and *searchcode.com* corroborates the fact that code search is of interest to developers and researchers. FIRE SOCO task[4] calls for similarity computation of source code. Our work focuses on improving code search in cases where existing vocabulary does not help with retrieval. Begel [6] argues that developers ask questions that can be answered with meta data about source code. Our approach uses much richer extraneous metadata coming from crowdsourced information. Moreover, we solve a fundamentally different problem of bridging the gap between NL and syntactic form of source code entities such as array, which will benefit all these research efforts.

Tagging Source Code. The closest work to this is on tagging source code for concept location using a variety of approaches [27, 20]. Pollock et al. [25] discuss the advances in NL program analysis research and associated tools. However, these techniques depend on vocabulary in source code and hence solve different problems. Once entity profiles are built, we do not depend on any other source or user defined terms to tag the source code.

Missing NL Terms. The problem of missing vocabulary is highlighted by Howard et al. [16]. They mine semantically similar word pairs from comments. They find synonyms such as "(search,find),(verifies,assert),..." by matching words from code comments with terms in a method name. David and James [33] show how programming constructs translate to English language. They argue in favor of a need for a programming language that is more natural. On these lines, we supplement existing source code with NL terms.

Role of NL in Retrieval Systems. Hill et al. [14] propose a system that improves source code search by allowing users to specify NL phrase representation of method signatures. Arnold and Lieberman [4] suggest that there exists a gap between unambiguous source code representation to a more ambiguous natural description of the functionality. They believe, programming environments should support developers to specify their purpose in natural form to help them to construct source code from such specifications. Exemplar is a search engine to find relevant executable applications for reuse. Exemplar links concepts to source code via the API calls they use. Prompter [26] converts the given code context automatically into a query to retrieve relevant posts from SO. Wursch et al. [34] discuss in detail, the need to support NL queries in IDEs. Our work supports such tools with an approach to link NL terms to source code.

Entity Recognition and Linking. Entity linking is a well-researched problem in IR. This is also referred to as Wikification and Record Linkage [11]. TwiNER system [19] performs Named Entity Recognition (NER) in Twitter streams. Their objective is to identify named entities, such as location and people in twitter streams in an unsupervised manner using a combination of local and global context of the n-grams in Twitter stream. In web text, Downey et al. [8] show that their statistical approach gives better results when compared with supervised approaches. While these apply to source code, the algorithms and approach need significant modification. In this work, we go beyond the common structural entities such as classes, methods and identifiers. We use entity linking approach to find and tag entities that have different NL and syntactic surface forms.

Automated Programming Assignment Feedback. Gulwani et al. [12] describe a dynamic analysis based approach to test whether a student's work matches teacher's specification. The availability of such specifications is a requirement for this approach. Functional correctness of assignments are checked in a variety of ways as surveyed by Ihantola et al. [17]. While we use grading as our use case, our goal is to facilitate search using NL terms.

9. CONCLUSIONS AND FUTURE WORK

In this work, we present a technique that leverages the structural similarities in how people phrase programming questions, and the repetition of syntactic structures in source code, to map source code lines to their programming concepts. This opens up new opportunities to support tools and techniques that connect natural language to source code. Search engines and IDEs can use this mapping to improve code search over NL queries. We show how such a mapping can help in academic assignment search through our tool prototype, ANNE.

Even though our approach, at least in theory, can be extended to program blocks with multiple lines of code, it might need more sophisticated code models for syntactic matching. Another interesting research direction will be to support longer NL phrases. Our approach is predominantly language independent. We look forward to thoroughly evaluating our work on a variety of languages, in addition to Java and C, especially in functional and scripting languages.

10. ACKNOWLEDGMENTS

This work is supported in part by Confederation of Indian Industries (CII), Microsoft Research, National Science Foundation grants HCC 1559657 and CCF 1253786, Infosys Foundation, and Infosys Center for Artificial Intelligence at IIIT-Delhi. Authors thank Ayushi Rastogi and Niharika Sachdeva for their feedback on initial drafts of the paper and anonymous reviewers for their valuable suggestions.

[4]http://users.dsic.upv.es/grupos/nle/soco/

11. REFERENCES

[1] Krugle. http://www.krugle.com/, (accessed 2016-07-30).

[2] Openhub. https://www.openhub.net/, (accessed 2016-07-30).

[3] Stack overflow. http://stackoverflow.com/, (accessed 2016-07-30).

[4] K. C. Arnold and H. Lieberman. Managing ambiguity in programming by finding unambiguous examples. OOPSLA '10, pages 877–884, New York, NY, USA, 2010.

[5] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. OOPSLA '06, pages 681–682, New York, NY, USA, 2006.

[6] A. Begel. Codifier: A programmer-centric search user interface. 2007.

[7] X. Ding, B. Liu, and L. Zhang. Entity discovery and assignment for opinion mining applications. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1125–1134, New York, NY, USA, 2009.

[8] D. Downey, M. Broadhead, and O. Etzioni. Locating complex named entities in web text. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2733–2739, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[9] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[10] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, NY, USA, 2010.

[11] A. Gruenheid, X. L. Dong, and D. Srivastava. Incremental record linkage. *Proc. VLDB Endow.*, 7(9):697–708, 2014.

[12] S. Gulwani, I. Radiček, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 41–51, New York, NY, USA, 2014.

[13] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010.

[14] E. Hill, L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 524–527, Washington, DC, USA, 2011.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012.

[16] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 377–386, Piscataway, NJ, USA, 2013.

[17] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, NY, USA, 2010.

[18] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988.

[19] C. Li, J. Weng, Q. He, Y. Yao, A. Datta, A. Sun, and B.-S. Lee. Twiner: Named entity recognition in targeted twitter stream. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 721–730, New York, NY, USA, 2012.

[20] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243, New York, NY, USA, 2007.

[21] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[22] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, Nov 2004.

[23] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 279–290, New York, NY, USA, 2014.

[24] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1–37:30, Oct. 2013.

[25] L. L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd. Natural language-based software analyses and tools for software maintenance. In A. D. Lucia and F. Ferrucci, editors, *ISSSE*, volume 7171 of *Lecture Notes in Computer Science*, pages 94–125. Springer, 2011.

[26] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111, New York, NY, USA, 2014.

[27] D. Poshyvanyk, M. Gethers, and A. Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4):23:1–23:34, Feb. 2013.

[28] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841, Piscataway, NJ, USA, 2013.

[29] H. Schildt. *Java: The Complete Reference, Ninth Edition*. The Complete Reference. McGraw-Hill Education, 2014.

[30] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[31] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, pages 173–180, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[32] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014.

[33] D. Vadas and R. J. Curran. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*, pages 191–199, 2005.

[34] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 165–174, New York, NY, USA, 2010.