Support Vector Machine Source Code Classification

Abstract

Abstract goes here, to be filled in later.

1. Introduction

The goal of the project is to apply machine learning and natural language processing techniques to efficiently search a corpus composed of programing language source code and comments. The input to the search will be a method signature as well as any commented lines directly above it. The system will then use the information contained within the input to search through a corpus comprised of programming language code and return a top-k list of method proposals intended to match the functionality of the original input search method. The unit of search is a single method and each method is a single document within the code corpus. The intended application of this system is to aid in the use of foreign APIs and SDKs to software developers who may not be familiar with all of the methods and their proper usage, but have an idea of exactly what they would like to achieve by using the API.

Figure 1 shows a sample program in which the user specifies the method call (highlighted) and the comments above the method call. The user first creates an unimplemented interface method (see Figure 2) and calls this method in the scope in which it is intended to be used. The data present in the comments, the method call and all of the variables within the scope of the method call will be used to accurately search for completion proposals. The intended end result is implementing the interface method by filling in the code in such a way that the method matches the functionality that the user intended.

```
public class TestClass{
    public static void main(String[] args) {
        TestInterface i1;
        i1 = new TestImplementation();

        int[] array = new int[]{3, 5, 2, 1, 4};

        /*
         * Sort an array of integers
         * @param array
         * @return sorted array
         */
        i1.testMethodTwo(array);
    }

}
```

Figure 1: Sample Inputs

2. Related Work

2.1. Language Model for Source Code

Hindle et al.[1] create and analyze a language model for source code.

2.2. SVM Text Classification

The success of support vector machines for text classification is well known. Joachims et al. explore the use of support vector machines using the tf-idf of the words in the text as features[2].

2.3. SVM Source Code Classification

Ugurel et al. explore the use of SVM classifiers to categorize source code archives by topic as well as by programming language[3]. They show promising results while classifying code at the project or archive level.

2.4. Semantic Clustering and LSI

Kuhn et al. introduce a method of semantic clustering based on the LSI model[4]. A LSI model is used to cluster source code by topic based on the comments and naming conventions(method and variable names) present in the code.

3. Model

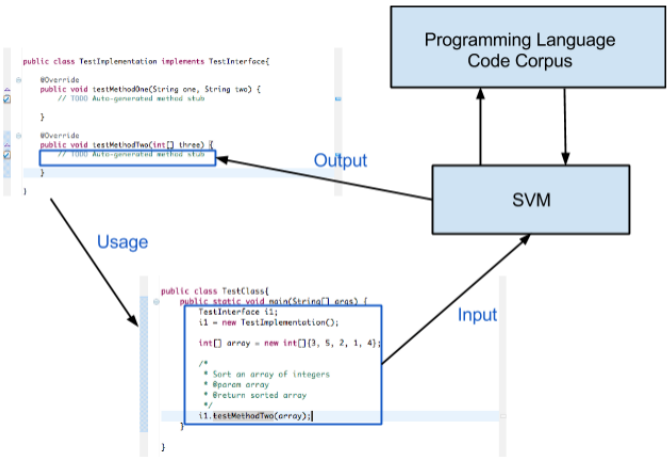

Figure 2: Overall Flow

3.1. Inputs

The inputs to the code completion project consist of all code present in the file when calling the project, the interface method to be filled in with the appropriate data as well as

any comments describing the desired functionality of the interface method to be implemented. See Figure 1 for a sample program input.Type information as well as word occurrences are used from this data in order to aid the model.

3.2. SVM Classification

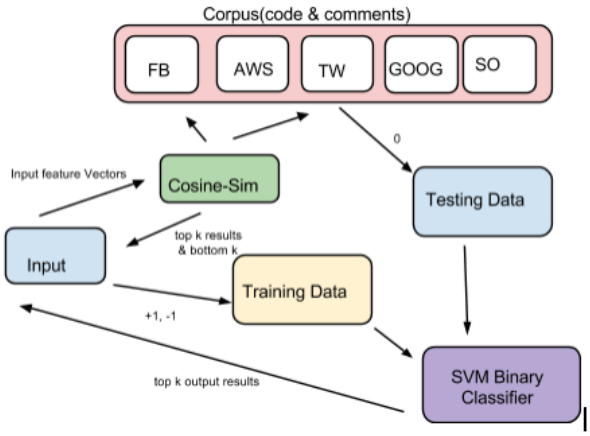

Figure 3: SVM Classification System

We use a support vector machine(SVM) to classify code within the corpus in a binary manner as being similar or non-similar to our search input. Figure 3 illustrates the overall SVM classification system. The purpose of the SVM system is to retrieve a set of methods from the entire corpus that are similar(in terms of functionality) to the input query.

The SVM system works as follows:

1. User Inputs comments and method to be completed(blue box in Figure 3)
2. System creates a bag of word feature representation set of the tf-idf of words in the input text
3. System calculates cosine similarity between input feature representations and the feature representations for all of the code in the corpus(green box)
4. A set of training data is created by returning the top k and bottom k results from the cosine similarity search and allowing the user to mark each one as correct(+1) or incorrect(-1) (yellow box)
5. The entire corpus is then used as the testing set and the SVM classifies each piece of code as matching or non-matching with a certain ranking. (blue and purple boxes)
6. The top k matches are returned

3.3. Cost Function

Once the SVM system returns the top results, a cost function is used to select a single or a set of methods that can be used to achieve the desired result of the input query. Variable types, usage/declaration, scope, location and method signature information will be used to determine which methods are best able to complete the desired functionality. The content of these methods will then be combined in a way to minimize this cost function and complete the unimplemented interface method.

4. Experiments

4.1. Source Code Corpus

Programming language source code was extracted from Github[1] as well as Grepcode [2]. The focus was on the Java programming language, due to the fact that it is widely used and documented in addition to having strong conventions. Publicly available SDK repositories of large corporations were cloned and parsed from Github. Logically these would be the best commented, cleanest and maintained. Additionally, Individual methods were pulled from classes retrieved by searching on Grepcode.com. Once the source code was downloaded all individual methods were parsed into individual files and the comments corresponding to the methods were also parsed out and linked to by filename.

After the source code was downloaded and methods were parsed, tests were created by labeling the methods as either belonging to a class[3] or not belonging. The size of the tests range from 20 to 600 methods. For each, test methods were classified as having a specific functionality(i.e. sorting function) or not. This data was split into testing and training data and then the classification methods were applied with the goal of determining whether a specific method under test had the desired functionality or not. For each test, baselines were calculated by calculating the accuracy of the classification method classifying all code into the same class. For example, given a test in which 90 methods belong to class a and 10 methods belong to class b, the baseline would be calculated as 0.90.

4.2. Software Packages

The support vector machine package SVMLight[5] was used.

4.3. Support Vector Machine Feature Vectors

We tested feature vectors representations based on different available data and the results are shown in Table 1 and Table 2 below. The tests comprise of training and test methods for different groups of code ranging in size from 20 to 80 methods. Many different feature vector representations were tested. A description of each of these representations can be found below. All of the representations show significant improvement over the baseline.

The following tests were used in testing linear kernel support vector machine classification:

[1] www.github.com
[2] www.grepcode.com
[3] In the sense of classification not object-oriented programming

- Binary feature vectors of code present in method as input features. 1 if word was present in the method, else 0. (Table 2)
- TF-IDF feature vectors of code present in method as input features. Calculate TF-IDF of each word present in the method. (Table 2)
- Binary feature vectors of compressed code present in method as input features. Compressed code refers to the removal of specific variable or method names and only contains the variable types(i.e. String), statements(i.e. 'if') and method input and output types present in the method. (Table 1)
- TF-IDF feature vectors of comments for each method as input features. Calculate TF-IDF of each word present in the comments. (Table 1)
- TF-IDF feature vectors of JavaDoc tags crossed with comments for each method as input features. JavaDoc tags such as @param were crossed with the corresponding variable in order to take advantage of the JavaDoc styled comments. (Table 1)

Test	TF-IDF	Types & Statements	JavaDoc	Baseline
Test_3	70 %	50%	70%	50%
Test_5	91.67%	75%	91.67%	75%
Test_7	97.62%	52.38%	100%	52.38%
Test_10	82.98%	74.47%	82.98%	85.11%

Table 1: Accuracies of SVM using Comments

Test	Binary	TF-IDF	Baseline
Test_3	95%	50%	50%
Test_5	91.67%	75%	75%
Test_7	100%	52.38%	52.38%
Test_10	82.98%	85.11%	85.11%

Table 2: Accuracies of SVM using Source Code

4.4. Cost Function Tests

Add in cost function table, showing that it is poor at classifying code, but good at determining if a method will 'fit' into the returned interface.

4.5. Overall System Tests

To be completed

5. Conclusion & Future Work

Conclusion goes here, to be filled in later.

6. References

[1] Hindle, Abram, et al. "On the naturalness of software." Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012.

[2] Joachims, Thorsten. Text categorization with support vector machines: Learning with many relevant features. Springer Berlin Heidelberg, 1998.

[3] Ugurel, Secil, Robert Krovetz, and C. Lee Giles. "What's the code?: automatic classification of source code archives." Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2002.

[4] Kuhn, Adrian, Stephane Ducasse, and Tudor Girba. "Semantic clustering: Identifying topics in source code." Information and Software Technology 49.3 (2007): 230-243.

[5] T. Joachims, Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning, B. Scholkopf and C. Burges and A. Smola (ed.), MIT-Press, 1999

7. Acknowledgements

Acknowledgements