INVOCMAP: MAPPING METHOD NAMES TO METHOD INVOCATIONS VIA MACHINE LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Implementing correct method invocation is an important task for software developers. However, this is challenging work, since the structure of method invocation can be complicated. In this paper, we propose InvocMap, a code completion tool allows developers to obtain an implementation of multiple method invocations from a list of method names inside code context. InvocMap is able to predict the nested method invocations which their names didnt 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 the 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 the accuracy rate up to 84 in F1-score depending on how much information of context provided along with method names, that shows its potential for auto code completion.

1 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 of all, developers need to remember the structure and the combination of invocations depending on their purpose. Secondly, the implementation of method invocation is also depending on the surrounding context of the code. Thus, the code developed by non-experience developers may be in the risks of being semantic error.

To help developers with interacting and analyzing by 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 is able to interact with the structure of each elements inside the source code. MI, which is defined as sub-type of Expression, is one of the fundamental AST Nodes 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. The reason for this issue is that expression can be composed by different types of AST Node including MI.

An example of a complicated MI is shown in Listing 1. Within this Listing, the outside MI contains four nested MI in its implementation. Additionally, there are five 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. Without doubt, the outer method name `set` is just one word while the respected MI is a deep AST tree.

The representation of MI also relies 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 1: Example in Android (2019a)

```
public void setOffsets(int
    newHorizontalOffset, int
    newVerticalOffset) {
  ...
    if (mView != null) {
      ...
      invalidateRectf.
          offset(-xoffset
          , -yoffset);
      invalidateRect.set((
          int) Math.floor
          (
          invalidateRectf
          .left),(int)
          Math.floor(
          invalidateRectf
          .top),(int)
          Math.ceil(
          invalidateRectf
          .right),(int)
          Math.ceil(
          invalidateRectf
          .bottom));
      ...
```

Listing 2: Example 2A in Android (2019b)

```
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 in Android (2019c)

```
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 the examples above, we recognize that implementing an effective method invocation requires strong background and experiences of developers. Even two MIs that belong to the same API and share the same context of local variables and literal still have ambiguous in the way of implementation like Listing 2 and Listing 3. These challenges hinders the ability of writing a appropriate MI and as well as developers need to spend time to remember or identify the correct structure of AST in MI for software development.

With 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 in order to obtain the complete code. For instance, in Listing 2, developer can write the list of method names including the name `getBooleanExtra`. The output for the suggestion 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 et al., 2014). For taking advantage from SMT, we propose a direction of 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 and foremost, 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. In order to evaluate our approach, we do experiments to check the accuracy of our code completion technique in two data sets collected from Github and from online forums. Resources of this paper can be found in (InvocMap, 2019). This research has following contributions:

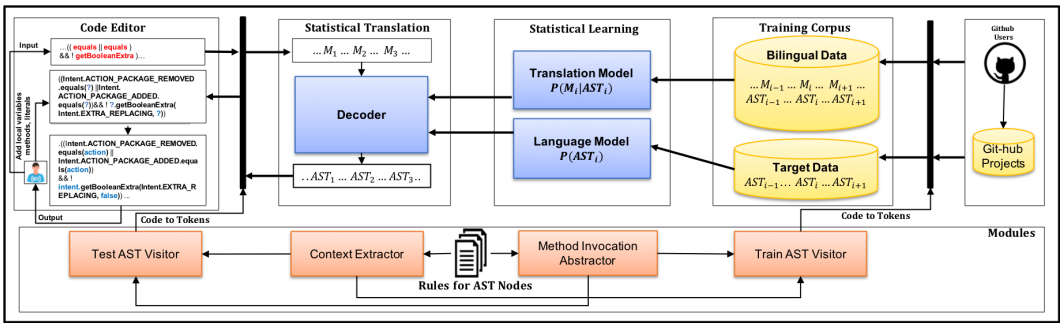


Figure 1: Overview Approach of InvocMap

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

2 Approach

We summarize the engines inside InvocMap on Figure 1. From the perspective of developers, InvocMap provides a plugin inside with Java code editor to allow them to write a single or multiple method names inside the code environment. Starting with 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 are abstracted at level 3 in our definition. That means they only require developers to add local variables, local methods or literals to obtain 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 from experienced developers, representing it by a machine learning model to help non-experienced 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 modules we developed. Inside these visitors, we handle each AST Node types by functions of module Context Extractor and MI Abstractor, which we discuss in next sections.

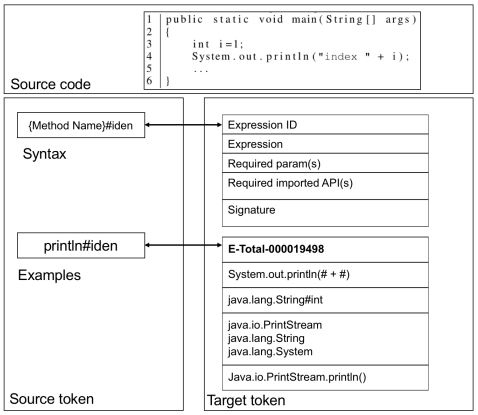
2.1 Levels of MI Abstraction in InvocMap

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

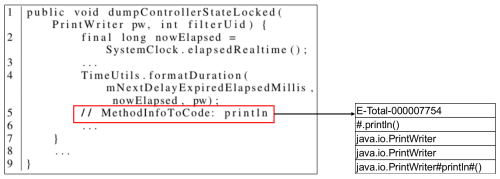
Definition 2 *Level 2 of abstraction of a method invocation is the information about type (or signature) of that method invocation.*

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.*

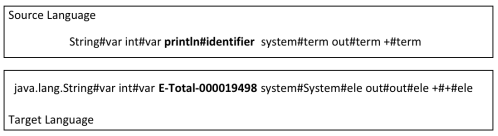
Definition 4 *Level 4 of abstraction of a method invocation is the complete Abstract Syntax Tree of that method invocation.*



(a) Example of AST_Level3 instance



(b) Example of inference from method name to MI at level 3 for code snippet in Nexus (2019)



(c) Example of source and target language

Figure 2: Representations of AST_Level3

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.'*

Along with 4 levels of abstraction in MI, we have the definition of local context provided for each MI. An example of 4 levels is shown in Figure 2(a). 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 compile-able. The level 3 is the AST that is having 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 to get level 4, 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. Therefore, developers could know which types of local variables to obtain the final code along with the set of imported APIs when they receive an AST at level 3 of abstraction. 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(a), we see 2 local entities, including the string literal `"index"` and the integer variable `i`. The suggested terms can be `"System"` and `"+"` sign.

2.2 Levels of Abstraction for Other AST Nodes

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.*

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). According to definitions of 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.

2.3 Source and Target Tokens Extraction

Other AST Nodes tokens. 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 Visitor and Test AST Visitor.

MI tokens. There are two types of information we want to embed for MI: the mapping between method name and the AST along with 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(a) is shown in Figure 2(c).

2.4 METHOD INVOCATION ABSTRACTION

Listing 4: Algorithm for Method Invocation Abstraction

```
1  AST_Level3 abstractMethodInvocation(
2    mi : MethodInvocation ,
3    dictionaryAST : Set<AST_Level3>,
4    visitor : InvocAbstractVisitor) {
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 }
```

Listing 5: Definition of AST_Level3 and InvocAbstractVisitor

```
1  class AST_Level3 {
2   // Fields to store AST in level 3
3   strCode : String;
4   // Other informations
5   listArguments : List<Type>;
6   setImportedAPIs : Set<Type>;
7   strSignature : Type;
8   // id is created by the uniqueness of 4 other
        fields
9   uniqueId : String;
10
11 }
12 class InvocAbstractVisitor extends ASTVisitor{
13  result : AST_level3;
14  ...
15  void visit(ASTNode node) {
16   // If visit local entity, abstract the place
17   if(isLocalEntity(node)) {
18    result.strCode.append("#");
19    result.listArguments.append(getType(node));
20   }
21   else {
22    // visit structure and update result
23    visitStructure(node);
24   }
25   // Add type of node to set of imported APIs
26   result.setImportedAPIs.add(getType(node));
27  }
28 }
```

We get information about level 3 of abstraction in MI by proposing an algorithm in Listings 4 and 5. 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 InvocAbstractVisitor defined in Listing 5 (line #12). 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 in Listing 5(#23). After visiting and abstracting of MI to an `AST_Level3`, this object is checked by the first four fields defined in Listing 5(#1-#10) 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(a).

2.5 STATISTICAL MACHINE TRANSLATION

To learn the mapping between source and target language, we apply the SMT (Green et al. (2014)). SMT was built from two models: the language model and the translation model.

Language Model (LM). LM is used to predict the next token given a sequence of previous tokens (Koehn et al., 2003). The more comprehensive corpus of target language we have, the higher quality of prediction the LM achieves. LM had been used widely in Software Engineering (SE) researches (Hindle et al., 2012; Hellendoorn et al., 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 training data. Here we use the n-gram language model, which proposed by Jurafsky & Martin (2009). Assume that we have

Table 1: Corpus, configurations and RQ3 result

(a) Corpus

Library	Pairs
JDK	820386
GWT	170435
Joda-Time	91072
Android	312822
Hibernate	149887
Xstream	159170

(b) Configurations

	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

(c) RQ 3 result

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%

m tokens in the target language $AST_1, ..., AST_m$, the probability provided by LM is shown in the above equation of Equations 1.

$$P_{LM}[AST_1, ..., AST_m] = \prod_{i=1}^{m} P[AST_i | AST_{i-n}, ..., AST_{i-1}]$$
$$D_{best} = argmax_D(p(S|D) * P_{LM}(D))) \quad (1)$$

Translation Model. This model calculates the probability of a phrase from source language that can be translated to a phrase in a target language. 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 by the below equation in Equations 1. Since we infer from method names to MIs which are consistent in order, we don't apply the reordering probability in the translation model.

3 EVALUATION

Data Preparation. 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 et al. (2014)). To generate corpus, we select 1000 highest stars Java projects from Github (Github (2019)), which have most files used APIs from libraries in Table 1a. For each Java project, InvocMap parses each Java source files by an the Train ASTVisitor module on Figure 1. The number of pairs respected to each method body we collect is shown in Table 1a.

Training and Testing Configuration. To train the SMT model, we use a high-end computer with core-i7 Intel processor and use 32 GB of memory. We apply our solution using Phrasal Green et al. (2014). We allocate Phrasal with phrase length equals to 7. The total training time requires about 6 hours. For testing, 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 1b. 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 they remember as part of the MI, but they don't remember the structure of AST. At 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.

Metrics. Information about tokens of method name and MI can be recognized by the annotation `#identifier` in 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. Out of Vocabulary (OOV) result is the case that the method name token does not in the corpus (Out of Source - OOS) or the expected AST in level 3 does not appear in the target corpus (Out of Target - OOT).

Table 2: Intrinsic Evaluation Result on Github projects and Extrinsic Evaluation Result on Online Forum Code

	Intrinsic Evaluation with Configuration 1						**Extrinsic Evaluation with Configuration 1**					
Library	**Correct**	**OOVoc**	**Total**	**Pre**	**Rec**	**F1**	**Cor**	**OOV**	**Total**	**Pre**	**Rec**	**F1**
GWT	39635	31522	93475	63.98%	55.70%	59.55%	58	9	102	62.37%	86.57%	72.50%
Joda-Time	27364	1743	39715	72.06%	94.01%	81.59%	36	17	75	62.07%	67.92%	64.86%
JDK	1053330	394317	1988644	66.07%	72.76%	69.25%	115	44	250	55.83%	72.33%	63.01%
Android	471347	54316	617416	83.71%	89.67%	86.58%	51	13	106	54.84%	79.69%	64.97%
Hibernate	53319	38877	117501	67.82%	57.83%	62.43%	125	40	226	67.20%	75.76%	71.23%
Xstream	4671	3019	9382	73.41%	60.74%	66.48%	44	14	64	88.00%	75.86%	81.48%
Total	1649666	523794	2866133	70.43%	75.90%	73.06%	429	137	823	62.54%	75.80%	68.53%
	Intrinsic Evaluation with Configuration 2						**Extrinsic Evaluation with Configuration 2**					
GWT	53042	31522	93475	85.62%	62.72%	72.40%	88	9	102	94.62%	90.72%	92.63%
Joda-Time	29028	1743	39715	76.45%	94.34%	84.45%	53	17	75	91.38%	75.71%	82.81%
JDK	1347221	394359	1988644	84.50%	77.36%	80.77%	177	44	250	85.92%	80.09%	82.90%
Android	470725	54321	617416	83.60%	89.65%	86.52%	85	13	106	91.40%	86.73%	89.01%
Hibernate	63275	38881	117501	80.48%	61.94%	70.00%	138	40	226	74.19%	77.53%	75.82%
Xstream	5145	3019	9382	80.86%	63.02%	70.83%	49	14	64	98.00%	77.78%	86.73%
Total	1968436	523845	2866133	84.04%	78.98%	81.43%	590	137	823	86.01%	81.16%	83.51%
	Intrinsic Evaluation with Configuration 3						**Extrinsic Evaluation with Configuration 3**					
GWT	55510	31522	93475	89.60%	63.78%	74.52%	89	9	102	95.70%	90.82%	93.19%
Joda-Time	31394	1743	39715	82.68%	94.74%	88.30%	55	17	75	94.83%	76.39%	84.62%
JDK	1435424	394359	1988644	90.04%	78.45%	83.84%	174	44	250	84.47%	79.82%	82.08%
Android	498708	54321	617416	88.57%	90.18%	89.36%	82	13	106	88.17%	86.32%	87.23%
Hibernate	65860	38881	117501	83.77%	62.88%	71.84%	146	40	226	78.49%	78.49%	78.49%
Xstream	5516	3019	9382	86.69%	64.63%	74.05%	50	14	64	100.00%	78.13%	87.72%
Total	2092412	523845	2866133	89.33%	79.98%	84.40%	596	137	823	86.88%	81.31%	84.00%

3.1 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 2. 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)). 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), which only infers the type information of MI.

We study an example in the Intrinsic Evaluation in Figure 2(b). 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()`.

3.2 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). 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 1a. The result for extrinsic evaluation is shown in Table 2. We see that with level 1, since the case that only method names are provided in the source language, our approach stills predict correctly 68.5% in F1-score. With the configuration levels that developers add more information, the F1-score increases to 84%. 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.

3.3 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. The result, which is shown in Table 1c, reveals 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% from low to high number of mappings, means that the prediction is still acceptable although the method names are too ambiguous.

4 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 & 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 are two reasons cause to this challenge. First, large scale code corpus contains noise data Pascarella & 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 & 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 et al. (2015) derive the code in C# language from code in Java language by machine translation. (Gvero & 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

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, July 2018. ISSN 0360-0300. doi: 10.1145/3212695. URL http://doi.acm.org/10.1145/3212695.

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

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

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

Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *CoRR*, abs/1707.02275, 2017. URL http://arxiv.org/abs/1707.02275.

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

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

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

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

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 631–642, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950334. URL http://doi.acm.org/10.1145/2950290.2950334.

Tihomir Gvero and Viktor Kuncak. 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, pp. 416–432, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814295. URL http://doi.acm.org/10.1145/2814270.2814295.

V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 157–167, May 2015. doi: 10.1109/MSR.2015.22.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337322.

InvocMap. Invocmap data. https://tinyurl.com/yxbrl3x7, August 2019.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. ISBN 0131873210.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. 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, pp. 48–54, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. doi: 10.3115/1073445.1073462. URL https://doi.org/10.3115/1073445.1073462.

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

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

A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1407–1410, June 2012. doi: 10.1109/ICSE.2012.6227236.

A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. 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)*, pp. 585–596, Nov 2015. doi: 10.1109/ASE.2015.74.

Luca Pascarella and Alberto Bacchelli. Classifying code comments in java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pp. 227–237, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-1544-7. doi: 10.1109/MSR.2017.63. URL https://doi.org/10.1109/MSR.2017.63.

H. Phan, H. Nguyen, N. Tran, L. Truong, A. Nguyen, and T. Nguyen. Statistical learning of api fully qualified names in code snippets of online forums. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 632–642, May 2018. doi: 10.1145/3180155.3180230.

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

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

Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 643–652, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568313. URL http://doi.acm.org/10.1145/2568225.2568313.