API Code Recommendation using Statistical Learning from Fine-Grained Changes

Anh Tuan Nguyen[1], Michael Hilton[2], Mihai Codoban[3], Hoan Anh Nguyen[1],
Lily Mast[4*], Eli Rademacher[2], Tien N. Nguyen[1], Danny Dig[2]
[1]Department of Electrical and Computer Engineering, Iowa State University, USA
[2]School of EECS, Oregon State University, USA
[3]Microsoft, USA
[4]College of Engineering and Computer Science, University of Evansville, USA

ABSTRACT

Learning and remembering how to use APIs is difficult. While code-completion tools can recommend API methods, browsing a long list of API method names and their documentation is tedious. Moreover, users can easily be overwhelmed with too much information.

We present a novel API recommendation approach that taps into the predictive power of repetitive code changes to provide relevant API recommendations for developers. Our approach and tool, APIREC, is based on statistical learning from fine-grained code changes and from the context in which those changes were made. Our empirical evaluation shows that APIREC correctly recommends an API call in the first position 59% of the time, and it recommends the correct API call in the top 5 positions 77% of the time. This is a significant improvement over the state-of-the-art approaches by 30-160% for top-1 accuracy, and 10-30% for top-5 accuracy, respectively. Our result shows that APIREC performs well even with a one-time, minimal training dataset of 50 publicly available projects.

CCS Concepts

•**Software and its engineering** → **Software evolution;** *Integrated and visual development environments;*

Keywords

API Recommendation; Fine-grained Changes; Statistical Learning

1. INTRODUCTION

Today's programs use Application Programming Interfaces (APIs) extensively: even the "Hello World" program invokes an API method. One great challenge for software developers is learning and remembering how to use APIs [10, 25, 38, 40, 45].

The state-of-the-practice support for working with APIs comes in the form of code-completion tools integrated with IDEs [11, 19, 20]. Code completion tools allow a user to type a variable and request a possible API method call recommendation. Code completion tools are among the top-5 most used features of IDEs [33]. Still, a developer learning an API (or trying to remember it) can waste a lot of time combing through a long list of API method names available on a receiver object. For example, invoking the code completion on an object of type `String` in JDK 8 populates a list of 67 possible methods (and 10 additional methods inherited from superclasses).

The state-of-the-art research in code completion takes advantage of API usage patterns [7, 12, 36, 44], which researchers mine via the deterministic algorithms such as frequent itemset mining, pair associations, frequent subsequence or subgraph mining. When a recommendation is requested, these approaches analyze the surrounding context. If the context matches a previously identified pattern, the recommender will suggest the rest of the API elements in the pattern. Other approaches [3, 12, 16, 24, 35, 37, 42, 52] use *statistical learning* via language models to recommend the next token, including API calls. They rely on the regularity of source code [16] and create a model that statistically learns code patterns from a large corpus. The model then can predict what token is likely to follow a sequence of given code elements. A key limitation to the approach is that it is difficult to determine which tokens belong to a project-specific code idiom. These tokens produce noise for recommendation.

We present a novel approach to code completion that leverages the regularity and *repetitiveness of software changes* [1, 34]. Our intuition is that when developers make low-level changes, even non-contiguous changes are connected. These connections exist because the developer made the changes with a *higher-level intent* in mind (e.g., adding a loop collector). Grouping fine-grained changes by higher-level intent allows us to cut through the noise of unrelated tokens that may surround the recommendation point. To find these groups of fine-grained changes, we use statistical learning on a large code change corpus. The changes that belong to higher-level intents will co-occur more frequently than non-related changes.

Additionally, we also consider the surrounding code context at the recommendation point. For example, when adding a loop collector, while the code tokens '`for`' and '`HashSet`' were not changed, they are good indicators for a tool to recognize that high-level intent. Thus, being aware of the code context, a tool would recommend correctly the next token, e.g., `HashSet.add`.

We implemented our approach in a tool, APIREC, that computes the most likely API call to be inserted at the requested location where an API call would be valid. APIREC works in three steps: (i) it builds a corpus of fine-grained code changes from a training set, (ii) it statistically learns which fine-grained changes co-occur, and (iii) it computes and then recommends a new API call at a given location based on the current context and previous changes.

For the first step, we trained our model on a corpus of fine-grained code changes from the change commits in 50 open-source projects from GitHub. APIREC iterates over 113,103 commits and detects

*Work done while being an intern at Oregon State University

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950333

the differences in 43,538,386 Abstract Syntax Trees (AST) nodes using the state-of-the-art AST diff tool GumTree [13].

For the second step, we developed an association-based inference model that learns which changes frequently co-occur in the same changed file. Additionally, the model operates on the code context of fine-grained changes (e.g., for loops, preceding method calls).

In the third step, using the change context of previous changes, the code context of the recommendation point, and the trained inference model, APIREC determines the likelihood of a user inserting an API method call at this location. If it determines that an API method insertion is indeed likely, then it returns a list of candidate API calls ranked by the computed likelihood of being selected by a developer.

To empirically evaluate the usefulness of our approach, we answer three following research questions.

RQ1: ***Accuracy*** How accurate is APIREC in suggesting API calls?

RQ2: ***Sensitivity Analysis*** How do factors such as the size of training data, the requested location, the sizes of the change context, and code context impact accuracy?

RQ3: ***Running Time*** What is the running time of APIREC?

To answer **RQ1** we measure the accuracy of the recommender. Top-*k* accuracy measures how likely the correct API is in the first *k* recommended APIs. We measure the accuracy in three different evaluation editions. In the *community edition*, we first train APIREC on 50 open-source projects and then measure APIREC's accuracy on a corpus of 8 projects that other researchers [16, 35] have previously used. In the *project edition*, we do a 10-fold validation on each of the 8 above projects. For the *user edition*, we also do a 10-fold validation on the same 8 projects as above, but only on the commits coming from a single user. To answer **RQ2**, we studied the impact of several factors on accuracy, e.g., the size of training data, previous changes, surrounding context, and the location of recommendation invocation. To answer **RQ3**, we look at the running time of APIREC. For each evaluation, we compare APIREC with the previous state-of-the-art learning approaches: *n*-gram [42], Bruch *et al.* [7], and GraLan [35]. This paper makes the following contributions:

1. **Approach.** We present a novel approach that uses statistical learning on fine-grained changes with surrounding code context to create a new code-completion tool. We set forth a new direction that takes advantage of the *repetiveness of both source code and fine-grained code changes*.
2. **Implementation.** We implemented our approach in a tool, APIREC, that computes the most likely API method call to be inserted at the requested location in the code.
3. **Empirical Evaluation.** Our empirical evaluation on real-world projects shows that APIREC has high accuracy in API code completion: 59.5% top-1 accuracy. This is an improvement over the state-of-the-art approaches: 30–160% for top-1 accuracy. Our evaluation shows that APIREC also performs well even with a one-time, minimal training dataset of 50 publicly available projects. Interestingly, we found that *considering code authorship, one could train with less data, yet achieve higher accuracy than when training with an entire project*. Training the model with the community corpus still results in higher accuracy than training with the data from the project or an individual developer. This finding suggests that *developers should obtain a community trained model, and then further refine it with their own change histories*.

2. MOTIVATING EXAMPLE

Figure 1 shows a real-world example that we collected from prior work [34] in mining fine-grained code changes. This is a common change pattern called *Adding a Loop Collector*. In this change, a

```
1  for (Task t : tasks) {
2    t.execute();
3  }                                                    a)
```

⇓

```
1  Set<TaskResult> results = new HashSet<>();
2  for (Task t : tasks) {
3    t.execute();
4    results.add(t.getResult());
5  }                                                    b)
```

Figure 1: A Change Pattern: Adding a Loop Collector

developer introduces a new variable that collects or aggregates the values processed in a loop. In the example, the fine-grained code changes include the changes at line 1 and line 4 in Figure 1b. Specifically, the changes at line 1 include the addition of the declaration of the variable results with the type Set<TaskResult>, and the addition of its instantiation via new HashSet<>(). Line 4 has the additions of two method calls results.add() and t.getResult().

Assume that the current editing location is at line 4 of Figure 1b after a developer has typed the changes at line 1 and the name of the variable results. (S)he then triggers the code-completion tool. Here is how different tools respond to their request. The de-facto modern IDE will present the list of methods and fields of HashSet in a pre-defined (usually alphabetical) order. (S)he must browse through a list of 37 methods to find the desired method. Advanced code completion engines [7, 35, 42] will recommend a list of API calls based on the API usage patterns that are mined from the code corpus. There are two common mining strategies. The first strategy relies on deterministic mining algorithms such as frequent itemset, common subsequence, or common subgraph mining [7, 36, 44]. The second one uses statistical learning from code [16, 42, 52]. Both strategies share the same principle that *source code is repetitive* [16]. Code suggestion techniques (e.g., *n*-gram model [16, 52]), which rely on the code context, identify the context as the sequence of tokens preceding the variable results (in our example, the sequence 't.execute();'). However, this code sequence is specific to this project, and is not part of any code pattern related to the method add. Thus, tools based on code patterns might not recommend the correct API call HashSet.add. More advanced approaches [35, 42], which also consider program dependencies among the entities (e.g, at line 1 and line 4), still might not see HashSet.add as a good candidate because other API calls from HashSet are just as likely to occur.

Key Ideas

Instead of relying on source code repetitiveness, APIREC is based on *code change repetitiveness* [34]. Hindle *et al.* [16] reported that software exhibits its naturalness: source code has a higher degree of repetitiveness than natural-language texts. We expect the same principle of *naturalness of software* [16] to occur on fine-grained code changes (i.e., *naturalness of code changes*) since similar changes may be performed to introduce similar behavior. In our example, the addition of a new HashSet (generally a Collection) is followed by the addition of the call HashSet.add on the same variable. Since APIREC observed the addition of a new HashSet in the current change context, it is able to correctly suggest HashSet.add at line 4. For APIREC to work, we rely on the following key ideas:

1) First, we develop an **association-based model** to *implicitly* capture change patterns, i.e., frequent, co-occurring, fine-grained code changes in our training data. Our insight for using associations among fine-grained changes is that such changes in a pattern do not need to have strict order as required in a traditional *n*-gram

model. For example, HashSet.add could be edited before or after the addition of the declaration of the HashSet variable.

2) Second, the recent fine-grained code changes in the **change context** of the current code lead the trained model to recommend the next method call (e.g., the addition of a HashSet object often leads to the call HashSet.add). Moreover, we use the **code context** surrounding the requested location, which might contain the code tokens that are part of the change patterns. For example, the token 'for' is part of both the code context and the change pattern of *Adding a Loop Collector* because users often collect the elements into a collection via a for loop. Thus, it helps suggest the method call add at line 4.

3) Third, not all the changes in the current context are useful in recommendation because they can be project-specific and considered as noise in the change patterns. Recent work [41] confirms that not all code changes are repetitive. To address this, we rely on the **basis of consensus**: given a large number of changes in many projects, the project-specific changes are less likely to appear frequently than the changes belonging to a higher-level intent pattern.

3. DEFINITIONS

3.1 API Call Completion

An *API (method) call* is a call to an API of an external or internal library, while a *method call* is a call to a method within a project. For brevity, we call both types API calls. We distinguish them if needed. In traditional code completion, a programmer invokes code completion by placing a dot after a variable, such as v. APIREC supports any recommendation where the addition of a method call would result in syntactically correct code. For example, after the = sign in an assignment, as in v = _, it recommends a method call.

3.2 Fine-grained Atomic Code Changes

We represent source code as ASTs to avoid formatting changes which are not helpful for suggestion. For a changed file, we compare the ASTs before and after the changes to derive the fine-grained changes. The state-of-the-art differencing tool GumTree [13] is used.

DEFINITION 1 (ATOMIC CHANGE). *A (fine-grained) atomic change is represented by a triplet of* **(<operation kind>, <AST node type>, <label>)**.

Table 1 shows the atomic changes for the editing scenario in Figure 1. Each atomic change corresponds to *a change to an AST node* in the program. We consider each change to be one of the following operations: change, add, delete, and move. The AST node types represent the Java AST nodes. The labels represent the textual information of the AST nodes. We only use labels for some types of AST nodes. For *method invocation*, *simple type*, and *simple name* (of a method invocation or a simple type), we use the labels to identify the name of a method or type. We use the name to find change patterns and to recommend. For example, in Table 1, two SimpleTypes have their labels of Set and TaskResult. We also keep the labels for *boolean constants* and *the null value*. These labels are special literals that help in detecting change patterns involving those values. For the AST nodes for which we keep the labels, in addition to node types and operation kinds, we use the labels when comparing ASTs.

DEFINITION 2 (TRANSACTION). *Atomic changes from the same changed file in a commit are collected into a* **transaction**.

We use a *bag* (or multiset, which allows for multiple instances of an element) to represent a transaction, rather than a list, since the atomic changes might be different depending on each programmer, even though they belong to the same change pattern. Thus, if we establish a strict order, we might not be able to statistically learn the patterns for recommendation. Moreover, because the atomic code changes are recovered from the committed changes in a code repository, we do not have the order in which they were written.

3.3 Change Context and Code Context

DEFINITION 3 (CHANGE CONTEXT). *The* **change context** *is the bag of fine-grained atomic changes that occurred before the requested change at the current location in the same editing session.*

The change context at the underlined location in Figure 1 contains atomic changes at line 1 (partially shown in Table 1), and the addition of results (not shown for space reason). They are useful in recommending the method add at line 4. The instantiation of a HashSet object and the call to the method add are part of a change pattern. Identifying these changes as the beginning of a pattern helps APIREC recommend the addition of HashSet.add. We also give a larger weight to the changes made to the program elements that have data dependencies with the current code element (i.e., results), because they are more likely to occur together in a change pattern than other elements with no data dependency. We currently consider only the dependencies between variables' definitions and their uses, and between the method calls on the same variable.

DEFINITION 4 (CODE CONTEXT). *The* **code context** *is the set of code tokens that precede the current editing location within a certain distance in terms of code tokens.*

We obtain the **code tokens** from the AST. For example, the tokens for, Task, t, tasks, and execute will be used as the code context to recommend the API call HashSet.add. The rationale is that the tokens surrounding the recommendation point might often go together with the API call as part of a pattern even though they might not be recently changed. Thus, the code context helps us recommend the correct API call. We do not consider separators and punctuation.

For both change and code contexts, we consider the **distance** and the **scope** of a change and token. We attribute a higher impact to the preceding changes or code tokens that are nearer to the current location. Thus, we give them higher *weights* in the decision process. We measure the distance by the number of code tokens in the program. Since we focus on recommending API calls in a method, we give higher weights to the changes and code tokens within the method under edit, and lower weights to the changes/tokens outside of it.

4. CHANGE INFERENCE MODEL

To rank and recommend the candidates of API calls, APIREC uses our association-based change inference model. The model learns from a fine-grained change corpus to compute *the likelihood scores for each candidate change c to occur at the requested location given both the change and code contexts*. To do that, it computes the contributions of individual changes and code tokens in the contexts by counting in the corpus *the co-occurrence frequency of each atomic change in the change context with c* and *the frequency of each code token in the code context appearing in the change c*. Finally, the contributed scores are integrated and adjusted via the weighting factors for the distances between changes and the scope of changes.

4.1 Model Overview

The goal of our model is to compute the likelihood score that a change c occurs at the requested location given (i) the fine-grained code changes in the **change context** $\mathscr{C}$ preceding the current change c, and (ii) the code tokens of interest in the **code context** $\mathscr{T}$ surrounding the requested location. c is in the form <add, MethodInvocation, methodName> and APIREC needs to predict methodName.

Table 1: Fine-grained Atomic Code Changes

Ind.	Oper.	AST Node Type	Label	Content of AST's Sub-Tree
c_1	Add	VariableDeclarationStatement (VDS)	VariableDeclarationStatement	Set <TaskResult> results = new HashSet<>();
c_2	Add	ParameterizedType (PT)	ParameterizedType	Set <TaskResult>
c_3	Add	SimpleType (ST)	Set	Set
c_4	Add	SimpleName (SN)	Set	Set
c_5	Add	SimpleType (ST)	TaskResult	TaskResult
c_6	Add	SimpleName (SN)	TaskResult	TaskResult
c_7	Add	VariableDeclarationFragment (VDF)	VariableDeclarationFragment	results = new HashSet <>();
...	...	...	...	...

The occurrence likelihood $Score(c,(\mathscr{C},\mathscr{T}))$ of a change c is a function of c, $\mathscr{C}$, and $\mathscr{T}$. Generally, learning that function from data is challenging. To make it computable, we assume that both change and code contexts have independent impacts on the occurrence of the next token. We denote the impacts of change and code contexts on the occurrence of c by $Score(c,\mathscr{C})$ and $Score(c,\mathscr{T})$, respectively. We aim to learn the occurrence likelihood $Score(c,(\mathscr{C},\mathscr{T}))$ of change c in the form of a *weighted, linear combination of two impacts*:

$$Score(c,(\mathscr{C},\mathscr{T})) = w_C \times Score(c,\mathscr{C}) + w_T \times Score(c,\mathscr{T})$$

where w_c and w_T are the weights corresponding to the impacts of contexts. Other types of combinations can be explored in future.

To compute the impact of the change context via $Score(c,\mathscr{C})$ with $\mathscr{C}$ consisting of the atomic changes $c_1,c_2,...,c_n$, we adapted the concept of trigger pairs by Rosenfeld [23,46] in language models for natural-language texts. That is, if a word sequence A is significantly correlated with another word sequence B, then $(A \rightarrow B)$ is considered as a trigger pair. When A occurs before (does not need to be within a n-gram), it triggers B, causing its probability estimate to change [46].

While Rosenfeld considers words, we consider each of the changes in the context $\mathscr{C}$ as a trigger where the order among c_is is not needed as explained in Section 2. However, instead of using maximum entropy [21], in APIREC, we approximate $Score(c,\mathscr{C})$ by the product of each trigger pair $Score(c,c_i)$ $i = 1...n$ because the number of changes in a change context is much smaller than the number of words in a document. We then compute each trigger pair $Score(c,c_i)$ via the conditional probability $Pr(c|c_i)$. This probability can be estimated with their association score, i.e., the ratio between the number of transactions having both changes c and c_i over the number of transactions having c_i. Finally, we also incorporate the weights for distance between the two changes and the scope of a change. We similarly compute the impact $Score(c,\mathscr{T})$ of the code context via the trigger pair $Score(c,t_i)$ with t_i is a token in $\mathscr{T}$.

4.2 Details on Model Computation

4.2.1 Computing $Score(c,\mathscr{C})$

The value of $Score(c,\mathscr{C})$ represents the impact of the atomic changes in the change context for predicting c. One could compute:

$$Score(c,\mathscr{C}) = Score(c,\{c_1,c_2,...,c_n\}) = \frac{N(c,c_1,c_2,...,c_n)}{N(c_1,c_2,...,c_n)} \quad (1)$$

where $N(c_1,c_2,...,c_n)$ is the number of transactions containing the changes $c_1,c_2,...,c_n$, and $N(c,c_1,c_2,...,c_n)$ is the number of transactions containing the changes $c,c_1,c_2,...,c_n$ including change c.

However, we cannot always compute those numbers. This is because in the training data, we might not frequently encounter the cases where those changes occurred in the same transaction. That is, we might have a small number of such co-appearance. Thus, we adapted the trigger pair concept to compute Equation 1 as follows.

$$\begin{aligned} &Score(c,\mathscr{C}) = Score(c,\{c_1,c_2,...,c_n\}) \\ &\simeq Score(c,c_1) \times Score(c,c_2) \times ... \times Score(c,c_n) \end{aligned} \quad (2)$$

In maximum entropy [21], the impact of a term on the presence of another is modeled by a set of constraints. Their intersection is the set of probability functions that are consistent for all the terms. The function with the highest entropy in that set is the ME solution [46].

In APIREC, because the number of changes in $\mathscr{C}$ is usually small, we do not aim to find the combined estimation. Instead, we estimate $Score(c,c_i)$ (i.e., the probability that c occurs given that c_i occurred) as the following and then take the product of all scores:

$$Score(c,c_i) \simeq Pr(c|c_i) \simeq \frac{N(c,c_i)+1}{N(c_i)+1}$$

where $N(c,c_i)$ is the number of transactions in which the changes c and c_i co-appear, and $N(c_i)$ is the number of transactions having c_i. The ratio represents the **association score** between two changes.

To account for the distance between a change c_i and the current change c and to avoid underflow, we use the logarithmic form:

$$\log(Score(c,c_i)) \propto \frac{1}{d(c,c_i)} \times \log \frac{N(c,c_i)+1}{N(c_i)+1}$$

where $d(c,c_i)$ is the distance between c and c_i, which is measured as the number of tokens in the code between the two tokens corresponding to the two changed nodes of c and c_i. We sort the changes c_i according to their distances to c. The smaller the distance, the higher the change ranks. Then, we use the rank for a change c_i as its distance $d(c,c_i)$. The *log* form is to avoid underflow in computation.

Because c = <add, MethodInvocation, methodName> and we want to recommend an addition of a method invocation, the method name (denoted by mname) is the only variable in c. Thus, we have

$$\log(Score(c,c_i)) \propto \log Score(c_{mname}|c_i) \simeq \frac{1}{d(c,c_i)} \times \log \frac{N(c,c_i)+1}{N(c_i)+1}$$

c_{mname} is an addition of a method call with the name of mname.

To consider the scope of the changes (e.g., the changes outside of the current method are weighed lower than those occurring inside it), we set different constants for the weights of the factors:

$$\log(Score(c,c_i)) \propto \log Score(c_{mname}|c_i) \simeq \frac{w_{scope_{ci}}}{d(c,c_i)} \times \log \frac{N(c,c_i)+1}{N(c_i)+1}$$

$w_{scope_{ci}}$ is the weight accounting for the scope of the change c_i. It equals 1 if c_i occurs in the current method of c, and equals 0.5 if c_i occurs outside of the method. Similarly, we set the values of the weights $w_{dep_{ci}}$ for the changes to code elements having data dependencies with the current element. Finally, from Equation 2:

$$\begin{aligned} &\log(Score(c,\mathscr{C})) \propto \log(Score(c_{mname},\mathscr{C})) \\ &\simeq \log(Score(c,c_1)) + \log(Score(c,c_2)) + ... + \log(Score(c,c_n)) \\ &\simeq \textstyle\sum_{i=1..n} \frac{w_{scope_{ci}} \times w_{dep_{ci}}}{d(c,c_i)} \times \log \frac{N(c,c_i)+1}{N(c_i)+1} \end{aligned} \quad (3)$$

4.2.2 Computing $Score(c,\mathscr{T})$

We estimate the likelihood score $Score(c,\mathscr{T})$ of c given the code context $\mathscr{T}$ in the same manner as the computation for the score

$Score(c,\mathscr{C})$. $Score(c,\mathscr{T})$ is estimated according to:

$$\begin{aligned}&\log(Score(c,\mathscr{T})) \propto \log(Score(c_{mname},\mathscr{T}))\\ &\simeq \log(Score(c,t_1)) + \log(Score(c,t_2)) + ... + \log(Score(c,t_m))\\ &\simeq \sum_{i=1..m} \frac{w_{scope_{ti}} \times w_{dep_{ti}}}{d(c,t_i)} \times \log \frac{N(c,t_i)+1}{N(t_i)+1}\end{aligned} \quad (4)$$

In this formula:

1. $t_1, t_2, ..., t_m$ are m code tokens of interest in the code context.

2. $Score(c,t_i)$ is the likelihood score that the code token t_i in the surrounding context (e.g., the token for) indicates the occurrence the change c (e.g., the addition of HashSet.add).

3. $w_{scope_{ti}}$ is the weight on the scope of the token t_i. It equals 1 if t_i is within the current method of c, and equals 0.5 if t_i is outside.

4. $d(c,t_i)$ is the distance between the token t_i and the requested location. It is computed similarly as $d(c,c_i)$.

5. $N(c,t_i)$ is the number of transactions in which the token t_i is in the nearby code context of the change c in the change history. $N(t_i)$ the number of transactions in which t_i appeared.

From Formulas 3 and 4, we compute $Score(c,\mathscr{C})$ and $Score(c,\mathscr{T})$.

5. TRAINING AND RECOMMENDATION

Before APIREC can recommend API calls, we must first train it. We will explain how we train and use the model for recommendation.

5.1 Learning Change and Code Co-occurrences

According to Formulas 3 and 4, to be able to compute $Score(c,\mathscr{C})$ and $Score(c,\mathscr{T})$, APIREC needs to learn three types of parameters:

1) the numbers of (co-)occurrences of the fine-grained atomic changes, i.e., $N(c,c_i)$ and $N(c_i)$ in Formula 3

2) the numbers of (co-)occurrences of atomic changes and code tokens of interest, i.e., $N(c,t_i)$ and $N(t_i)$ in Formula 4.

3) The weights $w_{\mathscr{C}}$ and $w_{\mathscr{T}}$, one of which we fix by using $w_{\mathscr{C}} + w_{\mathscr{T}} = 1$.

We use hill-climbing adaptive learning [50] to learn the value for $w_{\mathscr{C}}$ from a training set. The idea of the training algorithm to adjust that weight is via gradient descent as follows. First, it is initialized with a value. We train on $(k-1)$ folds and test on one fold. The parameters of the trained model is used to estimate the scores $Score(c,\mathscr{C})$ and $Score(c,\mathscr{T})$. The combined score is computed with the current value of the weight $w_{\mathscr{C}}$. The candidates for c are ranked. We compute the goal function MAP (m_{actual}, P_{list}) between the list of predicted method calls, P_{list}, and the actual one, m_{actual}. The weight is then adjusted. The process is repeated. Finally, the optimal weight corresponding to the highest value of MAP is used.

5.2 API Call Recommendation

After all above parameters were trained, we use the Formulas 3 and 4 with all the occurrence counts obtained during training to estimate the likelihood of a change c, i.e., an addition of a method invocation with the method name mname. We compute the occurrence likelihood for all candidate changes in the vocabulary that satisfy the following: (i) it is an addition of a method invocation, and (ii) it has appeared in at least one transaction with at least one change in $c_1, ..., c_n$, or in at least one transaction with at least one token in $t_1, ..., t_m$. The second condition avoids the trivial cases of zero occurrences. Finally, we rank the candidates by their scores.

Let us explain this computation using the example in Figure 1. Let us assume that the programmer finishes the changes and stops right after typing the variable results at line 4 of Figure 1b. They request APIREC to complete the code with an API method invocation. Our goal is to recommend a method call for the variable results.

Table 2 shows the computation of the scores. All the atomic changes that preceded the current location are collected into the change context $c_1, c_2, ..., c_{13}$ (assume that in this example, there are no other changes outside of the method). The tokens prior to the current location are considered including $t_1, t_2, ..., t_6$ (i.e., the code context). The token types and labels are presented in Table 2.

Table 2: Example of Score Calculation for Candidates (bold are highest component scores, see operation notations in Table 1)

Ind.	<Operation,Type,Label>	*Score*(a candidate given c_i)				
	Candidates	add	remove	contains	addAll	clear
c_1	Add, VDS, VDS	0.02	0.01	0.02	0.02	0.03
c_2	Add, PT, PT	0.01	0.02	0.02	0.03	0.02
c_3	Add, ST, Set	**0.20**	**0.11**	**0.13**	**0.08**	**0.10**
c_4	Add, SN, SN	0.01	0.01	0.02	0.01	0.01
...	...	...	...	...	...	...
c_{11}	Add, ST, HashSet	**0.22**	**0.12**	**0.12**	**0.09**	**0.09**
...	...	...	...	...	...	...
c_{13}	Add, ES, ES	0.02	0.02	0.01	0.02	0.01
Ind.	<Token, Type, Label>	*Score*(a candidate given t_i)				
t_1	Token, FOR, for	**0.15**	**0.01**	**0.13**	**0.02**	**0.02**
t_2	Token, Type, Task	0.00	0.00	0.00	0.00	0.00
t_3	Token, Var, t	0.00	0.00	0.00	0.00	0.00
...	...	...	...	...	...	...
t_6	Token, MI, execute	0.00	0.00	0.00	0.00	0.00
	Combined score	0.40	0.20	0.24	0.11	0.13
	Rank	1	3	2	5	4

The candidate method invocations that satisfy the above conditions (i) and (ii) are listed in the columns including the method calls add, remove, contains, addAll, and clear.

The scores in Table 2 shows the likelihood scores $Score(c,c_i)$ (representing how likely the change c occurs given c_i occurred in the change context of interest) and $Score(c,t_i)$ (representing how likely the change c occurs given the token t_i appear in the code context). $Score(c,c_i)$ and $Score(c,t_i)$ are computed by Formulas 3 and 4.

For example, the scores for the candidates with respect to the previous change c_3 (<Add, ST, Set>: adding a simple type Set) and to the previous change c_{11} (<Add, ST, HashSet>: adding a simple type HashSet) are higher than others since in the training data, the changes involving adding a variable with the type Set or HashSet often co-occur with the changes involving the add method of the variable of that type. That is, the change pattern consists of an addition of a variable of the type Set or HashSet followed by an addition of a call to the add method of that variable. Both of the scores for Set and HashSet are high, since in the training data, programmers might declare the type of HashSet in some cases, and that of Set in others.

The scores for the candidates with respect to the prior tokens are computed similarly. For example, in Table 2, the scores for the candidates with respect to the token 'for' is higher than those of the other tokens because a 'for' iteration and the API method call HashSet.add is part of the change pattern *Adding a Loop Collector*.

Among the candidate method calls, (more precisely, the candidate changes with the change kind of add and the change AST node of MethodInvocation), the method call add has the highest scores. This is reasonable because programmers often use the method to collect elements into a collection via a loop. Finally, the combined score for each candidate API call is computed according to the Formulas 3 and 4. The method call add is ranked highest when other factors such as distance, scope, and dependency are considered as well.

6. EMPIRICAL METHODOLOGY

To evaluate APIREC, we answer the following research questions:

RQ1: *Accuracy* How accurate is APIREC when recommending API calls? How does its accuracy compare to the state-of-the-art ap-

proaches Bruch *et al.* [7] (set-based approach on code only, available as an advanced feature [12] in Eclipse), *n*-gram [42] (sequence-based statistical approach with program dependencies), and GraLan [35] (statistical learning without modeling changes)?

RQ2: ***Sensitivity Analysis*** How do factors such as the size of training data, the requested location, the sizes of the change context and code context impact accuracy?

RQ3: ***Running Time*** What is the running time of APIREC?

6.1 Corpora

We compiled two disjunct corpora to train and test APIREC.

Large Corpus. This corpus consists of 50 randomly selected Java projects from Github that have long development histories (+1,000 commits each). Table 3 shows the number of commits contained in this corpus. Based on previous research [6], in order to avoid large commit size, we do not select repositories which were migrated to GitHub from a centralized version control system. We extract the atomic changes from all the commits in the corpus. To do this, we iterate over all the files in all the commits. We then use GumTree [13] to compute the *atomic changes* (see definition in Section 3.2) between the before and after versions of each file.

Community Corpus. This smaller corpus contains eight projects from GitHub that have been used by previous researchers [16, 35]. The third column in Table 3 lists the statistics about this corpus. We extract atomic changes from this corpus in the same manner.

6.2 Evaluation Setup

We aim to investigate the foundation of our hypothesis, the repetitiveness of changes. We posit that changes performed on different projects and by different programmers have different degrees of repetitiveness. Thus, to evaluate the impact that the project's culture and individual developer's habits play, we designed three scenarios:

Community Edition. We trained APIREC with the Large Corpus and then we tested it against the Community Corpus.

Project Edition. For each project in the Community Corpus, we trained APIREC on the first 90% of commits, and then we tested on the remaining 10% of commits (10-fold validation).

User Edition. This is similar to the Project Edition scenario, but we only used the commits from one user from each project. We selected the user who authored the most commits in each project.

6.3 Procedure, Metrics, and Settings

To measure APIREC's accuracy, we reenact real-world code evolution scenarios. We use the real API calls from the corpora as the "oracle" for determining the correct recommendation.

For each transaction we choose an atomic change to be used as *the prediction point*. The prediction point represents the change that APIREC will try to predict. This mimics real development where a developer has typed part of the changes in a commit, and at some point, they invoke APIREC. We used the following procedure to choose the prediction point. First, we order the atomic changes in the transaction according to their locations in the file. Let n be the transaction's size, i.e. it contains n changes. We assume the change is at different positions $l = 1...n$ to study the impact of the prediction point on accuracy. If the change at position l is the addition of an API call m, we will use m as a prediction point. Otherwise, we will check the atomic change at $l+1$ and so on until we encounter a method addition. If no such change is found, we skip that transaction.

We use the atomic changes in the current transaction preceding the prediction point m as the change context. We collect all the code tokens prior to m in the current file as the code context. The changes and code tokens that are outside of the method containing m are assigned lower weights (Section 4). We invoke APIREC with this context to recommend the candidate list L of API calls. If the method m (the API call from the oracle) is in the top k positions of the list L of API calls, we count it as a hit for top-k accuracy. Otherwise, it is a miss. The top-k accuracy for a project is calculated as the ratio between the number of hits over the total number of hits and misses.

Table 3: Collected Data on Fine-grained Code Changes

	Large Corpus	Community Corpus
Projects	50	8
Total source files	48,699	8,561
Total SLOCs	7,642,422	1,331,240
Number of commits	113,103	18,233
Total changed files	471,730	63,962
Total AST nodes of changed files	714,622,846	86,297,938
Total changed AST nodes	43,538,386	4,487,479
Total detected changes	1,915,535	252,522
Total detected changes with JDK APIs	788,741	117,481

Model Parameters. APIREC has two parameters: the weights w_C and w_T for the change and code contexts. However, we use adaptive learning [50] to learn the optimal weights $w_{\mathscr{C}}$ and $w_{\mathscr{T}}$ from a training data (Section 5.1). The other parameters are $w_{scope_{ci}}$, $w_{scope_{ti}}$, $w_{dep_{ci}}$, and $w_{dep_{ti}}$, which are the weights for the scopes and dependencies of the elements in the two contexts in relation to the prediction point. Since we focus on recommending the API call in the current method, we use two levels of weights for the contexts inside and outside of the containing method, and two levels of weights for having or not having data dependencies (Section 4).

7. EMPIRICAL RESULTS

7.1 Accuracy: Community Edition

In this experiment, we evaluate APIREC's recommendation accuracy when it is trained on the Large Corpus and tested on the Community Corpus. We compare APIREC with the state-of-the-art API completion approach by Raychev *et al.* [42]. We implemented their *n*-gram API recommendation model according to the description in their paper. We also compare APIREC with Bruch *et al.* [7] (which uses association among APIs in a *set* for recommendation), and with GraLan [35], a graph-generative model. We trained all those *n*-gram-based, set-based, and graph-based models with the source code of the entire last snapshots of the projects in the Large Corpus.

We compared the approaches in two settings: 1) on all APIs in all libraries in the corpora, and 2) on the APIs of the JDK library.

7.1.1 Accuracy Comparison for General API calls

APIREC is a statistical, data-driven approach, which could not predict an API call that it has not seen in the training data. This is often referred to as *out-of-vocabulary* (OOV). An API call is considered to be OOV if it is neither declared nor used in the training data, but is in the testing data. This can occur when APIREC is trained on the Large Corpus, but is tested on the Community Corpus.

For a fair comparison, we measured in-vocabulary accuracy (IN) of APIREC and three above approaches (Figure 2). To compute in-vocabulary accuracy, we followed the same procedure as explained earlier except when we search for the prediction point we only stop when we find an API call m' that *previously* appeared in the training data (in our vocabulary). If we cannot find m' in our training data, we skip the current transaction and continue. Thus, APIREC only tried to predict methods which it had previously seen.

In this experiment, we used APIREC to predict the middle point of the change transaction $l = \lfloor n/2 \rfloor + 1$. The total number of recommendations for a project appears in parentheses after the project name.

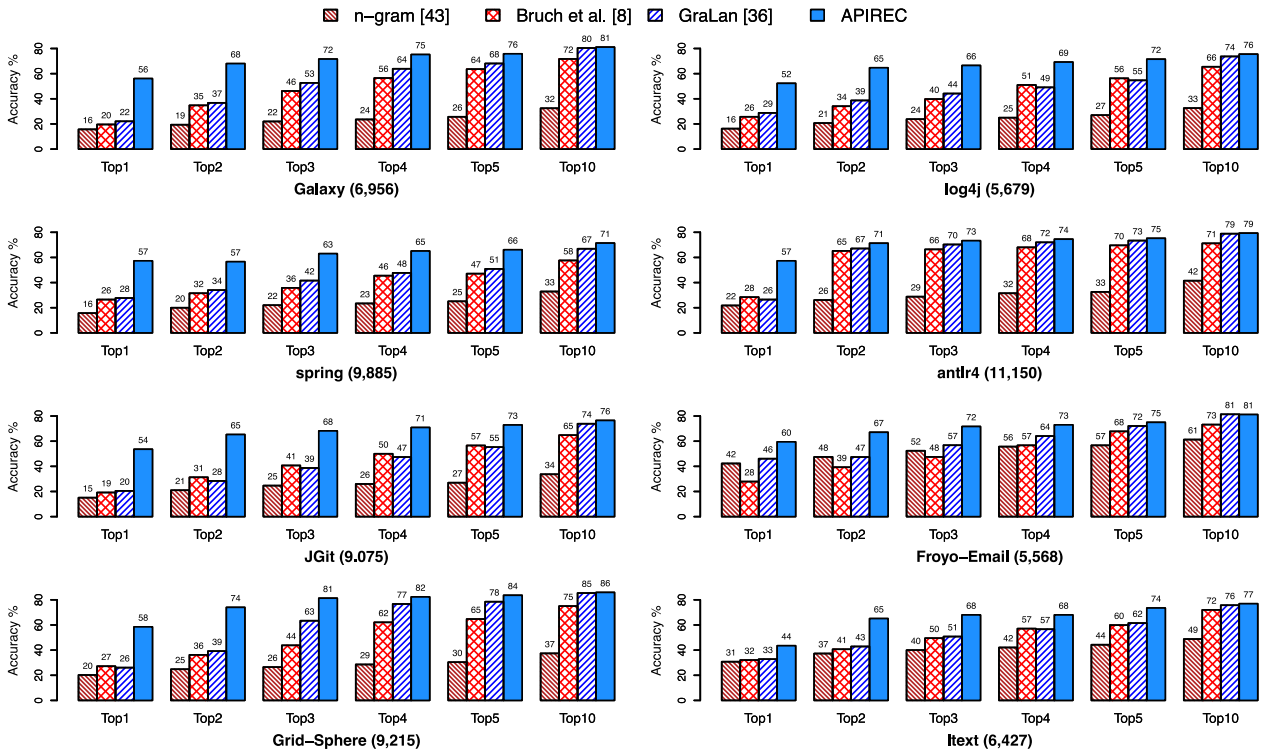

Figure 2: API Recommendation Accuracy for Community Edition (%) per Project (the parentheses are the number of recommendations)

APIREC *outperforms others across the board. At Top-1, it is better from 30–160%, and at Top-5 from 20–30%.*

APIREC *achieves high accuracy. Top-1 is correct in 44.2–59.5% of the cases (IN). Top-5 accuracy is as high as 83.6% (IN).*

We investigated the differences in accuracy between the approaches. We found that *n*-gram model requires strict, unnecessary ordering between API calls. For example, in HashMap, clone→put→remove and clone→remove→put are not considered as the same pattern by *n*-gram. Thus, clone→put cannot be used to recommend remove. In contrast, a clone was changed/added together with an addition of either a put or a remove. APIREC uses the context with the modification/addition of clone to recommend either put or remove. That is, the atomic changes for the same high-level intent often co-occur in the same transaction (e.g., the tasks "clone a hash map and add to it" or "clone a hash map and remove from it").

Bruch *et al.* [7] does not consider the order of API calls for recommendation. This could lead to more noise due to its use of different subsets of API calls for recommendation. In contrast, GraLan uses partial orders among API calls, leading to better performance. For example, clone→put→remove and clone→remove→put belong to a pattern that clone appears before others, but there is no strict order between put and remove. APIREC has higher top-*k* accuracy than GraLan for small *k*s. There are two reasons: 1) APIREC relies on the repeated changes that a user has with high-level intents (e.g. adding a loop collector) that connect the fine-grained changes logically; and 2) APIREC's consideration of the impacts of the distance between context changes and the recommendation location (nearby changes or code tokens are better for recommendation). Their accuracies are comparable for larger *k*s as more candidates are considered.

7.1.2 Accuracy Comparison for JDK

In this section, we evaluate APIREC's accuracy in recommending a specific API library (in contrast to general API calls). We chose JDK for this experiment since it is frequently used in Java programs. We followed a similar evaluation procedure as before. However, while selecting the prediction point, we search for a method *m* that belongs to the JDK library after the location *l*. That is, the actual change must be the addition of a method call from JDK. We skip the transaction if we do not find such a method.

Table 4 shows the accuracy comparison. The accuracy for recommending API calls in JDK is high. *In 56.4–74.3% of the cases,* APIREC *can correctly recommend the JDK method call as the top recommendation. In 76.1-89.8% of the recommendation cases, the actual JDK method call is in the top five candidates.*

The accuracy in all the projects is higher than those for recommending general API calls (Figure 2). This is expected since JDK is popular. With the 50 projects for training, all the JDK API calls are in the vocabulary (IN). At Top-1, APIREC improves over *n*-gram by 200%, over Bruch *et al.*'s by 170%, and over GraLan by 120%.

7.2 Accuracy: Project Edition

In this experiment, we aimed to evaluate the impact of project's culture on APIREC's accuracy. We trained/tested it on the fine-grained changes of the same project. For comparison, we used the testing projects in the community edition. However, for each project, we sorted all the commits in the chronological order. We then used the oldest 90% of the project commits for training APIREC and the 10% most recent commits for recommendation.

Tables 5 and 6 show accuracy results for the project edition setting for recommending general and JDK API calls. The changes in individual projects do not repeat as much as those across projects. Thus, the accuracy is generally lower than those in the community edition (comparing Table 5 and Figure 2, Tables 6 and 4).

Since JDK is a popular Java library, the code changes involving JDK APIs still occur and repeat more frequently than the project-specific method calls. Thus, the accuracy in recommending JDK APIs is higher than the accuracy in recommending general API calls (comparing Tables 5 and 6). Note that, in this experiment, with our training projects, all JDK API calls are in the vocabulary. Thus, Table 6 only contains the result for APIREC.in.

Table 4: JDK Recommendation Accuracy for Community Edition (%) (No OOV APIs in JDK library in this experiment)

System	Model	Top1	Top2	Top3	Top4	Top5	Top10
Galaxy	APIREC	66.8	78.0	80.9	82.1	82.6	85.6
(6,956)	*n*-gram	17.6	23.5	26.1	29.1	31.9	39.3
	Bruch *et al.*	19.7	32.9	41.9	47.0	52.1	58.7
	GraLan	25.1	42.1	59.0	68.8	74.0	84.9
log4j	APIREC	57.3	70.7	72.6	75.0	76.1	78.4
(5,679)	*n*-gram	15.5	21.2	25.4	29.2	31.7	37.8
	Bruch *et al.*	22.7	36.9	45.4	52.5	57.1	64.6
	GraLan	29.5	41.7	48.3	53.3	57.8	76.0
spring	APIREC	61.1	70.6	76.1	77.4	79.0	80.6
(9,885)	*n*-gram	19.1	24.8	27.5	29.8	32.3	40.0
	Bruch *et al.*	24.6	38.2	47.7	55.1	57.8	70.9
	GraLan	29.4	42.1	50.2	56.7	60.8	75.9
antlr4	APIREC	67.9	81.9	83.6	84.0	84.5	85.5
(11,150)	*n*-gram	22.8	30.9	34.7	37.6	39.5	46.9
	Bruch *et al.*	24.7	50.6	65.3	72.2	74.6	78.6
	GraLan	30.3	76.1	80.5	81.7	82.7	87.2
JGit	APIREC	73.4	82.6	85.1	85.9	86.0	87.7
(9,075)	*n*-gram	17.6	25.0	28.3	31.0	33.0	40.8
	Bruch *et al.*	23.1	37.5	48.8	54.9	59.1	66.3
	GraLan	27.5	35.9	48.4	57.3	65.2	85.6
Froyo-E	APIREC	74.3	81.7	84.5	85.6	86.0	89.8
(5,568)	*n*-gram	39.9	46.7	48.5	51.6	54.9	60.4
	Bruch *et al.*	26.1	43.3	53.8	63.8	67.8	80.6
	GraLan	32.4	45.5	55.1	63.5	75.8	90.4
Grid-S	APIREC	61.6	80.2	88.0	88.8	89.8	91.1
(9,215)	*n*-gram	22.3	27.7	29.9	32.2	34.2	42.5
	Bruch *et al.*	25.3	41.6	54.3	62.5	64.5	67.6
	GraLan	27.6	42.4	68.9	82.9	83.9	90.8
Itext	APIREC	56.4	76.5	77.5	79.5	80.1	81.5
(6,427)	*n*-gram	31.4	37.5	40.8	44.0	47.9	55.6
	Bruch *et al.*	25.7	44.3	53.5	60.0	63.9	78.7
	GraLan	32.9	44.7	53.4	61.5	66.5	80.1

Table 5: API Recommendation Accuracy - Project Edition (%)

System	Model	Top1	Top2	Top3	Top4	Top5	Top10
Galaxy	APIREC.oov	11.1	20.3	23.9	30.3	35.4	51.9
(6,956)	APIREC.in	11.2	21.9	25.7	30.0	36.0	48.8
log4j	APIREC.oov	15.4	26.0	32.6	35.2	37.5	40.9
(5,679)	APIREC.in	18.1	29.7	36.2	39.1	42.1	45.8
spring	APIREC.oov	18.5	23.5	26.1	26.8	28.0	31.5
(9,885)	APIREC.in	21.7	29.7	32.9	33.4	34.4	40.2
antlr4	APIREC.oov	11.1	11.3	12.7	20.7	21.2	21.2
(11,150)	APIREC.in	11.1	11.1	12.7	20.7	21.2	21.2
JGit	APIREC.oov	7.0	18.1	21.2	23.3	26.8	41.4
(9,075)	APIREC.in	7.2	20.6	22.5	24.0	27.1	42.3
Froyo-	APIREC.oov	12.7	19.2	22.8	27.8	31.2	38.6
Email	APIREC.in	13.8	20.7	25.7	28.9	31.9	39.4
Grid-	APIREC.oov	33.1	42.9	48.3	49.2	49.8	55.0
Sphere	APIREC.in	41.4	45.8	51.0	51.6	51.9	58.7
Itext	APIREC.oov	9.6	18.2	18.8	21.6	22.9	25.6
(6,427)	APIREC.in	14.0	23.7	24.2	27.4	30.2	33.7

Table 5 shows that the values for APIREC.in are slightly higher than those for APIREC.oov. This is reasonable since most used method calls existed. Thus, if we use a change history in a project covering as many project-specific methods as possible, APIREC can be used to recommend the calls to those methods. The same trends apply for top-*k* accuracy numbers for other models (not shown in Table 5). APIREC outperforms the other models across the board.

7.3 Accuracy: User Edition

In this experiment, we evaluate how APIREC performs when being trained only on the commits from a single user. From each project in the test corpus, we selected the user who has the most commits. A user makes up a significant part of each project, ranging from 10% to 28% of all commits in a single project. To compare results, we used the same projects as the Project Edition. We used the same sorting technique so that 90% of the commits were used for training, and the most recent 10% were used for recommendation.

Table 6: JDK Recommendation Accuracy for the Project Edition (%) (No OOV APIs in JDK library in this experiment)

System	Model	Top1	Top2	Top3	Top4	Top5	Top10
Galaxy	APIREC	31.4	38.7	42.2	45.7	49.9	63.7
log4j	APIREC	24.2	34.4	38.5	41.1	42.9	51.5
spring	APIREC	34.3	47.0	52.6	53.7	55.0	59.5
antlr4	APIREC	24.8	24.8	24.8	33.4	33.6	33.6
JGit	APIREC	16.5	20.6	28.7	30.3	32.3	57.6
Froyo-E	APIREC	22.6	33.6	46.8	53.4	55.2	61.9
Grid-S	APIREC	60.0	68.6	75.6	76.7	77.0	80.4
Itext	APIREC	33.7	48.5	50.7	51.7	53.9	62.6

Table 7: API Recommendation Accuracy for User Edition (%)

System	User	Top1	Top2	Top3	Top4	Top5	Top10
Galaxy (6956)	dandiep (864)	27.0	46.5	49.2	53.0	55.8	60.7
log4j (5,679)	ceki (967)	27.6	37.4	42.2	43.1	47.5	59.4
spring (9,885)	jhoeller (2,542)	9.5	14.8	17.5	18.7	27.3	30.0
antlr4 (11,150)	parrt (1,412)	57.0	65.8	68.3	68.9	69.9	76.2
JGit (9,075)	spearce (938)	26.1	45.3	70.1	70.3	76.3	81.3
Froyo-E (5,568)	mblank (946)	23.3	32.7	38.7	40.0	49.3	58.7
Grid-S (9,215)	novotny (2,613)	34.4	40.3	43.6	44.6	44.8	45.2

Table 7 shows the result for the user with most commits from each project. When compared with Figure 2 and Table 4, accuracy in the User Edition is lower than that in Community Edition. We expect this result because more training should yield better results.

Interestingly, the User Edition's accuracy generally is higher than that of Project Edition. For the data we randomly selected, each user commits to only one project. This leads us to infer that there is a subset of the training data that is more important. *Considering code authorship, we could train with less data, yet have more precise results than when training with the entire project.*

7.4 Sensitivity Analysis: Out-Of-Vocabulary Data

As with all statistical learning methods, APIREC's results are affected by the sufficiency of the training data. Thus, we evaluated to what extent OOV impacts APIREC's accuracy. We chose a random project, Froyo-Email, from the Community Corpus. We conducted two executions with two experimental procedures to compare accuracy when OOV occurs and does not occur. In the first execution, we followed the same procedure in Section 6.3 to measure top-*k* accuracy. For this study, we use the prediction point at the middle of a transaction, i.e., $l = \lfloor n/2 \rfloor + 1$. When an API call is OOV, we counted it as a miss. In the second execution, we followed the same procedure to measure in-vocabulary accuracy as described in Section 7.1.1. To be able to compare against the two runs, APIREC made exactly 5,568 recommendations in each execution.

Table 8 shows the top-*k* accuracy for the two cases. Even with the OOV issue, APIREC is able to achieve high accuracy. With a single recommendation, it is able to correctly recommend the API call in almost 45% of the cases. In 56.8% of the cases, the actual call is in the list of only five candidates. APIREC's accuracy is even higher if trained with enough API calls (the IN case). Here, it is able to correctly recommend the call with a single recommended candidate in almost 60% of the cases. In 75% of the cases, it correctly recommends the call with only five candidate APIs.

Table 8: Impact of OOV on Recommendation Accuracy (%)

	Top-1	Top-2	Top-3	Top-4	Top-5	Top-10
OOV	**44.8**	51.6	54.7	55.9	**56.8**	62.5
IN	**59.5**	67.1	71.7	73.0	**75.0**	81.2

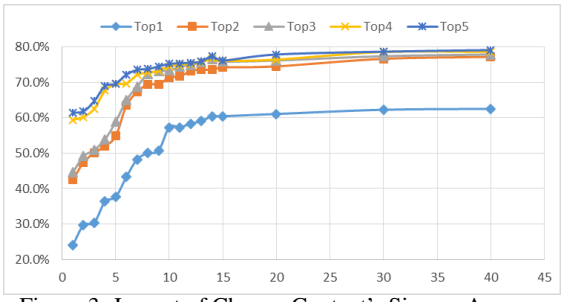

Figure 3: Impact of Change Context's Size on Accuracy

When APIREC uses in-vocabulary elements, it makes better recommendations ranging from 14.5% to 18.7% improvements in accuracy. We measured the *OOV rate*, defined as the percentage of predicted API call in the Community Corpus that are not contained in the Large Corpus. The OOV rate for Froyo-Email project is 28.1%, a lower bound of the loss in accuracy. This result shows that OOV has impact on accuracy. This is expected because as a learning model, APIREC needs to observe the API calls to recommend them.

7.5 Sensitivity Analysis: Change Context and Code Context

APIREC relies on both change and code contexts in recommendation. Thus, we also conducted an experiment to measure the impact of the size of the change context (number of changes) and that of the code context (number of code tokens) preceding the prediction point. We randomly chose a project, antlr, in Community Corpus. We varied the contexts' sizes and measured in-vocabulary accuracy.

Figures 3 and 4 show accuracy with various sizes of either contexts. As seen, when increasing either context's size, accuracy improves. However, the impact on accuracy for the size of the change context is higher when considering small sizes. For example, increasing the change context's size from 1 to 10, top-1 accuracy improves 34%, while increasing the code context's size from 1 to 10, accuracy improves only 3-5%. Thus, adding more changes to the context helps connect better the changes in a high-level intent, leading to higher accuracy. The increase is smaller when adding more code tokens to the code context. When contexts' sizes are greater than 15 (15 prior changes or tokens), accuracy improves only slightly since the information needed to correctly recommend was sufficient. Thus, we chose 10 as the default sizes for both contexts. This result also shows that APIREC *maintains reasonably high accuracy with small sizes of contexts of prior changes and code tokens*.

7.6 Sensitivity Analysis: Prediction Locations

Because APIREC is also based on the change context, i.e., prior changes, selecting a prediction point among n changes in a transaction might have impact on accuracy. Thus, we conducted another experiment to measure that. We first chose a random project, JGit, from the Community Corpus after training APIREC on the Large Corpus. We chose a prediction point at three locations among n changes in a transaction: the first quartile point $l_1 = \lfloor n/4 \rfloor + 1$, the middle point $l_2 = \lfloor n/2 \rfloor + 1$, and the third quartile point $l_3 = \lfloor 3 * n/4 \rfloor + 1$. We followed the same procedure to measure top-k accuracy. We used the in-vocabulary setting.

Table 9 shows the accuracy with different prediction locations. As seen, accuracy slightly increases if we move the point to a later

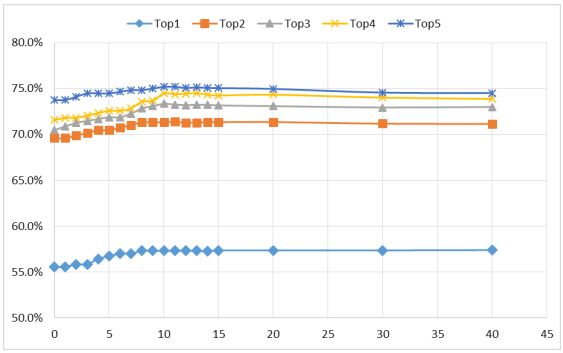

Figure 4: Impact of Code Context's Size on Accuracy

Table 9: Impact of Recommendation Points on Accuracy (%)

Location	Top-1	Top-2	Top-3	Top-4	Top-5
1st quartile	64.18	75.10	77.69	80.20	82.17
middle point	64.24	75.27	77.70	80.26	82.20
3rd quartile	67.00	75.45	77.96	80.33	82.36

part of a transaction from 1^{st} to 3^{rd} quartile point. This is expected because APIREC collects more prior changes in the change context.

7.7 Sensitivity Analysis: Training Data's Size

We want to analyze the impact of the size of the training dataset of fine-grained changes on accuracy with the test project, antlr, in the Community Corpus. We built 6 training datasets by increasing their sizes with additional projects in GitHub from 50 to 300 projects. We ran APIREC for each dataset (Table 10). As seen, top-1 accuracy increases from 57.3 to 58.6% with more training data. As expected, larger training data sets perform better, however the improvements are small. This shows that a minimum training set of 50 projects produces good results. Results get stable when we use +300 projects.

7.8 Running Time

All experiments were run on a computer with Xeon E5-2620 2.1GHz (configured with 1 thread and 32GB RAM). The time complexity is reported in Table 11. The training time is significant, but training can be done off-line for our model. The recommendation time is short (<1 second per recommendation), thus APIREC is suitable to be interactively used in an IDE.

7.9 Threats to Validity

Our corpus in Java only might not be representative. For different projects with different OOV rates, the results vary. For comparison, we ran all approaches on the same dataset and measured IN accuracy. We also evaluated the impact of OOV data (Section 7.4). We do not have the tool in Raychev *et al.* [42] (it is not publicly available), but we carefully followed the approach described in the paper.

The simulated procedure in our evaluation is not true editing. The choice of the prediction point at the middle of a change transaction could affect the accuracy as seen in Section 7.6. However, it is representative as it achieves neither the best nor the worst accuracy. The study on usefulness needs to involve human subjects, and will be part of our future work. The results for JDK might be different for other libraries. But APIREC is general for any library and language.

8. DISCUSSION

8.1 Limitations

Our approach also has shortcomings. First, out-of-vocabulary is an issue. However, as seen, even with only 50 projects in the pres-

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

Number of projects	Top-1	Top-2	Top-3	Top-4	Top-5
DataSet-1 (50)	57.3	71.3	73.3	74.5	75.2
DataSet-2 (100)	57.7	71.4	73.7	74.5	75.3
DataSet-3 (150)	58.3	71.9	74.5	75.6	76.2
DataSet-4 (200)	58.5	72.7	75.0	75.8	76.6
DataSet-5 (250)	58.6	72.9	75.2	75.9	76.7
DataSet-6 (300)	58.6	73.0	75.3	75.9	76.8

Table 11: Performance

		Cross Project	Within Project
Storage		680 Mbytes	10 - 30 Mbytes
Training time		15 hrs (50 projs)	10 - 30 min
Recommendation time	Avg	0.6 sec/change	0.2 sec/change
	Max	2s/change	1 sec/change

ence of OOV, APIREC's accuracy is high. Moreover, if trained with sufficient data, APIREC performs better than existing approaches (in-vocabulary accuracy). Second, using the file level as the scope for transactions may also lead to a loss of accuracy. However, the majority of change patterns appear in the same file [34]. In some cases files may contain tangled changes: changes from multiple tasks that are committed together [14]. This can introduce noise while learning, as spurious changes would be associated with a pattern. We also miss changes that span across files but are part of the same pattern. Finally, we handle code context by finding the associations between code tokens in transactions. A potential alternative is the combination between APIREC and a language model for code (e.g., GraLan [35], cache model [52] or RNN LM [53]).

8.2 Implications

We group implications by three categories of audiences: developers, tool builders, and researchers.

Developers. APIREC is data-driven, so choosing the right training corpus is important. Our results suggest that training on the Community Corpus leads to higher accuracy than when training on a project. *When training and testing on changes of one user, the accuracy was between that of the Community Corpus and each project*. Even though the Project Edition results are not directly comparable to the other two (because the user commits are fewer than the projects), the results suggest that with a user-specific change history, the user-trained model achieves better accuracy. The implication is that *users should obtain a community trained model, and then further refine it with their own changes*.

Tool Builders. Section 7.8 shows that a challenge when using statistical learning recommenders is the long training time for the model. The problem can be mitigated in different ways. First, the community models can be trained by the tool vendors and offered with the tool. Second, the user's continuous integration server can incrementally augment the community model each night with the changes that the user committed during the day.

Researchers. Our paper, together with recent related work [1,34], shows that fine grained code changes are highly repetitive. This opens up new research topics in mining fine-grained changes. First, researchers can mine changes to actively help developers during development: code completion, dynamically learned refactorings, record/replay of bug fixes, etc. Second, they can mine changes to learn and develop theories on the nature of software change, such as building catalogs of software change building blocks, incorporating change patterns into atomic changes via language and IDE design, predicting the changes required for a task, etc.

9. RELATED WORK

Code and API completion based on statistical language models. Hindle *et al.* [16] use the *n*-gram model [26] on code tokens to show that source code has high repetitiveness, and then use it to recommend the next token. Tu *et al.* [52] enhance *n*-grams with caching of recently seen tokens. Raychev *et al.* [42] and GraLan were explained earlier. White *et al.* [53] applied RNN LM on code tokens to achieve higher accuracy than *n*-gram. Mou *et al.* [32]'s tree-based convolutional neural network is applied to source code to recommend syntactic units. Allamanis *et al.* [4] use bimodal modeling for short texts and code snippets. Maddison and Tarlow [24] use probabilistic context free grammars and neural-probabilistic language models for source code. NATURALIZE [1] learns coding conventions to recommend identifier names and formatting conventions.

In comparison to those statistical approaches, APIREC has key differences. First, they are based on the principle of code repetitiveness, while APIREC relies on the *repetitiveness of fine-grained code changes*. Second, APIREC is tailored toward method invocation recommendation including API calls. Other models, except GraLan and Raychev *et al.* [42], are either for general tokens [16, 37, 52] or for special-purpose recommendations (e.g. AST structures [24], coding conventions [1], or methods/classes' names [2]).

Code and API completion based on mined patterns. Bruch *et al.* [7] propose best-matching neighbor algorithm for code completion that uses as features the *set* of API calls of the current variable *v* and the names of the methods using *v*. The set features in the current code is matched against those in the codebase. Grapacc [36] mines patterns as graphs and matches them against the current code. Robbes and Lanza [44] improve code completion using recent modified/inserted code during an editing session. We train our model on change histories, rather than the editing changes in a session.

There exist other deterministic approaches to improve *code completion* and *code search* by using editing history [18, 22, 44], cloned code [15], API examples, and documentation [8, 27, 28, 30, 31, 49], structural context [17, 29], parameter filling [5, 54], interactive code generation [39], specifications [43, 48], documentation [28], type information [47], Feature Requests [51]. APIExplorer [9] leverages structural relations between APIs to facilitate their discoverability.

10. CONCLUSION

This work is the first that leverages *the regularity of fine-grained code changes in the context of API code-completion*. Whereas the previous approaches used the regularity of idioms of code tokens, in this paper, we address the problem of API method recommendation by a statistical learning model that we train on fine-grained code changes. When we mine these in a large corpus, the changes belonging to higher-level intents will appear more frequently than project-specific changes. Our thorough empirical evaluation shows that APIREC improves over the state-of-the-art tools between 30–160% for top-1 recommendations. It performs well even with a one-time, minimal training dataset of 50 publicly available projects.

We found that training the model with the changes from individuals achieves higher accuracy than training with changes in their entire projects. Thus, the recommender could be trained from a large corpus of community projects, and an individual user could further refine the model with their own changes.

11. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their constructive feedbacks. This work was supported in part by the US NSF grants CCF-1553741, CCF-1518897, CNS-1513263, CCF-1413927, CCF-1439957, CCF-1320578, and TWC-1223828.

12. REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 281–293. ACM, 2014.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'15, pages 38–49. ACM, 2015.

[3] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories*, MSR'13, pages 207–216. IEEE CS, 2013.

[4] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning*, ICML'15, pages 2123–2132. ACM, 2015.

[5] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider. Exploring API method parameter recommendations. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME'15, pages 271–280. IEEE CS, 2015.

[6] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 322–333. ACM, 2014.

[7] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222. ACM, 2009.

[8] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792. IEEE CS, 2012.

[9] E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 79–104. Springer-Verlag, 2011.

[10] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 266–276. IEEE Press, 2012.

[11] Eclipse. www.eclipse.org.

[12] Eclipse code recommenders. http://www.eclipse.org/recommenders/.

[13] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324. ACM, 2014.

[14] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 392–401. IEEE Press, 2013.

[15] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 228–235. IEEE CS, 2004.

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

[17] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE'05, pages 117–125. ACM, 2005.

[18] D. Hou and D. M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 233–242. IEEE CS, 2011.

[19] Informer. http://javascript.software.informer.com/download-javascript-code-completion-tool-for-eclipse-plugin/.

[20] Intellisense. https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx.

[21] E. T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630, May 1957.

[22] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 1–11. ACM, 2006.

[23] R. Lau, R. Rosenfeld, and S. Roukos. Trigger-based language models: a maximum entropy approach. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'93, pages 45–48, 1993.

[24] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on Machine Learning*, ICML'14, pages 649–657, 2014.

[25] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[26] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT Press, Cambridge, MA, USA, 1999.

[27] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[28] C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending Source Code Examples via API Call Usages and Documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE'10, pages 21–25. ACM, 2010.

[29] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 502–511. IEEE Press, 2013.

[30] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 997–1016. ACM, 2012.

[31] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *Proceedings of the 28th International Conference on Automated Software Engineering*, ASE'13, pages 646–651. IEEE CS, 2013.

[32] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR*, abs/1409.5718, 2014.

[33] G. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[34] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 803–813. ACM, 2014.

[35] A. T. Nguyen and T. N. Nguyen. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE'15, pages 858–868. IEEE Press, 2015.

[36] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 69–79. IEEE Press, 2012.

[37] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'13, pages 532–542. ACM Press, 2013.

[38] S. Okur and D. Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE'12, pages 54:1–54:11. ACM, 2012.

[39] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 859–869. IEEE, 2012.

[40] M. Piccioni, C. A. Furia, and B. Meyer. An empirical study of API usability. In *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'13, pages 5–14. IEEE, 2013.

[41] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The uniqueness of changes: characteristics and applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR'15, pages 34–44. IEEE Press, 2015.

[42] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 419–428. ACM, 2014.

[43] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253. IEEE CS, 2009.

[44] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the International Conference on Automated Software Engineering*, ASE'08, pages 317–326. IEEE CS, 2008.

[45] M. P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[46] R. Rosenfeld. A maximum entropy approach to adaptive statistical language modelling. *Computer Speech and Language*, 10(3):187–228, July 1996.

[47] A. A. Sawant and A. Bacchelli. A dataset for API usage. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*, MSR'15, pages 506–509. IEEE, 2015.

[48] K. T. Stolee and S. Elbaum. Toward semantic search via SMT solver. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE'12, pages 25:1–25:4. ACM, 2012.

[49] 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.

[50] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 253–262. IEEE CS, 2011.

[51] F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic recommendation of API methods from feature requests. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 290–300. IEEE, 2013.

[52] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22nd Symposium on Foundations of Software Engineering*, FSE'14, pages 269–280. ACM Press, 2014.

[53] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories*, MSR'15, pages 334–345. IEEE CS, 2015.

[54] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 826–836. IEEE Press, 2012.