Using Change Context with Statistical Learning for API Code Recommendation

Anh Nguyen¹, Michael Hilton², Mihai Codoban ³, Hoan Nguyen ¹, Lily Mast⁴, Eli Rademaker⁵, Tien Nguyen¹, Danny Dig⁶
¹Department of Electrical and Computer Engineering, Iowa State University, USA
²School of EECS, Oregon State University, USA
³Microsoft, USA
⁴College of Engineering and Computer Science, University of Evansville, USA

ABSTRACT

Learning and remembering how to use APIs is hard. While code-completion tools list all the API methods available on a given object, reading through a long list of API method names and their associated documentation is tedious, and users can be easily overloaded with too many suggestions. While several researchers proposed techniques for recommending APIs, their accuracy is low.

We present a novel API recommendation approach that taps into the predictive power of repetitive code changes. 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. We trained APIRec on 43M changes from 100K commits in open-source projects. Our empirical evaluation shows that APIRec correctly recommends an API method in the first position 59% of the time, and it recommends the correct API method in the top 5 positions 75% of the time. This is a significant improvement over the best in class state of the art recommender by a factor of 2.4x for the first position, and 2.2x for the top 5 positions, respectively.

1. INTRODUCTION

Today’s programs use Application Programming Interfaces (APIs) extensively: even the “Hello World” program invokes an API method. One of the main challenges in software development is learning and remembering how to use APIs. Even developers who are familiar with some APIs are forced to re-learn them due to continuous API evolution.

The state-of-the-practice support for working with APIs comes in the form of code-completion tools integrated in the IDEs. 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. Still, a developer learning an API (or trying to remember it) can be wasting 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 from JDK 8 populates a list of 67 possible methods (and 10 additional methods inherited from superclasses). Carefully examining a long list of API method names and reading the associated Javadoc documentation is tedious.

The state-of-the-art research in code completion takes advantage of API usage patterns, which they deterministically mine from software repositories. These mining approaches analyze the context in which the recommender was invoked. If this context matches a previously identified pattern, then the recommender will suggest the rest of the API element(s) in the pattern. Other researchers employ statistical learning where they use language models to recommend the next code token, including API elements. The principle behind this line of work is that a model can be trained from a large code corpus to statistically learn common code patterns (often called idioms) with high regularity. It can then predict what token is likely to follow a sequence of given code elements in the current code. However, accuracy of previous approaches is low: even the best in class has an average accuracy of 20%.

In this paper, we present a novel approach to code completion centered around fine-grained code changes. The intuition behind our approach is that source code changes are repetitive and have a high degree of regularity; thus, we can harvest them to significantly improve the accuracy of recommendations. For example, when changing the code to instantiate a fresh List object, a developer often changes the code to add a new element into the fresh list via List.add. However, the developer is unlikely to call List.remove just after instantiating a fresh List. Without being aware of the context of the change, a code completion tool would not be able to differentiate between recommending add or remove. In our approach, we mine the fine-grained code changes as well as the context in which the recommender was invoked to build a statistical model for recommending the next API method call. Our approach can be thought as a language model for fine-grained source code changes.

We implemented our approach in a tool, APIRec, that computes the most likely API method call to be inserted at the requested location in a given part of the code. APIRec works in three stages: (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.

To accurately recommend API calls, we trained our statistical model on a large data set. We collected more than 43,538,386 fine-grained code changes coming from 113,103 change commits in 50 open-source projects from GitHub. APIRec iterates over each commit in every project, and detects the differences in the Abstract Syntax Trees (AST) using the state-of-the-art GumTree tool. These fine-grained changes provide a detailed representation of each change, from which APIRec can learn much more than from a simple textual diff.

We developed an association-based inference model that learns from the corpus the changes that frequently co-occur in the same changed file within a commit. APIRec also learns the surrounding context (e.g., for loops, preceding method calls) of the code changes. In addition, APIRec learns the appropriate weights to assign to the surrounding context and the previous changes.
work in mining fine-grained code changes. This is a common

we do a 10-fold cross-validation on each of the 8 projects used

validation on the same 8 projects as above, but only on the commits

code changes include the changes at line 1 and line 4 in Figure 1b.

Adding a Loop Collector

2. MOTIVATING EXAMPLE

2.1 Adding a Loop Collector Change Pattern

Let us start with an example to motivate our approach. Figure[1]
shows a real-world example that we have collected from our prior
work [18] in mining fine-grained code changes. This is a common
change pattern, called Adding a Loop Collector. In this change, a
developer introduces a new variable that collects or aggregates the
returned values processed in a loop. In this example, the fine-grained
code changes include the changes at line 1 and line 4 in Figure[1].

More 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<>(). The changes at line 4 include the additions of two method calls in which
one of them is the argument of the other.

Assume that the current editing location is at line 4 of Figure[1],
after the developer has typed the changes at line 1 and the variable
results itself. (S)he then requests the code recommendation tool.
In modern IDEs, the list of fields and methods of HashSet will be
presented to the developer in a pre-defined order. (S)he must browse
through a list of 37 methods to find the desired method. Advanced
code completion engines will recommend a list of API calls based
on the API usage patterns that are deterministically/explicitly or
statically/implicitly mined from the code corpus. Both of these
strategies are based on the fact that source code is repetitive [18].
Code recommendation techniques which rely on the context of
the change would identify this context as the sequence of code
tokens preceding the variable results, i.e., the sequence ‘t.execute();’;
However, this code sequence is in fact very specific to this project,
and is not part of any code pattern related to the method add. Thus,
those engines based on code patterns might not recommend the
correct API call HashSet.add. The approaches [18, 38] that 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.

2.2 Key Ideas

Instead of relying on the source code repetitiveness, our approach
is based on code change repetitiveness [18]. Hindle et al. [18]
reported that software exhibits its naturalness: source code is writ-
ten by human beings and it has a higher degree of repetitiveness
than natural-language texts. We expect the same principles of
naturalness of software [18] to occur on fine-grained code changes
(i.e., naturalness of code changes) because similar changes may be
performed to introduce similar behavior. Also, the reuse of software
libraries and frameworks leads to re-occurring changes.

When we apply our approach to the change pattern Adding a
Loop Collector, the addition of a variable declaration with a type
HashSet (generally a Collection) is often followed by the addition of
the API call HashSet.add of the same variable. Because we observe
the addition of HashSet.new in the recent change context, we are
able to recommend HashSet.add.

For our approach to work, we rely on two key ideas.
The first key idea is that we develop a statistical model to im-
plicitly capture a large number of change patterns (i.e., frequent
fine-grained code changes occurring together) in our training data.
The recent fine-grained code changes observed 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
method call HashSet.add). We also use the code context surrounding
the requested location, which might contain the code tokens that are
part of the change patterns. In our motivating example, the token
for is part of the auto-completion context and is also part of
the change pattern of Adding a Loop Collector because programmers
often collect the elements into a collection via a for loop. Thus, the
token for might help in recommending the method call add at line 4.
The second key idea is that 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. For example,
a change to a local variable (e.g., to t, not shown) is not part of
the above change pattern. We rely on the basis of consensus in
which project-specific changes are expected to appear less frequently
than the true changes in a pattern when we consider an ultra-large number
of changes. The true change patterns will emerge and be captured.

```java
a) for (Task t : tasks) {
  t.execute();
}

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

Figure 1: A Change Pattern: Adding a Loop Collector

Given previous changes, the context of the recommendation invo-
cation, and the inference model, APIREC computes the likelihood
of inserting an API method call. Finally, it ranks the candidate API
calls based on computed likelihood.

To empirically evaluate the usefulness of our approach we mea-
sure the accuracy of the recommender. We use top-k accuracy as
the likelihood that the correct API is in the first k recommended
APIs. We measured the accuracy in three different scenarios: in the
community evaluation we train on 50 open-source projects then we
measure APIREC’s accuracy on a corpus of 8 projects that other
researchers [18, 38] have previously used; in the project evaluation
we do a 10-fold cross-validation on each of the 8 projects used
by others [18, 38]; in the user evaluation we do a 10-fold cross-
validation on the same 8 projects as above, but only on the commits
coming from a single user. Whereas the first kind of evaluation is
standard in the community, our 3-pronged evaluation thoroughly
measures the impact of project culture and individual user’s habits.
In addition, we compare APIREC with the best in class approach
(n-gram [45]) from the statistical learning approaches.

This paper makes the following contributions:

1. Approach. We present a novel approach that combines fine-
grained changes with statistical learning to create a new gen-
eration of code-completion tools. We set forth a new direction
that takes advantage of both language models for source code
and of 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 a given part of the code.

3. Empirical Evaluation. Our empirical evaluation on real-
world projects shows that APIREC has high accuracy in API
code completion: top-1 average accuracy for in-vocabulary
is 59.5% and top-5 accuracy is 75%. This is a significant
improvement over the previous best-in-class approach: 2.4x
for top-1 and 2.2x for top-5.

2.2 Key Ideas

Instead of relying on the source code repetitiveness, our approach
is based on code change repetitiveness [18]. Hindle et al. [18]
reported that software exhibits its naturalness: source code is writ-
ten by human beings and it has a higher degree of repetitiveness
than natural-language texts. We expect the same principles of
naturalness of software [18] to occur on fine-grained code changes
(i.e., naturalness of code changes) because similar changes may be
performed to introduce similar behavior. Also, the reuse of software
libraries and frameworks leads to re-occurring changes.

When we apply our approach to the change pattern Adding a
Loop Collector, the addition of a variable declaration with a type
HashSet (generally a Collection) is often followed by the addition of
the API call HashSet.add of the same variable. Because we observe
the addition of HashSet.new in the recent change context, we are
able to recommend HashSet.add.

For our approach to work, we rely on two key ideas.
The first key idea is that we develop a statistical model to im-
plicitly capture a large number of change patterns (i.e., frequent
fine-grained code changes occurring together) in our training data.
The recent fine-grained code changes observed 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
method call HashSet.add). We also use the code context surrounding
the requested location, which might contain the code tokens that are
part of the change patterns. In our motivating example, the token
for is part of the auto-completion context and is also part of
the change pattern of Adding a Loop Collector because programmers
often collect the elements into a collection via a for loop. Thus, the
token for might help in recommending the method call add at line 4.
The second key idea is that 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. For example,
a change to a local variable (e.g., to t, not shown) is not part of
the above change pattern. We rely on the basis of consensus in
which project-specific changes are expected to appear less frequently
than the true changes in a pattern when we consider an ultra-large number
of changes. The true change patterns will emerge and be captured.
3. IMPORTANT CONCEPTS

In this section, we present the important concepts of our work.

3.1 API call Completion

In traditional code completion, the programmer invokes the code completion engine by placing a dot after a variable, such as v. However, there are other ways to invoke methods in Java. In APIREc, we also support the scenario in which the cursor can be placed at any location in the current code. For example, after the $= $ sign in an assignment, as in $v = w$, a user can request a method call and APIREc must be able to recommend a candidate API call or method call. 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 defined in a project (we’ll refer to both as API calls).

3.2 Fine-grained Atomic Code Changes

In APIREc, we represent source code as Abstract Syntax Trees (ASTs). It allows us to avoid cosmetic changes which are not helpful for recommendation. For a changed file, we compare the ASTs before and after the changes to derive the fine-grained AST changes. (We use GumTree [15] for this task).

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 code changes for the code editing scenario in Figure 1. Operation kinds represent the possible changes an AST node can have: change, add, delete, and move. The AST node types represent the Java AST nodes. The labels represent the textual information of the affected AST nodes. We use the labels only for the following nodes: method invocation, simple type, simple name (of a method invocation or a simple type), boolean constants, and the null value. The rationale is that the first three types of labels enable us to find change patterns and recommend the name for a method call of a class. For example, in Table 1, two MethodInvocation have their labels of add and getResult. The other simple names in Table 1 are project-specific names, which are simple names, the boolean values, null are special literals that might help in detecting change patterns involving those values. The labels for other nodes coincide with their AST node types.

We ignore the actual textual contents of the affected AST nodes to avoid making the changes too project-specific. For the aforementioned types of AST nodes for which we keep the labels, when comparing the atomic change, we compare their operation kinds, AST node types, and labels. For the other node types, we compare the labels only the operation kinds and AST node types.

Definition 2 (Transaction). The atomic changes from the same changed file in a commit are collected into a bag called a transaction.

We use a bag 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 changes are recovered from the committed changes in a code repository, we do not have the order they were written in.

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.

In Figure 1, the change context contains all atomic changes at line 1 ($c_1$–$c_{12}$), and the addition of results ($c_1$–$c_{13}$). 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 will help APIREc recommend the addition of HashSet.add. We also provide 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 go together in a change pattern than the other elements with no data dependency. Currently, we 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.

In APIREc, we obtain the code tokens from the AST node labels. For example, the code 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 change pattern. For example, the tokens for and HashSet.add are part of the Adding a Loop Collector. Thus, the code context helps us recommend the correct method 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. The distance is measured by the number of code tokens in the program. Moreover, 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 the method.

4. CHANGE INFERENCE MODEL

We develop APIREc, an association-based change inference model to recommend the API call at any requested location in the current code. APIREc takes as input the current code under editing, the requested location for completion, and the preceding changes. It produces the ranked list of the candidate API calls at the current location. We trained APIREc on the atomic changes from the transactions in our training corpus (Sec. 5).
of the next token. Thus, for simplicity we use a linear combination:

\[ \text{Score}(c, \mathcal{E}, \mathcal{T}) = w_c \times \text{Score}(c, \mathcal{E}) + w_T \times \text{Score}(c, \mathcal{T}) \]

where \(w_c\) and \(w_T\) are the weights corresponding to the contributions of both contexts. We will use the information in \(c = \langle \text{add}, \text{MethodInvocation}, \text{methodName}\rangle\) in a later computation of \(\text{Score}(c, \mathcal{E})\) and \(\text{Score}(c, \mathcal{T})\). Note that we know the first two components of \(c\) (add and MethodInvocation), and need to recommend candidates for methodName.

### 4.1.1 Computing \(\text{Score}(c, \mathcal{E})\)

The value of \(\text{Score}(c, \mathcal{E})\) represents the impact of the atomic changes in the change context for predicting \(c\). Assume that \(\mathcal{E}\) has \(c_1, c_2, \ldots, c_n\) where \(c_i\) is an atomic change. We compute it as follows:

\[ \text{Score}(c, \mathcal{E}) = \text{Score}(c, \{c_1, c_2, \ldots, c_n\}) = \frac{N(c_1, c_2, \ldots, c_n)}{N(c_1, c_2, \ldots, c_n, \mathcal{E})} \]

where \(N(c_1, c_2, \ldots, c_n)\) is the number of transactions containing the changes \(c_1, c_2, \ldots, c_n\), and \(N(c_1, c_2, \ldots, c_n, \mathcal{E})\) is the number of transactions containing the changes \(c_1, c_2, \ldots, c_n\) including change \(c\).

However, it is not always that we can compute those numbers 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 cases for such co-occurrences. Thus, we approximate the computation in Equation 1 as follows.

\[ \text{Score}(c, \mathcal{E}) \approx \text{Score}(c, \{c_1, c_2, \ldots, c_n\}) \approx \text{Score}(c, c_1) \times \text{Score}(c, c_2) \times \ldots \times \text{Score}(c, c_n) \]

in which, we estimate \(\text{Score}(c, c_i)\) (i.e., the likelihood that \(c\) occurs given that \(c_i\) occurred) as follows:

\[ \text{Score}(c, c_i) \approx \text{Pr}(c|c_i) \approx \frac{N(c_i, c) + 1}{N(c_i) + 1} \]

where \(N(c, c_i)\) is the number of transactions in which the atomic changes \(c\) and \(c_i\) co-appear, and \(N(c_i)\) is the number of transactions containing the change \(c_i\).

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

\[ \log(\text{Score}(c, c_i)) \approx \frac{1}{d(c, c_i)} \times \log \left( \frac{N(c_i, c) + 1}{N(c_i) + 1} \right) \]

where \(d(c, c_i)\) is the distance between the change \(c\) and \(c_i\). For simple computation, 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 \(d(c, c_i)\).

Because \(c = \langle \text{add}, \text{MethodInvocation}, \text{methodName}\rangle\) and because we know that we want to recommend an addition of a method invocation, the method name (denoted by \(\text{methodName}\)) is the only variable in \(c\). The operation add and the AST node type MethodInvocation are constant for all \(c_i\). Thus, we have

\[ \log(\text{Score}(c, c_i)) \propto \log(\text{Score}(c, \mathcal{E})) \]

\[ \propto \log(\text{Score}(c, \{c_1, c_2, \ldots, c_n\})) \propto \log(\text{Score}(c, \mathcal{E})) \]

where \(c_1, c_2, \ldots, c_n\) are the weights corresponding to the context.

Finally, from the Equation 2 we have:

\[ \log(\text{Score}(c, \mathcal{E})) \propto \log(\text{Score}(c, \mathcal{E})) \]

\[ \propto \log(\text{Score}(c, \{c_1, c_2, \ldots, c_n\})) \propto \log(\text{Score}(c, \mathcal{E})) \]

\[ \propto \sum_{i=1}^{n} \frac{w_{\text{scope}}, \log \left( \frac{N(c_i, c) + 1}{N(c_i) + 1} \right)}{d(c, c_i)} \]

### 4.1.2 Computing \(\text{Score}(c, \mathcal{T})\)

We estimate the likelihood score \(\text{Score}(c, \mathcal{T})\) of \(c\) given the code context \(T\) in the same manner as the computation for the score \(\text{Score}(c, \mathcal{E})\). \(\text{Score}(c, \mathcal{T})\) is estimated according to:

\[ \log(\text{Score}(c, \mathcal{T})) \propto \log(\text{Score}(c, \mathcal{E})) \]

\[ \propto \log(\text{Score}(c, \{c_1, c_2, \ldots, c_n\})) \propto \log(\text{Score}(c, \mathcal{E})) \]

\[ \propto \sum_{i=1}^{m} \frac{w_{\text{scope}}, \log \left( \frac{N(c_i, c) + 1}{N(c_i) + 1} \right)}{d(c, c_i)} \]

In this formula:

1. \(t_1, t_2, \ldots, t_m\) are \(m\) code tokens of interest in the code context.
2. \( \text{Score}(c, t_i) \) 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. \( w_{\text{scope}} \), 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 \( d(c, c_j) \).

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.

Finally, from Formulas 3 and 4 we derive the scores \( \text{Score}(c, \mathcal{C}) \) and \( \text{Score}(c, \mathcal{G}) \).

### 4.2 Training and Recommendation

For training, we need to process a large number of change revisions containing a large number of fine-grained changes. We collect the numbers of the (co-)occurrences of the fine-grained atomic changes (i.e., \( N(c, t_i) \) and \( N(c_i) \)) in Formula 5 and the number of the (co-)occurrences of the atomic changes and the code tokens of interest (i.e., \( N(c, t_i) \) and \( N(t_i) \)) in Formula 6.

For the weights \( w_{\mathcal{G}} \) and \( w_{\mathcal{C}} \), we fix one of them by using \( w_{\mathcal{G}} + w_{\mathcal{C}} = 1 \). We use Hill-climbing adaptive learning as in [52] to learn the values for \( w_{\mathcal{G}} \) from the training set. The idea of the training algorithm for that weight is 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 models are used to estimate the scores \( \text{Score}(c, \mathcal{C}) \) and \( \text{Score}(c, \mathcal{G}) \). The combined score is computed with the current value of the weight \( w_{\mathcal{G}} \). The candidates \( c \) are ranked. The goal function MAP \( (m_{\text{test}}, P_{\text{test}}) \) as in [52] between the list of predicted method calls, \( P_{\text{test}} \), and the actual one, \( m_{\text{test}} \), is computed. The weight value is then adjusted. The process is repeated. Finally, the optimal weight corresponding to the highest value of MAP is used.

For API call recommendation, 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., a method name or an addition of a method invocation. We compute the occurrence probability for all candidate changes in the vocabulary that satisfy the following: 1) it is an addition of a method invocation, and 2) it has appeared in at least one transaction with at least one change in \( c_1, \ldots, c_n \), or in at least one transaction with at least one token in \( t_1, \ldots, t_m \). The second condition ensures us to avoid the trivial cases of zero occurrences. Finally, we rank the candidates in a list according to their scores.

### 4.3 Running Example

Let us explain the computation for 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 1. (S)he requests the engine to complete the code with an API method invocation. Our goal is to recommend a method call for the variable results.

Table 1 shows the computation of the scores. First, all the atomic changes that preceded the current location are collected into the change context \( c_1, c_2, \ldots, c_{13} \) (assume that in this example, there is no other changes outside of the method). The tokens prior to the current location are considered including \( t_1, t_2, \ldots, t_4 \) (i.e., the code context). The token types and labels are presented in Table 1.

<table>
<thead>
<tr>
<th>Ind.</th>
<th>Operation, Type, Label</th>
<th>Score(a candidate given ( c_i ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1</td>
<td>Token, FOR, for</td>
<td>0.15</td>
</tr>
<tr>
<td>t_2</td>
<td>Token, Type, Task</td>
<td>0.00</td>
</tr>
<tr>
<td>t_3</td>
<td>Token, Var, t</td>
<td>0.00</td>
</tr>
<tr>
<td>t_4</td>
<td>Token, Expression, tasks</td>
<td>0.04</td>
</tr>
<tr>
<td>t_5</td>
<td>Token, Var, t</td>
<td>0.00</td>
</tr>
<tr>
<td>t_6</td>
<td>Token, MI, execute</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Combined score</td>
<td>0.40</td>
</tr>
</tbody>
</table>

The scores are computed using those values. For example, the association scores for the candidates with respect to the previous change \( c_3 \) (\text{Add, VDS, VDS}) and to the previous change \( c_1 \) (\text{Add, ST, Set}) are higher than others because in the training data, the changes involving adding a variable with the type Set or HashSet often go together with the changes involving the add method of the variable of that type. That is, the change pattern in this case consists of an addition of a variable of the type Set or HashSet followed by an addition of a method call to the add method of that variable. Both of the scores for Set and HashSet are high, because in the training data, programmers might declare the type of HashSet in some cases, and that of Set in other cases.

The scores for the candidates with respect to the prior tokens are computed similarly. For example, in Table 2 the association scores for the candidates with respect to the token for is higher than those of the other tokens because for iteration and the API method call HashSet.add is part of the change pattern \text{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 association 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.
5. EMPIRICAL EVALUATION

To evaluate APIRec, we answer the following research questions:

**RQ1.** How accurate is APIRec when recommending API calls?

**RQ2.** How does APIRec’s accuracy compare to the state-of-the-art approach [43]?

**RQ3.** How do different training corpora impact the accuracy of APIRec?

**RQ4.** What is the running time of APIRec?

### 5.1 Corpora

We compiled two disjunct corpora to train and test APIRec.

**Large Corpus.** This corpus consists of 50 randomly selected projects from Github that have long development histories (> 1,000 commits each). Table 3 shows the large number of commits contained in this corpus.

Based on previous research [6], in order to avoid large commit size, we did not select repositories which were migrated to GitHub from a centralized repository.

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 used GumTree [13] to compute the atomic changes (see definition in Sec. 3.2) between the before and after versions of each file. We collect the atomic changes from each changed file into a separate transaction (see definition in Sec. 3.2).

**Smaller Corpus.** This corpus contains eight projects from Github that have been used by previous researchers [18, 38]. The third column in Table 3 lists the statistics about this corpus. We extract atomic changes from this corpus in the same manner as with the larger corpus.

### 5.2 Evaluation Setup

In order to thoroughly evaluate APIRec and the impact of project culture and individual user’s habits, we designed three scenarios.

**Community Evaluation.** We trained APIRec with the Large Corpus and then we tested it against the Smaller Corpus.

**Project Evaluation.** For each project in the Smaller Corpus, we trained APIRec on the first 90% of commits, and then we tested on the remaining 10% of commits (10-fold cross validation).

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

### 5.3 Procedure, Metrics, and Settings

We want to replicate real-world code evolution scenarios and evaluate how successful APIRec would have been in giving recommendations. 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 (s)he invokes the recommendation engine to recommend a method call. We used the following procedure to choose the prediction point. First, we sort the atomic changes in the transaction according to their locations in the file. Then, we divide the changes in half. Let \( n \) be the transaction’s size. We begin by looking at the change at position \( l = \lfloor n/2 \rfloor + 1 \); 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.

### 5.4 Impact of Out Of Vocabulary Data

APIRec is a statistical, data-driven approach. Its results are affected by the sufficiency of the training data. We aim to evaluate how data that has not been observed during training impacts APIRec’s accuracy. This is often referred to as out-of-vocabulary (OOV).

In our data, a method 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 trains on the large corpus, but we test it against the smaller corpus. For example, a method such as calcCustomData defined in a project from the Smaller Corpus will not be learned by training on the Large Corpus. In contrast, any method used in the Large Corpus such as String.length() from JDK, will be in the vocabulary.

To evaluate the effect of OOV data, we chose a random project, Froyo-Email, from the Smaller 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 described earlier to measure top-k accuracy.

In the second execution, we followed the same procedure 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, in the second execution, APIRec only tried to predict methods which it had previously seen. To be able to compare against the two runs, APIRec made exactly

---

**Table 3: Collected Data on Fine-grained Changes**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Large Corpus</th>
<th>Small Corpus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>Total source files</td>
<td>48,699</td>
<td>8,561</td>
</tr>
<tr>
<td>Total SLOCs</td>
<td>7,642,422</td>
<td>1,331,240</td>
</tr>
<tr>
<td>Number of commits</td>
<td>113,103</td>
<td>18,233</td>
</tr>
<tr>
<td>Total changed files</td>
<td>471,730</td>
<td>63,962</td>
</tr>
<tr>
<td>Total AST nodes</td>
<td>714,622,846</td>
<td>86,297,938</td>
</tr>
<tr>
<td>Total changed AST nodes</td>
<td>43,538,386</td>
<td>4,487,479</td>
</tr>
<tr>
<td>Total detected changes</td>
<td>1,915,535</td>
<td>252,522</td>
</tr>
<tr>
<td>Total detected changes</td>
<td>788,741</td>
<td>117,481</td>
</tr>
</tbody>
</table>
Table 4: Impact of OOV on Accuracy

<table>
<thead>
<tr>
<th></th>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-3</th>
<th>Top-4</th>
<th>Top-5</th>
<th>Top-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOV</td>
<td>44.8%</td>
<td>51.6%</td>
<td>54.7%</td>
<td>55.9%</td>
<td>56.8%</td>
<td>62.5%</td>
</tr>
<tr>
<td>IN</td>
<td>59.5%</td>
<td>67.1%</td>
<td>71.7%</td>
<td>73.0%</td>
<td>75.0%</td>
<td>81.2%</td>
</tr>
</tbody>
</table>

Table 5: JDK Recommendation Accuracy for the Community Evaluation (No OOV API elements in JDK library in this experiment) (%)

<table>
<thead>
<tr>
<th>System</th>
<th>Model</th>
<th>Top1</th>
<th>Top2</th>
<th>Top3</th>
<th>Top4</th>
<th>Top5</th>
<th>Top10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaxy</td>
<td>APIRec</td>
<td>66.8</td>
<td>78.0</td>
<td>80.9</td>
<td>82.1</td>
<td>82.6</td>
<td>85.6</td>
</tr>
<tr>
<td>(6,956)</td>
<td>n-gram</td>
<td>17.6</td>
<td>23.5</td>
<td>26.1</td>
<td>29.1</td>
<td>31.9</td>
<td>39.3</td>
</tr>
<tr>
<td>log4j</td>
<td>APIRec</td>
<td>57.3</td>
<td>70.7</td>
<td>72.6</td>
<td>75.0</td>
<td>76.1</td>
<td>78.4</td>
</tr>
<tr>
<td>(5,679)</td>
<td>n-gram</td>
<td>15.5</td>
<td>21.2</td>
<td>25.4</td>
<td>29.2</td>
<td>31.7</td>
<td>37.8</td>
</tr>
<tr>
<td>spring</td>
<td>APIRec</td>
<td>61.1</td>
<td>70.6</td>
<td>71.6</td>
<td>77.4</td>
<td>79.0</td>
<td>80.6</td>
</tr>
<tr>
<td>(9,885)</td>
<td>n-gram</td>
<td>19.1</td>
<td>24.8</td>
<td>27.5</td>
<td>29.8</td>
<td>32.3</td>
<td>40.0</td>
</tr>
<tr>
<td>antl4</td>
<td>APIRec</td>
<td>67.9</td>
<td>81.9</td>
<td>83.6</td>
<td>84.0</td>
<td>84.5</td>
<td>85.5</td>
</tr>
<tr>
<td>(11,150)</td>
<td>n-gram</td>
<td>22.8</td>
<td>30.9</td>
<td>34.7</td>
<td>37.6</td>
<td>39.5</td>
<td>46.9</td>
</tr>
<tr>
<td>Test</td>
<td>APIRec</td>
<td>75.4</td>
<td>82.6</td>
<td>85.1</td>
<td>85.9</td>
<td>86.0</td>
<td>87.7</td>
</tr>
<tr>
<td>(9,075)</td>
<td>n-gram</td>
<td>17.6</td>
<td>25.0</td>
<td>28.3</td>
<td>31.0</td>
<td>33.0</td>
<td>40.8</td>
</tr>
<tr>
<td>Froyo-E</td>
<td>APIRec</td>
<td>74.3</td>
<td>81.7</td>
<td>84.5</td>
<td>85.6</td>
<td>86.0</td>
<td>89.8</td>
</tr>
<tr>
<td>(5,568)</td>
<td>n-gram</td>
<td>39.9</td>
<td>46.7</td>
<td>48.5</td>
<td>51.6</td>
<td>54.9</td>
<td>60.4</td>
</tr>
<tr>
<td>Grid-S</td>
<td>APIRec</td>
<td>61.6</td>
<td>80.2</td>
<td>80.8</td>
<td>84.0</td>
<td>89.8</td>
<td>91.1</td>
</tr>
<tr>
<td>(9,215)</td>
<td>n-gram</td>
<td>22.3</td>
<td>27.7</td>
<td>29.9</td>
<td>32.2</td>
<td>34.2</td>
<td>42.5</td>
</tr>
<tr>
<td>Text</td>
<td>APIRec</td>
<td>36.4</td>
<td>76.5</td>
<td>77.3</td>
<td>79.3</td>
<td>80.1</td>
<td>81.5</td>
</tr>
<tr>
<td>(6,427)</td>
<td>n-gram</td>
<td>31.4</td>
<td>37.5</td>
<td>40.8</td>
<td>44.0</td>
<td>47.9</td>
<td>55.6</td>
</tr>
</tbody>
</table>

Improvement Factor 3.03x 2.77x 2.59x 2.40x 2.26x 1.92x

5,568 recommendations in each execution.

Table 2 shows the top-k accuracy for the two cases. When APIRec uses in vocabulary (IN) elements it is able to make better recommendations ranging from 14.5% to 18.7% improvements in accuracy. We also measured the OOV rate, defined as the percentage of predicted API call in the Smaller Corpus that are not contained in the Large Corpus. The OOV rate for the Froyo-Email test project is 28.1%, a lower bound of the OOV in accuracy.

This result shows that OOV has considerable impact on suggestion accuracy. This is reasonable because like any learning model, APIRec needs to observe the API calls to be able to recommend them. This suggests that providing more training data would give APIRec a chance of achieving higher accuracy.

Even with the OOV issue, APIRec is able to achieve high suggestion 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 5 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 recommendation in almost 60% of the cases. In 75% of the cases, it correctly recommends the call with only 5 candidates.

5.5 Community Evaluation

In this section we compare APIRec with the state-of-the-art API completion approach by Raychev et al. [45]. We implemented their n-gram API recommendation model according to the description in their paper. For each API call in the data, we train an n-gram model [28] with the code of the method that contains the API call.

Unlike n-gram which is trained on the source code of the entire snapshot(s), APIRec is trained on bags (transactions) of atomic changes together with change and code contexts from the evolution of project(s).

We do not compare with GraLaN [38], another API recommendation approach, because it is a graph-generative model that operates on graphs built from source code and requires semantic information. However, a changed file alone might not contain such information. Therefore, we are unable to compare directly with their approach.

Figure 2 compares the accuracy between APIRec and n-gram. The total number of recommendations for each project appears in parentheses after the project name. APIRec outperforms n-gram across the board. At Top-1 it is better by a factor of 2.4x, and at Top-5 by 1.9x.

We investigated the differences in accuracy between the two approaches. We found that sometimes n-gram requires strict, unnecessary ordering between API calls. For example, after a FileInputStream.write call, in some programs, FileInputStream.read appears before FileInputStream.read and vice versa. Strict order in the n-gram model might miss a pattern in which open must come before read and write, but there is no required order between read and write. In this case, open→read→write and open→write→read are not considered the same pattern by n-gram. Thus, open→write cannot be used to recommend read. In contrast, we found many cases where in the same transaction an open was changed / added together with and addition of either a read or a write. APIRec used the context with the modification/addition of open to recommend either read or write. In other words, the co-changes in the same transaction seem to be relevant to the same change task (e.g., “open a file and read from it” or “open a file and write to it”). The atomic changes for the same task often co-occur in the same transaction.

APIRec achieves high accuracy. Top-1 is correct in 28.7%–44.8% of the cases (OOV), and in 44.2%–59.5% of the cases (IN). Top-5 accuracy can be as high as 57.2% (OOV) and 83.6% (IN).

5.6 Recommendation 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.

To evaluate APIRec’s accuracy in recommending JDK API calls, 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 don’t find such a method.

Table 5 shows the accuracy comparison between APIRec and n-gram when recommending JDK calls. 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-5.

The accuracy values in all the projects are higher than those for recommending general API calls (Figure 2). This is expected because JDK is a popular library for Java programs. With the same 50 projects for training, all the JDK API calls are in the vocabulary (IN). General method calls are project-specific and do not repeat as often, so are more likely to be out of vocabulary (OOV).

At Top-1, APIRec is better then n-gram by a factor of 3x, and at Top-5 by a factor of 2.3x.

5.7 Project Evaluation

In this experiment we aim to evaluate APIRec’s recommendation accuracy in a project when it is trained on the fine-grained changes within the same project. For comparison reasons we used the same set of testing projects as in the community evaluation. 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. We computed top-k accuracy as before.

Tables 6 and 7 show accuracy results for the project evaluation setting for: 1) recommending API calls, and 2) recommending JDK calls.
API method calls. The changes in individual projects do not repeat as much as those across projects. Thus, the accuracy values for each project are generally lower than those in the community evaluation. Comparing Tables 6 and 7, we can see that the accuracy in recommending general API calls is higher than the accuracy in recommending JDK APIs. This is because the JDK APIs are more likely to be used consistently across different projects. However, since JDK is a popular Java library, the code changes involving JDK APIs still occurred and repeated 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 6 and 7). Note that, in this experiment, with our training projects, all JDK API calls are in the vocabulary (IN). Thus, Table 7 has only the result for APIRec.in.

Table 6 shows that the values for APIRec.in are slightly higher than those for APIRec.oov. This is reasonable since most used method calls already exist. Thus, if we use a long-enough change history in a project covering as many project-specific methods as possible, APIRec can be used to recommend the calls to those methods.

Due to the space limitation, we do not display the top-k accuracy numbers for the n-gram model in Table 6. However, the same trends apply. APIRec outperforms the n-gram model across the board.

5.8 User Evaluation

Figure 2: API calls Recommendation Accuracy for the Community Evaluation (%) per project (the parentheses are the number of recommendations).
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 select the user who has the most commits. Each 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 use the same projects as the Project Evaluation. We used the same sorting technique so that 90% of the commits were used for training, and the most recent 10% of commits were used for recommendation. We then similarly compute top-k accuracy.

Table 8 shows the result for the largest user from each project. When compared with Figure 2 and Table 6 the user evaluation performs worse than the Community Evaluation. We expect this result because more training should yield better results.

The more surprising finding is that the User Evaluation generally performs better than the Project Evaluation. 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 than other training data. By using the author of the code, we can train with less data, and yet have more precise results then when training with the entire project.

### 5.9 Cases Studies on Co-changes

Our approach relies on capturing change patterns consisting of frequent fine-grained co-changes. Co-changes are changes that frequently occur together. Thus, we investigate the method calls that often appear together in the same transaction.

After training ApiRec, we stored several internal data structures for the model. An important one is the association values among changes $N = (c, c')$, i.e., the numbers of transactions containing the pairs of changes found in the training data. We then measured the Jaccard similarity coefficient (Jaccard index) between two method calls in each pair as the co-change likelihood. Table 2 shows the top-ranked pairs of JDK method calls with higher Jaccard indexes in our training data. They make sense within the common JDK idioms and are useful in recommending API calls.

### 5.10 Running Time

All experiments were run on a computer with Intel Xeon E5-2620 2.1GHz (configured with 1 thread and 32GB RAM). The running time is reported in Table 10. The training time is significant. However, the recommendation time is short (1-2 seconds per recommendation), thus, making ApiRec could be used in an IDE interactively.

### 5.11 Threats to Validity and Limitations

**Threats to Validity.** Our corpus might not be representative. For different projects with different OOV rates, the results might vary. However, for comparison, we ran all approaches on the same dataset and we evaluated the impact of OOV data (Sec. 5.4). We do not have the tool in Raychev et al. 45 (it is not publicly available), but we followed the approach described in the paper. The simulated procedure in our evaluation is not a true editing session. The study on usefulness needs to involve human subjects. The results for JDK might not be representative for other libraries.

**Limitations.** Our approach also has shortcomings. First, out-of-vocabulary is an issue, as explained earlier. However, if trained with sufficient data, ApiRec performs better than the code-based language model on API calls in 45. Second, using the file level as the scope for transactions may also lead to a loss of accuracy. In some cases files may contain tangled changes: changes from multiple tasks that are committed together. 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(s). Moreover, missing/incorrect semantic information (types, dependencies) leads to ApiRec’s inaccuracy. Finally, ApiRec handles code context by finding the associations between code tokens in transactions. A potential alternative is the combination between ApiRec and a language model in the source code space such as GraLan 38.

### 5.12 Implications

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

**Developers.** ApiRec relies on a trained model, so choosing the right training corpus is important. Our results suggest that training on the community corpus leads to a higher accuracy than when training on the project corpus. This difference is likely to grow if more projects are included in the community corpus. The implication for this observation is that developers using a statistical learning recommender should use a community trained model rather than a project one.

When using the user trained model (training and recommending on the changes of one user) the accuracy was between that of the community trained corpus and the project trained corpus. Even though the project evaluation results are not directly comparable to the other two (because the user commits are fewer than the projects), the results suggest that with a long enough user specific history, the project trained model may prove to be the best of the three. The implication for this observation is that developers using a statistical learning recommender should obtain a community bootstrapped model, and then further refine it with their own changes.
Tool Builders. Section 5.10 shows that a challenge when using statistical learning recommenders is the long time training time for the model. This problem will become more pressing in the future, as more and more development tools need to train models on user data. For example, Dias et al. [10] present EpiceaUntangler, a tool that learns from developer atomic changes in order to recommend untangled changes. As these tools move from research into practice, development tool builders will have to accommodate the means to iteratively train models on fine grained user data.

The problem can be mitigated in several 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 over the day.

Researchers. Our paper, together with recent related work [2,7], shows that fine grained code changes are highly repetitive. This opens up a 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 catalogues of software change building blocks, incorporating change patterns into atomic changes via language and IDE design, predicting the changes required for a task, etc.

Based on our experience with ApiRec’s evaluation, we recommend researchers to evaluate machine learning development tools on multiple data sets: between projects, within projects, within users, within libraries, within methods / classes, etc.

6. RELATED WORK

Traditional code completion engines. In modern IDEs, code completion is an important feature. Eclipse [14], IntelliJ [22], and other IDEs [21] complete the member of a variable under editing. The list of candidate API calls is often presented in a pre-defined order that might not be relevant to the current code context.

Code and API completion based on statistical language models. Hindle et al. [18] use the n-gram language model [28] on lexical tokens to show that source code has high repetitiveness, and then use it to recommend the next token. SLLAMC [41] defines n-grams of code tokens with semantic annotations including data and token types. Tu et al. [53] enhance n-grams with caching of recently seen tokens. Raychev et al. [45] use common sequences of per object API calls to recommend the next call. GraLan [38] is a generative graph-based model that can recommend the next API call. It uses API usage graphs to train a Bayesian inference model to learn what nodes are likely to be added to an existing completion site.

White et al. [54] applied RNN LM on lexical code tokens to achieve a higher accuracy than n-gram. Mo et al.’s tree-based convolutional neural network (TBCNN) is applied to source code to recommend syntactic units. Allamanis et al. [3] use bimodal modeling for short texts and source code snippets. Maddison and Tarlow [29] use probabilistic context free grammars and neuro-probabilistic language models for source code. Their model works on lexical and syntactic contexts.

In comparison to those statistical approaches, ApiRec has key differences. First, they are based on the principle of source code repetitiveness, while ApiRec relies on the repetitiveness nature of fine-grained source code changes. Second, ApiRec is tailored toward method invocation recommendation including API calls. Other models, except GraLan and Raychev et al. [45], are either for general tokens [18][41][53] or for special-purpose recommendations (e.g., AST structures [26], coding conventions [2], or methods/classes’ names [3]). Further studies are needed to investigate the combination of the two directions: taking advantage of both code-based and change-based recommendation models.

Code and API completion based on mined patterns. Bruch et al. [7] propose three code completion algorithms to recommend the method call for a single variable under editing based on code examples in a database. The first one, FreqCCS, recommends the method that is most frequently used in the database. The second one, ArCCS, mines the associate rules $A \rightarrow B$ in which if method A is used, method B is often called and will be recommended. The third one, called best-matching neighbor, 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 for API recommendation. Grapecq [59] mines patterns as graphs and matches them against the current code.

Robbes and Lanza [7] propose 6 strategies to improve code completion using recent modified/inserted code during an editing session. In comparison, ApiRec uses a statistical approach, rather than simple matching. Moreover, we train our model on the entire change histories of a large number of repositories rather than the editing changes in a session.

There exist other deterministic approaches to improve code completion / recommendation and code search by using recent editing history [20],[47], cloned code [17], developers’ editing history [24], API usages, examples, and documentation [8][29][30][31][34][51], structural context [19][32], padding parameter filling [55], interactive code generation [43], specifications on constraints between input and output [46][50], API documentation [31], type information [49], etc.

7. CONCLUSION

In this paper we address the problem of API method recommendation for code-completion tools. Our approach uses a statistical learning model that we train on fine-grained code changes. Our thorough, 3-pronged empirical evaluation shows that our approach and tool, ApiRec, outperform the previous state of the art by a factor of at least 2x.

Surprising findings. Although we knew for a few years [37][40] that code changes are repetitive, this work is the first that leverages 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, there is still too much noise surrounding the token that needs to be recommended at a given code location. Our approach relies on the fact that when developers change non-contiguous lines of code, the developers have higher-level intents (e.g., add loop collector) that connect those fine-grained changes logically. When we mine these in a large corpus, the changes belonging to higher-level intents will appear more frequently than project-specific changes. They help us to counteract the noise of unrelated tokens preceding the location of the cursor.

This current result puts us on a long-term trajectory to recommend a whole sequence of code changes that is suitable for a given change intent. We hope that others will find novel uses of this promising direction.

Acknowledgements

We gratefully acknowledge the funding from NSF through grants CCF-1439957, CCF-1442157, CCF-1413927, CNS-1513263, CCF-1518897. We especially thank Nicholas Nelson for helping us with data visualization.
8. REFERENCES


