Evaluation of Type Inference with Textual Cues

Amirreza Shirani[†], A. Pastor Lopez-Monroy[†], Fabio Gonzalez[‡], Thamar Solorio[†], Mohammad Amin Alipour[†]
[†]Department of Computer Science, University of Houston, TX
[‡]Department of Systems and Computer Engineering, National University of Colombia, Bogota, Columbia

Abstract

Type information plays an important role in the success of information retrieval and recommendation systems in software engineering. Thus, the absence of types in dynamically-typed languages poses a challenge to adapt these systems to support dynamic languages.

In this paper, we explore the viability of type inference using *textual cues*. That is, we formulate the type inference problem as a classification problem which uses the textual features in the source code to predict the type of variables. In this approach, a classifier learns a model to distinguish between types of variables in a program. The model is subsequently used to (approximately) infer the types of other variables.

We evaluate the feasibility of this approach on four Java projects wherein type information is already available in the source code and can be used to train and test a classifier. Our experiments show this approach can predict the type of new variables with relatively high accuracy (80% F-measure). These results suggest that textual cues can be *complementary* tools in inferring types for dynamic languages.

Introduction

Several recommendation and information retrieval systems have been designed to help programmers in various software development tasks; for example, code completion (Bruch, Monperrus, and Mezini 2009), retrieval of related Stack Overflow posts (Ponzanelli et al. 2014), and recommendation of bug fixes (Chen and Kim 2015). Type information plays an important role in improving the performance of these systems.

Types define categories of entities with similar properties (Mitchell 2003), which make them highly discriminant features in any information retrieval tasks. Suppose the scenario in which a novice programmer seeks to learn about finding an element in a list of type `ArrayList` in Java. Let us consider two queries: "Java list find an element", and "Java arraylist find an element" for this information retrieval task. Figure 1 illustrates the search results of execution of queries on the Stack Overflow search engine.

In the example given in Figure 1, type `ArrayList` in the second query is specifying the data type where the sorting needs to be done. In contrast, the first query has only the general term `list`. Therefore, the results returned by Stack Overflow reflect the specificity of the query and this is clearly reflected in the number of results returned by both queries, the more general one has around five times more matches. It is likely that a useful answer is still somewhere in the list of $\sim$ 880 results returned, but the user will have to spend more time searching in that list than in the more specific query. Glancing at the results, it is obvious that the results of the second query are more relevant to the programmer's intent than the first query. It is due to the discriminant type information `ArrayList` that has narrowed down the search results significantly.

Lack of type information can significantly degrade the performance of information retrieval systems. It is particularly problematic for the dynamic programming languages that the variables are not annotated by type information.

In this paper, we explore the viability of reducing the type inference problem to a classification problem. It is based on two insights: (1) types can be viewed as labels for variables with similar properties, such as valid operations on the variables of a particular type, (2) programmers usually tend to embed some hints about the type of variables in the variable names (Martin 2009). Thus, using the textual features in the source code, a classification technique can be able to distinguish between variable with different properties.

In this paper, we evaluate the *viability* of type inference with classification based on *textual cues*. Textual cues include the identifier name, meaning, and association with other tokens in the source code. More specifically, we train and evaluate multiple models with different feature sets using two classification techniques to predict the types of variables in four Java projects from two open-source organizations. We evaluate the intra-project accuracy classifiers for various feature sets.

We choose Java because it relieves us from the burden of type annotation for training and testing of classifiers. However, it has a limitation that a variable can have only one type in a scope of the program, which is not the case in dynamic languages such as JavaScript.

This work is a part of our ongoing effort to build an information retrieval system for Python. We hope that these experiments help us in devising techniques for a reliable approximation of types in dynamically-typed languages.

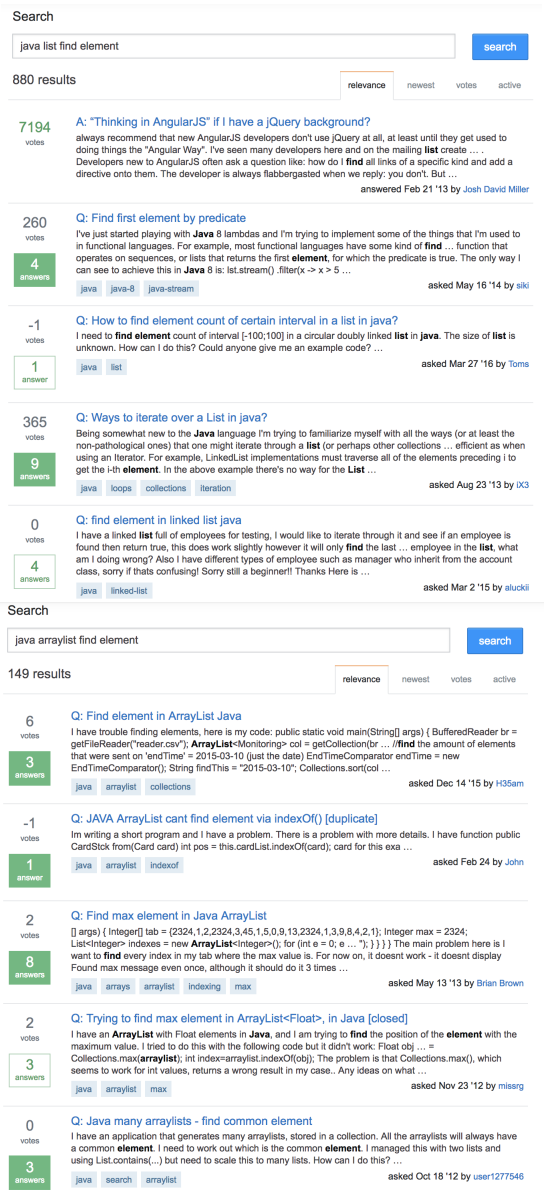


Figure 1: Search results for "Java list find element", and "Java arraylist find element", on the Stack Overflow search engine. The second query produces more relevant results.

Methodology

This section presents the classification model for type inference as well as the experimental setup for its evaluation. The classification model receives features that characterize a variable and produces a prediction of its corresponding type. In our experimental evaluation, we use Java source files from large open-source projects. Since Java is a statically-typed language, it is straightforward to determine the type of a variable by parsing the source code. The goal of the experimental evaluation is to determine whether a machine learning model is able to infer the type from a limited set of textual cues.

The representation of instances is the first step in the strategy, which involves the extraction of relevant textual features. The second step is the classification step, which is the part to infer the types. The following two sections present the aforementioned steps.

Feature Extraction and representation

Given a variable name in source code, we extract the following textual information from the text:

1. *Normalized variable names:*

   After extracting pairs of variable-type by using Java parser library, to be able to extract textual features we need to follow a couple of preprocessing steps. First, we transform all names to lower-case. In most cases, variables are in the compound form. (combination of words e.g. "myString" or "my_string") By considering different cases, we split them up into atomic words (to "my string").

2. *Associated method calls:* We add all instance methods that co-occur (by dot operator) with the variable in the source code. For example, in one sample the variable `text` typed as `String`, can have the following four different method calls in the body of function: `text.trim()`, `text.toLowerCase()`, `text.replace(x,y)`, and `text.indexOf(y)`.

Derived Features

The textual information extracted for each variable is processed to derive additional relevant features. For this, we consider traditional textual features and also automatically learned contextual features.

- We use the following traditional features: *n-gram*: n-gram is a contiguous sequence of n items from a given sequence of text. The items can be words or characters. For word sequences, an n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram"; size 3 is a "trigram" and so on. For character sequences, a char n-gram of size 1 is referred to as a "char unigram"; size 2 is a "char bigram"; size 3 is a "char trigram" and so on. We also used binary n-gram which instead of the frequency of word sequences it considers binary representation. We use different sets of traditional features on normalized variable names including unigram (U), bigram (B), trigram (T), binary unigram (bi-U), binary-bigram (bi-B), binary-trigram (bi-T), char trigram (C3), char four-gram (C4), char five-gram

```
uni-gram: helper, class,  name
bi-gram: helper class, class name
trigram: helper class name
char-trigrams: hel, elp, lpe, per, er[space], ...
```

Figure 2: Derived features for `helperClassName`

Table 1: Projects Characteristics.

Project	Organization	Version	#Pairs	#Types
CLI	Apache	1.4	323	51
IO	Apache	2.5	1156	18
Ant	Apache	1.10	12260	334
JDT	Eclipse	9.3.1	52461	2958

(C5). We also included associated method calls (bi-MC) for each variable as features.

Figure 2 illustrates a couple of derived traditional features for the normalized identifier `helperClassName`.

- We also used *word2vec*, W2V for short, of the words to capture the approximate semantic relation between terms (Mikolov et al. 2013). A W2V model embeds a word into a vector that captures the semantic value of the word in terms of the distribution of words that co-occur with it in a large corpus. For example, a vector suggested for `counter` can be very close to `numberOf` vector because they are associated with similar concepts. In this paper, we use two W2V models, one trained on Google News (Mikolov et al. 2013) and the other on Stack Overflow (Fu and Menzies 2017). Stack Overflow W2V is a domain-specific word embedding that has been trained from Stack Overflow Java text.

Classification Techniques

We use two classification techniques: Support Vector Machine, and Logistic Regression. These classification techniques are common approaches in such classification tasks. We used popular classification API for `SKLearn` with default configuration for this experiment.

Data

In this section, we describe data that used in our experiments. Table 1 illustrates the projects that we used in our experiments. In this table, #Pairs denotes the total number of pairs in training and testing, #Types is the number of types used in the projects. We chose four large, mature open-source Java projects: `CLI`, `IO`, `Ant`, `JDT` from two major open-source organizations: Apache foundation, and Eclipse foundation. `CLI` is a command line library that provides interface to integrate the command-line arguments, `IO` is a Java library to assist IO functionality such as reading and writing files. `Ant` is a Java library and command tool for building programs mainly in Java programming language. `JDT` is the Java development tool for the Eclipse integrated development environment.

Sets	Features
1	U, 'bi-U', 'C3', 'C4', 'C5', bi-MC, G-W2V
2	U, 'bi-U', 'C3', 'C4', 'C5', G-W2V
3	U, 'bi-U', 'C3', 'C4', 'C5', bi-MC, SO-W2V
4	U, 'bi-U', 'C3', 'C4', 'C5', SO-W2V
5	U, 'bi-U', 'C3', 'C4', 'C5', bi-MC, G-W2V, SO-W2V
6	U, 'bi-U', 'C3', 'C4', 'C5', G-W2V, SO-W2V

Table 2: Features used in each Features Sets. In the table, U=Unigram, bi-U=Binary unigram, C3=Char tri-gram, C4=Char four-gram, C5=Char five-gram, bi-MC= Binary method call, G-W2V=Google W2V, SO-W2V=Stack Overflow W2V

Results and Evaluation

In this section, we present and discuss the results of the experiments. We evaluate the performance of different feature sets using two classifiers. We train the classifiers on 60% of data and we test them on the remaining 40%.

Table 2 describes the feature sets that we evaluated in this paper. We tried feature sets that empirically found outperform other sets.

Figure 3 compares the results of the classification using six selected feature sets. We use weighted F-measure, which is a common performance metric for classifiers.

Research Questions We seek to answer the following research questions.

1. How different feature sets perform in type inference?
2. Which models perform better for type inference?

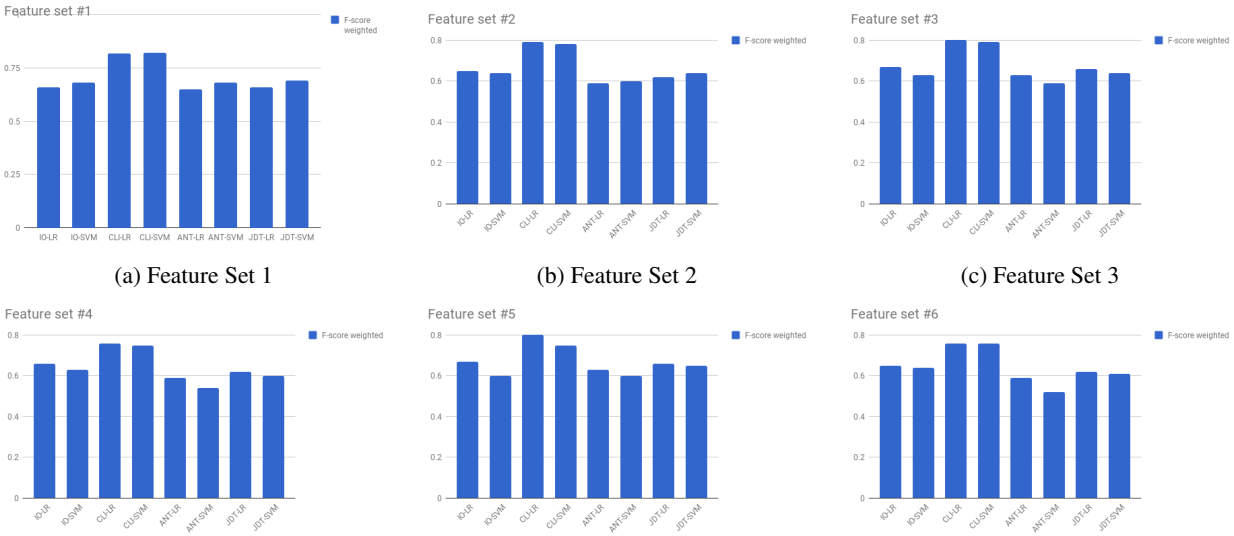
Performance of Feature sets

Models based on feature sets 2,4 and 6, which do not include information about the associated methods manifest lower accuracy compared to models based on feature sets 1, 3, and 5. Intuitively, the set of associated functions represent the operations allowed on the variable which can be a good predictor of the type of the variable.

In our experiments, W2V improves the overall accuracy of models which is the reason that we included it as one of the main features in classification. Between the feature sets using W2V based on Stack Overflow data and W2V based on Google News, there is no noticeable difference in the accuracy of models. It may suggest that *there is little need to train domain-specific V2W models and the models based on natural language can be used instead.*

Performance of Classifiers

In our experiments, models could predict a large portion of types in the test data (F-measure higher than 0.6 and as high as 0.8). In our experiments, logistic regression performs slightly better than SVM for most feature sets. It can be due to the sparseness of the representation of a large number of features used in classification.



(a) Feature Set 1 (b) Feature Set 2 (c) Feature Set 3

(d) Feature Set 4 (e) Feature Set 5 (f) Feature Set 6

Figure 3: Weighted f-measures of different feature sets. Columns ending with LR and SVM denotes F-measure for Logistic Regression and SVM models, respectively.

Related Work

The most related work are (Raychev, Vechev, and Krause) and (Xu et al. 2016) that use probabilistic graphical models to statistically infer types of identifiers in JavaScript and Python programming languages, respectively. These works take into account the control and data dependence into code. In contrast, we treat the programs as natural text and ignore the underlying semantic and syntactic relations between elements.

Conclusion and Future Work

We presented the approach of type inference with textual cues. We evaluated different models with various configurations for predicting the type of variables in a program using the textual features in the source code. Our results suggest that textual features can be good predictors of types.

In this work, we only evaluated the performance of classification for type classification in intra-projects. We evaluated the performance of classification in across-projects and across-organizations settings in a recent technical report (Shirani et al. 2017) where we discuss the limits and opportunities in carrying over the type classification model of a project to another project. Here, we evaluated our model using Java projects. However, our final goal is to devise such system for dynamic languages, particularly for Python.

In this work, we used only two classic classifiers with ad-hoc feature selection. In future, we plan to try other techniques such as random forests, and neural networks. Furthermore, we plan to use a more systematic feature selection approach such as (Bhalerao and Rajpoot 2003) to find the more effective feature sets.

Acknowledgements We thank Andrew Truelove and Victor Florintsev for comments on the earlier drafts of this paper.

References

Bhalerao, A. H., and Rajpoot, N. M. 2003. Discriminant feature selection for texture classification. In *(BMVC2003*.

Bruch, M.; Monperrus, M.; and Mezini, M. 2009. Learning from examples to improve code completion systems. In *ESEC/FSE '09*.

Chen, F., and Kim, S. 2015. Crowd debugging. In *ESEC/FSE 2015*.

Fu, W., and Menzies, T. 2017. Easy over hard: A case study on deep learning. In *ESEC/FSE 2017*.

Martin, R. C. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119.

Mitchell, J. C. 2003. *Concepts in programming languages*. Cambridge University Press.

Ponzanelli, L.; Bavota, G.; Di Penta, M.; Oliveto, R.; and Lanza, M. 2014. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *MSR 2014*.

Raychev, V.; Vechev, M.; and Krause, A. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50.

Shirani, A.; Lopez-Monroy, A. P.; Gonzalez, F.; Solorio, T.; and Alipour, M. A. 2017. Type inference without a type system; evaluation of type prediction with textual hints. Technical report, University of Houston, Department of Computer Science.

Xu, Z.; Zhang, X.; Chen, L.; Pei, K.; and Xu, B. 2016. Python probabilistic type inference with natural language support. In *FSE 2016*.