Abstract—Automatic code completion helps improve developers’ productivity in their programming tasks. A program contains instructions expressed via code statements, which are considered as the basic units of program execution. In this paper, we introduce AUTO\textsc{Sc}, which combines program analysis and the principle of software naturalness to fill in a partially completed statement. AUTO\textsc{Sc} benefits from the strengths of both directions, in which the completed code statement is both frequent and valid. AUTO\textsc{Sc} is first trained on a large code corpus to derive the templates of candidate statements. Then, it uses program analysis to validate and concretize the templates into syntactically and type-valid candidate statements. Finally, these candidates are ranked by using a language model trained on the lexical form of the source code in the code corpus. Our empirical evaluation on the large datasets of real-world projects shows that AUTO\textsc{Sc} achieves 38.9–41.3\% top-1 accuracy and 48.2–50.1\% top-5 accuracy in statement completion. It also outperforms a state-of-the-art approach from 9X–69X in top-1 accuracy.

Keywords—Code Completion; Statement Completion; Statistical Language Model; Program Analysis;

I. INTRODUCTION

Code completion tool helps improve developers’ productivity by filling in the code during their editing. A program contains instructions in source code to perform certain tasks. The procedure to achieve a task is expressed via program statements, each of which is considered as the basic unit of execution in a program. A statement can declare a variable, define an expression, perform a simple action by calling a method, control the execution flow of other statements, create an object, or assign a value to a variable, attribute, or field [4]. Thus, in this work, we aim to support automated code completion to help developers fill in their current statements. During writing the body of a method, if a developer finishes one or more tokens of the current statement, the tool as requested will fill in the remaining tokens of that statement. If the tool finishes a statement, the tool will suggest the entire next statement (next-statement completion). Let us call it a statement completion (SC) tool. SC encompasses next-statement completion.

To build an effective and efficient SC tool, one would face the following key challenges. First, the tool must predict the statement that a developer intends to type next to perform the programming task at hand. Second, the resulting code after completion must conform to the syntactic and semantic constraints defined by the programming language in use.

To address the first challenge, one can rely on the principle of software naturalness [9]. Source code is naturally written with certain regularity, i.e., it is repetitive and does not occur randomly. The code elements appear together because they are intended by developers to achieve a programming task(s). Hindle et al. [9] showed that such regularity in source code can be captured by statistical language models (LMs), e.g., \( n \)-gram model [16] can be leveraged to support code completion for the next token. Thus, one could train an LM with a large code corpus and use it to predict each token at a time until a complete statement is suggested. However, the frequent code fragments learned from different contexts might make the code after completion syntactically or semantically incorrect. For example, after \( i = \), if the most frequent variable in a corpus is \( i \), the resulting code is \( i = i \), which is invalid. A naive solution that uses program analysis (PA) to enforce the constraints in such output with multiple tokens would face combinatorial explosion. For example, assume that at each step, a model predicts and maintains \( n \) most likely valid tokens, the number of statements with \( m \) code tokens is \( n^m \).

To address the second challenge, an SC tool can apply PA with program constraints on the candidate statements to eliminate the invalid ones, the number of the remaining, valid candidates is still large. The accuracy of such a naive solution is very low due to the confounding effect of the accuracy of a prediction model for each token (see Section VIII). Another solution to this issue is to search for an entire code statement. However, it is ineffective since statements are project-specific and do not repeat often across different methods or projects as reported in PCC [27]. In fact, learning to suggest entire statements is less effective than an SC tool that is capable of filling the remaining token(s) of the current statement.

This paper proposes AUTO\textsc{Sc}, which combines program analysis and statistical LM in the process of statement completion. We aim to benefit from the strengths of both directions in which LM produces natural code sequences and PA enforces syntactic and type constraints. AUTO\textsc{Sc} works in three phases. First, it uses the \( n \)-gram LM on an abstraction level higher than lexical code to learn to derive the most likely candidate templates for the current statement. A candidate statement is modeled by a sequence of special annotations called extended code tokens (excode for short). An excode for a token is an annotation representing the token type and/or data type,
if available. The token type encodes whether the token is a variable, a field access, a method call, a type (class), etc. Such information in a template helps a LM predict better the next token, e.g., a variable cannot be next to another. Data types help distinguish the code fragments having the same meaning but with different variables’ names, e.g., “int len = s.length();” and “int l = str.length();” have the same meaning of “Retrieving the length of a String and assign it to an int variable”. With the types instead of the variables’ names, the two fragments have the same template. Thus, AUTOSC can learn templates from one place to suggest for the other. Data types also help distinguish the cases in which two fragments with the same lexical tokens having different meaning. For example, x.next means a variable x of a Scanner accessing the field next, while in another place, it means a variable x of a LinkedList calling the method next. Data types thus help determine the accessible method calls or field accesses for a variable.

At the second step, the candidate templates are syntactically and type validated. Then, the valid templates are concretized into code sequences. Finally, all valid suggested code sequences are ranked based on their occurrence likelihoods given the partial code. To do this, we train another n-gram model on the lexical form of the source code in a code corpus.

We conducted several experiments to evaluate AUTO SC in statement completion on a dataset used in the existing approaches [9], [20], [17] with +460K statements with a total of +1M suggestion points. Our results show that AUTO SC is very effective with top-1 accuracy of 40% and top-5 accuracy of 49.4% on average. That is, in 4 out of 10 cases, when a user requests to complete his/her currently-written statement, (s)he can find the remaining of the desired statement in the top of the suggestion list. Importantly, AUTO SC significantly improves over the baseline model using only n-gram on lexical code (up to 142X in top-1 accuracy) and the model using lexical n-gram+PA (up to 117X in top-1 accuracy). It also improves over the state-of-the-art tool PCC [27] with 69X higher in top-1 accuracy. In brief, our contributions include

1. A model with PA+LM to complete the current statement,
2. An empirical evaluation showing our model’s effectiveness and much better accuracy than the state-of-the-art tool.

II. MOTIVATING EXAMPLE

Figure 1 partially shows a method in Apache Ant [2]. Assume that the cursor is at line 5, right after the “=” sign. If a user requests a statement completion (SC) tool, it will complete the current statement, i.e., the assignment to the variable len. A SC tool would predict the intention to the variable len. A SC tool would try to complete the current statement, e.g., int len = children.getLength(). A SC tool would suggest the entire assignment statement, e.g., int len = children.getLength() (next-statement suggestion). That is, SC includes the functionality of next-statement suggestion (e.g., PCC [27]). Note that, the SC tool is automatically invoked as the user finishes typing a token in the middle of a statement, e.g., after int, len, “=”, etc.

To support statement completion, a model needs to consider the nature of source code. Source code is strictly defined by the syntax and semantics of the programming language. Source code is also repetitive [9]. Thus, the methods for SC can be realized in the following: information retrieval (IR) and pattern mining, program analysis, and statistical language model.

For IR and pattern mining, a model suggests to complete the current statement by searching for the same/similar statement(s) that have been seen in a corpus. When the retrieved statements have occurred frequently, they can be viewed as code patterns. Such a pattern or a retrieved statement can be used as the candidate for completion. However, the tokens need to be filled for the current statement might not be a pattern, leading to ineffectiveness of such approach. Moreover, while as single tokens, code is repetitive; as entire statements, they are quite unique for specific projects. This phenomenon was reported by Yang et al. [27]. Indeed, in our experiment (Section IX), the portion of repeated statements in our dataset is 25.9%. That is, 3 out of 4 cases on average cannot be correctly suggested by searching for the same statements in the corpus of the previously-seen statements. As an example, the statement int len = children.getLength(); is not used in any other project in our dataset. As an implication, to suggest or complete a statement, a model cannot rely solely on searching for the repeated statements as a whole.

For the program analysis (PA) direction, although the number of valid candidates for the next token is limited, the number of possible valid (complete) statements at the suggestion point might be combinatorially explosive or even infinite. For the right side of the assignment at line 5, the valid next-token candidates include the appropriate prefix operators (e.g., “++” and “--”), the open parenthesis, field access, method call, and local variable (e.g., children, l, etc.). However, there is an infinite number of valid statements at that point. In brief, program analysis direction could produce a large number of candidates with equal occurrence likelihoods, despite that the candidates are syntactically or semantically valid.

The statistical language models (LM) leverage the fact that code is highly repetitive and predictable [9]. The next tokens to be filled are based on the frequent sequences of tokens and the partial code. Solely relying on those to fill in a statement, a model could face the following issues. The first issue is caused by the fact that the code in different places with the same lexical code sequence have different meaning. For example, in one place, x.next means the variable x of a Scanner in

```java
Nodelist listChildNodes(Node parent, NodeFilter f) {  
    NodeListImpl matches = new NodeListImpl();  
    NodeList children = parent.getChildNodes();  
    if (children != null) {  
        int len = //Expected: "children.getLength();"  
        //Candidate 1: "children.getLength();"  
        //Candidate 2: "parent.numChildren();"  
        //Candidate 3: "0;"  
    }  
    return matches;  
}
```

Figure 1: A partial method in class DOMUtil of Apache Ant
JDK accessing the field `next`, while in another place, it means the variable `x` of a `LinkedList` calling the method `next`. In this case, a LM can mistakenly use one to suggest for another, e.g., it could recommend `?` after `x.next` for the field access of a `Scanner`, which results in a semantic error. Second, in contrast, to represent the same meaning in different places in the same or different projects, one could use different names of the variables. For example, the statement at line 5 is a code fragment that performs the task of “retrieving the size (length) of a list of nodes”. In other places, we might see `int size = children.getLength()`. Those two code fragments might be deemed as not performing the same task if only the lexical tokens are considered. Thus, an LM cannot learn from one place to complete the statement in the other place.

Third, the names of method calls and field accesses might not appear in the training data, leading to the out-of-vocabulary (OOV) issue. This also applies to local variables due to their method-specific nature. In natural language, human can understand a sentence even an OOV word is missing. However, OOV could cause the code un-compilable. Finally, even OOV does not occur, the completed code by a LM could violate syntactic and semantic constraints. At line 5, the most likely next sequences of tokens include `i;`, `(`, or `)`, which are frequent in a corpus. That would induce an “undeclared variable” error.

From the above discussion, it is natural to combine PA and LM to benefit from the strengths of both directions in completing the current statement. For example, PA can be used to derive/select the syntactically and type-valid candidate statements from the list produced by statistical LM, while the latter can apply the principle of code repetition [9] to rank the valid and most likely statements higher in the candidate list.

A naive LM+PA solution would use a statistical LM to predict the next token one by one, and then use PA to filter out the invalid ones and rank the remaining ones according to their occurrence likelihoods. However, doing so, the number of valid statements is still large. Our experiment (Section IX) showed that among those valid ones, the correct one is rarely in the top 5 most likely candidates: top-5 accuracy is ≈ 0.83%.

III. KEY IDEAS AND APPROACH OVERVIEW

We develop AUTOSC, which combines program syntax and type constraints and the naturalness principle of source code [9] in the process of code statement completion. First, we use an LM on an abstraction level higher than lexical source code to learn to derive the most likely candidate templates for the current statement. Second, the candidate templates are syntactically and type validated, and concretized into one or more code sequence candidates. After all, we rank the candidate code statements accordingly to their occurrence likelihoods by another LM trained on lexical source code.

To overcome the issues of a LM on OOV and capturing high-level abstraction of source code, we design a template as a sequence of extended annotation code tokens (`excode` for short). An `excode` for a token is an annotation representing the token type and/or data type, if available (details in Section IV). For an identifier, its `excode` captures its token type, i.e., a variable, a field access, a method call, a type (class), etc. Token types in a template helps a LM predict better the next token, e.g., an `(` is needed after a method call. `excode` also captures the data type if available. For example, `children` is of the type `NodeList` at line 5. The data type facilitates a model to restrict possible method calls or field accesses. However, the variable names are not kept in an `excode` because we want to capture the code pattern at a higher level. In contrast, `excode` keeps the name of the class that is declared (e.g., `NodeList` in `NodeList children`), the method that is called (e.g., `getLength`), the field that is accessed (e.g., `next`). The rationale is that those elements are designed to be (re)used in different classes, methods in the same or different projects (e.g., libraries/frameworks). Such reused names would be useful for a model to learn to apply in different places. The literals are not kept because they tend to be project-specific except if they are special literals, such as `null` or `0`. The other kinds of tokens are kept intact.

These treatments help AUTOSC learn better the candidate templates. At line 5, the template has the left-hand side of `TYPE(int) VAR(int), OP(ASSIGN)`, and the right-hand side of `VAR(NodeList) OP(ACC) CALL (NodeList, getLength, 0, int) LP RP`. By raising the abstraction from the code, we aim to increase the regularity/repetition to help a LM learn from other places to better find the statement templates. For example, while the fragment `len = children.getLength()` has never appeared in the project, the above template occurs 6 times.

Our process of learning templates and concretizing into code helps our model overcome OOV and the nature of locally-used variable names. The templates at higher level are learned from one place and applied to another, and PA is used to concretize them with concrete accessible variables at the new place. The step of learning at template level helps AUTOSC cover more candidates (improving recall), while the use of PA helps retain more valid ones (improving precision).

To enforce syntax and type constraints, we train an LM with the sequences of `excode` to learn the statement templates, and use that LM to suggest each `excode` by `excode` to form the candidate templates. During that, syntactical and type rules are applied to those candidate templates to enforce their validity.

The second LM on lexical source code at the last step helps select the variable names when there still exist multiple candidates of code sequences. When several valid variables are valid, the lexical LM selects the names that come naturally and frequently at the place. For example, the tokens `children, node, parent, etc.` often go together. Thus, at line 5, the variable name `children` likely occur than `student, network, etc.`

IV. EXTENDED CODE ANNOTATION

A. Design Strategies

We present extended code annotation (`excode`), a code representation designed for SC. Let us explain what information needs to be encoded. We first aim to encode `token type` of a code token. That is, we need to encode whether a code token is a keyword, separator, operator, method call, field access, variable, etc. This enables AUTOSC to learn program syntaxes on the validity of a next code token, e.g., “A left parenthesis
must appear after the method call next, not after the field next”. Additionally, for the validation of the type constraints, the data type of code tokens, especially of method call, field access, and variable, also needs to be encoded. For example, the RHS expression of the assignment at line 5 must be of the type int or Integer because the LHS variable is of the type int.

Because local variables are used locally, their names and meaning might be different in different methods. Thus, they can not be learned by a LM in a method to apply to the local variables but with different variable names in other methods. Thus, the names of local variables should be abstracted in the representation to better capture code regularity. Meanwhile, the names of data types, methods, and fields are kept since those elements are designed to be reused in other places. Thus, those names can be learned from one place and be applied to others.

B. Extended Code Tokens Annotation and Concretization

Definition 1 (Token Type). The token types in a program with regard to a programming language include keyword, operator, separator, data type, method call, field, variable, and literal.

For children.getLength(), the token types of children and getLength are variable and method call, respectively, while . (access) is an operator, and ( and ) are separators LP and RP.

Definition 2 (Excode Token). An excode token is an annotation corresponding to a code token, that represents its syntactic and type information, including its token type and data type.

Table I shows the rules to construct excode tokens for popular kinds of code tokens. For children in children.getLength(), which has the role of a variable, its corresponding excode token consists of the annotations “VAR”, “,”, its data type NodeList, and “)”. For method calls and field accesses, the information including the enclosing type name, return type, and the arguments, are additionally incorporated in the excode tokens. For example, the excode of getLength in children.getLength() is CALL(NodeList, getType, 0, int).

Definition 3 (Excode annotation function α). The annotation function $\alpha(C)$ on a code sequence $C = e_1 e_2 ... e_n$, defines the corresponding excode sequence $E = e_1 e_2 ... e_n$, such that $e_i$ is the corresponding excode token of $c_i$ defined in Table I.

Since $C$ is the current partial code, to realize $\alpha$, we perform partial program analysis using PPA [5] to get token types and data types in a best-effort fashion.

Definition 4 (Excode token concretization function). The concretization function $\pi(c, V)$ on an excode token $c$ and the set $V$ of the accessible variables and fields of the current class of the method, defines the set of code tokens as follows:

$$\pi(c, V) = \begin{cases} \{v : v \in V, type(v) = type(c)\} & \text{if } c \text{ is a variable} \\ \{c\} & \text{otherwise} \end{cases}$$

where, $c$ is the respective non-variable token listed in Table I.

In Figure 1, $\pi(\text{VAR(NodeList)}, V) = \{\text{children}\}$, where $V$ contains the set of accessible (global/local) variables of the method listChildNodes. Note that, literals will not be concretized except if they are special literals, such as null or 0.

Definition 5 (Excode sequence concretization function). The sequence concretization function $\Pi(E, V)$ on an excode sequence of length $n$, $E_n = e_{1 ... e_n}$, in a method and the set of the method’s accessible variables and fields $V$, defines a set of code sequences of length $n$, in which each code sequence $C_n = e_1 e_2 ... e_n$, $c_i \in \pi(e_i, V)$, for $\forall i \in [1, n]$.

Definition 6 (Excode expression). In a method having the set of accessible variables $V$, an excode expression $expr$ is an excode sequence with one or more excode tokens, such that there is at least one code sequence $C$ in $\Pi(expr, V)$ that is a valid code expression according to the programming language.

In our example, the excode sequence $\text{VAR(NodeList)}$ $\text{OP(notEquals)}$ $\text{NULL}$ is an excode expression since there exists a concretization to obtain a valid expression $\text{children} = \text{null}$.

Definition 7 (Excode statement). In a method having the set of accessible variables $V$, an excode statement $stm$ is an excode sequence with one or more excodes, such that there is at least one code sequence $C$ in $\Pi(stm, V)$ that is a valid statement.

We use excodes to represent a statement template. For example, $\text{TYPE(int)}$ $\text{VAR(int)}$ $\text{OP(ASSIGN)}$ $\text{VAR(NodeList)}$ $\text{OP(ACC)}$ $\text{CALL(NodeList, getLength, 0, int)}$ $\text{LP}$ $\text{RP}$ is a template.

V. IDENTIFYING CANDIDATE TEMPLATES

Given the partial code $P$, AUTO-SC first parses $P$ to build the excode sequence $E = e_1 e_2 ... e_n$. It uses the $n$-gram model [16] that is trained on the excode sequences built from a code corpus to predict each excode one by one that most likely follows $E$. It also uses rules for program constraints to derive the valid candidates of excode tokens and sequences. The resulting excode sequences represent templates. Let us detail it.
Algorithm 1 Identifying Candidate Templates

1: function IDENTIFYTEMPLATES(partialCode, project)
2:   E = o(partialCode) \triangleright Def. 3
3:   genSeqs = expandExcodeSeq(E, project)
4:   templates = extractRemainingParts(genSeqs, E)
5:   return templates

6: function EXPANDEXCODESEQ(exSeq, proj)
7:   if isEnded(exSeq) \lor reachMaxLen(exSeq) then
8:     return (exSeq)
9:   C = getNextValidExcode(exSeq, proj) \triangleright Def. 10
10:  if C = \emptyset then return \emptyset
11:  topCands = rank(C, exSeq, \phi_{excode}, K) \triangleright Form. 1
12:  exSequences = \emptyset
13:  for all cand \in topCands do
14:     newSeq = concat(exSeq, cand)
15:     newTemps = expandExcodeSeq(newSeq, proj)
16:     exSequences.addsAll(newTemps)
17:   return exSequences

A. Training n-gram LM with excodes to predict next excode

To predict the next excode, any statistical LM is applicable [26], [17], [6], [25]. Without loss of generality, we use n-gram LM [16]. The model is trained on the excode sequences built from a corpus. For prediction, given E and an excode candidate \epsilon, the likelihood that \epsilon is the next excode token following E, is estimated using the trained n-gram LM, \phi_{excode}:

\[ P(\epsilon | E) = \phi_{excode}(e_1,e_2,\ldots,e_n,\epsilon) \]  (1)

B. Deriving the next excode sequence for statement template

Next, using \phi_{excode}, AUTO\text{SC} identifies the most likely valid excode one at a time, and then composes them to obtain the candidates for statement template. Specifically, Algorithm 1 shows how AUTO\text{SC} identifies candidate templates. In this algorithm, the partial code is first parsed into the corresponding excode sequence (line 2). The next sequences are suggested by expanding the excode sequence token-by-token until encountering the end-statement token “;” or the length of the expanded sequence reaches the pre-defined maximum length of code statements (lines 3, 7–9). For each expansion step, AUTO\text{SC} applies the syntax rules and accessibility rules (will be explained later) to enforce program constraints. The set of valid candidates of the next excode token is stored in \mathcal{C}. Then, it selects the top K (predefined value) most likely tokens (line 11). These excodes are concatenated with E to form new candidates that are recursively expanded (lines 13–16).

C. Enforcing syntax rules and accessibility rules to decide the candidates for the next excode token

A vocabulary \mathcal{Y} is a set of all distinct excode tokens. Since code has strict syntax and semantics, for excode sequence E, the valid next excode token following E is restricted by program constraints/rules: Syntax rules and Accessibility rules.

Definition 8 (Program syntax rule). Given the excode sequence \( E = e_1e_2\ldots e_n \), the vocabulary \mathcal{Y} of all excode tokens, a program syntax rule \( r_{syntax} \) when applying on \( E \) will return a set \( \mathcal{J} \) of excode tokens in the vocabulary such that the resulting excode sequence \( E' = e_1e_2\ldots e_n, e_{n+1} \) does not violate a syntax rule of a programming language. Mathematically, a program syntax rule \( r_{syntax} \) is a relation \( r : (\mathcal{Y})^* \rightarrow 2^\mathcal{J} \), \( r_{syntax}(E) = \mathcal{J} \), where \( \mathcal{J} \subseteq \mathcal{Y} \) is the set of tokens, such that \( \forall e \in \mathcal{J} \), \( E' = e_1e_2\ldots e_{n+1} \) does not violate a syntax rule.

For example, the code \textbf{int len = } has the excode sequence of \textbf{TYPE(int) VAR(int) OP(ASSIGN)}. The excode tokens OP(ASSIGN) and OP(ACC) are excluded from \( r_{syntax}(E) \) because an “=” or “.” cannot occur after the “=” sign. In this example, \( r_{syntax}(E) \) includes literal, variable, method call field access, data type, prefix operators, or open parenthesis. Note that to check for \epsilon, instead of checking all syntax rules on \( E' = e_1e_2\ldots e_n \) at each expansion step, for efficiency, we could check the validity of \epsilon based on the last token \( e_n \), and finally, check on the syntactic validity of entire sequence at the last step when the end of statement is reached.

Definition 9 (Accessibility rule). Given the excode sequence \( E = e_1e_2\ldots e_n \), the vocabulary \mathcal{Y} of all excode tokens, an accessibility rule \( r_{access} \) when applying on \( E \) will return a set \( \mathcal{A} \subseteq \mathcal{Y} \) of the excode tokens in the vocabulary that are accessible at the current state of \( E \). That is, \( r_{access} \) is a relation \( r_{access} : (\mathcal{Y})^* \rightarrow 2^\mathcal{A} \), \( r_{access}(E) = \mathcal{A} \) such that \( \mathcal{A} \) includes the excode tokens which correspond to the following cases:

1) All declared local variables within the current scope are valid. In Figure 1, accessible local variables are VAR(Node), VAR(NodeFilter), VAR(NodeListImpl) and VAR(NodeList).
2) All the accesses to the fields and the calls to the methods in the enclosing class are accessible.
3) The accessible field accesses and method calls of a variable. For example, for a sequence \( E \) ending with VAR(NodeList) OP(ACC), all accessible field accesses and method calls in NodeList are included in \( r_{access}(E) \).
4) All data types and literals are valid.
5) All keywords, separators, and operators are valid.

Definition 10 (Valid next excode token). For a sequence, a excode token is considered as valid if it satisfies all Syntax rules and Accessibility rules. That is, given an excode sequence \( E = e_1e_2\ldots e_n \), the set of valid candidates \( \mathcal{C} \) is \( r_{syntax}(E) \cap r_{access}(E) \) for all syntax rules and accessibility rules.

VI. VALIDATING CANDIDATE TEMPLATES

Fully semantic checking with respect to the current programming language (e.g., Java) is always desired. However, it is impossible to do so for the candidate templates, which are expressed as the sequences of excode tokens and do not contain concrete lexemes of variables. Because our design is to have excodes contain data type information, we focus on performing type checking. With type checking, we can eliminate a large number of templates with incorrect and inconsistent types.

In general, one could use a type checker for the current programming language, e.g., Java type checker. However, we are
Table II: Key Type Check Rules for Excode Sequences

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Type</th>
<th>Type Check (e; T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>T</td>
<td>e.g., LIT(String): String</td>
</tr>
<tr>
<td>Variable</td>
<td>T</td>
<td>e.g., VAR(int); int or VAR(Unk): Unk</td>
</tr>
<tr>
<td>Assignment</td>
<td>e1, e2</td>
<td>T1 ⊆ T2 if (T1 = 1 = Unk) and (T2 = 1 = Unk) else if T1:Unk T2:Unk T2 ≠ Unk Unk</td>
</tr>
<tr>
<td>Prefix op</td>
<td>e</td>
<td>if op = not and (T=boolean) or (T=Unk) num types include char, short, int, etc.</td>
</tr>
<tr>
<td>Postfix op</td>
<td>e</td>
<td>if (T = num) or (T=Unk) num types include char, short, int, etc.</td>
</tr>
<tr>
<td>Comparison</td>
<td>e</td>
<td>e1 = T1, e2 = T2</td>
</tr>
<tr>
<td>Infix</td>
<td>e</td>
<td>e1 = T1, e2 = T2</td>
</tr>
<tr>
<td>Method Call</td>
<td>e</td>
<td>decl: RT T::m (T1 p1, T2 p2, ..., Tn pn) if (T1 ⊆ T) or (T=Unk) for i = 1...n</td>
</tr>
<tr>
<td>Constructor Call</td>
<td>e</td>
<td>decl: T::m (T1 p1, T2 p2, ..., Tn pn) if (T1 ⊆ T) or (T=Unk) for i = 1...n</td>
</tr>
<tr>
<td>Field Access</td>
<td>e</td>
<td>T::m(S) or T::m(S1)</td>
</tr>
<tr>
<td>Statement</td>
<td></td>
<td>e1: T</td>
</tr>
<tr>
<td>Variable Decl</td>
<td></td>
<td>if (T ⊆ T) or (T=Unk)</td>
</tr>
<tr>
<td>ForStmt S:=e</td>
<td>void</td>
<td>t1: T1, ..., t3: Tn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e: T, T=boolean or T=Unk, m::T, T1: T</td>
</tr>
<tr>
<td>S:=while (e) S1</td>
<td>void</td>
<td>e: T, T=boolean or T=Unk, S1::T1</td>
</tr>
<tr>
<td>S:=if (e) S1</td>
<td>void</td>
<td>e: T, T=boolean or T=Unk, S1::T1, S2::T2</td>
</tr>
<tr>
<td>ExprStmt S:=e</td>
<td>void</td>
<td>t1: T1, ..., t3: Tn</td>
</tr>
<tr>
<td>Block S:=;e1...en</td>
<td>void</td>
<td>decl: RT T::m (T1 p1, T2 p2, ..., Tn pn) T ⊆ RT</td>
</tr>
<tr>
<td>Return S:=return e</td>
<td>T</td>
<td>e: T</td>
</tr>
</tbody>
</table>

dealing with partially complete code and there are potentially program entities whose types cannot be resolved by PPA [5]. In those cases, the variables without type information are annotated with Unknown type. Thus, we build a type checker for excode with the accommodation of the Unknown type.

AUTOVEC performs type inference at the same time as type checking on excode statements and expressions using the rules in Table II. The process of type checking is similar to type checking for the source code in Java. However, there are two key differences. First, it works at the excode statements/expressions corresponding to the statements/expressions at the source code level (note: variables’ names are not there). Second, due to unresolved types, AUTOVEC has to consider Unknown type in a flexible manner, e.g., that type does not violate any subtype constraint. Let us explain the key type-checking rules.

1. Literal. When seeing the excode LIT(T) that represents a literal with a type T, we consider T as its type.

2. Variable. When seeing a VAR, if the type of excode is available, we use it. Otherwise, the resulting type is Unknown.

3. Assignment. The LHS and RHS expressions are type-checked first. If both types are known, the type of RHS must be a subtype or equal to the type of LHS. If either of them are Unknown, we consider the assignment as valid with the known type. If both are Unknown, the resulting type is Unknown.

4. Prefix. If the operator is a negation and if the type of e is available, it must be boolean, otherwise, it must be convertible to a numeric type (char, short, int, etc.) The resulting type is boolean or a numeric type, accordingly. If the type of e is Unknown, the result depends only on the operator (Table II).

5. Postfix. The type of e must be convertible to a numeric type or it is unavailable. The resulting type is numeric.

6. Comparison. The type of one side must be a sub-type or equal to the type of the other side, or the type of at least one of them must be Unknown. The resulting type is boolean.

7. Infix. Both expressions on two sides need to be type-checked. If both types are not Unknown, the type of one side must be a subtype or equal to the other, and the expression is assigned with the super type. If the type of one of the two sides is Unknown, the expression is assigned of the type of the known one. Otherwise, the type of the expression is Unknown.

8. Method Call. The expressions for the receiver and the arguments need to be type-checked first. The type of each argument (if available) must be a subtype or equal to the type of the corresponding formal parameter in the declaration of the method. The return type is used as the type of the call.

9. Constructor Call. A constructor call is handled similarly as a method call except that the declared type is used and the method name is the same as the class name.

10. Field Access. The receiver needs to be type checked. The class of the field must be the same as the respective type stored in the excode.

11. Variable Declaration. The RHS expression (if any) needs to be type checked, and its type (if available) must be a subtype or equal to the type stored in the excode VAR(T).

12. For/While/If statement. The components in the excode of such a statement need to be type checked. The conditional control statement must be of the type boolean or Unknown.

13. Expression/Block Statement. Each statement in each of those compound statements needs to be type checked.

14. Return statement. The expression needs to be typechecked and its type must be a subtype or equal to the return type of the enclosing method.

Definition 11 (Type-correct candidate template). Given an excode sequence E representing the current partial code, the template T (as an excode sequence) is considered as a type-correct candidate template if the sequence concatenated by E and T is type checked by our rules.
In Figure 1, both candidates 0 and VAR(NodeList) OP(ACC) CALL(NodeList.getLength,0,int) LP RP are type-correct.

VII. CONCRETIZING STATEMENT TEMPLATES AND RANKING CODE CANDIDATES

This section describes how the type-correct candidate templates as execute sequences are converted to code candidate sequences with the accessible variables in the current scope. The most likely code sequences are ranked based on their occurrence likelihoods computed by an LM. Let us detail it.

Algorithm 2 Concretizing Candidate Template

```
1: function CONCRETIZE(templ, V)
2:    codeCands = concretizeNext(templ, V, ϕ, 1)
3:    return codeCands

4: function CONCRETIZE_NEXT(templ,V,currCands,i)
5:    if i > len(templ) then
6:        return currCands
7:    codeCands = ϕ
8:    codeTokens = π(templ[i],V)  // Def. 4
9:    if currCands = ϕ then
10:       for all t ∈ codeTokens do
11:           newCand = concat(EMPTY_SEQ,t)
12:           codeCands.adds(newCand)
13:    else
14:       for all t ∈ codeTokens do
15:           for all cand ∈ currCands do
16:               newCand = concat(cand,t)
17:               codeCands.adds(newCand)
18:       return ConcretizeNext(templ,V,codeCands,i+1)
```

Concretization. Algorithm 2 shows our procedure. Each execute token is converted into code tokens using function π (Def. 4). These tokens are used to initiate a set of code sequences (lines 9–12) or concatenated with the current concretized code sequences to create the new ones (lines 14–17). The process recursively until the end of the template.

Training an LM on lexical code and Ranking candidate statements. To rank the candidate code statements, we train an n-gram model, φ.lexemes, on the lexical forms of the source code in a corpus. For training, all source files are tokenized based on naming conventions (Camelcase and Hungarian), and the obtained tokens are normalized to lowercase. The trained LM is used to estimate the occurrence likelihoods of the code sequence that is concatenated from the current code and the candidate statement. That is, given the current code C, the likelihood of the candidate statement γ is: φ.lexemes(concat(C.lexemes,γ.lexemes)), where C.lexemes and γ.lexemes are the lexical forms of C and γ, respectively.

VIII. EMPIRICAL METHODOLOGY

We have conducted several experiments to empirically evaluate AutoSC in statement completion. For that, we seek to answer the following research questions:

RQ1: Accuracy and Comparison. How accurate is AutoSC in current statement completion and next statement suggestion? How is it compared with the state-of-the-art tool PCC [27]?

RQ2: Intrinsic Accuracy. How accurate is AutoSC in completing code statement on various factors including code sequences’ lengths and code token types?

RQ3: Sensitivity Analysis. How do various factors affect our model, e.g., completion position, thresholds, and data’s sizes?

RQ4: Time Complexity. What is our training/testing time?

A. Subject Systems

In this study, we collected the same data set of Java projects used in the existing studies in code completion [9], [20] (Table III) (Large Corpus). In the dataset, the average number of code tokens in a statement is 8.8, whereas more than 85% of the code statements contain less than 12 code tokens.

For comparison on next-statement (NS) suggestion, we also used the same dataset as in PCC [27] (Small Corpus in Table IV). In the training data, 90% of statements contain less than 12 tokens. The test dataset contains only 10 individual files without their projects. Both training and test data are much smaller than our Large Corpus.

In our experiments, to balance between the completion effectiveness and efficiency, we set the maximum number of tokens in a statement of 12.

B. Evaluation Setup, Procedure, and Metrics

We used the same setting with data across projects as in existing work [9], [25], [20]. That is, we divided the source files of a project into 10 equal folds. We performed 10-fold cross-validation: each fold was chosen for testing, while the remaining folds and other projects were used for training.

Accuracy on statement completion is measured as follows. For a method in a source file in the test data, our evaluation tool traverses its code sequentially from the beginning. At a position i in a method with a code sequence Mn = c1c2...cn, a tool computes the top k most likely code sequences, s1,
\( s_2, \ldots , s_k \), for the remaining of the current statement based on the previous code sequence from the start of the method to the position \((i - 1)\): \( c_1c_2 \ldots c_{i-1} \). If the actual code sequence, from \( i \) to the end of the current statement \( s_i \) at the position \( t \) is among the above \( k \) suggested sequences, we count this as a hit. The top-\( k \) accuracy is the ratio of the total hits over the number of tokens. Top-\( k \) accuracy for a project is computed on all positions of its methods in cross validation.

Note that, for the compound statements including if-then, if-then-else, switch, for, while, and do-while statements, we run a model to complete/suggest their control expressions. Moreover, at a local variable declaration statement, AUTO\( \text{SC} \) suggests a placeholder and consider it matching with the actual name, because any new name can be used at that point.

To compare AUTO\( \text{SC} \) with PCC [27] in statement completion (SC) on Large Corpus, we use the following SC setting that works for PCC, which is aimed to suggest the next statement only (Section X). At a position \( i \), the previous code sequence is divided into \( C_1 = c_1c_2 \ldots c_i \) and \( C_2 = c_{i+1}c_{i+2} \ldots c_{i-1} \), where \( t \) is the ending position of the nearest completed statement. \( C_1 \) is used as the input of PCC to suggest the next statement. We collected into the list of the resulting suggestions the top \( k \) most likely code statements from PCC that begin with \( C_2 \). Among the list, if there exists a statement that is the actual code sequence, we count this as a hit.

To compare AUTO\( \text{SC} \) with PCC [27] in next-statement (NS) suggestion on Small Corpus, we uses the same setting as in PCC [27]. That is, instead of traversing source code sequentially token by token, we ran AUTO\( \text{SC} \) and computed the top-\( k \) accuracy only at the beginning position of every code statement. However, in Small Corpus, the test data includes individual files without containing the corresponding projects’ files. Meanwhile, AUTO\( \text{SC} \) is designed using program analysis on the code of currently developing projects. Therefore, we created dummy projects for each of the testing files.

**IX. EMPIRICAL RESULTS**

**A. Accuracy Comparison (RQ1)**

1) **Comparative Results of SC on Large Corpus:** We compared AUTO\( \text{SC} \) with PCC [27], which applies a statement-level \( n \)-gram LM and searches for similar statements for next statement completion (will be detailed in Section X). We also compared AUTO\( \text{SC} \) against two baseline approaches: \( n \)-gram LM and \( n \)-gram+PA. For \( n \)-gram LM, we trained a \( n \)-gram LM and used it to predict the next token by token, and rank the candidates for the code sequence according their occurrence likelihoods. The \( n \)-gram+PA model works similarly to \( n \)-gram LM except that PA is additionally applied to filter out the invalid candidate sequences. Then, the valid ones is ranked. We used the 6-grams for both \( n \)-gram LM and \( n \)-gram+PA. In AUTO\( \text{SC} \)’s \( n \)-gram LMs, \( \phi_{\text{code}} \) and \( \hat{\phi}_{\text{lexemes}} \), \( n = 6 \). We did not compare with a model that solely uses PA since it generates a huge number of equally-ranked candidates.

As seen in Table V, the top-1 accuracy for AUTO\( \text{SC} \) is 39.8–41.3%. That is, up to 4 out of 10 requests, users could find their expected next code sequence for the current statement at the top of our ranked list. For PCC, the top-1 accuracy is from 0.6–4.3%, that is 9X–69X lower than AUTO\( \text{SC} \)’s. Meanwhile, lexical \( n \)-gram achieves only from 0.02–0.28%. Even when we used PA to filter out invalid suggestions, the top-1 accuracy is still very low, 0.02–0.34%, that is more than 100X lower than AUTO\( \text{SC} \)’s top-1 accuracy. For top-5 accuracy, AUTO\( \text{SC} \) also achieves up to 51.0%, which is 7–14X and more than 50X higher than PCC and both \( n \)-gram LM and \( n \)-gram+PA.

There are two key reasons for their low accuracy. First, code statement as its entirety is relatively project-specific [27]. Indeed, on average, the portion of the code statements in a project that can be found in others is only 3.2%. That leads to the low accuracy of PCC [27], which relies on the repetition of entire code statements. Second, for \( n \)-gram baselines, because the next sequence is suggested by predicting next token one at a time, the accuracy of next sequence suggestion is affected by the confounding effect of the accuracy of a single next-token suggestion. The highest top-1 accuracy of an \( n \)-gram LM for next code token suggestion is about 0.5 [17]. Therefore, for predicting a next code sequence containing 6 tokens (on average), the maximum top-1 accuracy is 0.5\(^6\) \approx 1.6\%.

Table V: Code Statement Completion Accuracy

<table>
<thead>
<tr>
<th>Project</th>
<th>Top-k</th>
<th>AUTO( \text{SC} ) ( n )-gram LM</th>
<th>AUTO( \text{SC} ) ( n )-gram LM+PA</th>
<th>PCC [27]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>Top 1</td>
<td>41.3%</td>
<td>0.09%</td>
<td>2.13%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>48.7%</td>
<td>0.33%</td>
<td>4.65%</td>
</tr>
<tr>
<td>Batik</td>
<td>Top 1</td>
<td>39.8%</td>
<td>0.28%</td>
<td>4.24%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>50.1%</td>
<td>0.78%</td>
<td>7.24%</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Top 1</td>
<td>40.5%</td>
<td>0.19%</td>
<td>5.50%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>49.3%</td>
<td>0.44%</td>
<td>5.89%</td>
</tr>
<tr>
<td>Log4J</td>
<td>Top 1</td>
<td>40.4%</td>
<td>0.02%</td>
<td>1.12%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>48.2%</td>
<td>0.10%</td>
<td>3.94%</td>
</tr>
<tr>
<td>Lucene</td>
<td>Top 1</td>
<td>39.3%</td>
<td>0.23%</td>
<td>3.87%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>49.5%</td>
<td>0.54%</td>
<td>6.60%</td>
</tr>
<tr>
<td>Maven-2</td>
<td>Top 1</td>
<td>40.2%</td>
<td>0.10%</td>
<td>0.58%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>51.0%</td>
<td>0.25%</td>
<td>3.32%</td>
</tr>
<tr>
<td>Maven-3</td>
<td>Top 1</td>
<td>38.9%</td>
<td>0.16%</td>
<td>1.35%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>48.7%</td>
<td>0.28%</td>
<td>4.86%</td>
</tr>
<tr>
<td>Xalan</td>
<td>Top 1</td>
<td>40.1%</td>
<td>0.17%</td>
<td>2.91%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>49.2%</td>
<td>0.41%</td>
<td>6.12%</td>
</tr>
<tr>
<td>Xerces</td>
<td>Top 1</td>
<td>39.7%</td>
<td>0.22%</td>
<td>2.35%</td>
</tr>
<tr>
<td></td>
<td>Top 5</td>
<td>49.9%</td>
<td>0.49%</td>
<td>6.31%</td>
</tr>
</tbody>
</table>

Note that, in this experiment, we used Large Corpus and the statement-completion (SC) setting that are different from Small Corpus and the next-statement (NS) setting used in PCC [27]. Thus, this leads to a different accuracy for PCC than the one reported in its paper [27].

2) **Comparative Results of next-statement (NS) suggestion on Small Corpus:** As seen in Table VI, AUTO\( \text{SC} \) does not perform next-statement suggestion as good as PCC. The main reason is that the test set contains individual Java files without the project-specific files and information such as the fields and methods of the classes. Thus, the components of AUTO\( \text{SC} \) relevant to program analysis, e.g., identifying the valid candidates for the next token, and type-checking, cannot be performed as expected.

3) **Analysis:** We analyzed the correct results and found that AUTO\( \text{SC} \)’s high accuracy can be attributed to the fol-
The application of PA to filter out the type-incorrect templates help AUTO\textsc{SC} achieve high accuracy. In the above example, the set of 137 valid candidates among 2,000 candidates that are learned from other places, are adapted to fit with the current context using PA. For example, AUTO\textsc{SC} concretized \texttt{VAR(ColumnFamilyStore) OP(ACC)...LP RP RP;} by using the accessible variable \texttt{cfs} for \texttt{encode} token \texttt{VAR(ColumnFamilyStore)} instead of \texttt{cfs} or filter as in other models. This adaptation ability to the current method with PA is the third reason for AUTO\textsc{SC}'s high accuracy.

Another reason for our high accuracy is that in AUTO\textsc{SC}, OOV is addressed by enforcing accessibility rules to avoid missing the valid file-specific or project-specific tokens when producing the candidate templates.

Finally, AUTO\textsc{SC} leverages the naturalness of source code in the lexical context, to effectively rank the candidate code sequences. In Figure 3, given a partial statement starting with \texttt{reports.addAll()}, where the type of \texttt{reports} is \texttt{List}, the expected sequence is \texttt{getReportExecutions();}. In fact, method \texttt{getReportExecutions} is declared inside the current class and has never been seen in the training data. This accessible call is still used to produce template. Since the type restriction for the argument of method \texttt{addAll} there are a few type-valid candidates, such as \texttt{getReportExecutions();} or \texttt{null};. Then, the candidate \texttt{getReportExecutions();} is ranked on the top by the lexeme-based LM, \texttt{\$latexcodes\$}, because the tokens \texttt{reports} and \texttt{Report} in \texttt{getReportExecutions} frequently go together, such as: \texttt{reports.contains(report)} and \texttt{reports.add(reportMojo)}.

We further studied the cases in which AUTO\textsc{SC} did not suggest well. We found that the majority of them are the cases whose the completion position is near the beginning of the current statement, especially the cases of suggesting entire statement (will be explained in Section IX-C1). Since the next-token prediction accuracy is not 100\%, the more next tokens predicted, the lower next-sequence completion accuracy.

B. Intrinsic Evaluation Results (RQ2)

We further studied the complexity and diversity, and AUTO\textsc{SC}'s effectiveness on different kinds of code tokens and different lengths of the statements completed by AUTO\textsc{SC}. We randomly sampled 10,000 results from 460K total results.

First, we classified the sampled results into 12 categories corresponding to the size (1–12 tokens) of the remaining code sequence of the currently completed statement (the maximum number of tokens to be completed is set to 12). Figure 4 shows the number of correct results over the total number of results for each category. As expected, the longer the remaining sequence, the more number of tokens to be completed, the less number of correct results. As seen, AUTO\textsc{SC} correctly handles complex completed statements with various lengths. Also, through the similar shapes of two types of columns from left to right, we see that the proportions of correct results over the total ones for all categories are quite uniform. Thus, our model is effective for various lengths of the remaining sequences, even for long sequences. A correct example in Maven-3 is as follows. The given partial code is the fragment \texttt{Activ activ = new Activ();}\_ The correct suggestion is \texttt{activ.setActiveByDefault(settingsActivation.isActiveByDefault());}, which has a total of 11 tokens after the cursor.

Second, to study the results by AUTO\textsc{SC} with respect to different kinds of tokens, we classified all the tokens in

| Table VI: Next-Statement Suggestion Accuracy |
|-----------------|-----------------|-----------------|-----------------|-----------------|
|                | Top 1 | Top 3 | Top 6 | Top 10 |
| AUTO\textsc{SC} | 20.3% | 28.5% | 32.0% | 42.2% |
| PCC [27]        | 28.9% | 51.1% | 54.8% | 59.3% |

Figure 2: A partial method in Cassandra

Figure 3: A partial method in Apache Maven

Figure 4: Accuracy on length of remaining code sequences
the sampled results into several categories corresponding to different syntactical token types. As seen in Figure 5, the proportions of the correct results over the total ones for all categories are relatively similar. Thus, AUTO SC is equally effective for diverse kinds of tokens. A correct example is a conditional expression, *null && file.isDirectory()* (containing a *null* literal, infix operators, identifiers, and separators.) for a partial condition of an *if-then* statement, if (*jarFile ==_-*

C. Sensitivity Results (RQ3)

1) Completion position: Because AUTO SC is based on the given code sequence, a completion point in the code sequence of a method $M_n = c_1c_2...c_n$ has impact on accuracy. Thus, we conducted an experiment to measure that. We first chose a random project, *Lucence*. For each method, we chose a completion point at three locations: 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 $l_3 = \lfloor 3n/4 \rfloor + 1$.

As seen in Table VII, accuracy slightly increases if we move the point to a later part of a method from $1^{st}$ to $3^{rd}$ quartile point. This is expected as AUTO SC has more information.

We also computed the accuracy as the completion points at the beginning of a new statement (i.e., next-statement suggestion). The percentage of the cases in which the next statement being correctly ranked on the top of the suggestion list is $8.2\%$ (top-1 accuracy, not shown). This is expected because any type of statement is valid at those beginning points. However, AUTO SC’s accuracy is still $3.9X$ better than top-1 accuracy in next-statement suggestion of PCC [27].

2) Threshold $K$: AUTO SC also relies on the pre-defined number $K$ of most likely tokens for next *excode* to identify templates. Figure 6 shows the accuracy and running time per completion request when we varied $K$. As seen, when $K$ is small, the accuracy are very low because the correct next *excode* token might be dropped out of top $K$. The accuracy increases when we keep a larger number of top candidates. However, with a large $K$, $K=9–10$, the number of the predicted code sequences is very large. This lead to a slower increasing trend as $K$ is larger. Regarding the running time, since the number of the predicted code sequences exponentially increases when we increase $K$, the running time for each request also grows exponentially when $K$ is larger.

3) Value of $n$ in the *n*-gram LMs: We also measured the impact of the size $n$ in the *n*-gram LMs $\phi_{excode}$ and $\phi_{lexemes}$, on AUTO SC’s accuracy. We varied $n$ for both $\phi_{excode}$ and $\phi_{lexemes}$ from 2–6 and computed top-1 accuracy when we ran AUTO SC on a randomly selected project.

As seen in Figure VIII, the accuracy grows from $29.7\%$ to $40.4\%$ for $n=2–6$. The reason is that the *n*-gram LMs with larger $n$ is able to capture more precisely the current context and rank better the correct next *excode* tokens (for $\phi_{excode}$) and the correct next code sequences (for $\phi_{lexemes}$). Meanwhile, the running time for each completion request increases linearly from 1,095ms to 5,447ms, because longer sequences need to be computed as $n$ is increased from 2 to 6.

4) Training data’s size: For training data’s size, we randomly selected a project, *Ant*, and divided its source files into 10 folds. We used one fold for testing and increased the sizes of the training data by adding into a dataset of 8 other projects one fold at a time until 9 remaining folds are added. Top-1 accuracy increases from $28.6\%$ to $41.3\%$ when we increase training data (Table IX). As expected, with larger training data sets, the model has observed more and performs better.

D. Time Complexity (RQ4)

All experiments were run on a Windows with 16 Intel Xeon 3.7GHz, 32GB RAM. AUTO SC took 10 minutes for training. The average running time for a request is 5.5s. On average, in the results in which the remaining code sequence is in top 5 of the ranked list, the average number of tokens in the remaining code sequences is 3.1 tokens. This equals the typing speed of

![Figure 5: Accuracy on various token types](image)

**Table VII: Impact of Completion Points on Accuracy**

<table>
<thead>
<tr>
<th>Location</th>
<th>Top 1</th>
<th>Top 2</th>
<th>Top 3</th>
<th>Top 4</th>
<th>Top 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st quartile</td>
<td>24.5%</td>
<td>30.1%</td>
<td>35.8%</td>
<td>39.2%</td>
<td>41.1%</td>
</tr>
<tr>
<td>2nd point</td>
<td>38.6%</td>
<td>43.0%</td>
<td>46.6%</td>
<td>48.9%</td>
<td>51.2%</td>
</tr>
<tr>
<td>3rd quartile</td>
<td>56.8%</td>
<td>58.6%</td>
<td>59.0%</td>
<td>60.6%</td>
<td>62.4%</td>
</tr>
</tbody>
</table>

![Figure 6: Impact of threshold $K$ on accuracy and running time](image)

**Table VIII: Impact of $n$ in $n$-gram LMs on Accuracy**

<table>
<thead>
<tr>
<th>$n$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top 1</td>
<td>29.7%</td>
<td>30.3%</td>
<td>31.3%</td>
<td>35.4%</td>
<td>40.4%</td>
</tr>
<tr>
<td>Running time</td>
<td>1095ms</td>
<td>2287ms</td>
<td>3169ms</td>
<td>4388ms</td>
<td>5447ms</td>
</tr>
</tbody>
</table>
Table IX: Impact of Training Data’s Size on Accuracy

<table>
<thead>
<tr>
<th>#Folds</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top 1</td>
<td>28.6%</td>
<td>31.4%</td>
<td>34.9%</td>
<td>37.8%</td>
<td>41.3%</td>
</tr>
<tr>
<td>Top 5</td>
<td>30.2%</td>
<td>35.4%</td>
<td>38.5%</td>
<td>42.0%</td>
<td>48.7%</td>
</tr>
</tbody>
</table>

about 0.56 tokens per sec, which is slightly slower than the average human typing speed, 0.68 tokens/sec [22].

E. Ineffective Cases

For incorrect cases, we classified them into the categories based on their number of code tokens that are in the remaining of the expected sequences. We found that the portion of the cases which contain redundant tokens is up to 23%. For the percentages of the cases of 1, 2 and 3 missed-tokens are 29%, 12% and 26%, respectively. Meanwhile, the portion of the cases of more than 3 missed-tokens is only 10%. For example, the correct one is `commits.get(readFrom);` (the suggested one is `commits.get(writeTo);`) for the given partial code `commits = _`. Thus, these results show that even for the ineffective cases, AUTOSC’s suggestion lists are still reasonable.

F. Threats to Validity

Our selected projects are not representative and different from PCC [27]’s dataset. However, we chose a high number of projects with large numbers of statements. For PCC, we used its default setting for the comparison. Our simulated code suggestion procedure is not true code editing. Inaccuracy is from the fact that AUTOSC cannot correctly resolve types/roles sometimes due to incomplete code.

X. RELATED WORK

AUTOSC is related to PCC by Yang et al. [27]. In comparison, there are fundamental differences between AUTOSC and PCC. First, PCC focuses on suggesting the next statement when a user finishes the previous statement, while AUTOSC supports both filling a partially typed statement (SC) and generating a next statement (NS). PCC can be used to support statement completion when the partially typed statement is matched against the suggested statement s, and the remaining tokens of s will be recommended for users. Second, while PCC is based solely on statistical LM, AUTOSC combines PA and LM. Third, the way PCC used an LM is also different. PCC combines all lexical tokens belonging to a statement into a pseudo-token called IR, for the statement. In training, it converts source code into sequences of IRs and trains a n-gram model to learn to recommend an entire statement. Because the entire statements do not repeat often, PCC has to consider similar IRs as the same, causing inaccuracy. AUTOSC uses LM+PA to predict token by token and compose them. We showed that AUTOSC outperforms PCC in both SC and NS.

There exists a rich literature of approaches on CC. The approaches can be broadly classified into the following categories. The first category relies on program analysis. IDEs support the completion of method calls/field accesses. Eclipse [7] and IntelliJ IDEA [13], [12] also support template-based completion for common constructs and APIs (for/while, iterator).

The second category uses code pattern mining [3], [8], [10], [11], [14], [19], [21], [24], [28], [29]. Grapacc [19] uses API patterns to match them against the current code. Bruch et al. [3] suggest a call based on frequent methods, co-occurrent calls, and best matching and their calling structures.

The third category relies on statistical LMs [15]. Hindle et al. [9] use n-gram on lexical tokens to predict the next token. Later, Tu et al. [25] improve n-gram model with caching for recently seen tokens. Raychev et al. [23] use n-gram to predict API call. SLAMC [20] associates code tokens with sememes, including token roles and data types. In comparison, there are key differences. First, `excode` is designed for template statements while sememes are abstractions over source code to predict the next token. Second, AUTOSC has a type checker for `excode` with `Unknown` type, while sememes do not have it. Third, n-gram topic model is used in sememes to provide the context for prediction, while AUTOSC uses PA+LM. Finally, SLAMC suggests only the next token. GraLan [18] is a graph-based LM that captures usage patterns to suggest API calls.

Recent advances in deep learning have been used in next token suggestion. White et al. [26] use Recurrent Neural Network (RNN) to learn the context to predict the next token, while Dam et al. [6] rely on LSTM. DNN4C incorporates syntactic information for better prediction using DNN LM [17].

Despite the success of using statistical LMs, those existing approaches are still limited to support only next token. They do not combine LM with PA as in AUTOSC.

XI. CONCLUSION

We introduce AUTOSC [1], which combines PA and the principle of software naturalness complete partial statements. We aim to benefit from the strengths of both directions. AUTOSC is trained on a code corpus to learn the candidate templates. Then, it uses PA to validate and concretize the templates into valid code statements. Finally, they are ranked by using a LM trained on the lexical form of the source code.

We conducted several experiments to evaluate AUTOSC in statement completion and next-statement suggestion on datasets with +460K statements with a total of +1M suggestion points. Our results show that AUTOSC is very effective with top-1 accuracy of 40% and top-5 accuracy of 49.4% on average. That is, in 4 out of 10 cases, when a user requests to complete his/her currently-written statement, (s)he can find the remaining of the desired statement in the top of the suggestion list. Importantly, AUTOSC significantly improves over the baseline model using only n-gram on lexical code (up to 142X in top-1 accuracy) and the model using lexical n-gram+PA (up to 117X in top-1 accuracy). It also improves over the state-of-the-art tool PCC [27] with 69X higher in top-1 accuracy.

ACKNOWLEDGMENT

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

[22] Ratatype. Average typing speed.