InvocMap: Mapping Method Names to Method Invocations via Machine Learning

Anonymous Author(s)

Abstract

Implementing correct method invocation is an important task for software developers. However, this is a challenging work, since the structure of method invocation can be complicated. In this work, we propose InvocMap, a code completion tool that allows developers to get the correct implementation structure of multiple method invocations from a list of method names inside code context. InvocMap can predict the nested method invocations which their names didn't appear in the list of input method names given by developers. To achieve this, we analyze the Method Invocations by four levels of abstraction. We build a Machine Translation engine to learn the mapping from first level to the third level of abstraction of multiple method invocations, which only requires developers to manually add local variables from generated expression to get the final code. We evaluate our proposed approach on six popular libraries: JDK, Android, GWT, Joda-Time, Hibernate, and Xstream. With the training corpus of 2.86 million method invocations extracted from 1000 Java Github projects and the testing corpus extracted from 120 online forums code snippets, InvocMap achieves accuracy up to 84 in F1-score depending on how much information of context provided along with method names, which shows its potential for auto code completion.

Introduction

Writing code is a challenge for non-experienced software developers. To write the code that implements a specific task in a programming language, developers need to remember the syntax of that language and be familiar with how to implement method invocations. While the syntax of the language is easier to learn since it contains a permanent set of words in the vocabulary, implementing Method Invocations (MI)s is more challenging due to the following reasons. First, developers need to remember the structure and the combination of invocations depending on their purpose. Second, the implementation of method invocation is also depending on the surrounding context of the code. These challenges cause the code developed by non-experience developers to be in the risks of being semantic error.

To help developers with interacting and analyzing a given Java source code snippet, Java Development Tool (JDT) library defines a list of Abstract Syntax Tree (AST) Node types (Eclipse 2019). With the list of these AST Node types, JDT can be able to interact with the structure of each elements inside the source code. Method invocation, which is defined as sub-type of Expression, is one of the fundamental AST Node that developers need to implement. MI has been used to make Application Programming Interface (API) calls from other libraries or from other methods inside a Java project. The structure of a syntactically correct MI contains method name, receiver and the list of arguments which could be empty. Since receiver and arguments are types of expression (Eclipse 2019), the structure of an MI could be complicated as a deep AST tree since expression can be composed by different types of AST Node including MI.

Listing 1: Example of invocation of API `java.awt.Rectangle.set()` in (Android 2019c)

```
1  public void setOffsets(int newHorizontalOffset, int
       newVerticalOffset) {
2      ...
3          if (mView != null) {
4              ...
5              invalidateRectf.offset(-xoffset, -yoffset);
6              invalidateRect.set((int) Math.floor(invalidateRectf.
                   left),(int) Math.floor(invalidateRectf.top),(int
                   ) Math.ceil(invalidateRectf.right),(int) Math.
                   ceil(invalidateRectf.bottom));
7              ...
8          }
9      ...
10 }
```

An example of a complicated MI is shown in Listing 1. In this Listing, the outside MI contains four nested MI in its implementation. Besides, there are 5 positions that requires local variables inside the expression. Type casting to integer is embedded to this MI to provide a semantically correct MI. This MI is used along with other calculated MIs inside the body of method, providing the a specific surrounding context for this MI. Certainly, though the outer method name `set` is just one word, the respected method invocation is a deep AST tree.

The representation of method invocation is also depending on code context. Consider examples 2A and 2B on Listing 2 and Listing 3.

These Listings show the implementation of API `android.content.Intent.getBooleanExtra()`. Although 2 MIs share the same information about context of using the same local variable `Intent` and the `false` boolean literal, they are differ in the structure of AST. Since the MI in Listing 2 associates with the action of add or remove an application package from an android device, the MI on Listing 3 associates with actions of network status checking. The difference in contexts brings 2 MIs, which represents in 2 static Field Accesses `Intent.EXTRA_REPLACING` and `ConnectivityManager.EXTRA_NO_CONNECTIVITY`.

Listing 2: Example 2A of invocation of API `android.content.Intent.getBooleanExtra()` in (Android 2019a)

```
public void onReceive(Context context, Intent intent) {
    ...
    if ((Intent.ACTION_PACKAGE_REMOVED.equals(action) ||
            Intent.ACTION_PACKAGE_ADDED.equals(action))
            && !intent.getBooleanExtra(Intent.EXTRA_REPLACING,
                    false)) {
        ...
    }
}
```

Listing 3: Example 2B of invocation of API `android.content.Intent.getBooleanExtra()` in (Android 2019b)

```
public void onReceive(Context context, Intent intent) {
    ...
    if (activeNetwork == null) {
        ...
    } else if (activeNetwork.getType() == networkType) {
        mNetworkUnmetered = false;
        mNetworkConnected = !intent.getBooleanExtra(
                ConnectivityManager.EXTRA_NO_CONNECTIVITY, false);
        ...
    }

}
```

From these examples, we see that implementing an effective method invocation requires strong background and experiences from developers. Even two Method Invocations that belongs to the same API and share the same context of local variables and literal stills have ambiguous in the way of implementation like Listing 2 and Listing 3. These challenges hinders the ability of writing a appropriate MI and cause developers to spend time to remember or finding the correct structure of AST in MI for software development.

In this work, we want to tackle this problem by providing InvocMap, a code completion tool for helping developers to achieve the implementation of method invocation efficiently. InvocMap accepts input as a sequence of method names inside the code environment of a method declaration, then produce the output as the list of ASTs as translation results for each input method names. The generated ASTs will only require developers to input information about local variables and literals to get the complete code. For example, in Listing 2, developer can write the list of method names including the name `getBooleanExtra`. The output for the suggestion for `set` will be `#.getBooleanExtra(Intent.EXTRA_REPLACING,#)`, which can be completed manually by a variable of type `android.content.Intent` in the first `"#"` and a boolean literal in the second `"#"`.

Statistical Machine Translation (SMT) is a well-known approach in Natural Language Processing (NLP) for translating between languages (Green, Cer, and Manning 2014). To take advantage from SMT, we propose a direction for code completion for Method Invocation by a Statistical approach, which learn the translation from the abstract information of MIs to the their detail information, which are represented by AST with complicate structure. First, we analyze the information inside a typical MI. We divide the MI by four levels of abstraction. We also define information of context for each MI which can help to predict the AST structure. Next, we build an SMT engine specified for our work to infer from the very abstract layer of MI, means Method Name, to the third level of MI, which is an AST tree that requires to be fulfill by local variables and literals. To evaluate our approach, we do experiments to check the accuracy of our code completion technique in two datasets collected from Github and from online forums. This paper has following contributions:

1. Design a strategy for classifying Method Invocation by levels of abstraction, from very abstract to details level.
2. Design rules for extracting code tokens for representing abstract level and details level for various types of AST nodes.
3. Proposing an algorithm for visiting an method invocation inside the code environment to abstract and encode their structure in AST as an object for statistical learning.
4. Building a Phrase based Statistical Machine Translation (PBMT) system for learning from the context of code environment, including Method Invocations from large scale Github high quality projects. This PBMT system is able to predict the sequences of AST structure given sequences of method name and context.

Resources of this paper can be found at (InvocMap 2019).

Approach

Overview

We summarize the engines inside InvocMap on Figure 1. From developer view, InvocMap provides a plugin inside with Java code editor to allow (s)he to write a singleor multiple method namess inside the code environment. From this input, InvocMap translates each method names to respective ASTs. These ASTs reflect the complex structure of method invocations which might be inconvenient for developers to remember. They were abstracted at level 3 in our definition, means they only require developers to add local variables, local methods or literals to get the final code. We will discuss about MI at level 3 of abstraction in the next section. The ability of inferring ASTs for code completion relies on the Statistical Translation module. The training process is done by the Statistical Learning module. This module learns information from the data extracted from large scale Github code corpus (Github 2019). In general, our statistical approach takes advantages of the knowledge of implementing MIs of experience developers, representing it by a machine

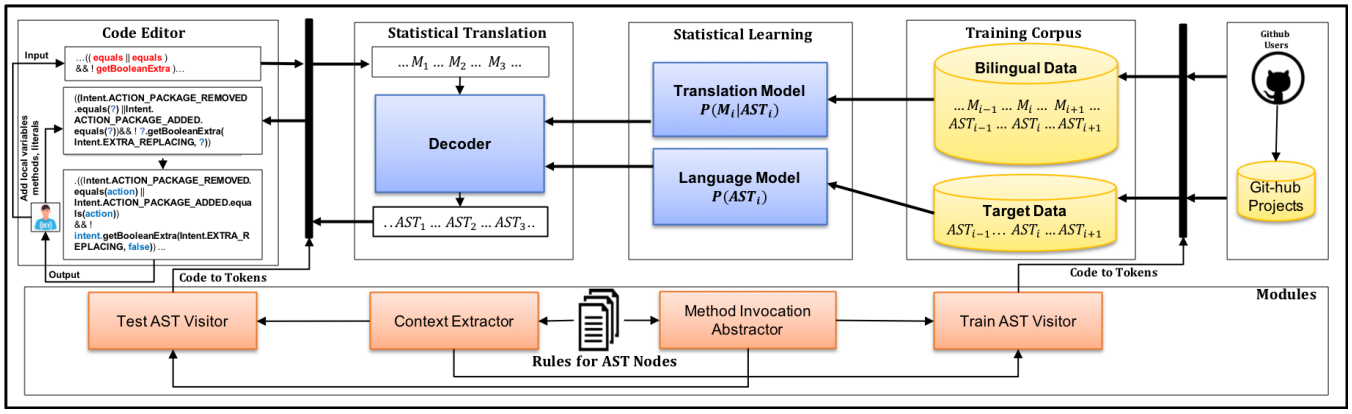

Figure 1: Overview of InvocMap Approach

learning model to help non-experience developers in retrieving effective implementation of MIs. Both the source code at developers side and code corpus are analyzed to extract sequences of tokens by the Train AST Visitor and Test AST Visitor module we developed. Inside these visitors, we handle each AST Node types by function of modules Context Extractor and MI Abstractor, which we discuss in next sections.

Levels of MI Abstraction in InvocMap

In this section, we describe our theorems for method invocation abstraction.

Definition 1 *Level 1 of abstraction of a method invocation is the information about method name of that method invocation.*

Definition 2 *Level 2 of abstraction of a method invocation is the information about type (or signature) of that method invocation.*

Definition 3 *Level 3 of abstraction of a method invocation is the Abstract Syntax Tree of that method invocation with abstracted place holder for local variables, local methods and literal.*

Definition 4 *Level 4 of abstraction of a method invocation is the complete Abstract Syntax Tree of that method invocation.*

Along with 4 levels of abstraction in MI, we have the definition of local context provided for each MI.

Definition 5 *Local Context of a method invocation is the information about types of local entities and suggested terms. Local entities are local variables, local method invocations and literals inside the MI. The suggested terms are the words that appeared inside the MI.'*

An example of how we see 4 levels is shown in Figure 2. In this code snippet, we have level 1 as method name `println`. The level 2 of abstraction brings us information about type, which is `java.io.PrintStream.println`.The level 4 is the final source code which is compilable. The layer 3 is the AST that is having 2 places which are local entities are abstracted by their type information. In the implementation, we represent this AST in level 3 by 4 fields: the code with abstracted places for local entities, the list of types of required arguments to add, the list of imported APIs and the type of MI. These 4 fields will make an unique identification for the expression, which will serve as a representative token for the AST. Because of this, when developer receives an AST at level 3 of abstraction, (s)he will know what types of local variables (s)he can get to get the final code, along with the set of imported APIs. In our work, we focus on the inference from level 1 to level 3 by translation. We will use information of local context to help developers who already remember what variables should run inside the MI and some words inside the MI to better retrieve the AST of implementation. In Figure 2, we see 2 local entities, including the string literal `"index"` and the integer variable `i`. The suggested terms can be `"System"` and `"+"` sign.

Levels of Abstraction in Other AST Nodes

In the context of this work, we call other AST Nodes as all kinds of AST except the MI that are defined in (Eclipse 2019).

Definition 6 *Level 1 of abstraction of other AST Nodes is the information about the Partial Qualified Name (PQN) of type of those nodes.*

Definition 7 *Level 2 of abstraction of other AST Nodes is the information about Fully Qualified Name (FQN) of type of those nodes.*

According to definition in (Phan et al. 2018), an example is the API `java.io.File`. In this API, we have `File` as PQN while we have `java.io.File` as FQN.

Source and Target Tokens for Other AST Nodes We extract information about other AST Nodes to provide useful context for MIs prediction. In the source language, we extract all tokens of level 1 of abstraction for each AST Node, and extract all tokens in level 2 of that AST Node to put into target language. The implementation of the extraction is the Context Extractor module, which is called inside Train AST

Table 1: Rules for extract source and target side for AST Nodes

AST Node	Syntax / Source()/ Destination()
Instanceof Expr	E1 instanceof T1
	S(E1) instanceof PQN(T1)
	D(E1) instanceof FQN(T1)
SimpleName	I1
	PQN(I1)
	FQN(I1)

Visitor and Test AST Visitor. The rules for handling each type of AST are shown in Table 1. In this Table, the `PQN()` and `FQN()` ran when the AST Visitor visit elements that have type. The function for extracting tokens `S()` and `D()` can be recursive to sub elements of a given AST Node.

Source and Target Tokens for Method Invocations There are two types of information we want to embed for MI: the mapping between method name and the AST and the information relate to local context. For the first type of information, the source language will store information about token as level 1 of abstraction of MI, while the target language stores information about level 3 of abstraction of MI. Besides, information about local context will be stored by level 1 of abstraction in the source and level 2 of abstraction in the target language. A sequence of tokens for MI in Figure 2 is shown in Figure 3.

Method Invocation Abstraction We get information about level 3 of abstraction in MI by proposing an algorithm in Listing 4. The `abstractMethodInvocation()` function is invoked when the Train AST Visitor or Test AST Visitor visit a MI and return the abstraction in level 3 by an instance of AST_Level3 class. This function will use the child class of ASTVisitor called MICVisitor defined in Listing 4 (Line #45). This visitor will visit each element inside the MI, check and abstract if the element is a local entity. This visitor also stores other information about the code of AST, the list of required types for each local entities and the set of imported APIs. The handling strategy for each types of AST Node inside the MI is implemented in the `visitStructure()` function at Line #61. After visiting and abstracting of MI to an `AST_Level3`, this object is checked by the first four fields defined in line #32 to line #38 to see if its exist in the dictionary or not. If yes, it will have the id of the existing object in the dictionary. Otherwise, it will generate a new unique id and will be added to the dictionary. The dictionary stores information about abstraction at layer 3 of MIs in the training step. An example of `AST_Level3` object is shown in Figure 2.

Listing 4: Algorithm for Method Invocation Abstraction

```
1  AST_Level3 abstractMethodInvocation(
2    mi : MethodInvocation ,
3    dictionaryAST : Set<AST_Level3>,
4    visitor : MICVisitor) {
5   AST_Level3 result=new AST_Level3(mi);
6   visitor.result=result;
7
8   //visit receiver if exist
9   Expression receiverExpr=getReceiver(mi);
10  if(receiverExpr not null)
11    visitor.visit(receiverExpr);
12
13  // add method name and open parenthesis
14  result.strCode.append(getMethodName(mi)+"(");
15
16  // visit content of arguments
17  Expression[] listArguments=getParams(mi);
18  for each Expression argExpr in listArguments:
19    visitor.visit(argExpr);
20
21  // add close parenthesis
22  result.strCode.append(")");
23
24  // set uniqueId
25  setUniqueId(result,dictionaryAST);
26
27  return result;
28 }
29 ...
30 class AST_Level3 {
31  // field to store AST in level 3
32  strCode : String;
33
34  // Other informations need to
35  // add to get final code
36  listArguments : List<Type>;
37  setImportedAPIs : Set<Type>;
38  strSignature : Type;
39
40  // id is created by the uniqueness of 4 other fields
41  uniqueId : String;
42
43 }
44 ...
45 class MICVisitor extends ASTVisitor{
46  result : AST_level3;
47  ...
48  void visit(ASTNode node) {
49   /*
50    * If visit local entity,
51    * mark as a place holder
52    * in the AST structure
53    */
54   if(isLocalEntity(node)) {
55    result.strCode.append("#");
56    result.listArguments.append(getType(node));
57   }
```

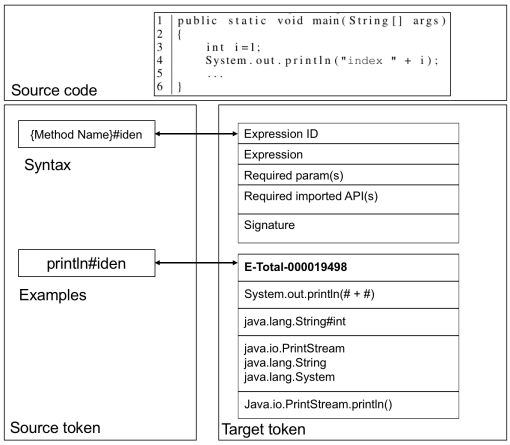

Figure 2: Example of AST_Level3 instance

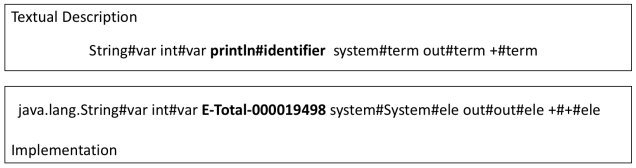

Figure 3: Example of tokens for source and target in Method Invocation

```
58      else {
59        // visit structure and update result
60        // can be recursive if node have children nodes
61        visitStructure(node);
62      }
63      // In every cases, add type of node to set of imported APIs
64      result.setImportedAPIs.add(getType(node));
65    }
66  }
```

SMT

To learn the mapping between source and target language, we apply the PBMT (Green, Cer, and Manning 2014). This approach works based on the ability to learn from two models: the language model and the translation model.

Language Model Language Model (LM) plays an important role in a PBMT system. LM is used to predict the next token given a sequence of previous tokens (Koehn, Och, and Marcu 2003). The more comprehensive corpus of target language do we have, the LM model quality for prediction is higher. LM had been used widely in Software Engineering (SE) researches (Hindle et al. 2012; Hellendoorn, Devanbu, and Bacchelli 2015; Liu 2016) with potential results. The most basic LM in NLP is uni-gram LM, which calculates the probability of each word based on the number of the appearance of that word in the corpus. This LM provides drawbacks that it doesn't take into account the history of how a word was used in a sequence from previous words. However, if we calculate the probability of a word from all previous words in a sequence of translation, the cost for doing such evaluation is expensive ((Jurafsky and Martin 2009)).
Here we use the n-gram language model, which proposed in (Jurafsky and Martin 2009). The n-gram LM is used to measure the probability of implementation of MI given a list of previous implementations. The n-gram LM overcome the cost of measuring history for each word by considering and approximate the history for the last n-words that appeared before the current word. Assume that we have m tokens in the target language $AST_1, ..., AST_m$, the probability provided by LM is:

$$P[AST_1, ..., AST_m] = \prod_{i=1}^{m} P[AST_i | AST_{i-n}, ..., AST_{i-1}] \quad (1)$$

In equation 1, the probability for each element of the production on the right side is estimated by the co-occurrence of the current word given the list of the previous words in the target corpus ((Jurafsky and Martin 2009)). In our model, we use 4-gram LM as the default configuration of LM in NLP.

Translation Model The translation model calculates the probability of a phrase from source language that can be translated to a phrase in a target language. In our problem, if we have a sentence D as the translated result of sentence S as tokens in the source language, the selection of D as the best candidate is calculated as follows:

$$D_{best} = argmax_D(p(S|D) * P_{LM}(D))) \quad (2)$$

Table 2: Statistics on Corpus

Library	Num of methods
JDK	820386
GWT	170435
Joda-Time	91072
Android	312822
Hibernate	149887
Xstream	159170

In equation 2, $p(S|D)$ is the probability of translation model to get the best accuracy calculated by the work of 3. In this formula, I is the number of phrase appeared in sentence D, while $\theta(s_i|d_i)$ is the probability functions between phrase calculated by (Green, Cer, and Manning 2014). Compare to the original translation between natural languages, the translation model in NL has an another type of distribution called reordering distribution. Since our corpus has consistent length between source and target, we don't apply this distribution to equation 3.

$$p(S|D) = \prod_{i=1}^{I} \theta(s_i|d_i) \quad (3)$$

Evaluation

Data Preparation and Metrics

To do the evaluation, we select corpus on six well-known libraries. They are Java Development Kit (JDK), Android, GWT, Joda-Time, Hibernate, and XStream. These libraries were selected to generate the corpus for other research works ((Phan et al. 2018; Subramanian, Inozemtseva, and Holmes 2014)). To generate corpus, we select 1000 highest stars Java projects from Github ((Github 2019)), which have most files used API from libraries in Table 2. For each Java project, InvocMap parses each Java source files by an the Train ASTVisitor module on Figure 1. We apply our solution using Phrasal (Green, Cer, and Manning 2014).The number of methods we collect is shown in Table 2.

Training Configuration To train the SMT model, we use a high-end computer with core-i7 Intel processor and use 32 GB of memory. We allocate Phrasal with phrase length equals to 7. The total training time requires about 6 hours.

Test Configuration We evaluate the ability of translation from a sequence of method names to ASTs in level 3 of abstraction. We simulate 3 configurations sequences of method names regarding to its local context defined in Table 3.

We can see the local context provided for method names is increasing from configurations at level 1 to level 3. At level 1, the input for translation contains only method names with the code context in the source language for translation. It simulates the case that developers write a list of method names inside the code environment. At level 2, information about partial class name of types of local entities is attached along with each method names. This case simulates the case developers remember and write method name and local variables (s)he needs to implement the MI but (s)he doesn't remember the structure of AST. At

Table 3: 3 Testing Configurations for Developers

	Meaning / Example
Config 1	Only method name
	println
Config 2	method name and variables
	println 10 5
Config 3	method name, variables and some suggested words
	println 10 + 5
Expected	System.out.println(#+#) Params: int and int

level 3, each method names in the source language will be attached the information about local entities and half of words appeared inside the MI. This case simulates the case that developers remember some words inside the MI along with local entities. In Table 3, at level 3 developer input `println` as method name, 2 integers as variables and an operator `+`. The input for predicting the MI is `int#var int#var println#identifier +#term`. And the translation engine needs to return as MI respected to `println#identifier`.

Metrics We evaluate the correct inference of method names. Information about tokens of method name can be recognized by the annotation `#identifier` for the source, and the expected results can be recognized by prefix `"E-Total"` of tokens in the target. We use Precision and Recall as 2 metrics for the evaluation. We category cases of predicting tokens of method name: True Positive (TP)- means the prediction is correct; False Positive (FP) - means the expected expression is in the training corpus but the translated expression is different and False Negative (FN) - means the expected expression is out of vocabulary (OOV). Out of Vocabulary is the case that the method name token does not in the corpus (Out of Source - OOS) or the expected expression token does not appear in the target corpus (Out of Target - OOV).

Research Question (RQ) 1: How InvocMap can perform to predict the implementation with Intrinsic Data?

We split the pairs of our parallel corpus for training and testing. We get 10% of the data for testing and the other with training and do ten-fold cross-validation to test the ability of prediction on our full data set. In total, there are 2.86 Million of MIs collected from 1000 projects from Github (Github 2019). The evaluation result for intrinsic data is shown in Table 4. We show that from configuration 1 to configuration 3, the F1 score increases from 73.06% to 84.4%. This seems to be feasible, since the fact that if we provide more local context information along with method names, the ability to predict correctly AST in level 3 for the translation model is better. We see one observation is that the number of Out of Vocabulary expressions are higher in percentage, cause decreasing in recall compare to the research work that applied Machine Translation for inferring Fully Qualified Name from incomplete code ((Phan et al. 2018)).

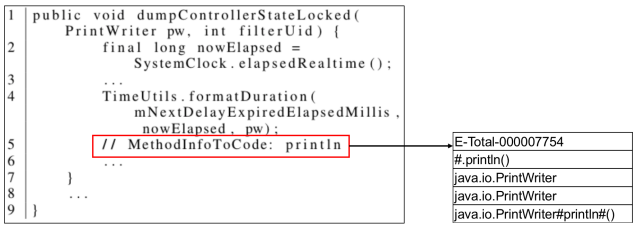

Figure 4: Example of query in configuration 1 for code snippet in (Nexus 2019)

This is reasonable, since our work requires to infer the MI in level 3 of abstraction, which contains detail structure compared to output of (Phan et al. 2018) as type information of MIs. We study an example in the Intrinsic Evaluation in Figure 4. This example is a function collected from (Nexus 2019) from our corpus. The testing for intrinsic evaluation simulates the case developers input only `println` inside the code environment, the output of this case will be the implementation of `java.io.PrintWriter.println()` function. We can see that the surrounding code is useful to infer the correct expression. If we do not have the context information, which means developer input `println` in an empty method, the translated result will return the most popular MI, `System.out.println()`.

RQ2: How well InvocMap can perform to predict the implementation with Extrinsic Data?

To do this experiment, we collect the data as code snippets from Online Forums ((StackOverflow 2019; ProgramCreek 2019; GeeksForGeeks 2019)). To do this, a Software Engineer who has 5 years of experience in Java programming was hired to collect code snippets from 120 posts in Online Forums, with 20 posts for each library in Table 2. Each code snippets are required to have the MIs of 6 libraries: JDK, Android, GWT, Joda-Time, Hibernate, and XStream. We run our Test ASTVisitor module to extract the source tokens with embedding MIs at 3 levels and use the SMT model learned from our parallel corpus for the prediction.

The result for extrinsic evaluation is shown in Table 5. We see that with level 1, since the case that only method names are provided in the source language, our approach stills predict correctly 62.54% in precision. With the configuration levels that developers add more information, the precision increases to 86.88%. For each library, we achieved the highest accuracy on GWT and lowest on Hibernate with input as detail information like configuration 3. This result seems reasonable, since Hibernate is a bigger library compared to GWT but it is not as popular as JDK, causes the variety of ASTs for APIs in this library.

RQ3: How well InvocMap can perform to predict ambiguous method names?

In this evaluation, we analyze the relation of the expression prediction result relates to the number of mapping of each method name from the parallel corpus. We use data collected for the Intrinsic Evaluation with configuration 3, means each

Table 4: Intrinsic Evaluation Result on Github projects

	Intrinsic Evaluation with Configuration 1								
Library	**Correct**	**Incorrect**	**OOSource**	**OOTarget**	**OOVoc**	**Total**	**Precision**	**Recall**	**F1-Score**
GWT	39635	22318	3082	28440	31522	93475	63.98%	55.70%	59.55%
Joda-Time	27364	10608	51	1692	1743	39715	72.06%	94.01%	81.59%
JDK	1053330	540997	3691	390626	394317	1988644	66.07%	72.76%	69.25%
Android	471347	91753	5654	48662	54316	617416	83.71%	89.67%	86.58%
Hibernate	53319	25305	4787	34090	38877	117501	67.82%	57.83%	62.43%
Xstream	4671	1692	70	2949	3019	9382	73.41%	60.74%	66.48%
Total	1649666	692673	17335	506459	523794	2866133	70.43%	75.90%	73.06%
	Intrinsic Evaluation with Configuration 2								
Library	**Correct**	**Incorrect**	**OOSource**	**OOTarget**	**OOVoc**	**Total**	**Precision**	**Recall**	**F1-Score**
GWT	53042	8911	3082	28440	31522	93475	85.62%	62.72%	72.40%
Joda-Time	29028	8944	51	1692	1743	39715	76.45%	94.34%	84.45%
JDK	1347221	247064	3691	390668	394359	1988644	84.50%	77.36%	80.77%
Android	470725	92370	5654	48667	54321	617416	83.60%	89.65%	86.52%
Hibernate	63275	15345	4787	34094	38881	117501	80.48%	61.94%	70.00%
Xstream	5145	1218	70	2949	3019	9382	80.86%	63.02%	70.83%
Total	1968436	373852	17335	506510	523845	2866133	84.04%	78.98%	81.43%
	Intrinsic Evaluation with Configuration 3								
Library	**Correct**	**Incorrect**	**OOSource**	**OOTarget**	**OOVoc**	**Total**	**Precision**	**Recall**	**F1-Score**
GWT	55510	6443	3082	28440	31522	93475	89.60%	63.78%	74.52%
Joda-Time	31394	6578	51	1692	1743	39715	82.68%	94.74%	88.30%
JDK	1435424	158861	3691	390668	394359	1988644	90.04%	78.45%	83.84%
Android	498708	64387	5654	48667	54321	617416	88.57%	90.18%	89.36%
Hibernate	65860	12760	4787	34094	38881	117501	83.77%	62.88%	71.84%
Xstream	5516	847	70	2949	3019	9382	86.69%	64.63%	74.05%
Total	2092412	249876	17335	506510	523845	2866133	89.33%	79.98%	84.40%

method names are embedded with local entities and local terms in the source language. The result is shown in Table 6.

The result shows that from the number of method name that has more than 100 mappings in the parallel corpus are about 72% of the total data. It proves the complexity of kinds of implementation for each method names. The total precision tends to decrease from 96.47 % to 87.68%. This proves the ability to generate expression from input queries.

Discussion

There are some threats to validity of our work. First, the data might not be representative. To alleviate this threat, we select high quality Github projects for training by selecting highest stars project which uses six popular libraries and select reliable online forums to get the code corpus. Second, we evaluate the accuracy of the translation system automatically. We consider the input for translation as all method names and evaluate the accuracy of their respected ASTs. In reality, developers might write full implementation of MI if (s)he remembered how to implement that function along with writing method names in case they need suggestion. Thus, the method names of already implemented MIs can be skipped if we consider about the accuracy of the inference of method names that developers need the suggestion. We will hire people to set up this type of evaluation in future.

There are a few challenges that we need to improve for the translation. First, we still require developers to remember the method names for the inference. This could be a challenges for developers at a very beginning level of expertise. We consider that the more appropriate input can be Natural Language (NL) text descriptions by developers instead of method names and local entities. An interesting future work should be extending the original SMT model to accept natural language description by processing the description to have the most appropriate method names and local entities automatically for the input of method name to AST translation. Second, since in reality developers can write the full MIs along with method names, InvocMap still doesn't take advantage of information provided by implemented MIs. This brings us to improve our work by a specific Machine Translation research for the case that some of source tokens we already know the result, and how we use those information to increase the accuracy, which can be useful in applying SMT in other areas. Third, our SMT model still not be able to generate the MI at the fully details level. We will combine SMT with program analysis for the inference from level 1 to level 4 of MIs in future works.

Related Works

Machine Learning has been applied widely in Software Engineering applications (Allamanis et al. 2018). Generating code by machine learning is an interesting but also confront challenges. There is a research by (Barone and Sennrich 2017) shows that the inference of code from documentation by machine translation achieved very low accuracy results on both SMT and Neural Machine Translation (NMT) models learned from practical large scale code corpus. There

Table 5: Extrinsic Evaluation Result on Online Forum Code Snippets

	Extrinsic Evaluation with Configuration 1								
Library	**Correct**	**Incorrect**	**OOSource**	**OOTarget**	**OOVoc**	**Total**	**Precision**	**Recall**	**F1-Score**
GWT	58	35	0	9	9	102	62.37%	86.57%	72.50%
Joda-Time	36	22	2	15	17	75	62.07%	67.92%	64.86%
JDK	115	91	0	44	44	250	55.83%	72.33%	63.01%
Android	51	42	0	13	13	106	54.84%	79.69%	64.97%
Hibernate	125	61	1	39	40	226	67.20%	75.76%	71.23%
Xstream	44	6	0	14	14	64	88.00%	75.86%	81.48%
Total	429	257	3	134	137	823	62.54%	75.80%	68.53%
	Extrinsic Evaluation with Configuration 2								
Library	**Correct**	**Incorrect**	**OOSource**	**OOTarget**	**OOVoc**	**Total**	**Precision**	**Recall**	**F1-Score**
GWT	88	5	0	9	9	102	94.62%	90.72%	92.63%
Joda-Time	53	5	2	15	17	75	91.38%	75.71%	82.81%
JDK	177	29	0	44	44	250	85.92%	80.09%	82.90%
Android	85	8	0	13	13	106	91.40%	86.73%	89.01%
Hibernate	138	48	1	39	40	226	74.19%	77.53%	75.82%
Xstream	49	1	0	14	14	64	98.00%	77.78%	86.73%
Total	590	96	3	134	137	823	86.01%	81.16%	83.51%
	Extrinsic Evaluation with Configuration 3								
Library	**Correct**	**Incorrect**	**OOSource**	**OOTarget**	**OOVoc**	**Total**	**Precision**	**Recall**	**F1-Score**
GWT	89	4	0	9	9	102	95.70%	90.82%	93.19%
Joda-Time	55	3	2	15	17	75	94.83%	76.39%	84.62%
JDK	174	32	0	44	44	250	84.47%	79.82%	82.08%
Android	82	11	0	13	13	106	88.17%	86.32%	87.23%
Hibernate	146	40	1	39	40	226	78.49%	78.49%	78.49%
Xstream	50	0	0	14	14	64	100.00%	78.13%	87.72%
Total	596	90	3	134	137	823	86.88%	81.31%	84.00%

Table 6: Analyze Relation between Accuracy and Num of distinct mapping of Config 3

Num of mapping	**1-10**	**11-20**	**21-50**	**50-100**	**>100**
Percentage	12.30%	4.20%	5.96%	4.90%	72.64%
Accuracy	**Prec**	**Prec**	**Prec**	**Prec**	**Prec**
GWT	96.58%	86.83%	88.71%	86.89%	86.59%
Joda-Time	93.38%	89.64%	76.19%	69.43%	74.11%
JDK	98.14%	96.02%	95.26%	91.92%	89.06%
Android	96.24%	92.51%	90.63%	89.08%	82.58%
Hibernate	92.61%	87.60%	87.50%	85.47%	78.98%
Xstream	97.78%	91.01%	78.81%	81.02%	80.80%
Total	96.47%	93.07%	92.05%	89.41%	87.68%

are two reasons cause this challenges. First, large scale code corpus contains noise data (Pascarella and Bacchelli 2017). Second, the structure of AST Node is complicate for a machine translation system to learn about the syntactically correct of generated code as shown in (Barone and Sennrich 2017). (Gu et al. 2016) propose an approach to achieve the implementation from in natural language description. However, the output of their tool consists only sequence of APIs which is in level 2 of our abstraction for MIs. In our work, we target the inference of MI in level 3 with the ability of complex AST structure of MIs.

There are several other inputs to get the complete code in other researches. (Nguyen, Nguyen, and Nguyen 2015) derive the code in C# language from code in Java language by machine translation. (Gvero and Kuncak 2015; Gu et al. 2016) generate the code from natural language descriptions. In these works, they consider the textual description as the full information for the inference. We consider our code generation problem in a different angle, which we take advantage of the surrounding context along with the textual description of method name in our work. (Nguyen et al. 2012) propose a graph based code completion tool that suggest the full code snippet when developers are writing an incomplete code. This work focuses on completing the code from a part of the code. We propose an inference from the skeleton of method invocations, which is in form of sequence of method names, to the implementation of method invocations.

Conclusion

In this work, we proposed InvocMap, a SMT engine for inferring the ASTs of method invocations from a list of method names and code context. By the evaluation on corpus collected from Github projects and online forums, we demonstrated the potential of our approach for auto code completion. A major advantage of InvocMap is that it is built on the idea of abstracting method invocations by four different levels. We provided an algorithm to achieve AST of method invocations for the method invocations inference. As future works, we will work on extending the SMT model to support inputs from multiple natural language descriptions of multiple method invocations, along with investigation of machine learning techniques for improving the accuracy.

References

Allamanis, M.; Barr, E. T.; Devanbu, P.; and Sutton, C. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* 51(4):81:1–81:37.

Android, P. 2019a. Example of method onreceive() in the android project. https://tinyurl.com/y49cxygr.

Android, P. 2019b. Example of method onreceive() in the android project. https://tinyurl.com/y334wqyw.

Android, P. 2019c. Example of method set() in the android project. https://tinyurl.com/yy9ge34m.

Barone, A. V. M., and Sennrich, R. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *CoRR* abs/1707.02275.

Eclipse. 2019. Abstract syntax tree documentation. https://tinyurl.com/y3gbe6jx.

GeeksForGeeks. 2019. Geeksforgeeks online forum. https://www.geeksforgeeks.org.

Github. 2019. Github repository. https://github.com.

Green, S.; Cer, D.; and Manning, C. D. 2014. Phrasal: A toolkit for new directions in statistical machine translation. In *In Proceddings of the Ninth Workshop on Statistical Machine Translation*.

Gu, X.; Zhang, H.; Zhang, D.; and Kim, S. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 631–642. New York, NY, USA: ACM.

Gvero, T., and Kuncak, V. 2015. Synthesizing java expressions from free-form queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, 416–432. New York, NY, USA: ACM.

Hellendoorn, V. J.; Devanbu, P. T.; and Bacchelli, A. 2015. Will they like this? evaluating code contributions with language models. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 157–167.

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

InvocMap. 2019. Invocmap data. https://tinyurl.com/y387n6xq.

Jurafsky, D., and Martin, J. H. 2009. *Speech and Language Processing (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Koehn, P.; Och, F. J.; and Marcu, D. 2003. Statistical phrase-based translation. 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, 48–54. Stroudsburg, PA, USA: Association for Computational Linguistics.

Liu, H. 2016. Towards better program obfuscation: Optimization via language models. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 680–682.

Nexus. 2019. nexu android framwork base example. https://tinyurl.com/y2ggtgj7.

Nguyen, A. T.; Nguyen, H. A.; Nguyen, T. T.; and Nguyen, T. N. 2012. Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool. In *2012 34th International Conference on Software Engineering (ICSE)*, 1407–1410.

Nguyen, A. T.; Nguyen, T. T.; and Nguyen, T. N. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 585–596.

Pascarella, L., and Bacchelli, A. 2017. Classifying code comments in java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, 227–237. Piscataway, NJ, USA: IEEE Press.

Phan, H.; Nguyen, H.; Tran, N.; Truong, L.; Nguyen, A.; and Nguyen, T. 2018. Statistical learning of api fully qualified names in code snippets of online forums. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 632–642.

ProgramCreek. 2019. Programcreek online forum. https://www.programcreek.com.

StackOverflow. 2019. Stackoverflow online forum. https://stackoverflow.com/.

Subramanian, S.; Inozemtseva, L.; and Holmes, R. 2014. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 643–652. New York, NY, USA: ACM.