Method to Suggest Similar API Calls Across Languages and Frameworks

Maesen Churchill (maesenc@stanford.edu)
Jess Fisher (jessf@stanford.edu)
Alex Valderrama (avald@stanford.edu)

Abstract

For software developers, finding equivalent API calls across different languages and frameworks can be a difficult and tedious task. However, doing so is often necessary for cross-platform development, or for educational purposes (e.g. programming classes). Currently, programmers must find equivalent API calls for different frameworks by searching documentation or reading code samples. We propose a novel way of suggesting similar methods, given a source and target API, by using classification. We trained two classifiers, using Perceptron and Support Vector Machines, to recognize functionally similar methods. We then test the classifiers on a set of equivalent methods from JavaMe and Android Graphics. Our classifiers are reasonably successful at identifying the most similar methods in the target API (in the SVM, the most similar method in the target API was the first method suggested 67.8% of the time). We also identified future work that could improve the classifier performance, including more accurate parsing of words from method names and more advanced relationship extraction from method descriptions.

I. Introduction

As a software developer, the ability to work across different languages and frameworks is a valuable skill. Most companies demand that their applications be available across several different platforms. Furthermore, computer science students are often expected to work with different languages for different classes.

Working in an unfamiliar framework can be difficult for even the most experienced programmers. Even if a developer suspects an API method exists, finding the right one for any given situation can be difficult. If a programmer wants to find the analog in API A of method from API B, they must do so by looking through documentation. This can be tedious and time-consuming, and at times challenging. For novice computer science students, working with language libraries was identified as one of the most difficult programming concepts, along with recursion, pointers and references, abstract data types, and error handling [1].

Amrutha Gokahale, Vinod Ganaparthy and Yogesh Padmanaban of Rutgers University proposed an automatic way to infer likely mappings between APIs by analyzing stack traces of implementations of the same application on different frameworks (for example, the TicTacToe game implemented on JavaME and Android). Gokahale et. al. inferred

relationships between methods in source and target APIs by looking at features of functions such as their call frequency, call position, call context and method name [2].

With this research in mind, we propose a novel way of suggesting similar methods in a target API to a function in some source API, using machine learning on API documentation. In this paper, we attempt to train classifiers to recognize similar function calls from function names, parameter names, and function descriptions. We then use the decision functions of the trained classifiers to suggest similar functions to a given method from a specified target API.

Method name	abs
Parameters	a
Description	Returns the absolute value of a double value.

Figure 1: The information from a single method we use in classification.

II. Methods

Our training set was compiled and labeled manually from 6 different API categories across 6 different languages. We selected only object-oriented languages, but we sampled from both scripting and compiled languages, and both weakly and strictly typed languages. Methods were given unique IDs so methods that were functionally equivalent from two different APIs were given the same ID. Methods were only compared in the training set of our classifiers if they were from the same type of API.

Library Types	Languages
Graphics	Python
Math	Java
I/O	C#
Strings	C++
Threads	Javascript
Date/Time	PHP

Figure 2: Sources of training data. We classified 236 methods to create 3022 data points (method pairs).

For each pair of methods, we extracted twelve features. We chose any features that might inform whether two methods were functionally equivalent. We did not include return or parameter type information so that our method selection system could be used on object-oriented languages that are not explicitly typed, such as Python.

The edit distance of the method names is defined as the number of insertions, deletions and substitutions needed to transform one method name to the other. An "uncommon" word in a given library type is defined as one that appears in fewer than 20 methods in libraries of that type in our training set. We also included features for the number of words the names have in common, and the number of distinct words in the method names. To parse words from method names, we used the following heuristic:

1. If the method name includes underscores, let underscores serve as word boundaries.
2. Otherwise, use changes in capitalization as indications of word boundaries, e.g. toLowerCase

would have words “to”, “lower” and “case.”

Features
The edit distance of the method names
Whether the methods had the same number of parameters
The difference in the number of parameters
The number of parameter names the methods share
The difference in the lengths of the method descriptions
The number of words the descriptions share
The number of words that appear in exactly one description
The “Manhattan distance” of the descriptions
The number of “uncommon” words the descriptions share
The number of “uncommon” words that appear in exactly one description
The number of words the method names share
The number of words that appear in exactly one method name

Figure 3: Features extracted from each pair of methods.

Each method pair was classified as a match if the methods had the same unique ID (indicating they are functionally equivalent).

The training set was used to train two classifiers: one using the Perceptron algorithm, and one Support Vector Machine. We then tested the classifiers on a test set composed of pairs of methods from the Java Mobile Edition and Android Graphics APIs. Our test set contained 34 pairs of equivalent methods. We tested the algorithm in the following way: for each method in JavaME, we paired it with all 34 possibilities in Android Graphics and ran both trained classifiers’ decision functions on that pair of methods. We used the outcome of the decision functions as a measure of similarity between the methods, and suggested the top five method matches in Android Graphics for each method in JavaME.

III. Results

Both the Perceptron and SVM performed reasonably well at recommending methods for the test set.

	Perceptron	**SVM**
Correct method was first suggestion	57.9%	67.6%
Correct method was in the top five suggestions	79.0%	85.3%

Figure 4: Classifier performance on the test set.

In both classifiers, edit distance of the names and the number of words the names shared were the most relevant features. In all cases of similar method names, the classifiers could correctly suggest the method from Android graphics.

Suggestions for isItalic (JavaME)
isItalic*
isMutable
isBold
isVisible
isShowing

Figure 5: The top five suggestions for JavaMe method isItalic in the Android Graphics API. These results were typical when paired methods had similar names. Asterisk indicates correct match.

For methods in JavaME that had matches in Android Graphics that didn't share similar names, the classifier had worse performance. However, the SVM still performed fairly well. Even if the target method wasn't the top suggestion, it often appeared in the top five.

Suggestions for createImage (JavaME)
createBitmap
create
setMessage
decodeByteArray*
setMessage

Figure 7: The top five suggestions for JavaME method createImage in the Android Graphics API. Results for suggestions when paired method names were dissimilar were less reliable.

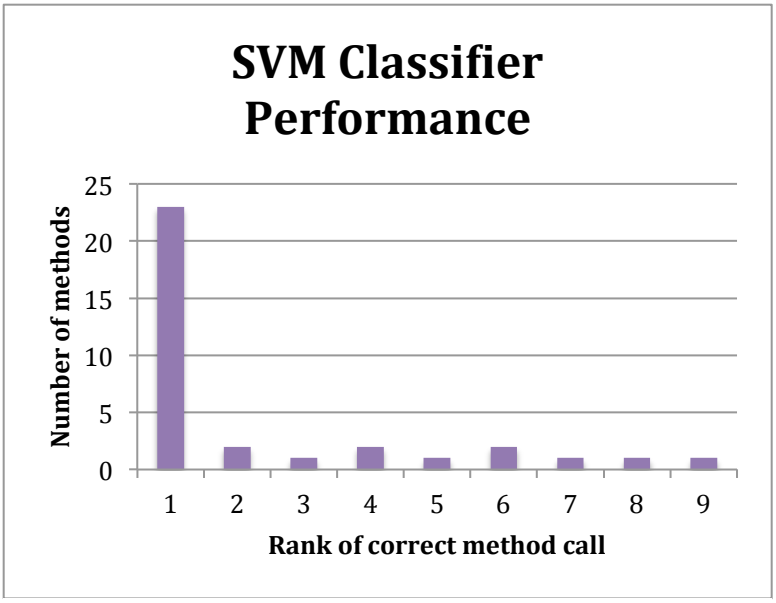

Figure 6: SVM performance for test set.

IV. Analysis

Classification of similar methods using documentation was generally successful. Our classifiers perform favorably compared to previous work in this field. Gokahale et. al. had a 40% success rate at predicting matching methods and a 70% success rate at having the matching method appear in list of the top ten suggestions when testing on 56 methods from the APIs we used – JavaMe and Android Graphics [2]. Though we cannot be sure how much our test set overlaps with that of Gokahale et. al., we are convinced that we have topped their performance.

Support Vector Machine performance topped the performance of the Weighted Classifier trained by the Perceptron algorithm. Perhaps this is because the SVM is a bit more resistant to over fitting, due to it keeping the magnitude of the weight vector small [3].

However, there are a few shortcomings of our classification scheme. Perhaps the most obvious pitfall of our method for similar function call suggestion is that it requires robust documentation from both source and target APIs. Being able to identify the target API and find documentation can be difficult, especially for novice programmers. Furthermore, if the API does not give adequate documentation for each method, our classification scheme will not work.

Another shortcoming of our classification scheme is the performance drop-off that

comes when two functionally equivalent functions do not have similar names. In both Perceptron and SVM, edit distance between method names is heavily weighted as the most important feature. However, this has negative implications for both specificity and sensitivity.

Placing such importance on method name can introduce some false positives to method pairs by chance. For example, before features for the number of words in common between method names, getFont was the top suggestion for setIcon by the SVM (rather than its true match, setImage), in large part because setIcon and getFont happen to have a low edit distance (due to the presence of "et" and "on" in both words, as well as the similarity in the lengths of the names).

Furthermore, the high weight of edit distance ensures that when true pairs have dissimilar names, they are often classified as negatives incorrectly. In these cases, our classification still does much better than random – for the SVM, the correct method is always in the top 10 suggestions. However, the distribution of the rank of the suggestion for the matching method is near uniform from the second to the ninth suggestion (see Figure 7). This suggests a lack of reliable prediction when method names differ significantly.

To try to correct for this, future work would include pruning back the feature set to prevent over-fitting, and trying to compile a list of more informative features.

For the future, the following extensions could be made to create more informative features:

1. Compile a list of synonyms to be able to more accurately quantify name and description similarity.
2. Use Named-Entity-Recognition to extract parts-of-speech from method descriptions.
3. Use more advanced Natural Language Processing to extract the return value "category" (e.g. number, image, void) from each method and create a feature for whether these categories match.
4. Use more advanced Natural Language Processing to reason about parameter-return value relationships.

We predict that more accurate feature sets could lead to better classification performance.

References

[1] Lahtinen, Essi, Ala-Mutka, Kirsti, and Jarvian, Hannu-Matti, "A Study of the Difficulties of Novice Programmers," submitted to the 2005 Conference on Innovation and Technology in Computer Science Education.

[2] Gokahale, Amrutha, Ganaparthy, Vinod, and Padmanaban, Yogesh, "Inferring Likely Mappings Between APIs," published for the 35th International Conference on Software Engineering, May 2013.

[3] Ng, Andrew. "CS229 Lecture Notes: Support Vector Machines." http://cs229.stanford.edu/notes/cs229-notes3.pdf