Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums

Hung Phan
Iowa State University, USA
hungphd@iastate.edu

Hoan Anh Nguyen
Iowa State University, USA
hoan@iastate.edu

Ngoc M. Tran
University of Texas at Dallas, USA
ngoctran@utdallas.edu

Linh H. Truong
University of Texas at Dallas, USA
linh.h.truong@utdallas.edu

Anh Tuan Nguyen
Independent Researcher, USA
ntanhbk44@gmail.com

Tien N. Nguyen
University of Texas at Dallas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Software developers often make use of the online forums such as StackOverflow (SO) to learn how to use software libraries and their APIs. However, the code snippets in such a forum often contain undeclared, ambiguous, or largely unqualified external references. Such declaration ambiguity and external reference ambiguity present challenges for developers in learning to correctly use the APIs. In this paper, we propose StatType, a statistical approach to resolve the fully qualified names (FQNs) for the API elements in such code snippets. Unlike existing approaches that are based on heuristics, StatType has two well-integrated factors. We first learn from a large training code corpus the FQNs that often co-occur. Then, to derive the FQN for an API name in a code snippet, we use that knowledge and also leverage the context consisting of neighboring API names. To realize those factors, we treat the problem as statistical machine translation from source code with partially qualified names to source code with FQNs of the APIs. Our empirical evaluation on real-world code and StackOverflow posts shows that StatType achieves very high accuracy with 97.6% precision and 96.7% recall, which is 16.5% relatively higher than the state-of-the-art approach.

CCS CONCEPTS

• **Software and its engineering** **Software libraries and repositories**; **API languages**;

KEYWORDS

Type Resolution; Type Inference; Type Annotations; Partial Program Analysis; Statistical Machine Translation; Naturalness; Big Code

ACM Reference Format:
Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *ICSE '18: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180230

1 INTRODUCTION

Software systems reuse code via software libraries or frameworks. The public methods, classes, and fields of libraries are called *Application Programming Interface (API) elements*. However, not all libraries and frameworks provide well-maintained documentation [7, 8, 26]. To cope with that, software engineers take advantage of online forums, *e.g.,* StackOverflow or GitHub Gists. Such forums have provided good resources on API usages [22]. Unfortunately, due to the informal nature of such forums, the code snippets in the answering StackOverflow posts may contain declaration ambiguity and external reference ambiguity [27]. The code snippets often reference the fields whose declarations are not included or refer to external identifiers that are unqualified. As a result, this presents challenges for developers in learning to correctly use the libraries [27].

Existing work has been addressing those issues by *resolving the types* of API elements within StackOverflow (SO) code snippets or descriptions and then linking the code snippets to the API documentation [8, 24, 27]. The existing techniques to resolve FQNs in such snippets follow three main strategies: *partial program analysis* [8], *text analysis* [24], and *iterative, deductive analysis* [27]. Despite different strategies, they all share the same principle of applying *rules* and *heuristics* on an incomplete code snippet. For example, Partial program analysis (PPA) [6] and RecoDoc [8] use *heuristics on syntactic constructs*, while ACE [24] uses heuristics on texts in SO posts and Baker [27] applies the scope rules to the API elements. However, an important weakness of the existing approaches is that their *program analysis rules and heuristics* are often *ineffective due to the missing much information in incomplete code*. For example, in many cases (15–31% of the dataset of code snippets under study [27]), Baker cannot infer the FQNs. After its process of overlapping candidate lists via scoping rules, many API elements still have multiple potential FQNs (some have hundreds of candidates) [27].

Instead of relying on rules/heuristics, we propose StatType, a *statistical approach* to resolve the FQNs for the API elements of a code snippet in an online forum by leveraging two kinds of *context*. First, StatType relies on the *type context* of API elements in a large code corpus. API elements occur together because source code is composed of idioms: an API usually collaborates with other specific APIs to achieve some functionality. If we can train a model to learn the regularity of co-occurring APIs, it could derive the most likely FQNs for certain API names in a code snippet. Let us call it the principle of *regularity of API elements*. Second, to narrow the candidate FQN list for an API name *P*, we use the *resolution context* on how the types of API names surrounding *P* have been resolved.

That is, the decision on inferring a FQN for a name *P* depends on the decisions on the FQN inference for the API names surrounding *P*.

These two ideas are integrated into StatType when we treat the type inference problem as *statistical machine translation* (SMT) from the code with partially qualified API names to the code with FQNs of the APIs. To find the respective FQNs for the API names in a code snippet, *i.e.,* to find the respective sentence in the target language, an SMT model is based on three models. First, the language model is responsible for estimating the regularity of a sequence of FQNs for an API usage, *i.e.,* how likely a sequence of FQNs for the APIs occurs. Second, the mapping model is responsible for estimating the likelihood of observing the mappings between API names and the respective FQNs, *i.e.,* how likely a certain API name has a certain FQN. Both language and mapping models are trained from a large code corpus of well-typed, open-source projects that use the APIs under study. Finally, the third model is the decoder, which uses the trained language and mapping models, and a search strategy to efficiently find the most likely sequence of FQNs corresponding to the sequence of API names in the snippet. The strategy considers the contexts of the surrounding API names in deriving the resulting FQNs.

To train StatType, we choose a large training corpus of well-typed projects. For each method of a project in the training corpus, we build a pair of sentences: 1) the source sentence is the sequence of annotations representing the original source code of the method, and 2) the target sentence is the sequence of annotations representing the FQNs of the API and code elements in the method.

Our large training corpus contains a total of 1.085M methods, 9.562M API usages, and 244K unique APIs in six software libraries. We then performed an experiment for an intrinsic evaluation by using 10-fold cross-validation on the large corpus. The result shows that StatType achieves very high accuracy with 97.6% precision and 96.7% recall. We also confirmed that the two key contexts (type context and resolution context) contribute to high accuracy of our model. The contexts help StatType disambiguate the APIs with multiple FQN candidates and achieve 97% precision in those cases.

To compare StatType against the state-of-the-art approach Baker [27], we trained StatType with the above corpus and tested it in a corpus of real-world SO code snippets with 7,154 API usages and 1,139 unique APIs. Our result shows that while achieving almost the same precision, StatType has significantly better recall (13.7% absolutely and 16.5% relatively).

In this paper, we have the following contributions:

1. StatType, a machine translation approach that resolves the FQNs of the API elements in a code snippet. One could use StatType to build *the automated tools to resolve/infer type annotations for incomplete source code*, *e.g.,* plugins in IDEs, Web browsers, or in the API-related tools *e.g.,* pattern mining, bug detection *etc.*

2. An extensive empirical evaluation to evaluate the accuracy of StatType on real-world code and StackOverflow posts, showing that it achieves very high accuracy and outperforms the state-of-the-art approach Baker [27] (16.5% relatively).

2 MOTIVATING EXAMPLE

2.1 An Example of a StackOverflow Code Snippet

StackOverflow is a good resource of high quality code snippets that show correct API usages. However, due to the discussion context and the informal nature of the forum, code snippets rarely contain sufficient declarations and references to fully-qualified names.

```
1   Event.sinkEvents(iframe, Event.ONLOAD);
2   Event.setEventListener(iframe, new EventListener() {
3       public void onBrowserEvent(Event event) {
4           StyleElement style = Document.get().createStyleElement();
5           style .setInnerText(resources.css() .getText()) ;
6           iframe.getContentDocument().getHead().appendChild(style);
7
8           ParagraphElement pEl = Document.get().createPElement();
9           pEl.setInnerText("Hello␣World");
10          pEl.addClassName(resources.css().getText());
11          iframe.getContentDocument().getBody().appendChild(pEl);
12      }
13  }) ;
```

Figure 1: StackOverflow post #34595450 on GWT Widgets [10]

Let us use a real-world example to illustrate the challenges in resolving the FQNs for an SO code snippet. Figure 1 shows a code snippet of an answer for an SO post on Widgets in the GWT library.

First, the snippets are not often embedded in methods or classes. In Figure 1, the code does not have an enclosing method or class.

Second, they may refer to variables whose *declarations are not included*, and their *identifiers are largely unqualified*. For example, the variable iframe at lines 1, 2, 6, and 11 do not have a declaration because the responder assumes that users would add a declaration of type IFrameElement for iframe or add a method call to the API createIFrameElement to get an object of the type IFrameElement.

Third, *the external references in code snippets are also often undeclared*. Specifically, code snippets frequently refer to external types without specifying their FQNs since the responder assumes that those FQNs could be implicitly understood in the context of the post. For example, the types Event, EventListener, StyleElement, Document, and ParagraphElement are referenced by simple names only.

Fourth, *name ambiguity occurs in the references to external libraries*. For example, the type EventListener at line 2 is a common unqualified type name. In this snippet, it refers to com.google.gwt.user.client.-EventListener. However, it is ambiguous with java.util.EventListener in JDK. Similarly, the simple type name Document at line 4 is also popular (com.google.gwt.dom.client.Document, org.eclipse.jface.text.Document, org.w3c.dom.Document, *etc.*). In fact, external reference ambiguity is popular [8]. For example, the simple names getId and Log occur 27,434 and 284 times respectively in various Java libraries [27].

To address those ambiguities, existing approaches rely on rules and heuristics mainly from program analysis. RecoDoc [8] uses heuristics in the partial program analysis tool, PPA [6], however, in many cases, the heuristics help recover only partially qualified names, due to much missing information. For example, at line 6, Figure 1, the types of the receiver iframe and method call getContentDocument, are not known. In contrast, Baker [27] uses a dictionary of all the API names of interest. Each API name has multiple candidates of FQNs. Baker iteratively cuts off the candidates by overlapping the candidate lists of API names in the same scopes in the code snippet. However, scoping rule or other rules/heuristics are not always effective in incomplete code snippets. In this example, lines 4–6 in Figure 1 have only one scope with the popular API names having many potential FQNs (Document, getText, appendChild). Moreover, building and updating a dictionary of the APIs require much effort [7].

2.2 Key Ideas

In this work, we develop StatType, a technique to identify the FQNs for the simple names of *variables, API classes, method calls, and field accesses* in a code snippet in online forums. In our solution, we leverage two kinds of context in deriving FQNs for API elements.

The first one is the *type context* of API elements in a large code corpus. We call it the principle of *regularity of API elements*. That is, in the API usages, the API elements including API classes, method calls, and field accesses of software libraries do not occur randomly. They appear regularly with certain other API elements due to the intent of the libraries' designers to provide certain functionality in API usages. For example, to manipulate style information for a frame in GWT, at lines 4–6 in Figure 1, a StyleElement in GWT could be created via a call to createStyleElement of a Document object. Then, the style is assigned with a name via StyleElement.setInnerText and associated with an IFrameElement object. Those API elements of those types repeatedly occur together because such tasks are provided by the GWT library and intended via such API usages. Thus, if StatType observes the API usages involving those API elements in a large corpus of programs using GWT, it can use the context of the given code snippet to derive the most likely FQNs for the APIs in the snippet. For example, the context of the API names StyleElement, Document, and createStyleElement could give StatType a hint on the FQNs of the undeclared variable iframe, the method calls getContentDocument, appendChild, and vice versa.

To learn the context, we rely on the basis of regularity with a large corpus: the API elements regularly going together in API usages have higher impact in deciding the FQNs than the less regular ones.

The second context is called *resolution context* on how the types of API names surrounding an API name have been resolved. The undeclared/ambiguous references are expected to be disambiguated with *the most likely FQNs with regard to the FQNs of the surrounding names*. For example, at lines 3–4, when resolving Event, StyleElement, and Document, we could use an effective search strategy to maintain the sequences of FQNs with the highest probabilities. For Event, we have multiple candidates. However, if we chose com.google.gwt.user.client.Event for Event, then based on their regularity, we could maintain the option com.google.gwt.dom.client.Document for Document, with a higher score than org.w3c.dom.Document. Finally, the sequence with the highest score will be reported as the result.

2.3 Type Inference with Machine Translation

These two ideas are integrated into our approach when we treat the type inference problem as statistical machine translation (SMT) from source code with API names to the source code with FQNs of the APIs. SMT [14] is a natural language processing (NLP) approach that uses statistical learning to derive the translation "rules" from a training data (called a *corpus*) and applies the trained model to translate a sequence from the source language (L_S) to the target one (L_T). A phrase-based SMT [15] is a model in which the translation rules are expressed in term of phrases to phrases from the source language to the target one. Phrase-based SMT translates a sequence s in L_S into a sequence t in L_T by searching for the sequence t that has the maximum probability $P\left(t|s\right) = \frac{P(s|t).P(t)}{P(s)}$. Since s is given, $P\left(s\right)$ is fixed for all potential sequences t. SMT model translates s by searching for the sequence t that maximizes the product $P\left(s|t\right).P\left(t\right)$.

To achieve that, phrase-based SMT translates a source sequence by relying on three models. First, the Mapping Model computes the likelihood $P\left(s|t\right)$ of the mapping pairs from t to s, *i.e.,* the alignment between the words/sequences in two languages. If we use s as the source sequence of API names and t as the target sequence of the corresponding FQNs, we can train the mapping model from a large code corpus of well-typed projects that use the libraries. That is, we train the mapping model to estimate *how likely an API name has a certain FQN*. Second, the Language Model is used to compute $P\left(t\right)$, *i.e.,* how likely that the sequence t that is used for translation of s occurs in the target language. In this case, we use the language model to measure the *regularity of a sequence of FQNs*, that is, how likely a sequence of FQNs for API usages occurs, which constitutes a **type context**. We can train the language model with a large corpus of the client code that uses the APIs of the libraries. Third, during translation, an SMT model makes use of the language model and the mapping model to search for the sequence t that maximizes $P\left(s|t\right).P\left(t\right)$. To do so, during the process of deciding a FQN for an API name, SMT uses a search strategy that considers the *already-resolved FQNs/types of the previous API names*. Thus, both the Mapping Model and such search strategy realize our idea of **resolution context**.

3 EXTRACTING API SEQUENCES

To train our model to learn the mappings between API names and the corresponding FQNs, we parse a large code corpus to build a parallel training corpus of pairs of source and target sentences. The source sentence is a sequence of *annotation units* representing simple or partial API names, and the target is a sequence of *annotation units* representing the FQNs of the corresponding API names and tokens in the source sentence. For example, the source sequence Event.sinkEvents corresponds to the target sequence com.google.gwt.user.client.Event com.google.gwt.user.client.Event.sinkEvents. For better readability, we use *sentence* to denote a sequence for a method in the corpus.

We use a parser to collect the static FQNs and build such pairs. It traverses the AST of each method and extracts a *FQN annotation sequence* consisting of *annotation units* according to different syntactic units related to API elements, including the syntactic types: *literal*, *type*, *method call*, *class instance creation*, *field access*, etc.

Table 1 shows the key rules to build annotation sequences for Java source code. We design the rules with two key goals in mind. First, they produce the mappings between API partial/simple names and FQNs to train the mapping model. Second, the sequences of FQNs of APIs show the surrounding API elements, which help capture the regularity of API usages. Let us use $\Re^p\left(E\right)$ and $\Re^f\left(E\right)$ to denote the two functions that are used to generate the source and target annotation sequences, respectively. It is first applied on a method body and recursively called upon the syntactic units in the code until it reaches the base cases. The annotation sequences produced in the intermediate steps are concatenated to form a source or target sentence for a method. For each unit, we use its FQN from the parser to create the annotation for it in the target sentence as follows.

Literal: We use the partial name of the literal's type, returned from function PT(.), in the source sequence and the FQN of the literal's data type, returned from function FT(.), in the target sequence. For example, for literal "Hello World", we produce the annotation String in the source sequence and the annotation java.lang.String in the target.

Table 1: The Key Rules $\Re^p\left(E\right)$ and $\Re^f\left(E\right)$ to Build API Annotation Sequences for Java source code

Syntax	**Build Rule and Example**	
	Partial Qualified or Simple Name (Source)	**Fully Qualified Name (Target)**
Literal $E ::= literal$	$\Re^p\left(E\right) = PT\left(literal\right)$ *e.g.,* $\Re^p\left(\text{“}Hello\ World\text{”}\right)$ = `String`	$\Re^f\left(E\right) = FT\left(literal\right)$ *e.g.,* $\Re^f\left(\text{“}Hello\ World\text{”}\right)$ = `java.lang.String`
Type $E ::= Type$	$\Re^p\left(E\right) = String\left(Type\right)$ *e.g.,* $\Re^p\left(Event\right)$ = `Event`	$\Re^f\left(E\right) = FT\left(Type\right)$ *e.g.,* $\Re^f\left(Event\right)$ = `com.google.gwt.user.client.Event`
Method call $E ::= e.m\left(e_1,...,e_n\right)$	$\Re^p\left(E\right) = \Re^p\left(e\right)\ .m\left(n\right)\ \Re^p\left(e_1\right)\ ...\ \Re^p\left(e_n\right)$ *e.g.,* $\Re^p\left(pEl.setInnerText\left(\text{“}Hello\ World\text{”}\right)\right)$ = `.setInnerText(1)` `String`	$\Re^f\left(E\right) = \Re^f\left(e\right)\ Sig\left(E\right)\ \Re^f\left(e_1\right)\ ...\ \Re^f\left(e_n\right)$ *e.g.,* $\Re^f\left(pEl.setInnerText\left(\text{“}Hello\ World\text{”}\right)\right)$ = `com.google.gwt.dom.client.Element.setInnerText(java.lang.String)` `java.lang.String`
Class instance creation $E ::=$**new** $Type\left(e_1,...,e_n\right)$	$\Re^p\left(E\right) = new\ Type\left(n\right)\ \Re^p\left(e_1\right)\ ...\ \Re^p\left(e_n\right)$ *e.g.,* $\Re^p$(**new** *EventListener*()) = `new EventListener(0)`	$\Re^f\left(E\right) = new\ Sig\left(E\right)\ \Re^f\left(e_1\right)\ ...\ \Re^f\left(e_n\right)$ *e.g.,* $\Re^f$(**new** *EventListener*()) = `new com.google.gwt.user.client.EventListener()`
Field access/Qual.Name $E ::= e.f$	$\Re^p\left(E\right) = \Re^p\left(e\right)\ .f$ *e.g.,* $\Re^p\left(Event.ONLOAD\right)$ = `Event` `.ONLOAD`	$\Re^f\left(E\right) = \Re^f\left(e\right)\ Sig\left(E\right)$ *e.g.,* $\Re^f\left(Event.ONLOAD\right)$ = `com.google.gwt.user.client.Event` `com.google.gwt.user.client.Event.ONLOAD`

Type: For a type reference, we use its name appearing in the source code in the source sequence and its FQN in the target sequence. In Figure 1, for the type Event, we produce the token Event and the FQN com.google.gwt.user.client.Event in the source and target sequences.

Method call: We use the method name and the number of arguments in the source sequence. We do not include the types of the arguments *in the source sequence* because the type information might not be available in a SO code snippet. Yet the number of arguments is always available and could help distinguish overloading methods with different numbers/types of arguments. In the *target sequence*, we use the method's full signature including the FQN of the declaring class, the method name, and the list of FQNs of the data types of its arguments. If the method call has a receiver or an argument, the sequence build rule will be applied recursively on each component. In the 3rd row in Table 1, for pEl.setInnerText("Hello World"), the sequence .setInnerText(1) String is generated in the source since the call has one argument. In the target, the sequence com.google.gwt.dom.client.Element.setInnerText(java.lang.String) java.lang.String is generated. For $m(n())$, we recursively generate sequences for $n()$.

We do not have a rule for variables. If we use variables' names in the source, since API usages of the same type in different places might use different names, the model might not learn the type in one place and apply in another. We also could not use a variable's type in the source because for a given incomplete code snippet, we might not be able to derive its type. Moreover, if we encode a variable with a special annotation (*e.g.,*VAR), we would accidentally increase the regularity of that annotation and create noises in the usages involving variables. Thus, we use post-processing for variables, *e.g.,* the receiver objects or arguments of a resolved API call (Section 5.3).

Class instance creation: is treated similarly to method calls.

Field access: We treat a field access similarly to a method call with no arguments. We distinguish it with a method call by not including the open and close parenthesis at the end of the generated annotation.

4 TRAINING SMT MODEL

4.1 Training the Language Model

In StatType, we use the n-gram language model on the set of the target sentences built from a corpus with the procedure in Section 3. n-gram model has two assumptions. First, it assumes that a sequence is generated from left to right. Second, the generating probability of a unit in that sequence is dependent only on its *local context*, *i.e.,* a *window of previously generated units* (a special case of Markov assumption). Such dependencies are modeled based on the occurrences of sequences with limited lengths, called n-gram (n consecutive units). With the assumption of generating units from left to right, the generating probability of the sequence $t = t_1t_2...t_m$ is computed as

$$P\left(t\right) = P\left(t_1\right).P\left(t_2|t_1\right).P\left(t_3|t_1t_2\right)....P\left(t_m|t_1...t_{m-1}\right) \quad (1)$$

We need to compute $P\left(c|p\right)$ where c is a unit of annotation and p is a sequence of them. With Markov assumption, the conditional probability $P\left(c|p\right)$ is computed as $P\left(c|l\right)$ in which l is a subsequence made of the last $(n-1)$ units of p. We need to compute only the conditional probabilities involving at most n consecutive units, that is,

$$P\left(c|l\right) \simeq \frac{\text{count}\left(\left(l,c\right)\right) + \alpha}{\text{count}\left(l\right) + |V|.\alpha} \quad (2)$$

(l,c) is the sequence formed by l and c. $|V|$ is the vocabulary' size for the target sequences. α is a smoothing value for the cases of small counting values. count is the counting function for a sequence in a corpus. For practicality, we use the collection of target sentences explained in Section 3 as the corpus to train the language model. We also compute $P\left(t\right)$ only for the sequences encountered in the corpus. The result of the training process is those probability values, *e.g.,*

P (java.lang.String.equals(java.lang.Object)) = 0.981

P (java.io.PrintStream.println(java.lang.String)) = 0.976

P (java.util.Iterator.hasNext() java.util.Iterator.next()) = 0.944 ...

These probabilities of the sequences whose lengths are greater than one enable an SMT model to consider the type context of the API elements that often co-occur in source code.

4.2 Training the Mapping Model

The input of the training process for the Mapping Model is the parallel corpus consisting of the pairs of source and target sentences. The key output of training of Mapping Model is the *phrase translation table* or *phrase table* for short, which contains the mappings between the pairs of corresponding source and target (sub)sequences and their associated probabilities/scores. The values represent how likely the sequences in the source and target sides are matched, *i.e.,* $P\left(s|t\right)$.

To do that, the training process has two phases. The first phase aims to build the *single mappings* from the individual units in a

```
public void writeField (String name, Date value) throws IOException {
  if (value != null) {
    writeField (name, AtomDate.format(value));
  }
}
```

Single Mappings for the body of the above writeField method:

Code Token	Source Unit	Target Unit
null	null	null
writeField	.writeField(2)	org.apache.abdera.ext.json.JSONStream .writeField(java.lang.String,java.lang.String)
AtomDate	AtomDate	org.apache.abdera.model.AtomDate
format	.format(1)	org.apache.abdera.model.AtomDate .format(java.util.Date)

Figure 2: Examples of Single Mappings for JSONStream.java

source sentence to the individual units in a target one. The second phase aims to build the phrase translation table consisting of phrase-to-phrase mappings from the single mappings of individual units.

4.2.1 Single Mappings. The input of this phase is all the pairs of respective sentences. For each pair p, we have to determine the mappings for individual units in the source and target sentences in p. In natural language processing (NLP), deriving single mappings is non-trivial. It requires 1) an *alignment function* that maps a word at a source position to a word at a target position, 2) *re-ordering of words* since words may be re-ordered during translation, 3) a *weighting* scheme for each single mapping accordingly to the occurrence frequencies of that pair, 4) the *insertion of words* in the target sentence because a source word may translate into multiple target words, and 5) *dropping of words* since words may be dropped when translated. In many SMT tools, IBM Model [4] with an expectation-maximization algorithm is often used to build single mappings.

In StatType, this is straightforwardly done as part of sequence building. Specifically, for a pair p of source and target sentences:

1. The alignment function is the identity function that maps a position in the source sentence to the same position in the target one,
2. No re-ordering of units is needed,
3. Neither addition nor deletion of units is needed,
4. The mappings for the units in the pair p are known because the target sentence is produced from the source one with the only differences where the FQNs are replaced for the simple or partially qualified names (PQNs) (Section 3). Thus, their weights for the mappings in the pair p are set to 1. The global weights for a single mapping in the entire corpus are computed later (Section 4.2.2).

For example, for the code of the method writeField in org.apache.-abdera.ext.json.JSONStream (Figure 2), the source and target sentences are built following the procedure in Section 3. The mappings between source and target units are shown in the table in Figure 2.

4.2.2 Sequence-to-sequence Mappings. First, from the single mappings, to build sequence-to-sequence mappings, we extract the mappings between the (sub)sequences in each source and its target sentence (*Sequence Pair Extraction*). In NLP, a phrase in L_T can be mapped to another phrase with a different length. However, in StatType, a target sequence has an equal length with the corresponding source sequence. Moreover, to map a sequence in L_S to a sequence in L_T, no reordering, deletion, or insertion of units is needed. Therefore, for sequence pair extraction, for a given pair of corresponding source and target sentences s and t, we extract the sub-sequence

Table 2: One Source Sequence with Multiple Target Sequences

Source sequence	Target sequence	ϕ
Document (1-gram)	com.google.gwt.dom.client.Document	0.162
Document (1-gram)	org.eclipse.jface.text.Document	0.043
Document (1-gram)	org.w3c.dom.Document	0.053
Document (1-gram)	org.jdom2.Document	0.012

Source sequence	Target sequence	ϕ
StyleElement Document (2-gram)	com.google.gwt.dom.client.StyleElement com.google.gwt.dom.client.Document	0.192
StyleElement Document (2-gram)	com.google.gwt.dom.client.StyleElement org.w3c.dom.Document	0.072
StyleElement Document (2-gram)	Telerik.WinControls.UI.Export.HTML .StyleElement org.w3c.dom.Document	0.025

pairs with equal lengths from 1 to l (l is the length of s or t). That is, we extract the sub-sequences of size 1, size 2, and so on from s and map them with the sub-sequences at the corresponding positions in t. Note that the order of the units is preserved in translation.

For the example in Figure 2, the size-1 mappings are listed in the table. Examples of mappings for the size-2 sub-sequences include

1. null .writeField(2) ↔ null org.apache.abdera.ext.json.JSONStream.-writeField(...String,...String)
2. .writeField(2) AtomDate ↔ org.apache.abdera.ext.json.JSONStream.-writeField(...String,...String) org.apache.abdera.model.AtomDate
3. AtomDate .format(1) ↔ org.apache.abdera.model.AtomDate org.apache.abdera.model.AtomDate.format(java.util.Date)...

Second, we need to globally assign a probability to each of extracted sequence pairs (*Sequence Pair Scoring*). For each sentence pair, we extract a number of (sub)sequence pairs as explained. Then, we count in how many sentence pairs a particular (sub)sequence pair $(\bar{s},\bar{t})$ are extracted (count$(\bar{s},\bar{t})$). Finally, the translation probability $\phi\left(\bar{s}|\bar{t}\right)$ is estimated by the relative frequency over all $\bar{s_i}$ in the corpus:

$$\phi\left(\bar{s}|\bar{t}\right) = \frac{count\left(\bar{s},\bar{t}\right)}{\sum_{\bar{s_i}} count\left(\bar{s_i},\bar{t}\right)} \tag{3}$$

When a phrase is matched to multiple phrases in a sentence pair, we assign, for each of matches, fractional counts that adds up to one. Table 2 shows some examples of the mapping sequences.

5 INFER FQN VIA MACHINE TRANSLATION

Overview. Let us explain how we use the trained SMT model to translate the source sentence/sequence s built from a given code fragment. The model processes s from left to right. It considers breaking s into multiple phrases in all possible ways, and searches them in the phrase table. It repeatedly selects the units to be translated, finds the target sequence in the table, adds that sequence to the end of the partially translated sentence, and repeats the process until all units in s are translated. A probability is given to each target candidate t based on the probabilities of the aligned phrases within s (from the phrase table), and the occurrence probability $P\left(t\right)$ (from the language model). Such probability for a candidate t is gradually computed during translation. To avoid combinatorial explosion, a beam search strategy is used. The candidate with the highest probability is presented.

5.1 Formula

Mathematically, with the principle of phrase-based SMT [15], we aim to find the translated sentence t such that

$$t_{best} = argmax_t P\left(s|t\right) P\left(t\right) \ = argmax_t \prod_{i=1}^{I} \phi\left(\overline{s_i}|\overline{t_i}\right) P\left(t\right) \quad (4)$$

The factors contributing to the decision on t_{best} include the phrase translation table ϕ and the n-gram language model, which is responsible for computing $P\left(t\right)$ for a candidate sentence. Given all $i = 1...I$ phrases $\overline{s_i}$ of the input sentence s and respective phrases $\overline{t_i}$ and their positions in the target sentence, we compute the probability of the candidate translated sentence t using the phrase translation table ϕ (Formula 3) and $P\left(t\right)$ (Formula 1). Note that, in NLP, the probability in Formula 4 also depends on a reordering model. However, in StatType, we do not need to re-order the units. Neither addition nor insertion of units is needed. Thus, we need only ϕ and P in (4).

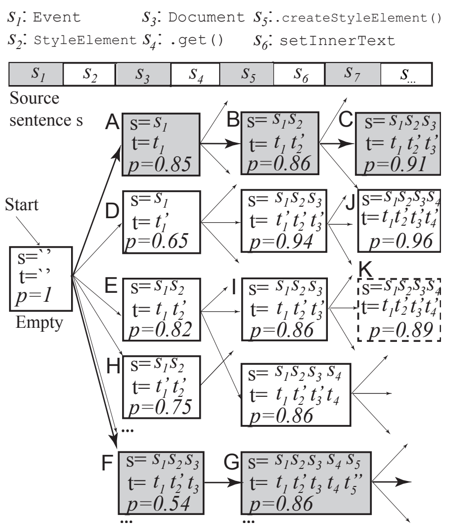
5.2 Algorithm

The translation algorithm, often called *decoding*, builds the translated sentence t on the target side from left to right. It starts with an empty sentence: no source unit covered, no target unit produced, and a probability of 1 for the empty sentence (Figure 3).

Initial Phase. In translating natural-language texts, any word or phrase could be selected for translation first because the new sentence might have a different order than the original one. However, for StatType, we could pick the *first unit* or any *sequence starting with the first unit* in the given sentence s to be translated first because the translated sentence t will have the same order of units as the given sentence s. Thus, StatType initially selects for translation either the unit s_1, the size-2 sequence $s_1 s_2$, ..., or the size-k sequence $s_1 s_2 ... s_k$ (s_i's are the units in s; $k \leq l$ where l is the length of the sentence). For each choice, it looks up in the phrase translation table ϕ for the corresponding unit or sequence. For example, in Figure 3, the first unit s_1 could be translated to either t_1 (choice A), t_1' (choice D), or t_1'', *etc.* The 2-gram $s_1 s_2$ could become either $t_1 t_2$, $t_1' t_2$, or $t_1 t_2'$ (choice E), *etc.* Similarly, the first 3-gram $s_1 s_2 s_3$ could become several alternative sequences (*e.g.,* choice F). The choices A, D, E, H, F,... in the first column are created in the initial phase. For each choice, it marks the units that have been translated in the source sentence s, and computes the probability for the current sequence using the formula $\prod_{i=1}^{I} \phi\left(\overline{s_i}|\overline{t_i}\right) P\left(t\right)$, as $\overline{s_i}$ and $\overline{t_i}$ are mapped phrases.

Expansion Phase. For each initial choice, StatType continues the process of expansion by considering one additional unit or sequence. For example, in Figure 3, choice B represents an additional selection of the unit s_2 to be translated after s_1. The choice G is the expansion of a sequence $s_4 s_5$ after $s_1 s_2 s_3$ was translated. Generally, there are many ways of expansion and translations for each (sub)sequence. After each expansion, the additional unit or the units in the additional sequence are marked as *covered*. StatType computes the probability for the current sequence similarly as in the initial phase. It expands further until all units in the original sentence s are translated.

Note that one path from the empty choice to the finishing node/-choice corresponds to one way of dividing the source sentence s for translation. For example, the path Empty $\rightarrow A \rightarrow B \rightarrow C...$ corresponds to breaking s into each individual unit and translates one by one. The path Empty $\rightarrow F \rightarrow G...$ represents the translation of $s_1 s_2 s_3$ as a sequence first, followed by the translation of $s_4 s_5$, *etc.*



s_1: Event
s_3: Document
s_5: .createStyleElement()
s_2: StyleElement
s_4: .get()
s_6: setInnerText
s_1 s_2 s_3 s_4 s_5 s_6 s_7 $s_{...}$
Source sentence s
Start
s=`' t=`' $p=1$
Empty
A s=s_1 t=t_1 $p=0.85$
B s=$s_1 s_2$ t=$t_1 t_2'$ $p=0.86$
C s=$s_1 s_2 s_3$ t=$t_1 t_2' t_3'$ $p=0.91$
D s=s_1 t=t_1' $p=0.65$
s=$s_1 s_2 s_3$ t=$t_1' t_2' t_3'$ $p=0.94$
J s=$s_1 s_2 s_3 s_4$ t=$t_1 t_2' t_3' t_4'$ $p=0.96$
E s=$s_1 s_2$ t=$t_1 t_2'$ $p=0.82$
I s=$s_1 s_2 s_3$ t=$t_1 t_2' t_3'$ $p=0.86$
K s=$s_1 s_2 s_3 s_4$ t=$t_1 t_2' t_3' t_4'$ $p=0.89$
H s=$s_1 s_2$ t=$t_1' t_2'$ $p=0.75$
s=$s_1 s_2 s_3 s_4$ t=$t_1 t_2' t_3' t_4$ $p=0.86$
F s=$s_1 s_2 s_3$ t=$t_1 t_2' t_3$ $p=0.54$
G s=$s_1 s_2 s_3 s_4 s_5$ t=$t_1 t_2' t_3 t_4 t_5''$ $p=0.86$
...

Figure 3: Type Inference via Translation

This process enables StatType to consider the *resolution context* of surrounding API elements in determining the FQN for an element.

Optimization. We aim to find the best sentence that covers all units in the source sentence. Unfortunately, the number of choices is exponential with respect to the sentence length. Translation has been known as NP-complete [13], thus, we need to reduce the search space. We adapted two strategies from phrase-based SMT [5] to address this complexity: *candidate recombination* and *beam search with stack.*

Candidate Recombination. In the search space, two different paths could lead to the same partial translation result. That is, two current result candidates could have the same original units translated, and the same units in the current output, but different probability scores. For example, in Figure 3, the partial translation results at J and K are the same, but come from two different paths $A \rightarrow B \rightarrow J$ and $E \rightarrow I \rightarrow K$. That is due to the use of different translation phrases in the translation table with different occurrence likelihoods computed by the language model. We recombine them and the worse choice with lower probability is dropped. The reason is that the worse choice can never be part of the best overall path: in any path that includes the worse choice, we can replace it with the better choice, and get a better score [14]. The worse path is not erased though. For the purpose of keeping the top-k candidates, a pointer is kept to connect its parent choice from the worse path to the better one. For J and K, we drop the choice K and connect the choice I to J.

Beam Search with Stack. The phrase-based SMT models [5, 14] address the complexity issue in finding the best path to completion by beam search with the use of a stack. In StatType, we adapt the beam search with stack by organizing the choices of partial candidates into stacks in which we prune out the worst candidates in the stack if it gets too large. We maintain a set of stacks based on the number of the units that have been translated. All partial translated candidates with the same number of units translated are placed in the same stack. For example, the stack #1 contains all partial translated candidates that have translated one unit in s, while the stack #2 is for the candidates translated two units in the path, and so on.

Table 3: Large Corpus from GitHub

Library	Sentence	#UsedAPIs			#Unique	Dens.	Sentence length	
	Pairs	Class	M.Inv	Field.Acc			Avg	Med
Android	450,743	2.53M	1.3M	445K	89,024	9.5	25.3	14
Joda Time	39,081	202K	144K	6.5K	4,086	9.0	33.3	18
XStream	6,178	25.4K	18.9K	1010	2,726	7.3	30.9	17
GWT	91,526	444.3K	316K	25.6K	48,721	8.6	25.2	12
Hibernate	100,979	613.8K	384K	64K	94,731	10.5	26.4	10
JDK	396,608	2.42M	638K	18.2K	5,258	7.8	34.7	21
Overall	1,085,114	6.252M	2.804M	506.9K	244,546	8.9	25.9	13

During translation, StatType iterates through all stacks, all candidates in each stack, and all expansion choices to create the new partially translated candidates. The expanded candidates are placed into a stack for larger sizes further down. For pruning, StatType keeps a maximum number k of the top-k candidates in a stack. If stacks are filled and all translation choices are feasible, the number of possible expansions is equal to the maximum stack size times the number of translation choices times the length of the input sentence. The pruning process could lead to the risk that the current bad partial-translated candidates might come back to produce the best one [14].

5.3 Post-Processing

This section presents how StatType performs post-processing to infer the FQNs for the types of the variables in a code snippet. It uses program analysis in the same fashion as PPA [6] to infer the type for a variable after it already resolved the FQNs of the other code elements. To do that, we rely on static analysis via the Java type system. Depending on the role of a variable v in a specific syntax rule, we determine the type. Here are some examples of our rules:

1. Variable declaration: we use the declared type of v.

2. Variable assignment, `v = rhs;`: in this case, we use the inferred type of the right-hand side and assign it to v.

3. Variable as the receiver in a method call, $v.m()$: since we know the FQNs for m, we easily know the type of v.

4. Variable as an argument of a call, $m(...,v,...)$: we use the type of the corresponding formal parameter of m at the position of v.

5. Variable as a formal parameter of a method declaration: we use the FQN of the type of the parameter.

6 EMPIRICAL EVALUATION

To evaluate StatType, we answer the following research questions:

RQ1: *Intrinsic Accuracy.* How accurate is StatType in identifying FQNs for the API elements in Java source code snippets?

RQ2: *Extrinsic Accuracy.* How accurate is StatType in identifying FQNs for the code snippets in StackOverflow posts? How is its accuracy compared to the state-of-the-art approach, Baker [27]?

RQ3: *Sensitivity Analysis.* How do contexts and other factors affect accuracy, *e.g.,* training data' sizes, cardinality, *etc.*?

6.1 Data Collection

We collected the following corpora to train and test StatType.

Large Corpus. For comparison purpose, we choose to evaluate StatType on the same set of libraries that were used in the evaluation of Baker [27], including Android, GWT, Hibernate, Joda Time, and XStream. They were also used in the earlier work [8]. We also added JDK library due to its popularity. For each library, we collected the top-200 Java projects from GitHub that use the library the most based on the numbers of files using its APIs. The projects have long, established development histories with the supporting libraries needed to resolve all APIs in their source code. We used only the projects with fully resolved FQNs. We used Eclipse JDT to parse each project's source code and resolve all the FQNs, and built the source and target sentences for all the methods to form *a parallel corpus* of pairs of sentences. The sentences that do not use any APIs of those 6 libraries were removed. Table 3 shows the statistics of this corpus. It contains 1.085M client methods and 9.562M usages of the APIs including types, method calls, field accesses with 244K unique APIs in the vocabulary. Each sentence has an average length of 25.9 (maximum length of 255) containing on average 8.9 APIs of those 6 libraries. The projects using Android and JDK have larger numbers of methods using the APIs due to their popularity.

Table 4: Community Corpus from StackOverflow Posts

Library	Sentence	#UsedAPIs			#Unique	Density	Sequence length		
	Pairs	Class	M.Inv	F.Acc	APIs		Avg	Med	Max
Android	73	617	361	44	297	14.0	30.4	19	122
Joda Time	57	384	263	5	129	11.4	26.4	24	69
XStream	50	250	211	2	76	9.3	34.7	21	94
GWT	90	698	519	26	246	13.8	24.7	18	123
Hibernate	75	517	317	6	97	11.2	27.4	22	145
JDK	371	2,180	614	140	294	7.9	26.3	19	145
Overall	716	4,646	2,285	223	1,139	10.0	22.5	16	145

Community Corpus. We aimed to have an extrinsic and comparative evaluation for StatType against Baker [27] on a corpus of StackOverflow posts. We followed the instructions in Baker paper to download the SO data repository provided as part of the 2013 Mining Software Repositories Challenge [2]. For each library L of interest, we searched via SO tags for all the posts that use L, further manually verified them, and then randomly selected 50 posts from the results. We considered code snippets in both questions and answers. In total, we collected 300 SO posts of those six libraries. The statistics on the sizes of the posts are in Table 4. There are 716 methods with 7,154 used APIs, 1,139 unique APIs. Each code snippet has an average of 10 APIs in those six libraries with an average of 22.5 code tokens.

6.2 Procedure, Settings, and Metrics

Intrinsic Evaluation. In this setting, we aim to evaluate StatType's accuracy in deriving the FQNs of 6 libraries on Large Corpus. We used 10-fold cross validation. We first randomly split the corpus into 10 folds with equal numbers of projects. We used 9 folds for training and 1 fold for testing, and repeated 10 times for 10 folds.

For each method in a project for testing, we built the source sentence and translated it. In the method, for each of the code tokens whose FQNs need to be derived, we compared the FQN produced by StatType against the expected FQN in the Large Corpus. Note that the corpus contains the fully-resolved FQNs for the APIs.

For each token, if the result returned by StatType correctly matches the FQN in the oracle, we counted as a *correct* case or a hit. As with all statistical learning methods, the model could not derive a FQN that was not seen in the training data. Thus, we also evaluated to what extent un-seen data affects accuracy. If a code token t is not in the vocabulary of the set of source sentences or the correct respective FQN T is not in the vocabulary of the set of target sentences in the training data, we count this as a *miss* case, *i.e.,* the tool cannot give a result. This case can be referred to as *out-of-vocabulary (OOV)*.

Table 5: Accuracy in Intrinsic Evaluation on Large Corpus

Library	Total APIs	Correct	Miss	Precision	Recall
Android	4,242,948	4,169,231	14,906	98.6%	98.3%
Joda Time	353,179	351,192	285	99.5%	99.4%
XStream	47,450	45,563	1,434	99.0%	96.0%
GWT	778,651	721,314	31,253	96.5%	92.6%
Hibernate	1,060,302	896,595	54,302	89.1%	84.6%
JDK	3,079,403	3,063,286	2,161	99.5%	99.5%
Overall	9,561,933	9,247,181	104,341	**97.8%**	**96.7%**

Note that, we also counted as a *miss case* where a mapping was not encountered despite that the source and target were seen in training because the tool cannot give a result. Moreover, for a variable's name, even if it does not appear in the vocabulary, we do not count it as a miss because we still infer its FQN via post-processing. If the result is not a miss and does not match the correct FQN, we counted it as an *incorrect case*. We compute *Precision* (Prec) as the *ratio between the number of correct cases over the total number of inferring/prediction cases*. *Recall* (Rec) is the *ratio between the number of correct cases over the total number of cases*. The tool infers either correctly, incorrectly, or it misses, exclusively.

Extrinsic Evaluation. We also evaluated StatType in its application: deriving the FQNs for SO code snippets in the Community Corpus. Thus, we trained StatType with Large Corpus, and then tested with Community Corpus. Note that the training data is from GitHub, while the testing data is from StackOverflow.

To build the oracle of the correct FQNs for the APIs in a SO snippet, two authors, who did not participate into solution development, copied the snippets into Eclipse, *manually added the required libraries* and made it compiled. They used Eclipse JDT compiler to resolve the FQNs in the snippets and used those FQNs as an oracle to check the tools' results. We then computed Precision and Recall.

7 EMPIRICAL RESULTS

7.1 RQ1. Intrinsic Evaluation Results

Table 5 shows the result when we performed cross validation on Large Corpus. As seen, StatType achieves very high accuracy. The overall Prec is 97.8% and Rec is 96.7% for all the libraries. For JDK, StatType correctly derives the FQNs better than for other libraries (99.5%). Recall of 99.5% means that with an average of 9 folds in Large Corpus for training, almost all JDK APIs in the test sentences are in the training. This is expected since JDK is very popular.

We investigated the "translated" sentences to gain the insights. We found that both the phrase table and the language model contain the sequences providing *the context* of the surrounding APIs for StatType to identify the FQNs of the code elements even if the external references are undeclared/ambiguous. In many cases, candidate lists are long. For example, getText(0) has 229 candidate FQNs in the phrase table. However, as considering the 2-gram [css(0), getText(0)] (line 5 of Figure 1), we have only 4 candidate 2-grams in which getText(0) has two possible FQNs: com.google.gwt.resources.client.TextResource.getText() and com.google.gwt.resources.client.CssResource.getText(). With the 3-gram setInnerText(1), css(0), and getText(0), StatType is able to use the context to pick the right sequence in the phrase table and then the right FQN (the latter one) for getText. Note that, at line 5 of Figure 1, to infer the type for getText, Baker [27] cannot rely on the scope with the same variable because the receiver of getText is another method call (css()), instead of a variable.

Table 6: Accuracy on Various Mapping Cardinalities

Lib	Bin	Precision for Bins with One-to-Many Mappings										
	1-1	Total	2	3	4	≤10	≤20	≤50	≤100	≤500	≤1K	1K+
And	39.3	60.7	12.8	6.0	3.4	8.5	7.3	8.2	8.7	4.9	0.7	0.2
Prec	100	98	98	97	96	97	99	99	98	96	98	99
Joda	31.5	68.5	10.8	7.9	3.7	12.1	10.7	6.5	7.1	5.6	4.0	0.3
Prec	100	96	95	94	96	97	94	96	97	92	94	81
XStr	58.4	41.6	6.3	4.3	1.6	17.7	2.2	2.4	1.6	3.4	2.0	0.3
Prec	100	98	95	98	100	99.7	95	99	97	97	93	67
gwt	62.3	37.7	8.9	3.3	2.0	8.4	1.3	2.2	2.9	4.1	4.4	0.1
Prec	100	99	100	100	99	99.7	98	97	98	97	96	99
Hib	26.7	73.3	11.7	5.2	3.5	9.5	6.8	23	3.6	6.8	1.6	1.4
Prec	100	84	79	84	85	84	89	92	75	77	95	72
JDK	37.3	62.7	11.8	4.6	5.8	10.1	17.4	4.4	3.5	1.1	3.7	0.2
Prec	100	98	100	97	100	99	99	99	97	99	98	98
All	**44.9**	**55.1**	11.2	5.2	3.1	9.0	5.9	7.5	5.9	4.8	2.4	0.3
Prec	**100**	**97**	**96**	**96**	**95**	**96**	**97**	**96**	**97**	**93**	**96**	**84**

As another example, at line 4 of Figure 1, to resolve the FQN for Document, StatType uses the sequences containing the surrounding FQNs of com.google.gwt.dom.client.Document, *e.g.,* Event (com.google.gwt.user.client.Event), StyleElement (com.google.gwt.dom.client.StyleElement), and setInnerText (com.google.gwt.dom.client.Element.setInnerText(java.lang.String)). In fact, those (sub)sequences appear in the phrase and language model tables with higher scores than the same sequence with org.w3c.dom.Document. Thus, this enables StatType to differentiate com.google.gwt.dom.client.Document from org.w3c.dom.Document.

7.2 Accuracy Analysis on Mapping Cardinality

To study StatType's ability to resolve the ambiguity, we performed another experiment to see how well it performs for the API elements with one or multiple possible mapping FQN candidates. We examined the prediction cases and divided them into bins. Bin #1 (**1-1**) contains the cases (shown in %) where an element has only one FQN candidate in the phrase table. In general, bin #*n* contains the cases where a code element has *n* possible FQN candidates for mapping (*n*=1–4). If *n* >= 5, we collected the bins for 5–10 (≤10), 10–20 (≤20),..., 500–1,000 (≤1K), and +1,000 (1K+) possible candidates. We computed the precision for each bin as individual by *taking the number of correct cases and dividing it by the number of cases in that bin*. We care only Prec since we cannot infer for missing cases.

In Table 6, for a library, the first row is the percentage of the cases in a bin over all the cases and the second row is corresponding StatType's Precision for that bin. As seen in the last two rows, in all 6 libraries, there are 44.9% of the APIs that have 1-to-1 mapping, *i.e.,* an API has a single FQN. Thus, StatType is 100% accurate for those cases. Among a total of 9.457M API elements with 244K unique APIs, there is a large percentage (55.1%) of the APIs with multiple FQN candidates. It was able to perform well with 97% precision. Interestingly, the precisions for all the bins except the last one are quite stable and very high (93–98%). That is, even with an *API element having a high cardinality (with multiple FQN candidates)*, StatType *is able to maintain a stable and high accuracy* in identifying the right FQN. A random guess would yield fast-decreasing precisions: 1/2 (for *n*=2), 1/3 (for *n*=3), *etc.* The accuracy drops only to 84% for 0.3% of the cases with +1,000 FQN candidates (*e.g.,*getId, toString). Thus, StatType handles well high-cardinality mappings.

Table 7: Accuracy on StackOverflow Code Snippets

Lib	StatType						Baker		
	Total	Cor	Miss	Prec	Rec	FS	Prec	Rec	FS
And	1,022	1,001	8	98.7	97.9	98.3	97.4	83.4	89.8
GWT	1,243	1,192	9	96.6	95.9	96.3	99.6	86.0	92.3
Hib	840	725	33	89.8	86.3	88.0	99.3	74.7	85.3
Joda	652	639	2	98.3	98.0	98.2	93.6	93.3	93.5
XStr	463	461	1	99.8	99.6	99.7	99.5	80.3	88.9
JDK	2,934	2,903	6	99.2	98.9	99.1	98.1	82.9	89.8
Over	7,154	6,921	59	**97.6**	**96.7**	**97.2**	**97.8**	**83.0**	**89.8**

7.3 RQ2. Extrinsic Evaluation and Comparison

Table 7 shows the result for the evaluation on StackOverflow posts. As shown, StatType also achieves very high accuracy. The overall Prec is 97.6%, Rec is 96.7%, with F-score is 97.2%. The result is consistent with the result from the intrinsic evaluation. The accuracy without considering JDK is one and a half percent lower: Prec of 96.4% and Rec of 95.2% (not shown). That is, StatType works well even with the libraries less popular than JDK.

The result has three implications. First, with Rec of 96.7%, our collected training data is sufficient for the model to encounter and correctly derive the FQNs of 96.7% of the cases in the SO dataset. Second, with a high Prec of 97.6%, the model disambiguates and derives well the FQNs for the APIs that it had learned in the training corpus. Thus, the data-driven approach works well for this problem. Third, with little effort, we were able to build a good training corpus by automatically collecting open-source repositories.

As seen in Table 7, while achieving similar high precision, StatType significantly improves recall over Baker (13.7% absolutely and 16.5% relatively). Regarding the harmony metric F-score, StatType outperforms Baker by 7.4% for all 6 libraries. For Baker [27], we found the following cases in which it does not perform well. First, since Baker relies on scoping rules on the same variables, it did not work for a cascading call, *i.e.,* a method call whose receiver is another call, rather than a variable. Second, for the popular names with a high number of candidates in incomplete code with a single scope, Baker cannot shorten the candidate lists, because the scoping rules are not effective with too little information. Third, Baker did not perform well for the SO posts where the declarations of variables are missing or outside of the current scope of a method.

7.4 RQ3. Sensitivity: Accuracy by Type Context

StatType relies on two contexts: the *type context* (*i.e.,* the APIs often occur together), and the *resolution context* (*i.e.,* the currently derived type depends on the decisions of surrounding APIs). In this experiment, we study the impact on accuracy of *the type context in term of the lengths of phrases* (*i.e.,* API annotation sequences) in the language model and phrase table. We performed intrinsic evaluation on Large Corpus. We varied the maximum lengths of the phrases and measured accuracy. As seen in the column for phrase's length of 1 in Table 8, the result without the context of surrounding elements is the lowest. With the context of one surrounding element (maximum phrase's length of 2), the result improves. With phrase's length from 3–7, a context helps StatType achieve highest accuracy. This is consistent with the result of a prior work which showed that the

Table 8: Impact of Type Context (Phrase Length) on Accuracy

Length	1	2	3	4	5	6	7
Precision	90.9	94.3	97.6	97.8	97.8	97.8	97.8
Recall	89.9	93.4	96.6	96.7	96.7	96.7	96.7

Table 9: Impact of Resolution Context (Stack Size) on Accuracy

Beam Size	1	5	10	20	50	100	200	500	1,000
Precision	85.8	93.4	94.5	95.9	96.3	96.5	97.6	97.7	97.8
Recall	85.1	92.6	93.7	95.1	95.5	95.7	96.7	96.7	96.7

Table 10: Impact of Training Data's Size on Accuracy

Dataset's Size	1 fold	3 folds	5 folds	7 folds	9 folds
Precision	97.3	97.6	97.6	97.6	97.8
Recall	83.1	90.7	94.0	95.9	96.7

n-grams with *n*=3–6 surrounding tokens provide the best context to predict the next API element [20]. In NLP, researchers also found that the *n*-grams for texts with *n*=3–6 also give best prediction for the next token [11]. This result also shows that our model fits well with the problem of type resolution on SO code snippets because they could be short with few API names, and our model does not need a long context to perform well. We used the value of 4 as the maximum length of phrases for the studies in Sections 7.1 and 7.3.

7.5 RQ3. Accuracy by Resolution Context

As explained, we realized resolution context via our beam search strategy. This experiment measures the impact on accuracy of different beam's stack sizes. Table 9 shows accuracy as we performed 10-fold cross-validation on Large Corpus with varied stack sizes. As the stack size is 1, accuracy is much lower. It is expected because pruning occurs many times (the number of translated choices entered on stacks gets smaller), and best candidates might be dropped out of stack. However, with stack size greater than 5, accuracy is much better. The result becomes stable when the size is greater than 200.

7.6 RQ3. Sensitivity: Accuracy by Training Data

As any statistical learning approach, the size of the training corpus affects accuracy. We used the Large Corpus and picked one fold for testing. We then built different training datasets by increasing their sizes with additional data from 1 to 9 folds used for training. One fold is equal to 1/10 of the projects in Table 3. We ran StatType with each dataset. As seen in Table 10, *Recall increases* from 83.1% to 96.7% with more training data because StatType *encountered more APIs and their FQNs*. This also indicates that StatType scales well from small to large datasets. Importantly, as more training data added, *Prec is quite stable* (97.3–97.8%). Thus, it *performs well on the APIs that were learned regardless of the sizes of training data.*

7.7 Running Time and Storage

All experiments were run on a Linux computer with Xeon E5-2620 2.1GHz (with 1 thread, 32GB RAM). We first fixed the phrase length of 7 and varied the data sizes from 1 to 9 folds. Then, we fixed the

Table 11: Statistics on Time and Space Complexity

	Data size (w. Phrase length=7)					Phrase Length (w. 9-fold Data)					
	1-fold	3-folds	5-folds	7-folds	9-folds	1	3	4	5	6	7
Train. time (hrs)	0.8	2.7	5.3	7.6	9.5	5	8.2	8.5	8.8	9	9.5
Testing time (ms)	2.5	2.8	3.2	3.6	4.4	1.9	2.5	3.2	3.5	3.9	4.4
Voc. size (K words)	178	445	604	698	844	844	844	844	844	844	844
Lang model (M phrases)	1.8	4.7	6.5	7.6	9.0	0.5	5.2	9.0	13.4	18.1	21.7
Lang model (GB)	0.2	0.6	0.8	0.9	1.2	0.04	0.6	1.2	2	3.1	3.8
Phrase table (M pairs)	0.6	1.8	2.9	4.0	5.0	0.2	1.4	2.2	3.1	4.0	5.0
Phrase table (GB)	0.2	0.5	0.8	1.1	1.4	0.03	0.3	0.5	0.7	1.0	1.4

data size of 9 folds and varied the phrase lengths from 1–7. As seen, training time is significant, however, it can be done off-line. As we added more folds in training or used longer contexts, training time increased much. However, *suggestion time is less than 4.4ms per suggestion*, thus, our tool is suitable to be interactively used. The vocabulary's size increases as more data added. The number of phrases in the language model increases as more folds added or as the length is increased since more phrases are added. Similar observations are true for the size of a phrase table. In brief, storage cost is reasonable.

7.8 Potential Applications

We built StatType based on the phrase-based SMT tool Phrasal [5]. Let us discuss a few applications of StatType. First, one could build a plugin for a Web browser or a component within StackOverflow system to automatically resolve the FQNs for the code snippets and link all the APIs to their online documentation as in Baker [27].

As another application, one could use StatType to build a plugin in an IDE to help resolve the types of APIs in a code snippet when it is first copied to the IDE. The plugin will suggest users to download the right libraries. For example, at line 8 of Figure 2, the type ParagraphElement is unknown to an IDE since a developer does not know that the ParagraphElement belongs to GWT, and did not declare it in the classpath of the IDE yet. Such a plugin with StatType would find a match for it with the FQN com.google.gwt.dom.client.ParagraphElement, thus, save developer's effort in searching for types.

In general, StatType could be used in automated tools requiring type resolution for incomplete code, *e.g.,* in API usage pattern mining. Higher recall and precision in type resolution leads to more meaningful patterns and less false positives in anomaly detection.

7.9 Limitations

StatType also has shortcomings. First, out-of-vocabulary affects the recall of our tool. However, as seen, simply using open-source projects that have most usages of a library allows us to achieve high accuracy. Second, we found that in 82.3% of the wrong cases, StatType chose another FQN in the same package with the correct one, *e.g.,* the sub-class android.os.Bundle.putString() was chosen, while the correct one is the parent class android.os.BaseBundle.putString(). This happened often in Hibernate project, leading to lower accuracy. Among them, a small percentage of incorrect cases is due to the incorrect selection of the same type but in a different version of the library. Third, in 17.7% of the wrong cases, the tool picked the incorrect FQNs from different packages. Other incorrect cases occurred when equivalent classes could be used but a less popular one is the correct one. Finally, we did not take advantage of dependencies among types in a code snippet. A potential alternative is to combine statistical learning in StatType and program analysis [6].

Threats to Validity. Our corpus in Java only might not be representative. For dynamic language, *e.g.,* JavaScript, our approach could be applicable if we use the same technique as in Baker by using JS parser (*e.g.,* ESPRIMA parser) to construct the static types.

We do not study the usefulness involving human subjects. The results for other libraries might vary. But for comparison, we used the same set of libraries and SO dataset as in Baker [27]. From their paper, we re-implemented Baker and ran on our 300 SO posts. The replicated results are comparable with the results in their paper.

8 RELATED WORK

Our work is closely related to Baker [27]. It builds a candidate list for each name. It tracks the scopes of the names and the candidate lists are combined according to the scoping rules and get smaller. The process starts again when Baker visits the entire AST. In comparison, StatType uses SMT with a parallel corpus. We also do not need to build partial ASTs, which is not always feasible. Moreover, we use the context of surrounding API names to decide the FQNs with SMT. Another related solution is Partial Program Analysis (PPA) [6] where it resolves the types by applying a set of heuristics on the syntactic constructs to infer the declared types of expressions.

RecoDoc [8] uses PPA to infer links between APIs and documentation. We could integrate PPA in pre- or post-processing to improve accuracy. ACE [24] analyzes the texts surrounding the code snippets in SO posts. It focuses on the code elements embedded within texts, rather than code snippets. We could extend StatType to integrate the knowledge of surrounding texts in SO posts during translation. JSNice [23] leverages conditional random fields to perform joint prediction of type annotations of variables in JS code. An interesting direction is to combine such prediction and our SMT-based solution.

Several approaches including Information retrieval (IR) are used in linking texts in documentation to code, *e.g.,* text matching [3], Latent Semantic Indexing [9, 18], revised Vector Space Model [28, 30], Latent Dirichlet Allocation [1, 21], Structure-oriented Information Retrieval [19, 25], feature location [17]. Others use learn-to-rank [29], deep learning [16], classifiers [12], etc.

9 CONCLUSION

We propose StatType, a statistical approach to resolve the fully qualified names for the APIs in SO code snippets. StatType derives the FQN for an element by using the surrounding code elements as the context, taking advantage of the FQNs that often co-occur in a large code corpus. Our empirical evaluation on real-world code snippets shows that StatType achieves high accuracy and performs 16.5% relatively higher than the state-of-the-art approach Baker [27].

StatType relies heavily on the principle of code regularity, however, achieves a very high accuracy (96–97% precision and recall) and even higher than the existing rule-based approaches that use program analysis [6, 8, 27]. That is, the data-driven approach works very well to derive the types despite that we did not hardwire much program semantics. An interesting future direction is to combine StatType and program analysis to achieve the best of both worlds.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 95–104. ACM, 2010.

[2] A. Bacchelli. Mining challenge 2013: Stackoverflow. In *the 10th Working Conference on Mining Software Repositories*, page to appear, 2013.

[3] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 375–384. ACM, 2010.

[4] P. F. Brown, V. J. Della Pietra, S. A. Della Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguist.*, 19(2):263–311, June 1993.

[5] D. Cer, M. Galley, D. Jurafsky, and C. D. Manning. Phrasal: A toolkit for statistical machine translation with facilities for extraction and incorporation of arbitrary model features. In *Proceedings of the NAACL HLT 2010 Demonstration Session*, HLT-DEMO '10, pages 9–12, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[6] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328. ACM, 2008.

[7] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 127–136. ACM, 2010.

[8] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 47–57. IEEE Press, 2012.

[9] A. De Lucia, R. Oliveto, and G. Tortora. Adams Re-trace: Traceability Link Recovery via Latent Semantic Indexing. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 839–842. ACM, 2008.

[10] Add GWT Widgets inside an iFrame in UIBinder? http://stackoverflow.com/questions/34595450/add-gwt-widgets-inside-an-iframe-in-uibinder/.

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

[12] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[13] K. Knight. Decoding complexity in word-replacement translation models. *Comput. Linguist.*, 25(4):607–615, Dec. 1999.

[14] P. Koehn. *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[15] P. Koehn, F. J. Och, and D. 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, pages 48–54. Association for Computational Linguistics, 2003.

[16] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 476–481. IEEE CS, 2015.

[17] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243. ACM, 2007.

[18] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135. IEEE Computer Society, 2003.

[19] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120. ACM, 2011.

[20] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE'15. IEEE CS, 2015.

[21] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 263–272. IEEE CS, 2011.

[22] C. Parnin and C. Treude. Measuring API documentation on the Web. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, Web2SE '11, pages 25–30. ACM, 2011.

[23] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124. ACM, 2015.

[24] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841. IEEE Press, 2013.

[25] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, pages 345–355. IEEE CS, 2013.

[26] J. Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 139–. IEEE CS, 1998.

[27] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 643–652. ACM, 2014.

[28] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE CS, 2014.

[29] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 689–699. ACM, 2014.

[30] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24. IEEE Press, 2012.