Improving Code Search with Named Entities

Chaitanya Kumar (2012031) and Shuktika Jain (2012163)

***Abstract*— Source code contains syntactic redundancies which are usually discussed using natural language terms. For example, the syntax "(int) a" is referred to as type casting in natural language. Now, if a person searches for *type-cast* in this code, he/she will not find this statement because the terms are different. Our project aims at capturing such relations between syntax and the terms developers use to refer to them.**

I. Introduction

A number of code search engines like Krugle and Ohloh typically perform a text search. On firing the query "initialize" on these two systems, the output had only those results that had the text "initialize" in code (class/method titles), or in comments surrounding the code. From a natural language perspective, an "initialize" would mean more than that - from trivial assignments to even creation of object instances. Likewise, other queries like "assignment", "increment", "add", "check" yielded similar results, having little to do with their actual syntactic patterns. Thus, we attempt to find a solution to this problem by using named entities in code search to optimize code based search systems. Currently, our project has been implemented only for the Java programming language, but it can easily be expanded to incorporate other languages. The project could also be extended to auto-comment and summarize code.

From the StackOverflow data dump, we extract the frequently occurring natural language words related to syntactic patterns in source code. These form the *Named Entities* for our project. Prior to the first milestone, for each entity, we extracted the relevant code for that entity from the StackOverflow posts and converted it into an *Abstract Intermediate Representation (AIR)* which captured the syntactic pattern of a line of code. The most common representations for an entity were then combined to form an *Entity Profile* for that entity. Next, for a code snippet, we converted it into AIR and used the search system *Lucene* to index and search for the matching entity profile. This process constituted the *Entity Linking* aspect. Post the first milestone, instead of using *AIR*, we directly made use of term frequencies, incorporating document length normalization. Also, stopwords specific to the code obtained for each entity were eliminated. For the *Entity Profiling* and *Entity Linking* part, Language Modeling was used. Details on this constitute Section IV.

II. Dataset and Queries

Our project makes use of freely available anonymized dump of user-contributed content on the Stack Exchange network [1], generated every three months. From the dump, the StackOverflow posts corresponding to Java programming language were extracted and parsed to form the final dataset. We use this dataset to learn the entities and their profiles.

For queries, we used code snippets from the textbook "Java The Complete Reference - 7th Edition". We believe these are suitable for evaluation because the dataset for testing is different from the training (StackOverflow) dataset. Also, the textbook contains a variety of snippets related to different concepts which will be representative of the Java language.

We used about 100 lines of code with each line acting as a query snippet. We have attached the text file *Oracle_Code.txt* which contains the query code lines and *Oracle_Code_ManuallyAnnotated.txt* which contains the ground truth, i.e. the manually annotated entities for each line against which we compare our system's annotation.

III. Related Work

The related work on these lines includes the work by Li et al, wherein named identities such as location and people were identified in twitter stream (TwiNER) [2]. The idea seems applicable to source code, although the algorithms and approach need some modification. Hill et al. [3] propose a system which 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 and help them to construct source code from such specifications. Their

system, Zone, is a step in this direction. Prompter [5] converts the given code context automatically into a query to retrieve relevant posts from StackOverflow. To the best of our knowledge, this is the first work to tag source code with natural language terms using an Entity Linking based approach.

IV. Technique and Implementation

This section contains the detailed description of our approach.

A. Named Entities

We begin by extracting named entities from the StackOverflow dump. We parsed the xml file containing Java posts to extract all titles and obtained 14,026 titles. Then, we tokenised the titles into unigrams (single words) and computed frequencies for the unigrams. We removed English stop-words from the set of unigrams because they cannot be named entities explaining some syntactic pattern in a code snippet. We also removed Java keywords as they can be found simply by doing a text search in the code and the focus of our project is to search for patterns having natural language terms not present in the code. Next, we used Porter Stemmer algorithm to map inflected words to their root word and combined the frequencies. We then ordered these unigrams in decreasing order of frequency. Out of these unigrams, we manually picked top 25 words where each word was a natural language term for at least one code snippet. These words consisting of "array", "loop", "add", "check" etc. formed our *named entities*.

B. Entity Profiles

Before the first milestone, we used the following approach without incorporating language models. For each entity, we needed some syntactic pattern which would be representative enough of all the code snippets corresponding to the particular entity. These patterns are called *entity profiles*. We extracted entity profiles in the following manner:

- *Code Extraction Per Entity:* For each title in the StackOverflow database, we computed its unigrams. If a unigram was also a named entity, then we extracted code from the body of that title and appended the code separated by a newline character in a file corresponding to that entity. This was done for all the titles to get code corresponding to all entities. The idea is that a post about the entity will have some code representative of the entity. For example, a post about "array" will have some code snippets containing square brackets. This way, we obtained a file containing code for each entity.
- *Abstract Intermediate Representation:* After extracting the code snippets, we had to convert them into a new representation which could generalize the lines. For example, code snippets "int a" and "int b" should correspond to a single representation "type var". Thus, for each entity, we found the AIR for all lines of extracted code corresponding to that entity. This was done using the Eclipse JDT API, which creates an Abstract Syntax Tree, for a given Java source. For a single line of code input, it was checked for the kind of node representations it had in the AST, and corresponding keywords were assigned to the statement. For instance, **int[] arr = {1,2,3,4}; :== type[] var operator literal**.
- *Clustering:* By this stage, we have the files containing AIR of the code snippets for every entity. We need to find the most common syntactic patterns for each entity from these AIR snippets which will form our *entity profiles*. To do this, for each entity, we computed n-grams of each AIR code snippet and ordered these n-grams in descending order of tf-idf score using equation 1.

$$score = tf * log(N/df) \quad (1)$$

We then removed the n-grams which were substrings of other n-grams of same frequency. Next, we took the top *k* n-grams according to calculated scores to get entity profiles for each entity. The value of *k* was empirically set to 10.

The first precision-recall numbers were not so good and to improve them, the entity profile construction needed to be modified. So, after the first milestone, we used a new method to construct profiles which is as follows:

- *Step 1:* The same process of code extraction per entity is followed to extract code for each entity.
- *Step 2:* For each entity, the code snippets are tokenized into unigrams and the frequency of each unigram is stored in a map. Similarly, global term frequencies are also stored, which correspond to the frequencies for the combined code of all entities. The unigrams with high frequencies in this map will be stop words as they'll be present in almost all the entities.
- *Step 3:* The term frequencies for each entity are then normalized so that these stop words have

a lower frequency. Also, document length normalization is done by simply dividing each term frequency by the number of unigrams for that entity.

- *Step 4:* The normalized term frequencies for each entity are then stored in a csv file.

C. Entity Linking

Now, we have an entity profile knowledgebase which is a set of entity profiles with at least one pattern each for each named entity. With this knowledgebase, our goal is to tag source code lines.

Before the first milestone, following is the process we used to link entities to source code lines. We convert each line in the input source code to AIR. We compare AIR with entity profiles and select the best profiles. We use Lucene to index these entity profile patterns. We then search this index with the n-grams of AIR of source code to be tagged and find the number of matches for every entity. If the ratio of number of matches to the count of n-grams is greater than a threshold, then we tag the entity to the snippet, i.e. write it in front of the line. This way, we tag every line of code and link the entities. The threshold acts as a sensitivity factor.

After the new method of entity profile construction, we used a new way of linking entities too. We used a language modeling approach. For each line of the source code, we tokenized it into unigrams. Then, to check if an entity should be tagged to the line, we use the normalized term frequencies. We have a top-k factor which is used to consider only the top-k unigrams in terms of normalized term frequencies. For each entity, we get the term frequencies of all the unigrams in the source code line, add the weights and then normalize them by dividing by the maximum total weight that could have been generated (using the term frequencies of combined code). Thus, we generate a number between 0 and 1 which is a probability of the source code line (query) being generated by the model (entity). Thus, we use a language model to tag source code lines. Next, we check if the attained probability is above a cut-off and tag the entity to the line if this is the case. This cut-off again acts as our threshold or sensitivity factor. Therefore, we have two factors to change - sensitivity factor and top-k factor.

V. Evaluation

To evaluate our system, we use well distributed and representative 100 lines of source code from Java programming language bestseller text book, "Java - The Complete Reference, 7th Edition". We manually tag the lines with the 25 named entities wherever appropriate. We consider this as our ground truth and compare the system output against this manually tagged version. In order to repeatedly perform this validation, for instance, to compute the impact of various values of sensitivities, we create an Oracle. Oracle is a system which reads the ground truth, compares with system output, and calculates precision, recall and F1 scores for each line of code and then outputs the average values of these measures. Precision and recall for a line of code are defined as follows:

- *Precision:* Out of all the system tagged entities, how many are also tagged in the manually annotated file, i.e. ratio of number of correctly tagged entities to the number of system tagged entities.
- *Recall:* Out of all the manually tagged entities, how many are tagged by the system, i.e. ratio of number of correctly tagged entities to the number of manually tagged entities.

For example, if the system tagged line has entities "array", "declare" and "number", and the manually tagged line has entities "array" and "declare", then precision for the line is 66% and recall is 100%.

Before the first milestone, upon evaluating on different levels of sensitivities, the precision and recall values obtained were 15% and 21% respectively. The F1 score was 0.175 or 17.5%.

After the first milestone, we evaluated the system annotation with different values of k and sensitivity. On taking all 25 entities together, the best precision and recall values obtained were 16.2% and 29.7% respectively implying an F1 score of 21%. When we took just the entity *array*, the best F1 value obtained was with a cut-off factor of *0.6* with top-100 unigrams. The precision and recall values obtained were 66.7% and 93.3% implying an F1 score of 77.8%. Figure 1 shows the precision, recall and F1 values for the entity *array* with top-100 unigrams at various sensitivity levels.

VI. Work Distribution

The initial work involving data collection, problem analysis and detailed study was a collaborative effort with equal contribution by both members. The implementation was also equally divided between the members. Chaitanya implemented the code for named entities extraction, converting code into AIR and Oracle's precision-recall computation while Shuktika implemented the code for code extraction per entity, clus-

Cut-off Factor	Precision	Recall	F1 measure
0.0	0.182	0.982	0.307
0.1	0.182	0.982	0.307
0.2	0.182	0.982	0.307
0.3	0.182	0.982	0.307
0.4	0.238	0.976	0.383
0.5	0.4	0.96	0.565
0.6	0.667	0.933	0.778
0.7	0.571	0.786	0.662
0.8	0.333	0.417	0.37
0.9	0.167	0.25	0.2
1.0	0	0	?

Fig. 1: Precision, Recall and F1 values for the entity *array* with top-100 unigrams at various sensitivity levels

tering to create entity profiles and code to link entities. Thus, three parts for implementation were done by each member. Both the members manually annotated the textbook code for evaluation to not make the tagging biased. After the first milestone too, the work was equally divided between the members with one taking up the entity profile construction part and the other concentrating on the entity linking implementation. The report is also a result of joint effort by both members.

VII. Demonstration Plan

After the instructions in the file Readme.txt:

1) Any chosen snippet from some source is taken, and fed to the tool.
2) For best results, the code should:
   - be a valid typical Java snippet
   - have an association with the entities ”array” and/or ”remove”
3) Clicking submit, shall produce two windows, with the first one containing the input code fed in the step above.
4) The second window will have the same code annotated by the tool, with the two entities ”array” and ”remove”, wherever appropriate.

VIII. Acknowledgments

We would like to thank Venkatesh Vinayakarao, for guiding us whenever needed, and enhancing our exposure towards the domain of Information Retrieval by means of discussion and brainstorming.

References

[1] https://archive.org/details/stackexchange
[2] http://researcher.ibm.com/researcher/files/us-heq/sigir12twiner.pdf
[3] Emily Hill, Lori 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 524527, Washington, DC, USA, 2011. IEEE Computer Society.
[4] Kenneth C. Arnold and Henry Lieberman. Managing ambiguity in programming by finding unambiguous examples. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 10, pages 877884, New York, NY, USA, 2010. ACM.
[5] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele 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 102111, New York, NY, USA, 2014. ACM.