A Statistical Semantic Language Model for Source Code

Tung Thanh Nguyen
tung@iastate.edu

Anh Tuan Nguyen
anhnt@iastate.edu

Hoan Anh Nguyen
hoan@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University
Ames, IA 50011, USA

ABSTRACT

Recent research has successfully applied the statistical n-gram language model to show that source code exhibits a good level of repetition. The n-gram model is shown to have good predictability in supporting code suggestion and completion. However, the state-of-the-art n-gram approach to capture source code regularities/patterns is based only on the lexical information in a local context of the code units. To improve predictability, we introduce SLAMC, a novel statistical semantic language model for source code. It incorporates semantic information into code tokens and models the regularities/patterns of such semantic annotations, called *sememes*, rather than their lexemes. It combines the local context in semantic n-grams with the global technical concerns/functionality into an n-gram topic model, together with pairwise associations of program elements. Based on SLAMC, we developed a new code suggestion method, which is empirically evaluated on several projects to have relatively 18–68% higher accuracy than the state-of-the-art approach.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Documentation, Experimentation, Measurement

Keywords

Statistical Semantic Language Model, Code Completion

1. INTRODUCTION

Previous research has shown that source code in programming languages exhibits a good level of repetition [5, 8]. Studying 420 million LOCs in 6,000 software projects in SourceForge, Gabel *et al.* [5] reported *syntactic redundancy* at the levels of granularity from 6–40 tokens. For example, a for loop such as "for (int i = 0; i < n; i++)" or a printing statement "System.out.println(...)" occur frequently in many source files. Hindle *et al.* [8] found that such code regularities/patterns can be captured by the n-gram statistical language model [15] via training on existing codebases. The model is then leveraged to support code suggestion and completion [1].

The state-of-the-art statistical n-gram language model for capturing such code repetitions/patterns and code suggestion relies on the *lexical* information and *local context* of code tokens [8]. Lexical analysis is performed on source code to break it into tokens. The sequences of the tokens, called *n-grams*, are collected with different sizes. For a token, only its textual representation, called *lexeme*, is extracted. The n-grams with high occurrence counts correspond to highly frequent code, called *code regularities/patterns*.

Using only lexical information, the n-gram model focuses on capturing code patterns at the lexical level. However, source code written in programming languages has well-defined semantics. Programming patterns at the higher levels of abstraction would be useful for code suggestion/completion as well. For example, let us consider two simple statements "int len = str.length()" and "int l = s.length()", when len and l are of the same type int, and str and s are of the same type String. Both of them are the instances of the same pattern of *getting the length of a String object and assigning it to an int variable*. It could not be captured at the lexical level due to the differences of lexemes (e.g. str versus s, len versus l).

Furthermore, such lexical n-grams can provide only the local context. However, several programming regularities/patterns might involve program elements that scatter apart and cannot be captured within n-grams with reasonable sizes. The first kind of such patterns includes the *pairs of program tokens that are required to occur together* due to the *syntactic rules* of a programming language (e.g. the pair of try/catch in Java) or due to the *usage specification* of a software library (e.g. lock and unlock in the mutual exclusion library). Let us call it *pairwise association* among tokens.

The second kind of such patterns involves *multiple co-occurring tokens* that often come together to realize the *same technical functionality/concerns*. The API elements such as methods and data types that are used to implement certain functionality/concerns will appear together more frequently in the files related to those concerns. For example, in a source file relevant to *file I/O* functionality, the related APIs such as File, fopen, fread, etc would be more likely to occur

[1]Code completion refers to completing a partially typed-in token. Code suggestion means the suggestion of a complete code token following a code portion [8].

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491458

than the APIs for other concerns. Moreover, with software modularity, a source file often involves a few technical functionality. Thus, knowing the technical concerns of a source file could benefit for the prediction of the next token.

In this paper, we introduce SLAMC, a novel statistical Semantic LAnguage Model for source Code that incorporates semantic information into code tokens and models the regularities on their semantic annotations (called *sememes*), rather than their lexical values (lexemes). A token is annotated with its data type and semantic role if available. For example, the token str is semantically annotated as the sememe VAR[String], denoting it to have the role of a *variable* and String data type. Scopes and dependencies among tokens are also used. In addition to the local context of code tokens, we also consider the global technical concerns of the source files and the pairwise associations of code tokens. We combine n-gram modeling and topic modeling into a novel *n-gram topic model* to capture the influence of both local context and global concerns on the next token's occurrence.

Based on SLAMC's ability to suggest next tokens, we developed a new code suggestion engine that is configurable for Java or C#. Unlike traditional code completion with template code, our engine suggests a ranked list of *sequences of tokens* that would complete the current code to *form a meaningful code unit* and *most likely appear next*. Meaningful code units are defined based on the language, and appearance likelihood is computed based on the generating probabilities of the sequences. The top-ranked sequences are unparsed into lexemes and suggested. Our empirical evaluation on several subject systems shows that our code suggestion engine improves from 18–68% accuracy over the state-of-the-art lexical n-gram approach. Our key contributions are

1. SLAMC, a novel statistical semantic language model for source code with the integration of semantic n-grams, global concerns, and pairwise association (Section 3);

2. A code suggestion engine based on SLAMC (Section 4);

3. An empirical evaluation on its accuracy and comparison to the state-of-the-art lexical n-gram model (Section 5).

2. BACKGROUND

Statistical language models are used to capture the regularities/patterns in natural languages by assigning occurrence probabilities to linguistic units such as words, phrases, sentences, and documents [15]. Since a linguistic unit is represented as a sequence of one or more basic symbols, language modeling is performed via computing the probability of such sequences. To do that, a modeling approach assumes that a sequence is generated by an imaginary (often stochastic) process of the corresponding language model. Formally:

DEFINITION 1 (LANGUAGE MODEL). *A language model L is a statistical, generative model defined via three components: a vocabulary V of basic units, a generative process G, and a likelihood function $P(.|L)$. $P(s|L)$ is the probability that a sequence s of the elements in V is "generated" by the language model L following the process G.*

When the context of discussion is clear regarding the language model L, we use $P(s)$ to denote $P(s|L)$ and call it *the generating probability of sequence s*. Thus, a language model could be simply considered to have a probability distribution of every possible sequence. It could be estimated (i.e. trained) from a given collection of sequences (called a *training corpus*).

Table 1: Lexical Code Tokens from "len = str.length();"

Lexeme	Token Type
len	Identifier
=	Equal (Symbol)
str	Identifier
.	Period (Symbol)
length	Identifier
(	Left parenthesis (Symbol)
)	Right parenthesis (Symbol)
;	Semicolon (Symbol)

2.1 Lexical Code Tokens and Sequences

Statistical language models have been applied to software engineering, such as in code suggestion/completion [8]. To apply such a model to source code, one first needs to define the vocabulary, i.e. the collection of basic units (also called *terms* or *words*) that are used to make up a sequence. A vocabulary can be constructed via performing lexical analysis on source code (as a sequence of characters), i.e. breaking it into *code tokens* based on the specification of the programming language. The lexemes (lexical values) of the tokens are then collected as the basic units in the vocabulary. The input source code is represented as a sequence of lexical code tokens, which is called *lexical code sequence*. Formally:

DEFINITION 2 (LEXICAL CODE TOKEN). *A* lexical code token *is a unit in the textual representation of source code and associated with a* lexical token type *including identifier, keyword, or symbol, specified by the programming language.*

DEFINITION 3 (LEXEME). *The* lexeme *of a token is a sequence of characters representing its lexical value.*

DEFINITION 4 (LEXICAL CODE SEQUENCE). *A* lexical code sequence *is a sequence of consecutive code tokens representing a portion of source code.*

For example, after lexical analysis, the piece of code "len = str.length();" is represented by a lexical code sequence of eight tokens, with their token types and lexemes shown in Table 1. len, str, and length are three Identifier tokens, while the other tokens have different types of symbols. In lexical analysis, no semantic information (e.g. data type) is available. For example, str is not recognized as a String variable, and length is not recognized as the name of a method in the String class.

2.2 Lexical N-gram Model for Source Code

A n-gram model is a language model with two assumptions. First, it assumes that a sequence could be generated from left to right. Second, the generating probability of a word in that sequence is dependent only on its *local context, i.e. a window of previously generated words* (a special case of Markov assumption). Such dependencies are modeled based on the occurrences of word sequences with limited lengths. A sequence of n words is called a n-gram. When n is fixed at 1, 2, or 3, the model is called unigram, bigram, or trigram.

DEFINITION 5 (LEXICAL N-GRAM). *The lexeme of a sequence of n consecutive code tokens is called a lexical n-gram. It is defined as the sequence of the lexemes of those tokens.*

Those assumptions are reasonable for source code. That is, the next code token could be predictable/dependent on

the previously written code tokens [8]. For example, in a source file, the code sequence "for (int i = 0; i <n; " is considered as the local context of the next token. This piece of code could be recognized as a for loop with i as the iterative variable, and thus, in many cases, the next code token is i.

With the assumption of generating tokens from left to right, the generating probability of code sequence $s = s_1 s_2 ... s_m$ is computed as

$$P(s) = P(s_1).P(s_2|s_1).P(s_3|s_1 s_2)....P(s_m|s_1...s_{m-1})$$

That is, the generating probability of a code sequence is computed via that of each of its tokens. Thus, a language model needs to compute all possible conditional probabilities $P(c|p)$ where c is a code token and p is a code sequence. With Markov assumption, the conditional probability $P(c|p)$ is computed as $P(c|p) = P(c|l)$ in which l is a subsequence made of the last $n-1$ code tokens of p. With this approximation, a model only needs to compute and store the conditional probabilities involving at most n consecutive code tokens. $P(c|l)$ is often estimated as:

$$P(c|l) \simeq \frac{\text{count}(\text{lex}(l,c)) + \alpha}{\text{count}(\text{lex}(l)) + V.\alpha}$$

lex is the function that builds the lexeme of a code sequence. For instance, the code sequence i < n might occur in several places, e.g. in a for or if statement. The same lexeme sequence (i lparen n) would be created for them, and they are all counted as the occurrences of the same code sequence. α is a smoothing value for the cases of small counting values.

2.3 Discussions and Motivation

The lexical n-gram model has been shown to capture well code regularities/patterns at the lexical level to support code completion/suggestion [8]. Code patterns at higher levels of abstraction could be also useful for that task. However, such patterns with well-defined semantics could not be captured well with the lexical n-gram model. Moreover, n-grams provide only the *local context* for code suggestion, while other influence factors such as the global context of the source files could be also useful to predict the next token. In this work, we aim to address those by 1) adding semantic information, and 2) adding global influence factors in the model.

2.3.1 Motivation on Adding Semantic Information

Let us consider the following statement "len = str.length();". The lexical n-gram model represents it as a lexical sequence (Table 1). However, an editor with semantic analysis capability will recognize it as an *assignment statement*, with the left-hand side being a *variable*, and the right-hand side being an *expression*, which in turn contains a *method call* to a method named length. If the code under editing is sufficiently complete, further semantic analysis such as typing and scoping will help identify the data types, e.g. len being of the type int and str being of the type String. Semantic analysis will also help verify the *applicability* of the method call to length on the variable str, and *type compatibility* of the assignment of the returned value from length to the variable len.

A language model could benefit from such semantic information to detect the pattern *"getting the length of a String object and assigning it to an int variable"*. Without semantic information, it is challenging for a language model to detect that pattern if the variable names are different in different places. Moreover, the pattern could then help in code suggestion. For example, assume that the statement was incomplete as "len = str." and code completion is requested. If the above semantic information is available, a model could determine that a method of a String object is sought and the returned value will be assigned to an int variable. Thus, the method length would be a candidate for the next suggested token. In brief, using semantic information, a language model would capture better the code patterns at higher abstraction levels, thus, produce better code suggestion.

2.3.2 Motivation for Adding Other Influence Factors

During programming, the next code token could be chosen based on not only the *local context*, but also the broader factors of source code. The first factor is the system-wise, global *technical concerns/functionality*. For example, if the current source file involves the *file I/O* functionality, the API functions related to I/O operations such as fopen, fread, fwrite, and fclose would be more likely to occur than the ones related to other concerns, such as *graphics* or *database*.

Another factor is the *pair-wise association* of program elements. In source code, some program elements often go together, due to the syntax specification of the programming language or the usage specification of the APIs in libraries. For example, the pairs of API functions such as lock/unlock, and fopen/fclose often co-occur. Thus, the occurrence of one token would likely suggest that of the other. Pairwise association complements to the local context factor in n-gram. The rationale is that two associated tokens might locate so far apart that the local context in n-grams cannot capture their association. For example, there are often many code tokens in-between the pair fopen and fclose.

Combining these factors could help detect local and global trends/patterns and recommend better the next code tokens. For example, the local context could suggest that, in the currently editing code, there is likely a function call after an IF token. Then, if the current concern is about file I/O, the functions feof or fread would be the better candidates since they relate to file I/O and appear frequently after an IF token. Moreover, if fopen appeared previously, the pairwise association could suggest fclose to be the next token.

3. SEMANTIC LANGUAGE MODEL

Let us present SLAMC, a statistical Semantic LAnguage Model designed for source Code. SLAMC encodes semantic information of code tokens into basic semantic units, and captures their regularities/patterns. It also combines local context with global concern information as well as the pairwise association of tokens in the modeling process.

3.1 Overview and Design Strategies

Let us first explain our design strategies in selecting the kinds of semantic information to be incorporated into our model. The first semantic information is the **role** of a token in a program with respect to the written programming language, i.e., whether it represents a variable, data type, operator, function call, field, keyword, etc. With that, SLAMC will be able to learn the syntactical regularities/patterns such as *"after a variable, there is often an assignment operator"*. Second, it is useful to include the **data types** of the tokens, especially the types of variables, fields, and literals. Data types would help us capture both syntactical patterns and the patterns at higher abstraction levels, e.g. *"the parameter of function System.out.println is often a String or Inte-*

Table 2: Construction Rules for Sememes of Semantic Code Tokens

Token Role	Construction Rule	Example
Data type T	TYPE[T]	String → TYPE[String]
Variable x	VAR[typeof(x)]	str (String) → VAR[String]
Literal v	LIT[typeof(v)]	"Java" → LIT[String]
Function decl m	FUNC[type(m),lexeme(m),paralist(m),rettype(m)]	indexOf → FUNC[String,indexOf,PARA[String],Integer]
Function call m	CALL[type(m),lexeme(m),paracount(m),rettype(m)]	length → CALL[String,length,0,Integer]
Parameter x	PARA[typeof(x)]	str (String) → PARA[String]
Field f	FIELD[type(f), lexeme(f)]	left → FIELD[Node,left]
Operator o	OP[name(o)]	= → OP[assign], . → OP[access]
Cast (T)	CAST[T]	(Integer) → CAST[Integer]
Keyword	*To corresponding reserved token*	if → IF, class → CLASS
Block open & close	*To corresponding reserved token*	} (of a for loop) → FOREND
Special literal	*To corresponding reserved token*	"" → EMPTY, null → NULL
Unknown	*To special lexical token* LEX	abc → LEX[abc]

ger literal". For a method, the **return type**, the declared **parameter list**, and the **number of passing parameters** are important to identify and characterize the method. Thus, for a method, we incorporate its signature, including its name, class name, return type, and parameter list.

Those kinds of information are encoded as the *semantic values* of the code tokens, which we call **sememes** (will be formally defined later). The sememes are included in the vocabulary of our model and used to construct the n-gram sequences and associated pairs in the modeling process. To extract meaningful sequences and pairs, SLAMC uses the **scopes** and **dependencies** of code tokens. That is, it considers only the associated pairs of the code tokens that have dependencies and in the proper scopes. For example, only the pairs of function calls having data dependencies are considered. Moreover, it considers only the sequences that are in appropriately structural scopes. For example, the sequences spanning across block, function or class boundaries are excluded. Let us formally describe the concepts.

3.2 Semantic Code Tokens and Sequences

3.2.1 Semantic Code Tokens

Definition 6 (Semantic Code Token). *A semantic code token is a lexical code token with associated semantic information including its ID, role, data type, sememe, scope, and structural and data dependencies.*

Definition 7 (Role). *The* role *of a semantic code token refers to the role of the token in a program with respect to a programming language. The typical token roles include type, variable, literal, operator, keyword, function call, function declaration, field, and class.*

For example, in "str.length()", after semantic analysis, str is recognized as a semantic code token with its role of a variable, while the role of length is a function call.

Definition 8 (Sememe). *The* sememe *of a semantic code token is a structured annotation representing its semantic value/information, including its token role and data type.*

Definition 9 (Vocabulary). *A* vocabulary *is a collection of distinct sememes of all semantic code tokens.*

Table 2 lists the construction rules to build the sememes for the popular types of semantic code tokens. For example, length in str.length() has the semantic role of a function call, its sememe consists of the annotation "CALL", "[", its class name String, its name length, the number of passing parameters(0), the returned type Integer, and "]" as shown in the fifth row. That sememe represents the semantic value of that semantic code token, i.e. a method call to length. The separator tokens, e.g. semicolons and parentheses, are not associated with semantic information, thus are excluded.

Semantic information might be unavailable, e.g. when the current code is incomplete, leading to no typing information or un-deciding whether an identifier is a variable, data type, or a method name. In such cases, the lexical token is kept and annotated with the sememe of type LEX (the last row).

For a variable, its sememe does not include its name, e.g. the variable str is encoded as VAR[String] to denote it as a String variable. This allows us to capture more general code patterns involving variables because variables' names are often individuals' choices and the naming convention might be even different across projects. For example, two statements "len = str.length()" and "l = s.length()" express the same code pattern when l and len are of type int, and s and str are of type String, although the variables' names are different (e.g. len versus l, and str versus s). To capture that pattern and improve predictability, SLAMC represents those statements by two code sequences with the same sememe sequence:

VAR[Integer] OP[assign] VAR[String] OP[access] CALL[String, length, 0, Integer]

Similarly, the concrete literals' values could vary in concrete usages. For example, the pattern of printing a string could be instantiated with different string literals in different usages (e.g. System.out.println("Hello World!"), or System.out.println("File not found!")). To capture code patterns with higher abstraction levels and enhance predictability for code suggestion, SLAMC annotates the sememe of a literal with its data type rather than its lexeme. Thus, those two printing statements will have the same sememe sequence.

In other cases, programming patterns could involve special literal values. For example, many functions use a 0 (zero) as the returned value indicating a successful execution. Objects are frequently checked for *nullity* before being processed, for example "if (node != null)". Thus, SLAMC has also special sememes representing such values (including null, zero, and empty string). For instance, the expression "if (node != null)" is captured as the sequence IF VAR[Node] OP[neq] NULL.

Definition 10 (Scope). *A* scope *associated with a semantic code token identifies the block containing that token.*

Table 3: Associated Information of Semantic Code Tokens for len=str.length()

ID	Role	Sememe	Lexeme	Scope	Depend
T1	Variable	VAR[int]	len	C1.M2.B3	[T3,T5]
T2	Operator	OP[assign]	=	C1.M2.B3	NA
T3	Variable	VAR[String]	str	C1.M2.B3	[T1,T5]
T4	Operator	OP[access]	.	C1.M2.B3	NA
T5	Method	CALL[String,...]	length	C1.M2.B3	[T1,T3]

For a program, a scope is modeled by a sequence of blocks' identifiers in its abstract syntax tree (AST). For example, the scope C1.M2.B3 identifies the third block in the second method of the first class in the current source file.

DEFINITION 11 (DEPENDENCY). *The* dependency *set of a semantic code token t is a set of IDs of the other code tokens that have structural or data dependencies with t.*

Structural dependencies are defined as child-parent relations in an AST. Data dependencies are defined among program elements and currently computed via data analysis on variables. Table 3 illustrates the semantic code tokens for the example "len = str.length();". The variable len (with the ID of T1) depends on the String variable str (T3), and the method call length (T5), thus its dependency set is [T3, T5].

Note that lexemes of semantic code tokens are used in code suggestion (Section 4). The patterns with sememes only suggest the token role and data type of the next token (e.g. a variable of type String). SLAMC needs to find the most suitable semantic token and use the lexeme to fill in the code.

3.2.2 Semantic Code Sequences

DEFINITION 12 (SEMANTIC CODE SEQUENCE). *A* semantic code sequence *is a sequence of semantic code tokens.*

DEFINITION 13 (SEMANTIC N-GRAM). *The sememe of a semantic code sequence of size n, called a* semantic *n*-gram, *is the sequence of the sememes of the corresponding tokens.*

For example, SLAMC represents the piece of code "if (node != null)" as the semantic code sequence of four semantic tokens (keyword if, variable node, operator !=, and special literal null). The sememe of this sequence, computed from those of its tokens, is a semantic 4-gram IF VAR[Node] OP[neq] NULL. For brevity, we will use the terms *code token*, *code sequence*, and *n-gram* to refer to the semantic counterparts.

3.3 N-gram Topic Model

Prior research [1, 17] shows that the latent topics recovered by topic modeling on source files correspond well to the technical concerns in a system. Thus, if the topics of the code sequences are recovered, they could provide a global view on the current concern/functionality of the code, thus, could help in predicting the next token. Inspired by topic modeling by Wallach [20], we have developed an *n*-gram topic model that integrates the information of both local contexts (via *n*-grams) and global concerns (via topics). The key idea is that the probability that a token c appears at a position is estimated simultaneously based on the global information k and the local sequence of $n-1$ previous tokens.

Our model assumes a codebase to have K topics (corresponding to its concerns/functionality). Since a source file might involve several concerns, SLAMC allows a code sequence to contain all K topics with often different percentages (some might potentially be zero as well). It represents the topics of a code sequence p as a multinomial distribution θ sampled from the Dirichlet distribution $\text{Dir}(\alpha, K)$. θ is called *topic proportion* of p and θ_k is the proportion of topic k in p. The proportion of topic k could be measured via the ratio of the number of tokens on topic k over the total number of tokens. For example, a code sequence representing a source file could have 40% of its tokens about I/O, 50% about string processing, and 10% on GUI. Each token in a code sequence is assigned with a topic.

In SLAMC, *the generating probability of a code token c is dependent on its topic assignment k as well as on its local context l.* This dependency is modeled by a multinomial distribution $\phi_{k,l}$ (called *token distribution*), which is a sample of the Dirichlet distribution $\text{Dir}(\beta, V)$.

Then, to compute the probability $P(c|p)$ for any given code token c and code sequence p, we need to do two tasks: 1) training, i.e. estimating the multinomial distribution $\phi_{k,l}$ for all possible topic k and local context l from a training codebase, and 2) predicting, i.e. estimating the multinomial distribution θ for p to compute $P(c|p)$. We have developed two algorithms for those two tasks based on Gibbs sampling.

```
1  function Train(B, α, β, K, N, N_t)
2      for each source file f in training codebase B
3          extract its semantic code sequence s
4          collect available sememes into V
5          randomly initiate its θ, z
6      loop N_t times
7          for each available topic k and n−gram l
8              for each token c ∈ V
9                  φ_{k,l}(c) = (count(l,c,k)+β) / (count(l,k)+KVβ)
10         for each code sequence s in B
11             [θ, z] = Estimate(s, φ)
12     return φ
13
14 function Estimate(s, φ)
15     repeat
16         for each position i in s
17             sample z_i where P(z_i = k) = θ_k · φ_{k,s_{n,i}}(s_i)
18         for each topic k
19             θ_k = (count(z_i=k)+α) / (length(s)+Kα)
20     until θ is stable
21     return θ, z
```

Figure 1: N-gram Topic Model Training/Predicting

3.3.1 Training N-gram Topic Model

Figure 1 illustrates our training algorithm. The input includes a codebase B, containing a collection of source files, and other pre-defined parameters, such as the number of topics K, hyper-parameters of Dirichlet distributions α and β, the maximal size of n-grams N, and the number of training iterations N_t. The output includes the vocabulary V containing all collected code tokens, the token distributions $\phi_{k,l}$ for every topic k and every possible n-gram l. In addition, for each source file represented as a code sequence s, the output also includes its topic proportion θ and the topic assignment z_i for every position i in s.

The training algorithm first parses all source files in the codebase, and builds a semantic code sequence for each of them. It collects all code tokens into the vocabulary V and randomly initiates all latent variables (e.g. θ, ϕ, z) (lines 2-5). Then, it performs two-phase processing as follows.

Phase 1. SLAMC uses the existing topic assignments of all sequences (variables z) (or randomly initiates for the first iteration) to estimate the token distributions (i.e. $\phi_{k,l}$ for every possible topic k and n-gram l). They are estimated as in line 9: $\phi_{k,l}(c) = \frac{\text{count}(l,c,k)+\beta}{\text{count}(l,k)+KV\beta}$.

In this formula, function $\mathsf{count}(l, c, k)$ counts every position i in every sequence s where $s_i = c, s_{n,i} = l$ and $z_i = k$, i.e. the token at position i is c and is assigned to topic k, and $n-1$ previous tokens make up the sequence l. Similarly, $\mathsf{count}(l, k)$ counts such positions but does not require $s_i = c$. The positive parameter β is added to all the counts for the smoothing purpose for the computation in later iterations.

Phase 2. SLAMC uses the estimated token distributions (ϕ variables) to estimate the topic proportion θ and topic assignment z for every code sequence s (each for a source file) in the codebase (lines 11,14-21). First, a topic is sampled and assigned for each position i in s. The probability that topic k is assigned to position i is computed as in line 17:

$$P(z_i = k|s, \theta, \phi) \sim \theta_k \cdot \phi_{k,s_{n,i}}(s_i)$$

where s_i is the token of s at position i and $s_{i,n}$ is the sequence in s of $n-1$ tokens before i. Once topics are assigned for all positions (i.e. z_i is sampled for every i), the topic proportion θ is re-estimated as line 19: $\theta_k \simeq \frac{\text{count}(z_i=k)+\alpha}{\text{length}(s)+K\alpha}$.

That means, SLAMC counts the number of tokens assigned to topic k, and approximately estimates the proportion of topic k by the ratio of the number of tokens with topic k over the length of sequence s. The positive parameter α is added to all the counts for the smoothing purpose.

This sampling and re-estimating process is repeated on each sequence until the topic proportion θ is stable (i.e. converged, line 20). When every sequence in the codebase has a stable topic proportion, the algorithm goes back to phase 1. It stops when the latent variables θ and ϕ are stable or the number of iterations reach the maximum number N_t.

Representation and Storage. To save storage costs and improve running time, SLAMC does not directly store the token distributions $\phi_{k,l}$. It instead stores all n-grams and their counts in a tree. Each tree node has the following fields: a pointer to its parent, a sememe in the vocabulary as its label c, a counting vector φ of size K for the counts, and the total count σ. The root node is an empty node. The path from a node to the root corresponds to an n-gram. Let us use l to denote the n-gram from the parent node b of node c to the root. The value φ_k is equal to $\mathsf{count}(l, c, k)$. $\mathsf{count}(l, k)$ is computed by summing over φ_k in all children nodes of b.

This tree is created when the training algorithm constructs the semantic code sequences. When a new semantic code token c is built, the algorithm extracts all possible n-grams l that end right before c (n varies from 1 to $N-1$). Then, it traverses the tree to find the path that corresponds to each n-gram l. If the last node of that path does not have a child with the label c, such a child is created and its total count σ is assigned with the value of 1. Otherwise, its total count is increased by 1. Then, the tree is updated at the beginning of every phase 1 in the training process. The algorithm processes each code token in a sequence in the training set similarly to when it creates the tree. However, if the topic assignment for that token is k, it updates φ_k instead of σ.

3.3.2 Predicting with N-gram Topic Model

The prediction algorithm has the input of a trained n-gram topic model ϕ, which contains the token distributions for all topics and n-grams, and a code sequence p. It first uses function Estimate (Figure 1) to estimate the topic proportion θ of p. Then, it estimates the generating probability $P(c|p)$ for any available token c and sequence p_n of the last $n-1$ code tokens of p, using the following formula:

$$P(c|p) = \max_n(\textstyle\sum_k \theta_k.\phi_{k,p_n}(c))$$

3.4 Pairwise Association

We use a conditional probability to model this pairwise association factor. $P(c|b)$ is the probability that c will occur as the next code token if a code token b has previously appeared. This probability is estimated as: $P(c|b) = \frac{\text{count}(c,b)}{\text{count}(b)}$.

To avoid the pairs that co-occur by chance, we consider a pair of tokens (c, b) only if they have data dependencies. For example, if two function calls open and close are performed on the same file (i.e. having a data dependency), they are counted. If they are used on different files, their co-occurrences might not be semantically related.

To reduce the storage and computational cost, we do not compute and store the probability $P(c|b)$ for any pair c and b (it would be a huge cost to compute/store V^2 such probabilities for the entire vocabulary). We instead consider only the tokens for control structures (including branching, loop, and exception handling statements) and API entities (including classes, methods/functions, and fields). We also consider only the pairs of tokens within the boundary of a method.

There might be several code tokens b associating with c in a code sequence p. We choose the one with the highest conditional probability $P(c|b)$. Then, we combine this conditional probability with the generating probability $P(c|p)$ computed via our n-gram topic model (Section 3.3). Currently, we choose the higher between two probabilities.

4. CODE SUGGESTION

Based on SLAMC's ability of next-token suggestion, we have built a code suggestion engine. Let us detail it next.

4.1 Semantics, Context-sensitive Suggestion

Overview. Instead of suggesting code in a template, our engine suggests *a sequence of tokens* that is *best-fit to the context* of the current code and *most likely to appear next.* It provides a *ranked list of such sequences.* SLAMC is semantic-based, thus we define a set of suggestion rules that are based on the current context and aim to complete a meaningful code sequence (Table 4). The idea is that such a suggested sequence would *complete the code at the current position to form a meaningful code unit and likely appear next.* Currently, we implement the rules to define a meaningful code unit in term of a member access, a method call, an infix expression, or a condition expression. For example, if the code context is recognized as an incomplete binary expression such as in "x + ", the suggestions will be an expression for the remaining operand with a data type compatible with x in the addition. If the context is an incomplete method call, a suggestion will be an expression with a compatible type for the next argument. If it does not match with any pre-defined context, the token with highest probability is suggested.

To illustrate our algorithm, let us consider an example (Table 5). Assume that a developer is writing a statement "if (node" and requests a suggestion (see (1)). Our engine first converts the code into a semantic code sequence p (see (2)). Analyzing this sequence, our engine recognizes that

Table 4: Rules of Context-sensitive Suggestion

Context	Example	Suggestion
Member Access	node.	a token for method or field name, e.g. size or value
Method Call	map.get(or Math.max(x,	a type-compatible expression for the next argument, e.g. y
Infix Expression	x +	a type-compatible expression for the other operand, e.g. y
Condition Expression	if (or while (	a Boolean expression e.g. x != y or !set.isEmpty()
Other	for (int i = 0;	a next token, e.g. i

Table 5: Semantic, Context-sensitive Completion

	Current code	Suggestions
Lexical tokens	(1) if (node	!= null (4) == null .isRoot()
Semantic tokens	(2) IF VAR[Node]	OP[neq] NULL (3) OP[equal] NULL OP[access] CALL[Node, isRoot,0,Boolean]

it matches the rule for an incomplete condition statement. Then, it searches for potential code sequences q that connect with the current code to form a boolean expression. Such sequences are ranked based on the score $P(q|p)$. Assume that the search returns a ranked list of three semantic code sequences as in (3). Those sequences are transformed back to lexical forms and presented to the user as in (4).

Our code suggestion algorithm has three main steps (Figure 2). In the first step (lines 2-3), it analyzes the code in the current method and produces its semantic code sequence. Since the current code might not be complete or syntactically correct, it uses Partial Program Analysis (PPA) [3] for code analysis and then recognizes the matched code context. PPA parses the code into an AST, which is then analyzed by SLAMC to produce the semantic code tokens with their sememes and other associated semantic information. If PPA cannot parse some tokens, it marks them as Unknown nodes and SLAMC creates the semantic tokens of type LEX for them. It also estimates the topics using n-gram topic model (line 3). In step 2 (lines 4-5, 11-22), it predicts the next code sequences that connect with the current code to form a type-compatible code unit as described in the rule of the matched context. All such sequences are ranked based on their scores using a search-based method. In step 3 (lines 6-9), those sequences are transformed to lexical forms and presented to users for selection and filling up. Let us detail the steps 2-3.

4.2 Predicting Relevant Code

Let us use s to denote the semantic code sequence for the entire source file under editing, and θ for its estimated topic proportion. Since the current editing position edpos might not be at the end of s, the engine starts the search from a sub-sequence p of s, containing all tokens prior to edpos. It looks for sequence(s) $q = c_1c_2...c_t$. The relevant score of q is:

$$P(q|p) = P(c_1|p,\theta).P(c_2|pc_1,\theta)...P(c_t|pc_1c_2...c_{t-1},\theta) \quad (*)$$

This suggests that we could expand the sequences token-by-token and compute the score of a newly explored sequence from the previously explored ones. Thus, our engine generates relevant next sequences by searching on a lattice of

```
1  function Recommend(CurrentCode F, NGramTopicModel φ)
2      s = BuildSequence(F) //sequence of semantic code tokens
3      θ = EstimateNGramTopic(s, φ) //topic proportion of F
4      p = GetCodePriorEditingPoint(s, edpos)
5      L = Search(p, θ)
6      foreach q ∈ L
7          lex[q] = Unparse(q)
8      u = UserSelect(lex)
9      Fillin(u)
10
11 function Search(p, θ)
12     L = new sorted list of size topSize, Q = new queue
13     Q.add('''', 1) //empty sequence, score = 1
14     repeat
15         q = Q.next()
16         if length(q) ≥ maxDepth then continue
17         C(q) = ExpandableTokens(p, q)
18         for each c ∈ C(q) Q.add(qc)
19         if ContextFit(p, q) then L.add(q, Score(q, p, θ, φ))
20     until Q is empty
21     if L is empty then add the top relevant tokens to L
22     return L
```

Figure 2: Code Suggestion

tokens of which each path is a potential suggestion using a depth-limited strategy. That is, it keeps a queue Q of exploring paths and chooses to expand a path q if it has not reached the maximum depth (maxDepth), which is a pre-defined maximum length for q (lines 15-18). If q satisfies a context rule, its score will be computed and it will be added to the ranked list L of suggested sequences (line 19). If no sequences satisfy the context, the top relevant tokens are added (line 21).

4.2.1 Expanding Relevant Tokens

Theoretically, at each search step, every token should be considered. However, to reduce the search space, we choose only the tokens "expandable" for the current search path q (function ExpandableTokens at line 17). To do that, we use the trained n-gram topic model ϕ to infer *the possible sememes $V(q)$ for the next token of q*, and then choose semantic tokens matching those sememes. Assume that the current search path is $q = c_1c_2...c_i$. To find the set of possible sememes $V(q)$ of the next token c, we connect p and q and extract any possible n-grams l ending at c_i (l might have tokens in both p and q). Then, we look for l on the prefix tree of n-grams (see Section 3.3.1). If l exists, all sememes of its children nodes are added to $V(q)$.

For each sememe $v \in V(q)$, we create a corresponding semantic code token and put it into the set of expandable tokens $C(q)$. We use the rules in Table 2 to infer necessary information, e.g. role or lexeme. For instance, if the sememe is CALL[Node,isRoot,0,Boolean], the semantic code token has the role of *function call* and the lexeme of isRoot. It has the same scope as the previous token c_i in q and no dependency.

The sememes of variables and literals in n-gram topic model do not have lexemes. Thus, we infer the lexemes for sememes of variables using a **caching** technique. If v is a sememe for a variable, we select all existing semantic code tokens in the sequence s that represent variables. Then, all tokens for variables that belong to the same or containing scope of the last code token c_i of the search path and have the same type as specified in the sememe v will be added to $C(q)$. For example, if c_i has the scope C1.M2.B3 and v is a VAR[String], all String variables in the scopes C1.M2.B3, C1.M2, and C1 are considered. For a literal sememe, we create a semantic token with the default value for its type (e.g. 0, null).

4.2.2 Checking of Context Fitness

Our engine uses the rules in Table 4 to check if a recommended sequence q produced by the above process fits with the context of the current code sequence p (function ContextFit, line 19). For example, from analyzing the current code via PPA to build semantic tokens, our engine knows that the last method call in the current code p has less number of arguments than that of parameters specified in its sememe, the context is then detected as an incomplete method call.

Then, based on the type of context of p, our engine checks if q fits with p as they are connected. If an expression is expected, our engine will check if q is a syntactically correct expression and has the expected type in the context p. If the context is a method call, it will check if q contains the expression that has the correct type of the next parameter for the method in p. If the context is an infix expression, then the result statement of connecting p and q must have the form of $X \diamond Y$, where X and Y are two valid expressions and have data types compatible with operator $\diamond$. Similar treatment is used for a condition statement in which a boolean expression is expected to be formed. If a context cannot be recognized due to incomplete code, ContextFit returns false.

4.2.3 Computing Relevance Scores

The relevance score of a new path qc is computed incrementally by (*) as $P(qc|p) = R(c).P(q|p)$, in which $R(c)$ is the relevance score of the token c to the current search path. Initially, $R(c)$ is computed as $P(c|pc_1c_2...c_i, \theta)$ using the n-gram topic model ϕ (see Section 3.3.2). Since ϕ models only local context and global concern, $R(c)$ is adjusted for other factors. First, if c is a token for a control keyword, or a method call, the maximal pair-wise association probability $P(c|b)$ for every $b \in pc_1c_2...c_i$ is selected for adjusting (Section 3.4). Otherwise, if c is a token for a variable, $R(c)$ is adjusted based on the distance r, in term of tokens, from the position of its declaration to the current position. In our current implementation, $R(c)$ is multiplied by $\lambda = 1/log(r+1)$. That is, the more distant the declaration of a variable, the lower its relevance to the current position.

4.3 Transforming to Lexical Forms

The transformation of a sequence q is done by creating the sequence of lexemes for the tokens in q. This task is straightforward since the lexeme is available in a token. However, our engine also adds the syntactic sugars for correctness (line 7). For instance, in CALL[String,length,0,Integer], the lexeme is length, and the method call has no argument. Thus, the lexical form length() is created with added parentheses. Finally, the lexical forms will be suggested in the original ranking.

5. EMPIRICAL EVALUATION

We conducted several experiments to study SLAMC's code suggestion accuracy with various configurations and to compare it with the lexical n-gram model [8]. Experiments were conducted on a computer with AMD Phenom II X4 965 3.0 GHz, 8GB RAM, and Linux Mint. For comparison, we collected the same data set of Java projects with the same revisions used in Hindle *et al.* [8] (Table 6). The data set consists of nine systems with a total of more than 2,039KLOCs. To evaluate on C# code, we also collected nine C# projects.

Procedure and Setting. We performed 10-fold cross validation on each project. We first divided the source files of a project into 10 folds (with similar sizes in term of LOCs). Each fold was chosen for testing, while the remaining ones were used for training. We performed training and testing for both SLAMC and the lexical n-gram model [8]. To evaluate the impact of different factors in SLAMC, we integrated various combinations of factors and performed training and testing for each newly combined model. For comparison, all models are configured to produce a single next lexical token.

Suggestion accuracy is measured as follows. For a source file in the test data, our evaluation tool traverses its code sequence s sequentially. At a position i, it uses the language model under evaluation to compute the top k most likely code tokens $x_1, x_2, ..., x_k$ for that position based on the previous code tokens. Since the previous tokens might not be complete, we used PPA tool [3] to perform partial parsing and semantic analysis for the code from the starting of the file to the current position to build semantic code tokens. If the actual token s_i at position i is among k suggested tokens, we count this as a hit. The top-k suggestion accuracy for a code sequence is the ratio of the total hits over the sequence's length. For example, if we have 60 hits on a code sequence of 100 tokens for a test file, accuracy is 60%. Total accuracy for a project is computed on all positions of its source files in the entire cross-validation process.

Table 6: Subject Systems

Java Proj.	Rel.Time	LOCs	C# Proj.	Rel.Time	LOCs
Ant	01/23/11	254,457	Banshee	01/23/13	166,279
Batik	01/18/11	367,293	CruiseControl	07/25/12	260,741
Cassandra	01/22/11	135,992	db4o	05/22/08	218,481
Log4J	11/19/10	68,528	Lucene.Net	03/08/07	169,413
Lucene	03/19/10	429,957	MediaPortal	01/19/13	922,765
Maven2	11/18/10	61,622	NServiceBus	03/09/12	31,892
Maven3	01/22/11	114,527	OpenRastar	09/28/11	52,018
Xalan-J	12/12/09	349,837	PDF Clown	11/13/11	66,308
Xerces	01/11/11	257,572	RASP Library	01/08/08	62,932

Table 7: Accuracy (%) with Various Configurations

Model	Top-1	Top-2	Top-5	Top-10
1. Lexical n-gram model ([8])	53.6	60.6	66.1	68.8
2. Seman.	58.0	65.8	72.7	76.3
3. Seman. + cache	58.7	66.9	75.7	80.3
4. Seman. + cache + depend.	58.8	67.0	75.8	80.4
5. Seman. + cache + depend. + pair.assoc	59.3	67.5	76.1	81.4
6. Seman. + cache + depend.+ LDA	58.9	67.1	76.0	81.3
7. Seman. + cache + depend. + n-gram topic	63.0	70.8	77.1	81.8
8. Seman. + cache + depend. +pair.assoc +n-gram topic [**SLAMC**]	64.0	71.9	78.2	82.3

5.1 Sensitivity Analysis: Impact of Factors

In our first experiment, we evaluated the impact of different factors on code suggestion accuracy. We chose Lucene, our largest Java subject system. Table 7 shows accuracy with different combinations of factors. The first row corresponds to the lexical n-gram model [8]. The second row shows accuracy of the model with the n-grams of semantic tokens, i.e., only semantic tokens and n-gram local context are considered. The 3^{rd} model is similar to the second one, however, the recently used variables' names are cached (Section 4.2.1). The 4^{th} row is for the model similar to the third one, however, the data dependencies among tokens in an n-

Table 8: Accuracy of Code Suggestion (Java)

Java Proj.	Rec. size	Lexical n-gram [8]	SLAMC	Abs. Improv	Rel. Improv.
Ant	Top 1	44.7%	63.5%	18.8%	42.1%
	Top 5	55.4%	79.5%	24.1%	43.5%
Batik	Top 1	44.7%	65.5%	20.8%	46.5%
	Top 5	55.4%	80.7%	25.3%	45.7%
Cassandra	Top 1	44.9%	65.9%	21.0%	46.8%
	Top 5	51.3%	73.5%	22.2%	43.3%
Log4J	Top 1	45.2%	67.4%	18.8%	41.6%
	Top 5	55.5%	79.2%	24.1%	43.4%
Lucene	Top 1	53.6%	64.0%	10.4%	19.5%
	Top 5	66.2%	78.2%	12.0%	18.1%
Maven-2	Top 1	41.3%	64.4%	23.1%	55.9%
	Top 5	51.0%	74.8%	23.8%	46.7%
Maven-3	Top 1	47.7%	65.0%	17.3%	36.3%
	Top 5	59.2%	74.1%	14.9%	25.2%
Xalan	Top 1	48.1%	68.6%	20.5%	42.6%
	Top 5	58.9%	82.4%	23.5%	39.9%
Xerces	Top 1	46.4%	66.6%	20.2%	43.5%
	Top 5	58.1%	81.8%	23.7%	40.8%

Table 9: Accuracy of Code Suggestion (C#)

Project	Rec. size	Lexical n-gram [8]	SLAMC	Abs. Improv	Rel. Improv.
Banshee (BS)	Top 1	37.2%	62.5%	25.3%	68.0%
	Top 5	47.8%	72.7%	24.9%	52.1%
Cruise Control (CC)	Top 1	42.8%	64.8%	22.0%	51.4%
	Top 5	54.4%	74.2%	19.8%	36.4%
db4o (DB)	Top 1	44.8%	65.0%	20.2%	45.1%
	Top 5	57.5%	77.3%	19.8%	34.4%
Lucene. Net (LN)	Top 1	47.0%	69.0%	22.0%	46.8%
	Top 5	58.6%	82.0%	23.4%	39.9%
Media Portal (MP)	Top 1	47.1%	66.7%	19.6%	41.6%
	Top 5	58.0%	79.4%	21.4%	36.9%
NService Bus (NB)	Top 1	44.5%	61.4%	16.9%	38.0%
	Top 5	55.6%	69.1%	13.5%	24.3%
Open Rastar (OR)	Top 1	36.3%	59.1%	22.8%	62.8%
	Top 5	46.1%	65.8%	19.7%	42.7%
PDF Clown (PC)	Top 1	44.8%	66.8%	22.0%	49.1%
	Top 5	56.2%	75.7%	19.5%	34.7%
RASP Library (RL)	Top 1	47.2%	68.3%	21.1%	44.7%
	Top 5	57.2%	77.6%	20.4%	35.7%

gram are considered. The n-grams with dependencies among their tokens are assigned twice as weights as the ones without dependencies. The 5^{th} model contains an addition of pairwise association factor to the 4^{th} one. To build the 6^{th} and 7^{th} models, we replaced pairwise association in the 5^{th} model with LDA and n-gram topic model, respectively. The last row corresponds to SLAMC model with all factors.

As seen, the models based on semantic tokens achieved better accuracy than the lexical n-gram model in [8]. With the addition of only semantic tokens, the relative improvement in accuracy is from 8.2% (top-1) to 10.9% (top-10) (comparing the first two rows). Adding the cache of recently used variables' names also improves over the lexical n-gram model, especially at top-5 (14.5%) and top-10 accuracy (16.7%). This suggests that for the practical use of SLAMC in code completion, considering the variables in the surrounding scopes helps much in filling in the next variable. We also found that requiring dependencies within an n-gram does not improve much accuracy (rows 3 and 4). This could be due to short sequences as n is 4 in this study. Interestingly, adding pairwise association improves slightly better than adding LDA topic model (rows 5 and 6). We examined concrete cases and found that pairwise association requires the co-occurrences of two tokens while LDA captures the topic via the co-occurrences of two or more tokens. Thus, LDA is too strict in those cases. Adding n-gram topic model improves better than adding pairwise association (rows 5 and 7). Importantly, SLAMC achieves even higher accuracy (last row). In comparing to the state-of-the-art lexical n-gram model [8], SLAMC has a good relative improvement in accuracy: 19.41% (top-1) and 19.62% (top-10).

5.2 Accuracy Comparison

Our second experiment was to compare SLAMC with the lexical n-gram model in two data sets of Java and C# projects. Tables 8 and 9 show the comparison results. First, for Java projects, accuracy with a single suggestion is 41.3–53.6% for the lexical n-gram model and 63.5–68.6% for SLAMC. For C# projects, top-1 accuracy with SLAMC is 59–69%, while lexical n-gram model achieves only 36.3–47.2%. With top-5 suggestions, SLAMC's accuracy could be as high as 82.4% (Java) and 82% (C#). Second, SLAMC is able to relatively improve over the lexical n-gram model from 18.1–55.9% (Java) and 24.3–68% (C#) in different top-ranked accuracy. Third, the suggesting time in both models is about a few seconds (not shown), and training time for all folds in the entire cross-validation process in SLAMC is much higher (2–15 times) (Table 10). However, it is still within a couple hours for the largest system. Finally, the result suggests different levels of code repetitiveness in different projects. It could be due to their nature and developers' coding style.

Table 10: Training Time Comparison (in seconds)

Model	BS	CC	DB	LN	MP	NB	OR	PC	RL
Lexical n-gram	46	150	117	80	957	9	14	10	11
SLAMC	300	592	1432	1150	4958	47	32	146	142

Examples. Here are some interesting patterns detected by SLAMC. The sememe sequence FOR TYPE[Map.Entry<String, Object>] VAR[Map.Entry<String,Object>] VAR[Map<String, Object>] CALL[Map<String,Object>,entrySet,0, Set<Map.Entry<String, Object> >] captures a pattern for accessing a Map object's entries. One of its instances is for (Map.Entry<String, Object> entry: map.entrySet()). Lexical n-gram model did not suggest correctly in an instance of this pattern with a different variable name: for (Map.Entry<String, Object> e: values.entrySet()).

The pattern WHILE VAR[StringTokenizer] CALL[StringTokenizer, hasMoreTokens, 0, boolean] VAR[String] OP[assign] VAR[StringTokenizer] CALL[StringTokenizer, nextToken,0, String] is frequently used for accessing all tokens of a StringTokenizer. One of its instances is while (st.hasMoreTokens()) { t = st.nextToken(); .... When the variable names change, as in while (qtokens.hasMoreTokens()) { tok = qtokens.nextToken();, SLAMC still recommends correctly, while the lexical n-gram model does not.

5.3 Cross-Project Training and Prediction

We performed another experiment to study SLAMC's accuracy when it is trained and used for prediction with data

Table 11: Cross-Project Prediction Accuracy

Java Project	Rec. Size	Lexical n-gram [8]	SLAMC	Abs. Improv.	Rel. Improv.
Ant	Top 1	44.5%	64.5%	20.0%	44.9%
	Top 5	56.6%	80.0%	23.4%	41.3%
Batik	Top 1	43.5%	66.5%	23.0%	52.8%
	Top 5	56.1%	81.1%	25.0%	44.6%
Cassandra	Top 1	45.4%	66.2%	20.8%	45.8%
	Top 5	57.7%	77.4%	19.7%	34.1%
Log4J	Top 1	47.5%	68.4%	20.9%	44.0%
	Top 5	59.6%	82.1%	22.5%	37.8%
Lucene	Top 1	53.6%	65.0%	11.4%	21.3%
	Top 5	66.2%	79.2%	13.0%	19.6%
Maven-2	Top 1	56.5%	70.4%	13.9%	24.6%
	Top 5	71.0%	83.9%	12.9%	18.2%
Maven-3	Top 1	54.2%	67.0%	12.8%	23.6%
	Top 5	68.6%	77.7%	9.1%	13.3%
Xalan	Top 1	49.6%	70.4%	20.8%	41.9%
	Top 5	61.0%	84.4%	23.4%	38.4%
Xerces	Top 1	46.6%	66.8%	20.2%	43.3%
	Top 5	59.5%	81.9%	22.4%	37.6%

across projects. For each Java project in Table 8, we performed 10-fold cross-validation as in Section 5.2. However, to predict for one fold, we used not only the other nine folds but also the other eight Java projects for training. As seen, when both models used the training data from other projects, SLAMC relatively improves over the lexical n-gram model from 13.3%–52.8% for top-1 and top-5 accuracy. This is consistent with the relative improvement of 18.1%–55.9% in Table 8 when training data was from only a single project.

Comparing SLAMC's accuracy in Tables 11 and 8, we can see that prediction accuracy is not improved much as using cross-project data for training (0.1%–9.1%). This is also true for the lexical n-gram model (also reported by Hindle *et al.* [8]). Similar accuracy implies that the degree of regularity across projects is similar to that in a single project.

Threats to Validity and Limitations. Our selected projects are not representative. However, we chose a high number of projects with large numbers of LOCs. Our simulated code suggestion procedure is not true code editing. We re-implemented lexical n-gram model, rather than using their tool. Inaccuracy is caused by the fact that SLAMC does not consider class inheritance and cannot correctly resolve types/roles sometimes due to incomplete code. It also faces out-of-vocabulary issue (code un-seen in training data).

6. RELATED WORK

Statistical language models [15] have been successfully used in software engineering. Hindle *et al.* [8] use n-gram model with lexical tokens to show that source code has high repetitiveness. Thus, the n-gram model has good predictability and is used to support code suggestion. In comparison, SLAMC has key advances. First, its basic units are semantic code tokens, which are incorporated with semantic information, thus providing better predictability. Second, SLAMC's n-grams are also complemented with pairwise association. It allows the representation of co-occurring pairs of tokens that cannot be efficiently captured with n consecutive tokens in n-grams. Finally, a novel n-gram topic model is developed in SLAMC to enhance its predictability via a global view on current technical functionality/concerns.

Code repetition is also observed by Gabel *et al.* [5]. They reported *syntactic redundancy* at levels of granularity from 6-40 tokens. They considered only syntactical tokens and renamed IDs in the code sequences, while SLAMC operates at the semantic level. Han *et al.* [6] have used Hidden Markov Model (HMM) to infer the next token from user-provided abbreviations. Abbreviated input is expanded into keywords by an HMM learned from a corpus. In comparison, their model has only local contextual information, while SLAMC has also n-gram topic modeling and pairwise association. n-gram model has been used to find code templates relevant to current task [13]. n-grams are built over clone groups.

Other line of approaches to support code completion relies on the *programming patterns* mined from existing code. Grapacc [16] mines and stores API usage patterns as graphs and matches them against the current code. The patterns most similar to the code are ranked higher. Bruch *et al.* [2] propose three algorithms to suggest the method call for a variable v based on a codebase. First, FreqCCS suggests the most frequently used method in the codebase. Second, ArCCS is based on mined associate rules where a method is often called after another. The third algorithm, best-matching neighbors, uses as features the set of method calls of v in the current code and the names of the methods that use v. The features of methods in examples are matched against those of the current code for suggestion. Precise [21] completes the parameter list of a method call. It mines a codebase to build a parameter usage database. Upon request, it queries the database to find best matched parameter candidates and concretizes the instances. Omar *et al.* [18] introduce active code completion in which interactive and specialized code generation interfaces are integrated in the code completion menu to provide additional information on the APIs in use.

Other strategies have been proposed to improve code completion. Hill and Rideout [7] use small *cloned fragments* for code completion. It matches the fragment under editing with small similar-structure code clones. Robbes and Lanza [19] introduced six strategies to improve code completion using *recent histories* of modified/inserted code during an editing session and on the methods and class hierarchy related to the current variable. Hou and Pletcher [10] found that ranking method calls by frequency of past use is effective. Eclipse [4] and IntelliJ IDEA [12, 11] support *template-based* completion for common constructs/APIs (for/while, Iterator).

MAPO [22] mines API patterns and suggests associated *code examples*. Strathcona [9] extracts *structural context* of the current code and finds its relevant examples. Mylyn [14], a code recommender, learns from a developer's personal usage history and suggests related methods.

7. CONCLUSIONS

We introduce SLAMC, a novel statistical semantic language model for source code. It incorporates semantic information into code tokens and models the regularities/patterns of such semantic annotations. It combines the local context in semantic n-grams with the global technical concerns into an n-gram topic model. It also incorporates pairwise associations of code elements. Based on SLAMC, we have developed a new code suggestion technique, which is empirically evaluated on open-source projects to have relatively 18–68% higher accuracy than the lexical n-gram model.

8. ACKNOWLEDGMENTS

This project is funded in part by US National Science Foundation (NSF) CCF-1018600 and CNS-1223828 grants.

9. REFERENCES

[1] P. Baldi, C. V. Lopes, E.J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 2008 Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'08*, pages 543–562. ACM Press, 2008.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC/FSE '09*, pages 213–222. ACM Press, 2009.

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

[4] Eclipse. www.eclipse.org.

[5] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the 2010 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156. ACM, 2010.

[6] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 332–343. IEEE CS, 2009.

[7] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the 2004 IEEE/ACM International Conference on Automated Software Engineering*, *ASE '04*, pages 228–235. IEEE CS, 2004.

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

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

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

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

[12] Intellisense. http://blogs.msdn.com/b/vcblog/archive/tags/intellisense/.

[13] F. Jacob and R. Tairas. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 104:1–104:6. ACM Press, 2010.

[14] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 2006 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, *SIGSOFT '06/FSE-14*, pages 1–11. ACM Press, 2006.

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

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

[17] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 2012 IEEE/ACM International Conference on Automated Software Engineering*, *ASE'12*, pages 70–79, ACM Press, 2012.

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

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

[20] H. M. Wallach. Topic modeling: beyond bag-of-words. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 977–984, ACM Press, 2006.

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

[22] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the 2009 European Conference on Object-Oriented Programming*, *ECOOP'09*, pages 318–343. Springer-Verlag, 2009.