Stepwise API Usage Assistance Using N-Gram Language Models

André L. Santos*, Gonçalo Prendi, Hugo Sousa, Ricardo Ribeiro
Instituto Universitário de Lisboa (ISCTE–IUL)

Abstract

Reusing software involves learning third-party APIs, a process that is often time-consuming and error-prone. Recommendation systems for API usage assistance based on statistical models built from source code corpora are capable of assisting API users through code completion mechanisms in IDEs. A valid sequence of API calls involving different types may be regarded as a well-formed sentence of tokens from the API vocabulary. In this article we describe an approach for recommending subsequent tokens to complete API sentences using n-gram language models built from source code corpora. The provided system was integrated in the code completion facilities of the Eclipse IDE, providing contextualized completion proposals for Java taking into account the nearest lines of code. The approach was evaluated against existing client code of four widely used APIs, revealing that in more than 90% of the cases the expected subsequent token is within the 10-top-most proposals of our models. The high score provides evidence that the recommendations could help on API learning and exploration, namely through the assistance on writing valid API sentences.

Keywords: API, usability, n-grams, code completion, IDE

1. Introduction

Software development and programming using third-party APIs (Application Programming Interfaces) can be considered inseparable activities, given that reliable and efficient software construction typically requires using several APIs. The study of API usability has gained attention in the research community [1], since efficient and correct use of third-party APIs is not easily achievable with mainstream development practices and tools. On the one hand, API usability was studied through empirical experiments (e.g., [2, 3, 4, 5, 6]), which revealed the types of barriers that API users face (e.g., relationships between types and instantiation of abstract types). On the other hand, recommendation

[*] Corresponding author
Email address: `andre.santos@iscte.pt` (André L. Santos)

systems were proposed as an aid to assist API users through IDE (Integrated Development Environment) code completion mechanisms, which rely either on structural analysis (e.g., [7, 8]) or on mining patterns from source code (e.g., [9, 10, 11, 12, 13]). Although code completion is an IDE feature that boosts programmer productivity with respect to code writing using a familiar API, empirical studies demonstrated that code completion is often used to explore, and hence, learn an unfamiliar API (e.g., [3, 6]).

Analogously to natural languages, a valid sequence of API calls may be regarded as a well-formed API *sentence*, which is composed of *tokens* from the API *vocabulary*. A user that is learning an API is in fact learning how to write valid API sentences for achieving a certain goal. More concretely, the API sentences are formed by instructions that either instantiate public types of the API or invoke public operations on those same types. API sentences typically involve more than one API type, and hence, a sentence often forms a collaboration between API types.

In this article we describe a recommendation system for stepwise API usage assistance through code completion, based on an API sentence model built from source code corpora using n-gram probabilistic language models [14]. N-gram models have been used before to study the "naturalness" of source code [15]. An important difference to this approach is that ours is a more semantic approach to program representation, as the extracted sentences are not programming instructions, but abstractions that represent them. The computed model and the recommendation method work over the abstraction of API sentence. On the other hand, our approach is specialized for building a language model for a well-defined API, providing code completion focused on that API accordingly. Constraining the scope of the language model to a well-defined API makes possible to have specialized models for APIs that assist its usage in isolation, as opposed to general source code writing.

The intelligent code completion approach initially proposed by Bruch et. al. [9] is based on statistical models and has been successfully transferred to an industry-scale solution, namely Code Recommenders[1] for Eclipse. This system can be considered to be one of the most advanced code completion systems available for developers. Although their system provides recommendations taking into account the context, i.e. the instructions that a developer is writing, the code completion recommendations are constrained to the calls on a particular API type given by a variable of the context. More concretely, the conventional code completion system of Eclipse is enhanced so that the valid operations on a type displayed in a code completion pop-up menu are ordered according to their likelihood given the context, instead of according to alphabetical order. Intelligent recommendation of operations for a type only helps an API learner partially, given that the code completion system does not provide assistance in the creation of sequences of API tokens involving different types.

[1] `www.eclipse.org/recommenders`

For example, considering the SWT[2] library, if one has a variable of type `Composite`, Code Recommenders may suggest invoking `setLayout(...)` as the first proposal, because upon instantiating this type the most likely step to take is setting the layout. However, the next typical step upon creating a `Composite` object is to instantiate one of the layout types (e.g., `GridLayout`), in order to use it through the `setLayout(...)` operation (otherwise no contents are displayed). While the recommendation of this operation hints into that direction, it is up to the user to find out about this requirement. In contrast to our approach, Code Recommenders provides no help with this respect, as their proposals are constrained to the API type on which the user requested code completion. In our approach, recommendations consist of API sentence completion involving different API types.

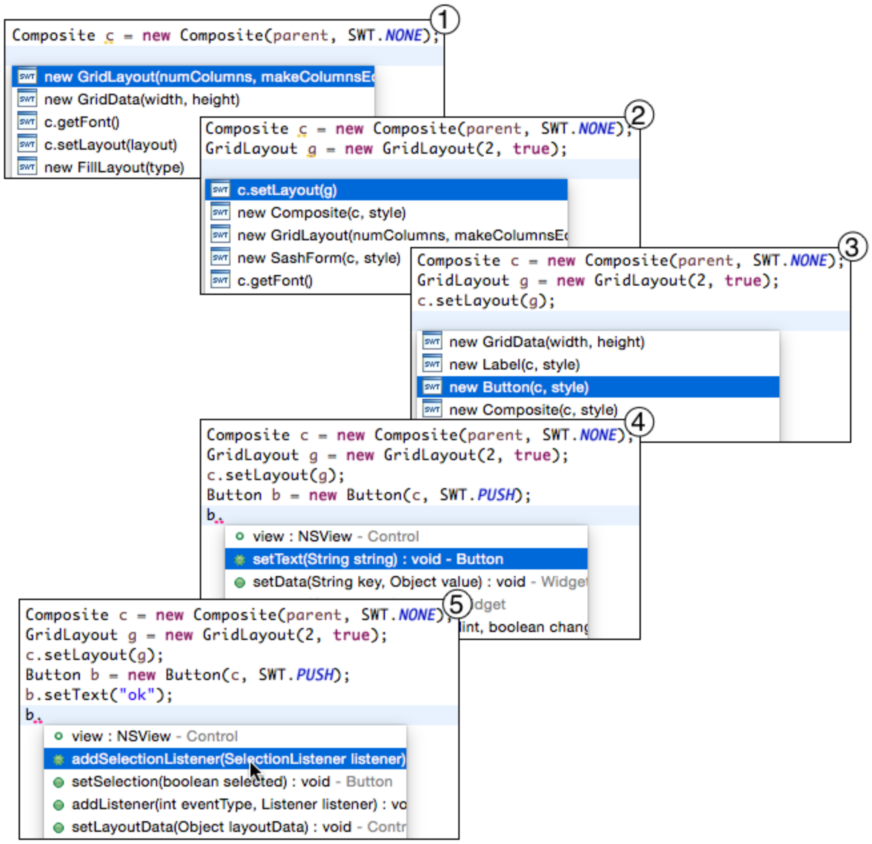
We implemented our approach for Java and the recommendation system was developed as a plugin to the Eclipse IDE. The plugin makes use of the available code completion facilities, namely the pop-up menus that are available when writing code in the editor. Figure 1 presents a usage scenario of our APISTA tool[3] where we can see that the code completion proposals adapt to the context, suggesting the use of related API types, using the previous example with the `Composite` object. Notice that as the context changes, proposals are progressively adapted to the previously written instructions (steps 1 to 3 in the figure). In addition to this kind of recommendations, our system is also suitable for recommending operation calls on a type given its variable (steps 4 and 5 in the figure), in a similar way as Code Recommenders.

The recommendation system was evaluated using 4 widely-used APIs, for which our models were trained and evaluated against existing source code harvested from public repositories of Github. The results of performing an evaluation of the model recommendations using cross-validation, revealed that in every API there is at least a 90% chance of having the expected subsequent instruction within the 10-top-most recommendations. Given this high score, we argue that the proposed recommendations consist of a valuable aid to programmers when having to learn, explore, and use an API, considering that the typical amount of code completion proposals that are visible in a code completion pop-up menu without scrolling is around 10–15 items.

In this article we describe the components and processes required for building the proposed recommendation system, namely how to mine software repositories, a method for building the *n*-gram language models for APIs, the integration of the latter in an IDE, and we present an evaluation experiment and its results. The main contributions of this article are: (1) an IDE-based code completion approach for assisting on writing API sentences, (2) a method for building an API sentence model using *n*-gram language models, and (3) a quantitative evaluation of the approach involving widely used APIs.

[2] Standard Widget Toolkit — `www.eclipse.org/swt`

[3] APISTA stands for *API Sentence Token Assistance*, whereas in Portuguese “a pista” means *the clue*.



1
new GridLayout(numColumns, makeColumnsE
new GridData(width, height)
c.getFont()
c.setLayout(layout)
new FillLayout(type)
2
c.setLayout(g)
new Composite(c, style)
new GridLayout(numColumns, makeColumnsE
new SashForm(c, style)
c.getFont()
3
new GridData(width, height)
new Label(c, style)
new Button(c, style)
new Composite(c, style)
4
view : NSView - Control
setText(String string) : void - Button
setData(String key, Object value) : void - Widge
dget
int, boolean chang
5
view : NSView - Control
addSelectionListener(SelectionListener listener)
setSelection(boolean selected) : void - Button
addListener(int eventType, Listener listener) : vo
setLayoutData(Object layoutData) : void - Contr

```
Composite c = new Composite(parent, SWT.NONE);
```

```
Composite c = new Composite(parent, SWT.NONE);
GridLayout g = new GridLayout(2, true);
```

```
Composite c = new Composite(parent, SWT.NONE);
GridLayout g = new GridLayout(2, true);
c.setLayout(g);
```

```
Composite c = new Composite(parent, SWT.NONE);
GridLayout g = new GridLayout(2, true);
c.setLayout(g);
Button b = new Button(c, SWT.PUSH);
b.
```

```
Composite c = new Composite(parent, SWT.NONE);
GridLayout g = new GridLayout(2, true);
c.setLayout(g);
Button b = new Button(c, SWT.PUSH);
b.setText("ok");
b.
```

Figure 1: Stepwise API usage assistance for writing API sentences using the code completion proposals of our APISTA tool in Eclipse. Users may request code completion either on an empty line or on a variable (after the dot). The immediate previous lines are considered as the context for computing the recommendations. Values of parameters for which a compatible variable was not available in the context were manually inserted by the user.

This article proceeds as follows. Section 2 presents an overview of our approach. Section 3 address the process of mining software repositories. Section 4 describes the language model in detail. Section 5 presents the evaluation of our system against existing APIs. Section 6 discusses issues pertaining to the integration of the recommendation system in an IDE. Section 7 compares our approach to related work. Section 8 describes limitations of our approach. Section 9 presents our conclusions and outlines future work.

2. Approach Overview

There are several components and stakeholders involved in our approach (see Figure 2). Providing a recommendation system for a given API requires mining source code repositories where the API is used. Several processes have

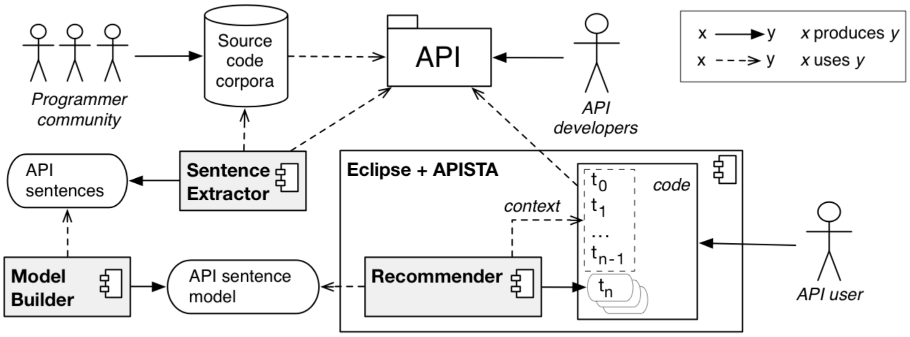


Figure 2: Overview of components and stakeholders involved in our approach. The three main components are depicted in gray. The recommender component is an Eclipse plugin (APISTA).

to be carried out for achieving robust and accurate recommendations. Namely, determining the API vocabulary in order to extract API sentences from source code repositories, building the API language model using the extracted information, and integrating the model in a code completion system that is able to recommend proposals of API usage that take into account the code that is being edited as context.

API developers write and maintain libraries and frameworks, whose API is used by several projects that are developed by a *programmer community*. Each project that uses the API is a candidate for being part of the source code corpus that is used to build the API *sentence model*. The broader the source code corpus is in terms of API coverage, the more complete the language model will be. The API source is used to determine its boundaries, in terms of which types belong to the API when mining the repositories.

There are two core processes for setting up our recommendation system for a particular API. The *sentence extractor* component mines a set of API sentences from the source code corpus. The *model builder* component uses the sentence set to build a API sentence model. In turn, the model is used by a *recommender component* in an IDE in order to assist API users with *proposals* that augment the API sentence that is being written in the code editor. The recommender is integrated with the code completion system of Eclipse, using the available facilities as the means to present the API usage proposals. When requesting code completion, the context in which the API user is writing code is given as input to the recommender, namely the tokens contained in the lines of code that precede the line where code completion is requested. API usage assistance is provided in a stepwise manner, through single-token proposals to augment the API sentence that is being written by the developer.

Invocation	Token type	Token operation
Class constructor	owner class	“new”
Static operation	owner class	operation name
Instance operation	target expression type	operation name

Table 1: Token kinds in API sentences.

3. Extracting API Sentences

The degree of robustness in the process of extracting API sentences from source code corpora affects the quality and accuracy of the recommendations. Poorly extracted snippets result in data with more “noise”, which in turn will lead to less accurate language models. Therefore, we handle with special care the process of sentence extraction sentences using different tactics. The sentence extractor is a parser of Java source files for this purpose, which we developed using the infrastructure of Eclipse's Java Development Tools. The sentence examples that we use in this section are based on the API of SWT[4].

3.1. Tokens

Within the realm of an API, we refer to *token* as a possible code instruction that involves a public API type and a public operation therein, considering both static operations and constructors as operations too. Tokens are semantic “chunks” that abstract the representation of code instructions. Currently we do not handle method overloading, and therefore, there is no distinction amongst methods of the same class whose name is equal but their parameters differ. The same applies to alternative public constructors of the same class. A token is uniquely identifiable by a tuple formed by an API type and an operation of that type. We consider the API *vocabulary* to be formed by a set containing all the possible tokens of the API[5], denoted by $\mathcal{V}$:
$\mathcal{V} = (t_0, t_1, \ldots, t_n) : \forall t \in \mathcal{V}, t = \langle \mathit{type}, \mathit{operation} \rangle$

Our extraction process considers three kinds of tokens: constructor invocations, static operations, and instance operations. Constructor invocation is in fact a static operation, but it has to have special treatment in Java and we use the keyword “new” to name the operation. Table 1 summarizes the kinds of tokens that are considered when extracting sentences.

Tokens are extracted from assignment statements and expressions. Tokens are extracted if their type (first element of the tuple) is part of the API, whereas all other tokens are ignored. The API boundaries are given by the API source code that takes part as input in the extraction process.

[4] `www.eclipse.org/swt`

[5] A member of a vocabulary is normally referred to as *type*, but we do not adopt such a terminology here to avoid confusion with the types of the API.

3.2. Sentences

An API *sentence* is a sequence of tokens of $\mathcal{V}$, whose instructions are related. API sentences are mined from a corpus (a set of projects that use the API), which is used to build the language model. A sentence consists of an observation found in the corpus (ω) and is treated as a vector of tokens from the vocabulary:

$$\omega = (t_0, t_1, \ldots, t_n) : \forall t \in \omega, t \in \mathcal{V}$$

The extracted sentences are mined by parsing every method instruction block found in the source code corpora. For instance, the following method would result in the extraction of the sentence below.

```
void method(...) {
  Label label = new Label(...);
  String s = ...;
  label.setText(s);
}
```

Extracted sentence: $\omega = (\langle Label, new\rangle, \langle Label, setText\rangle)$

The instantiation of `Label` was considered as an API token, the `String` assignment was ignored, and the operation `Label.setText` was also identified as a token.

3.3. Sentence sample vocabulary

Consider $\mathcal{S}_{\mathcal{V}}$ as the set of all possible sentences using vocabulary $\mathcal{V}$, i.e. the sample space. The set of extracted sentences is denoted by Ω, consisting of a sample of observations that is a subset of the possible sentences ($\Omega \subseteq \mathcal{S}_{\mathcal{V}}$).

$$\Omega = (\omega_0, \omega_1, \ldots, \omega_n) : \forall \omega \in \Omega, \omega \in \mathcal{S}_{\mathcal{V}}$$

Given that the sentences of the sample may not use every token of the vocabulary, the vocabulary of the extracted sentences $\mathcal{V}_{\Omega}$ is a subset of the API vocabulary ($\mathcal{V}_{\Omega} \subseteq \mathcal{V}$). $\mathcal{V}_{\Omega}$ is used to build the language model, and the accuracy of the latter depends on the quality of the former.

3.4. Composite expressions

Instructions may contain composite expressions that involve API tokens. Given that our goal is to capture sentences in terms of their semantics, rather than how they are expressed syntactically, we treat composite expressions so that the extracted sentences are decomposed as if having separate instructions. For instance, both the following code snippets result in the same extracted sentence.

```
Composite composite = new Composite(...);
composite.setLayout(new GridLayout(...));
```

```
Composite composite = new Composite(...);
GridLayout layout = new GridLayout(...);
composite.setLayout(layout);
```

Extracted sentence: $\omega = (\langle Composite, new\rangle, \langle GridLayout, new\rangle, \langle Composite, setLayout\rangle)$

The token order reflects the execution order, namely, the instantiation of `GridLayout` occurs prior to the invocation of `setLayout(...)`. Analogously, when having chained invocations, the expression is separated into different instructions. The following code snippet illustrates this case, where the two cases also result in the same extracted sentence.

```
new Label(...).getParent().getParent();
```

```
Label label = new Label(...);
Composite parent = label.getParent();
Composite parentParent = parent.getParent();
```

Extracted sentence: $\omega = (\langle Label, new\rangle, \langle Label, getParent\rangle, \langle Composite, getParent\rangle)$

3.5. Token dependencies

API calls are normally interleaved with other instructions that do not relate to the API, while unrelated API sentences might also be interleaved. The criteria for relating the tokens that form a sentence is based on dependency graphs between instructions. An instruction a is considered to depend on another instruction b in the following cases:

1. b is an assignment and a uses the assigned variable;
2. a is an invocation and b is used in its arguments.

Algorithm 1 describes in pseudo-code how the instruction blocks are processed with respect to token dependencies. Each instruction block will result in a set of sentences (possibly empty, if no API tokens are found therein). We start with an empty set of sentences. For every instruction of the block, the variables used therein are obtained and the set of sentences where at least one of those variables is used is computed (sv). If there are no matching sentences in the set, then a new sentence containing the instruction is created and added to the set. If there is only one matching sentence, then the instruction is appended to the end of that sentence. Finally, if there is more than one matching sentence, these are considered related and merged into a new sentence, using the order by which the instructions appear in the block. The merged sentences are removed from the set and the new sentence is added.

Consider the following example, where we have two unrelated sentences in a block. The instructions that form the two `Composite` objects are unrelated because the variables that are involved do not overlap, and hence, two sentences are extracted from the block.

```
Composite composite1 = new Composite(...);
composite1.setLayout(new RowLayout());
Composite composite2 = new Composite(...);
composite2.setLayout(new FillLayout());
Button button = new Button(composite1, ...);
Text text = new Text(composite2, ...);
button.setText(...);
text.setText(...);
```

```
input  : $block : \forall i \in block, i \in \mathcal{V}_\Omega$
         (a list of instructions)
output : $out \subseteq \mathcal{S}_\mathcal{V}$
         (a set of unrelated sentences)

$out \leftarrow \varnothing$
foreach $i : block$ do
    $sv \leftarrow \{s \in out : variables(s) \cap variables(i) \neq \varnothing\}$
    if $sv.isEmpty()$ then
        $sentences.add(new\ Sentence(i))$
    end
    else if $sv.size() = 1$ then
        $sv.get(0).append(i)$
    end
    else
        $out.removeAll(sv)$
        $s \leftarrow merge(sv)$
        $s.append(i)$
        $out.add(s)$
    end
end
```

Algorithm 1: Computing a set of unrelated sentences of an instruction block.

Extracted sentences:

$$\omega_1 = (\langle Composite, new\rangle, \langle RowLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Button, new\rangle, \langle Button, setText\rangle)$$

$$\omega_2 = (\langle Composite, new\rangle, \langle FillLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Text, new\rangle, \langle Text, setText\rangle)$$

3.6. Selections and loops

Selections (`if-else` blocks) are treated so that a sentence is extracted for each possible branch, using a similar strategy to the MAPO tool [10]. Once again, the extracted sentences follow the execution order, but in this case, with alternative execution sequences. Loops are treated in the same way as if they were conditionals, preserving the possible execution sequences.

Consider the following code snippet with an `if-else` block to illustrate this case. Given that there are two possible executions of the block, two sentences are extracted.

```
if(composite.isVisible()){
  composite.setFocus();
}
else {
  composite.setVisible(...);
  composite.redraw();
}
composite.layout();
```

Extracted sentences:

$$\omega_1 = (\langle Composite, isVisible\rangle, \langle Composite, setFocus\rangle, \langle Composite, layout\rangle)$$

$$\omega_2 = (\langle Composite, isVisible\rangle, \langle Composite, setVisible\rangle, \langle Composite, redraw\rangle, \langle Composite, layout\rangle)$$

4. API Sentence Model

The *model builder* component builds an API n-gram language model from the set of all observed sentences, Ω, of a given repository.

4.1. N-gram language models

A language model [14] is a statistical model that allows us to compute the probability of a sentence, or predict the next word in a sentence, of a given language. From a generative perspective, all sentences of a (natural) language can be described in terms of the product of a set of conditional probabilities. Hence, the probability of a sentence $\omega = (t_0, t_1, \ldots, t_n)$ is given by

$$P(\omega) = P(t_0)P(t_1|t_0)P(t_2|t_0t_1)\cdots P(t_n|t_0t_1\cdots t_{n-1}) \tag{1}$$

Formally, a language model is a probability distribution over a set of symbol (token) sequences, Ω, from a finite set of symbols, $\mathcal{V}_\Omega$. The probability of a sequence of symbols, ω is given by Equation 1, by applying the chain rule of probability.

Either to make predictions or to compute the probability of a given sentence, we need to estimate the conditional probabilities, $P(t|h)$, where t is known as the *prediction* and h as the *history*, found in Equation 1. A simple way to compute these probabilities is to use the maximum likelihood estimate, which is calculated by counting the number of sequences with history h that are followed by the prediction t and dividing by the number of sequences with history h (Equation 2).

$$P(t_n|t_0t_1\cdots t_{n-1}) = \frac{C(t_0t_1\cdots t_n)}{C(t_0t_1\cdots t_{n-1})} \tag{2}$$

However, since these models are computed using a limited set of data, in general, it is not reliable to estimate these probabilities for long histories. Instead, by making use of the (N-order) Markov assumption, we can approximate these probabilities by setting a limit, N, to the length of the history (Equation 3).

$$P(t_n|t_0t_1\cdots t_{n-1}) \approx P(t_n|t_{n-N+1}\cdots t_{n-1}) \tag{3}$$

The corresponding maximum likelihood estimate is given by Equation 4.

$$P(t_n|t_{n-N+1}\cdots t_{n-1}) = \frac{C(t_{n-N+1}\cdots t_n)}{C(t_{n-N+1}\cdots t_{n-1})} \tag{4}$$

Language models have been widely used in natural language processing for tasks such as speech recognition [16], spell-checking and correction [17], or machine translation [18]. We adapt this idea for code recommendation. Given a sequence of tokens, a code history, and using a language model, it is possible to compute the probability of each token of the vocabulary following that history. The list of predictions consists of a list of instructions associated to the tokens of $\mathcal{V}_\Omega$, sorted according to the probabilities given by the n-gram language model.

Tokens	**Count**
$\langle Composite, new\rangle$	3
$\langle RowLayout, new\rangle$	1
$\langle Composite, setLayout\rangle$	3
$\langle Button, new\rangle$	2
$\langle Button, setText\rangle$	2
$\langle FillLayout, new\rangle$	2
$\langle Text, new\rangle$	1
$\langle Text, setText\rangle$	1

(a) 1-gram counts (unigram).

Tokens	$\langle Composite, new\rangle$	$\langle RowLayout, new\rangle$	$\langle Composite, setLayout\rangle$	$\langle Button, new\rangle$	$\langle Button, setText\rangle$	$\langle FillLayout, new\rangle$	$\langle Text, new\rangle$	$\langle Text, setText\rangle$
$\langle Composite, new\rangle$		${}^{1}/_{3}$				${}^{2}/_{3}$		
$\langle RowLayout, new\rangle$			${}^{1}/_{1}$					
$\langle Composite, setLayout\rangle$				${}^{2}/_{3}$			${}^{1}/_{3}$	
$\langle Button, new\rangle$					${}^{2}/_{2}$			
$\langle Button, setText\rangle$								
$\langle FillLayout, new\rangle$			${}^{2}/_{2}$					
$\langle Text, new\rangle$								${}^{1}/_{1}$
$\langle Text, setText\rangle$								

(b) 2-gram counts and probabilities.

$$\begin{aligned}
\omega_0 &= (\langle Composite, new\rangle, \langle RowLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Button, new\rangle, \langle Button, setText\rangle)\\
\omega_1 &= (\langle Composite, new\rangle, \langle FillLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Text, new\rangle, \langle Text, setText\rangle)\\
\omega_2 &= (\langle Composite, new\rangle, \langle FillLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Button, new\rangle, \langle Button, setText\rangle)
\end{aligned}$$

(c) Sample of observations $\Omega = (\omega_0, \omega_1, \omega_2)$.

Figure 3: Example of the estimation of a 2-gram model (bigrams).

Figure 3 presents a small example of how to compute the probabilities for 2-grams given a set of sentences (subfigure 3c). Subfigure 3a presents the 1-gram (unigram) counts for the sentence set. Subfigure 3b presents the 2-gram counts and probabilities in a table that can be interpreted as "*column token* given *row token*". As shown in Equation 5, dividing each row by the frequency of the corresponding unigram (since the example is a 2-gram language model) we obtain the conditional probabilities, which are the parameters of our model.

$$P(t_1|t_0) = \frac{C(t_0 t_1)}{C(t_0)} \tag{5}$$

4.2. N-gram smoothing methods

As we can see from the example in Figure 3, there are several possible 2-grams that do not occur in the set of sentences used to compute the language model. This problem, known as sparse data problem, causes the maximum likelihood estimate to output zero probabilities for n-grams not occurring in the training set. In order to avoid zero probabilities, a smoothing method can be used to mitigate this problem.

Several smoothing methods have been proposed for n-gram language modeling [19]. Namely, methods like the Witten-Bell smoothing [20] and the Kneser-Ney smoothing [21] are still used in state-of-the-art work [22, 23, 24]. In general, these methods combine high-order models with lower-order ones.

In our experiments, the best results were achieved by the Witten-Bell smoothing method. This method uses an interpolated model, as shown in Equation 6 ($\tau(t_n|t_{n-N+1}...t_{n-1})$ is the distribution used when the n-gram has a non-zero count; $\gamma(t_{n-N+1}...t_{n-1})$ is a scaling factor to make the distribution sum to 1), to perform the combination of high-order models with lower-order ones.

$$\begin{aligned} &P_{\text{smooth}}(t_n|t_{n-N+1}...t_{n-1}) = \\ &\qquad \tau(t_n|t_{n-N+1}...t_{n-1}) + \gamma(t_{n-N+1}...t_{n-1}) P_{\text{smooth}}(t_n|t_{n-N+2}...t_{n-1}) \end{aligned} \tag{6}$$

In the Witten-Bell smoothing method, Equation 6 takes the form of Equation 7 (P_{ML} is the maximum likelihood estimate).

$$\begin{aligned} P_{\text{WB}}(t_n|t_{n-N+1}...t_{n-1}) = {} & \lambda_{t_{n-N+1}...t_{n-1}} P_{\text{ML}}(t_n|t_{n-N+1}...t_{n-1}) \\ & + (1-\lambda_{t_{n-N+1}...t_{n-1}}) P_{\text{WB}}(t_n|t_{n-N+2}...t_{n-1}) \end{aligned} \tag{7}$$

$\lambda_{t_{n-N+1}...t_{n-1}}$ parameters are defined such that

$$(1-\lambda_{t_{n-N+1}...t_{n-1}}) = \frac{|\{t_k : C(t_{n-N+1}...t_{n-1}t_k) > 0\}|}{|\{t_k : C(t_{n-N+1}...t_{n-1}t_k) > 0\}| + C(t_{n-N+1}...t_{n-1})} \tag{8}$$

Equation 9 shows how to compute the Witten-Bell estimate.

$$\begin{aligned} &P_{\text{WB}}(t_n|t_{n-N+1}...t_{n-1}) = \\ &\frac{C(t_{n-N+1}...t_n) + |\{t_k : C(t_{n-N+1}...t_{n-1}t_k) > 0\}| P_{\text{WB}}(t_n|t_{n-N+2}...t_{n-1})}{C(t_{n-N+1}...t_{n-1}) + |\{t_k : C(t_{n-N+1}...t_{n-1}t_k) > 0\}|} \end{aligned} \tag{9}$$

5. Evaluation

In order to evaluate our approach to code completion we carried out a quantitative experiment to investigate how well our API sentence models perform against existing client source code of APIs. We evaluated our models with 4 widely used APIs, which differ in terms of vocabulary size and domain, in order to assess if our models have similar properties across different APIs. For each API, we trained models with different orders of n-grams using the SRLIM tool[6]. This section describes this experiment in terms of goals, research questions, method, results, and threats to validity. All the procedures and measurements were performed on an Intel Core i5 2.5GHz machine with 4Gb of RAM running Linux Ubuntu 15.

5.1. Goals

The following are the two main goals of our evaluation:

- Measure the performance of API sentence models against existing client code, namely with respect to the prediction of subsequent API tokens;

- Analyze diverse APIs in order to characterize how well API sentence models can capture regularities in API usage in general.

5.2. Research questions

With respect to API sentence models and their performance, the following are our research questions:

1. How does the order of the n-gram API sentence model relate to the maximization of coverage (i.e. the expected token is included in the first x predictions of the ranking)?
2. How does the order of the n-gram API sentence model relate to the ranking of predictions with respect to token type (different token types are given equal weight regardless of their frequency)?

5.3. Data collection

Allamanis and Sutton [25] collected a large corpus from GitHub[7] composed of thousands of Java projects (more than 14,000). The set of projects was harvested from the public repositories and filtered according to the Github's social fork system, in order to have some confidence on the code quality level, given that low quality projects are rarely forked. We have used this corpus to obtain projects that contained client code to the APIs under evaluation, in order to have a set of popular APIs that vary in terms of vocabulary size and domain.

[6] SRI Language Modeling Toolkit, `http://www.speech.sri.com/projects/srilm`

[7] `www.github.com`

Table 2: APIs used in the experiment: vocabulary size, number of extracted sentences, number of corpus projects, vocabulary coverage, approximated duration of extraction process.

API	Vocabulary	Sentences	Projects	Voc. Coverage	Extraction
Swing	7,347	160,951	2,308	34%	21h
SWT	5,824	105,881	501	31%	17h
Jackson	551	9,306	73	33%	2h
JSoup	369	1,519	119	59%	1h

We started by analyzing the corpus to search for projects that used APIs that were familiar to the authors. In order to determine the corpus projects that used an API, we ran scripts to search in the projects' source code for import statements and fully qualified names that contain the APIs' root package. A preliminary analysis of the APIs was made to obtain their vocabulary from the source code of the associated library. If the corpus projects that used the API could cover more than 30% of its vocabulary, we considered that API for the evaluation. Since we aimed at APIs that substantially differ in terms of vocabulary size, we directed our criteria towards "filling the gaps" in with respect to vocabulary size in order to have a wide and balanced range of vocabulary sizes. Besides the selected APIs, we attempted to collect projects for other APIs such as JAXP (XML processing) and JDBC (Java Database Connectivity), but the vocabulary coverage was low. After determining which projects use the APIs, we used the method described in Section 3 to extract the API sentences.

The result of this extraction process was a set of 4 APIs, which we summarize in Table 2. APIs are sorted in the table by vocabulary size. Swing is a library that is part of Java's SDK to create graphical user interfaces, SWT is a cross-platform library to develop graphical user interfaces developed by Eclipse, Jackson[8] is library to process JSON data, and JSoup[9] is a library to fetch and parse and manipulate HTML.

5.4. Language model training

The evaluation experiment consisted of a 5-fold cross-validation [26] using the extracted set of API sentences (Ω). We randomly divided the sentences into 5 folds for training and testing. The evaluation consisted of 5 runs, which use 80% of the sentences for training and the remaining 20% for testing. Each one of the 5 folds served as part of the training set in 4 out of 5 runs, and as part of the test set in 1 out of 5 runs. The SRILM toolkit was used to train the n-gram language models from each subset of the extracted API sentences. The performance evaluation for each API was made under three forms: no smoothing method, Witten-Bell smoothing, and Kneser-Ney smoothing. Given

[8] `github.com/FasterXML/jackson`

[9] `jsoup.org`

that Witten-Bell smoothing[10] yielded better results in terms of coverage (1–5% improvements), here we only present the results using this smoothing strategy.

Each sentence of the test folds was an observation $\omega = (t_0, t_1, \ldots, t_n)$ that gave rise to $n-1$ test cases formed by the sentence itself and its subsentences of tokens ranging from t_0 to $t_{1 \cdots n-1}$. The last token of each test case was the *expected token* (t_e) against which the predictions of our model were compared. We refer to the train/test folds as $\Omega_{1 \cdots 5}$, and to the set expected tokens as $\mathcal{V}_e$.

Given that a query to the n-gram model yields several tokens ranked by a probability, we registered the position of the expected token in the ranking. A value of 1 means that t_e appeared first in the ranking, and hence, the lower the values are, the more accurate the model is. If the t_e was not proposed by the model, it can either mean that it did not occur in the training set at all or that the model learned it with a different context. This happens with rare tokens, i.e. tokens that are only present in the test set, or when the API does not have a minimum common usage pattern for that token. Finally, we did not evaluate the model with single-token sentences, since this implies an evaluation with an empty context in order to propose that token.

5.5. Measurements

We analyzed the API sentence models under two perspectives, *coverage* and *hit rank*, which we detail in the following sections.

5.5.1. Coverage

Coverage is a measure that indicates the percentage of predictions that matched the expected token within the x-top-most elements of the ranking of predictions. As x increases, the coverage percentage can never decrease. Given a test fold Ω_i and a model trained on $(\Omega_{1 \cdots 5} - \Omega_i)$, and having a function $\mathrm{topX}(\omega, x)$ that given a history context (ω) returns the top-most x predictions given by that model, we calculate the coverage dividing the number of test cases whose expected token was within the top-most x predictions by the total number of test cases (Equation 10).

$$\mathrm{coverage}(\Omega_i, x) = \frac{C(\omega_{t_0 \cdots t_{e-1} t_e} : \omega \in \Omega_i \wedge t_e \in \mathrm{topX}(\omega_{t_0 \cdots t_{e-1}}, x))}{C(\Omega_i)} \tag{10}$$

In turn, we obtain the overall coverage score of the experiment with a given n-gram by averaging the results of all the 5 runs (Equation 11).

$$\mathrm{coverage}_n(\Omega_{1..5}, x) = \mathrm{AVG}(\mathrm{coverage}(\Omega_1, x), \cdots, \mathrm{coverage}(\Omega_5, x)) \tag{11}$$

Figure 4 presents the complete results for each API and n-gram order. We trained models with 4 orders (2-gram, 3-gram, 4-gram, 5-gram) for each API,

[10] with SRILM parameters: `-interpolate`.

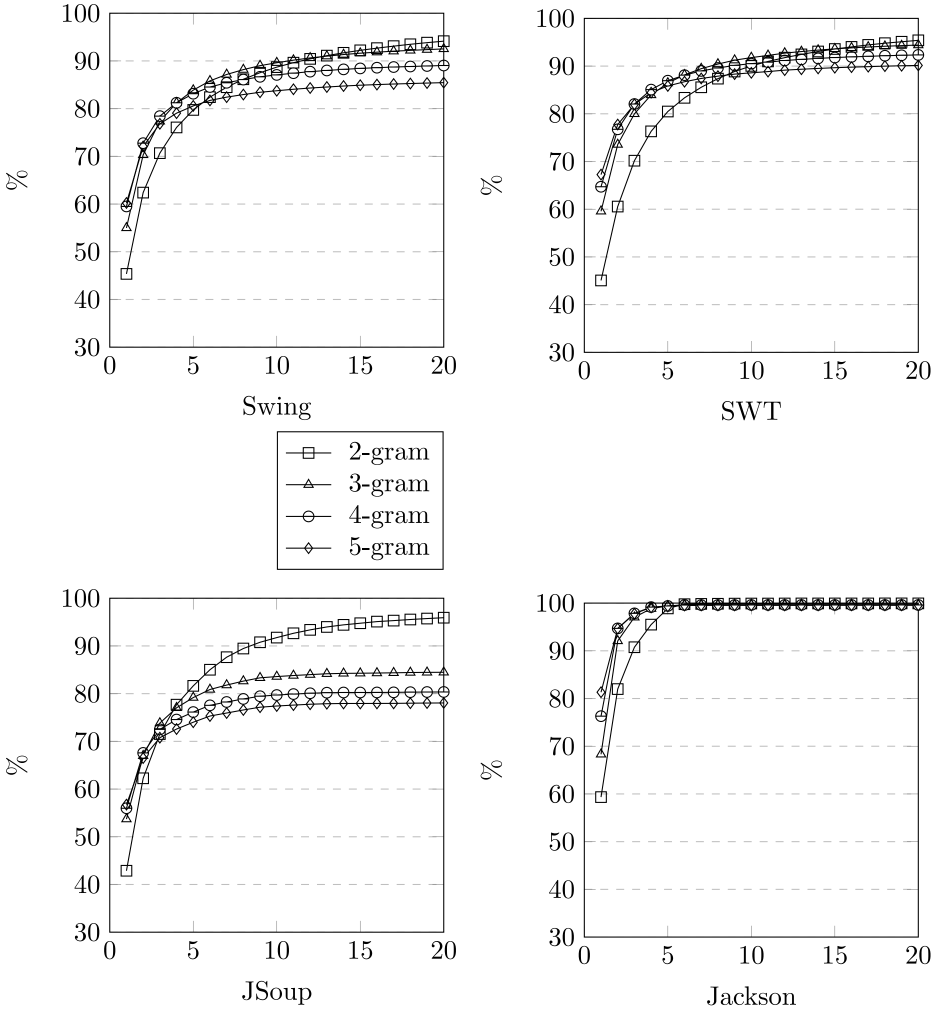


Figure 4: Coverage results for the several APIs using different n-gram orders with Witten-Bell smoothing. X-axis: number of ranking predictions (top-most). Y-axis: coverage percentage.

and the plot displays the values of coverage$_n$(...) for each one within the range [1, 20]. The x-axes of the plots represent the number of x-most ranking predictions, whereas the y-axes represent the coverage obtained with the first x predictions. We tested higher orders of n-gram (6-gram and 7-gram), however we omit those results here given that from 5-gram on the growth trends are similar, but the coverage becomes lower as we increase the n-gram order. This happens because longer histories strongly condition the prediction.

High coverage within a relatively low x is important with respect to exploration in code completion because programmers will typically drive their attention more often to the first proposals in the code completion pop-menu. For instance, Eclipse's code completion pop-up menu typically displays between 10 and 15 proposals at a time.

We found that for all APIs it is possible to reach a coverage value above 90% with the 10 top-most predictions of our models. Further, 80% can be reached with only the 5-top-most predictions. As expected, higher order n-gram models have overall lower coverages. In principle, by obtaining more training data, which in turn would provide more long history sentence contexts, larger coverage could be achieved for higher-order n-gram models.

Prediction models of 2-gram and 3-gram have a similar performance in terms of coverage in all APIs except one (JSoup), which might be due to the wide variability of sentences that can be written for JSoup. Despite the high coverage achieved by our models, we consider that the code completion itself (automation of code writing) is not the central benefit of our approach. Instead, we argue that main added value pertains to exploration and discoverability possibilities for the API user, given that in most situations a narrow set of possible directions (∼10) is enough to guide the developer on writing API sentences in most situations (∼90%).

5.5.2. Hit rank

Hit rank refers to the position of the expected token in the ranking of predictions. We analyze hit rank according to token type, meaning that the hits ranks are grouped by token and statistics are computed taking into account an equal weight for every unique expected token regardless of its frequency. This score indicates how the model performs across all the tokens it can propose, ignoring the fact that some combinations of API sentences will occur more often in the train/test sets. In this way, in contrast to the coverage measurement, here we avoid having results that are influenced positively or negatively by frequent tokens that are systematically predicted right or wrong (and having a significant impact on the overall scores). We only considered the 20-top-most hits obtained by our model.

Given a test set Ω_i, a trained model on the other sets $(\Omega_{1\cdots5} - \Omega_i)$, and assuming a function rankPosition(ω, t_e) that given a sentence (ω) and an expected subsequent token (t_e) returns its position in the prediction ranking obtained with the model, we gather the set of all positions for each unique expected token (Equation 12).

$$\text{rankSet}(\Omega_i, t_e) = \{\text{rankPosition}(\omega_{t_0 \cdots t_{e-1}}, t_e) : \omega_{t_0 \cdots t_e} \in \Omega_i\} \tag{12}$$

In turn, we obtain the overall hit rank score for an expected token by calculating the median of the set union of all predictions from the cross-validation runs (Equation 13).

$$\text{rankScore}(\Omega_{1\cdots5}, t_e) = \text{MEDIAN}\left(\bigcup_{k=1}^{5} \text{rankSet}(\Omega_k, t_e)\right) \tag{13}$$

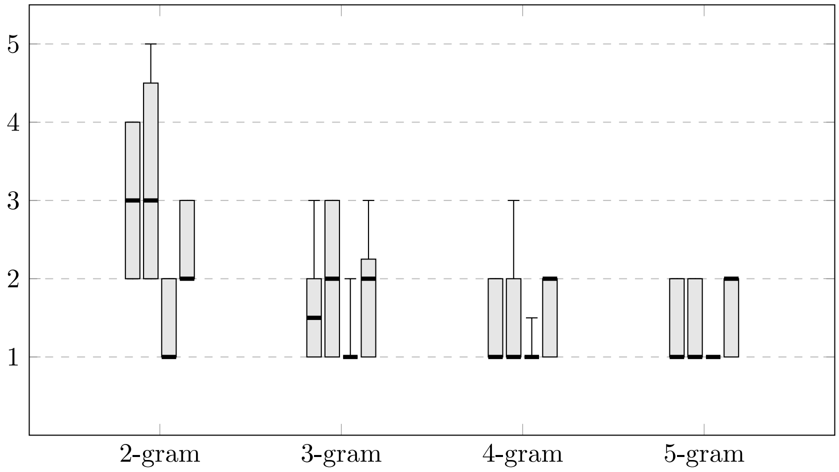


Figure 5: Hit rank score per token type on several orders of n-gram for each API. APIs are grouped by n-gram order, and within the group with the order (Swing, SWT, Jackson, JSoup). Box plot bars describe the quartiles (Q_1, median (thicker line), Q_3) and the whiskers are placed at the 10^{th} and 90^{th} percentiles. When certain elements are not visible is because their values are coinciding with a quartile.

Finally, the overall hit rank score for the experiment for an n-gram model is obtained by calculating the median of the rank scores of each unique expected token (Equation 14).

$$\text{rankScore}_n(\Omega_{1\cdots5}) = \text{MEDIAN}\left(\{\text{rankScore}(\Omega_{1\cdots5}, t_e) : t_e \in \mathcal{V}_e\}\right) \tag{14}$$

Figure 5 presents the data related to hit rank scores for every API grouped by the different model orders. Each box in a plot represents one API trained on a particular n-gram model, where the value of $\text{rankScore}_n(...)$ is represented by the thick line band that divides the box. The lower and upper edges of the boxes represent the quartiles Q_1 and Q_3, whereas the lower and upper whiskers represent the 10^{th} and 90^{th} percentiles.

We can observe that the results are similar for every API. As the order of n-gram increases the distribution of hit ranks narrows and the rank score (median) decreases. This means that the expected tokens are proposed higher up in the ranking of model predictions. Notice that with 2-gram models, 80% of the predictions include the expected token in the top-5, while from 3-gram on, 80% of the predictions achieve the same in the top-3. However, recall that as the order of n-gram increases, the coverage decreases, and hence, despite that the tokens that are being proposed match the expected token high up in the ranking, there are cases that the expected token is not being proposed at all (at most in $100\% - \mathit{coverage}$ of the cases). Notice also that from 2 to 3-gram models there are considerable improvements, from 3 to 4-gram as well but less expressive, and that from 4 to 5-gram there are practically no improvements.

5.6. Threats to validity

The obtained results quantitatively demonstrate that the model proposals would be useful to programmers when writing API sentences. However, it is important to consider some threats that might be influencing these results.

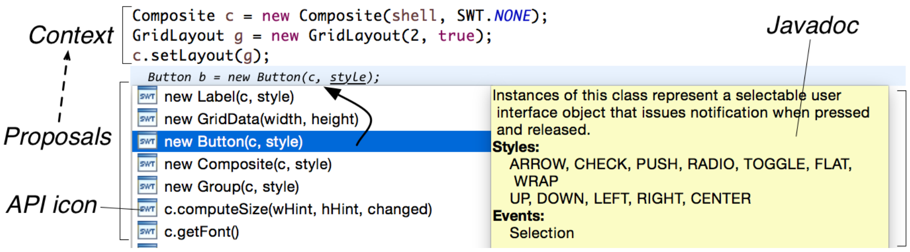
Construct validity. We decided to evaluate our models against existing code and we relied on a third-party previous work that gathered a corpus of projects from Github, ensuring a reasonable level of code quality. The evaluation is threatened by the possibility of this corpus not being representative, or not having enough sentences for the APIs under evaluation. We mitigate this issue by selecting APIs that were used by at least 100 projects in the corpus, ensuring a reasonable number of sentences for training and testing the models (more than 1500 for each API). Another problem is that despite having many projects that use an API, its usage could be narrow in terms of vocabulary coverage, and hence, we would be dealing with a small part of the API. We addressed this threat by ensuring that we selected API whose vocabulary had a reasonable proportion was used in the corpus (recall Table 2).

Internal validity. The selection of the APIs for evaluation may consist of a bias, given that the search was driven by the familiarity and knowledge that the authors previously had about certain APIs. We proceeded in this way because we were not aware of a source or well-established criteria for obtaining an adequate set of APIs for evaluation (to our knowledge, there are no benchmarks available for this purpose). Nevertheless, we ensured that the selected APIs were popular, as they were found in use on many projects.

External validity. Generalization of our results is threatened by the fact that the evaluation was performed on few APIs. Although our results were consistent with respect to the APIs that we tested, there might exist APIs with different characteristics on which API sentence models may perform differently. In other to mitigate this threat, our criteria for selecting APIs for evaluation was to obtain a set that differed substantially in size and domain. Therefore, we argue that it is reasonable to assume that these APIs are representative of what is commonly found in Java development.

6. Code completion tool (APISTA)

The probabilistic models of API sentences may be used for different purposes. In this article we focus on using the models for assisting API users, namely through code completion. IDEs typically offer facilities for code completion, commonly through pop-up menus in code editors containing proposals that the user may select. The selection of a code completion proposal inserts code at the caret position where the developer is typing, as well as additional code elsewhere in some cases (e.g., adding required import statements). An IDE such as Eclipse allows third-party plugins to contribute with code completion proposal engines. Using this mechanism, we extended Eclipse with code completion proposals provided by our API sentence model in a tool that we refer to as APISTA.



```
Composite c = new Composite(shell, SWT.NONE);
GridLayout g = new GridLayout(2, true);
c.setLayout(g);
Button b = new Button(c, style);
```

Instances of this class represent a selectable user interface object that issues notification when pressed and released.
Styles:
ARROW, CHECK, PUSH, RADIO, TOGGLE, FLAT, WRAP
UP, DOWN, LEFT, RIGHT, CENTER
Events:
Selection

Figure 6: User interaction with the APISTA tool. Parameters are matched with context variables (for instance, *c*). Unmatched parameters have to be completed manually (for instance, parameter *style*).

6.1. Selecting n-gram order

Our tool has to be parameterized with respect to the n-gram order of the API sentence models that will be used for obtaining the code completion proposals. Given that higher-order n-gram models suffer more from the sparse data problem than lower ones, they have a lower coverage (Section 5.5.1), which is not desirable from an API exploration perspective. On the other hand, higher-order n-gram models are more accurate in the predictions that they can generate (Section 5.5.2). Given that increasing the n-gram order progressively decreases coverage, we argue that the code completion engine should be set to use the lowest n so that $\text{rankScore}_n(...) \approx \text{rankScore}_{n+1}(...)$ (i.e. there is no major improvement regarding hit ranks when using the subsequent n-gram order available).

6.2. User interface

Figure 6 presents the user interface of our APISTA tool integrated with the code completion menu of Eclipse. As illustrated, the proposals are adapted to the context variables. Each proposal is accompanied by the Javadoc documentation text, as in regular settings, in order to help the user to select among the available proposals. We believe that having proposals that are adapted to the context variables helps the user to realize how the API objects are composed, besides accelerating code typing. For convenience, import statements are also automatically inserted if necessary, as in regular IDE code completion features. We extract the lines of the block in which the user is writing code that precede the line where code completion is requested. Depending on the order of the trained n-gram models that is being used, different context tail lengths will be used to query the model. A list of context variables is also extracted in terms of *identifier* and *type* so that this information can be used to adapt the code completion proposals to match them when possible.

In addition to the illustrated usage mode, another possible way of using the recommendations is through the existing code completion menu of Eclipse for displaying the available operations of a type given a variable (recall Figure 1, steps 4 and 5). We sort the Eclipse-provided results according to the ranking of our recommendations. The same ranking is used as in the other recommendation mode, but only the hits pertaining to types that are compatible with the object are considered.

The most common situation is that a software project makes use of several APIs. Therefore, it is relevant to consider that the proposals of different APIs have to coexist in the IDE. In our solution for Eclipse the support for each API can be plugged independently, defining a proposal category for each API with its own icon (as illustrated with SWT). The activation/deactivation of the proposal categories is managed by the IDE. Given that the proposals for each API are determined by the context, if the latter has no tokens of a certain API there will be no proposals with respect to that API.

6.3. Integration in software development

The described process for building an API language model is always targeted at a well-defined API. We envision that it should be a responsibility of the API developers to build a sentence model for it, and possibly package the model together with the libraries of that API (for instance, included as part of the JAR files of the libraries). Another option could be based on independent extensions that are installed separately. In either case, the owners of a software artifact should in principle have a good notion of which projects to select for building a model for their API. Although we currently have not designed a well-defined format for serializing the API model, this would be straightforward to achieve. The most important technical aspect here is to have a serialized model that is IDE-agnostic, so that different IDEs could seamlessly make use of the same artifact. As with our APISTA tool for Eclipse, a recommender component that is specific to an IDE has to load the model and implement the necessary behavior to query the model correctly. The recommender component has to extract the context of the code editor to a token format that is compatible with the model vocabulary.

7. Related Work

Robillard et. al. [27] present a survey of a wide range of approaches for automated API property inference using different techniques. The approaches also differ in terms of their goals, which may concern bug detection, documentation, API navigation, and recommendations, being the latter the focus of our work too. The survey classified the techniques into five categories, namely (1) unordered usage patterns, (2) sequential usage patterns, (3) behavioral specifications, (4) migration mappings, and (5) general information. Our approach fits in category (2), given that our recommendation are based on API token sequences that were mined from source code corpora. In this section we present related work related to API usage recommendations, divided into *method call recommenders*, which are the closest to our approach, *snippet recommenders*, which assist API users through examples, and *method chain recommenders*, which enable API users to find out how to reach a desired object type from a source object type.

7.1. Method call recommenders

Hindle et al. [15], applies Natural Language Processing techniques to code and try to find evidence that software is far more regular than, for example, English, and these techniques can be applied to code completion systems. To evaluate this, the authors implemented a system embodying a corpus-based n-gram [14] model suggestion engine that tries to guess the next token based on source code corpora. This approach motivated our work since it provides evidence that source code is predictable to a considerable extent. In our approach we focus on the goal of code completion for providing usage assistance for a well-defined API, rather than to source code in general.

SLAMC [12] is a statistical semantic language model for source code that introduces semantic information into the language models, which represents additional information in the language model that involves, for instance, the global context of source files to help predict the next token. Tu et al. [28] however, argue that code tokenization is enough for the n-gram language models. They also argue that n-gram models will not be useful when a particular context is not present in the source code corpora used to train the model.

SLANG [13] is a tool that uses n-gram language models and recurrent neural networks [29] to fill holes in partial programs that use a certain API. In our work we achieve a stepwise assistance that adapts and helps the programmer using and discovering the API, as opposed to filling holes in partial programs. Their approach also replaces very rare API calls with an unknown token in order compact the language model. Since their work focus on evaluating the tool and not the language models, we cannot compare the impact of this replacement. The token extraction method of SLANG is superior to ours, namely with respect to handling of parameter values.

The code completion system developed by Marcel Bruch et al. [9] mentioned in Section 1, uses a modified k-nearest-neighbours [30] called "Best Matching Neighbours", that recommends method calls for particular objects, by extracting the context of the variable, searching the codebase for similar situations and synthesize method recommendations. This code completion system led to an Eclipse Project named Code Recommenders, along with some additional features like "Override Completion" that recommends which methods are usually overridden when extending a certain API class, "Chain Completion" that suggests chains of method calls that return the desired type and "Adaptive Template Completion" that recommends multiple methods that frequently occur together on an object. Being this tool the most complete, and despite that the system provides several different recommendations programmers usually need with a certain object, it doesn't assist the programmer providing the next probable tokens when writing an API sentence. Given that sequential usage patterns are not addressed, the main limitation of Code Recommenders relates to assist in writing sequences that involve multiple types, which is the focus of our approach.

The Better Code Completion (BCC) system [31] sorts, filters and groups API methods. Sorting, has two options, first is a type-hierarchy-based sorting, which proposes the methods from the declared type before its super type, which is very useful since in certain contexts methods like `wait()` are never used. Second is a popularity-based sorting, that sorts based on the frequency of a method call, similar to our and other approaches. BCC allows developers to configure groups of methods that will be displayed together in the code completion pane, but this is performed manually and therefore suffers from scalability issues. Moreover, the assistance is constrained to the methods of a certain object and does not provide any additional aid on how write API sentences.

7.2. Snippet recommenders

When using an unfamiliar API, software developers often search for examples that can usually be found in its documentation or on the web. Snippet recommenders are tools that help on this process given that the amount of search results might make difficult to manually browse through the search hits. Although snippets show interactions with different objects, if they mismatch the developer's goal, it can become an obstacle [32]. Although snippet matching might be effective to learn parts of an API, it does not consist of a stepwise process where the programmer makes a decision on each instruction as in our approach. Both kinds have their strengths and weaknesses. Whereas a snippet is a ready-made sequence of instructions, it still has to be adapted to the programmer needs, and the programmers goal might be scattered among different code snippet recommendations.

MAPO is tool [10] used to mine API usage patterns to recommend associated code snippets, using the structural context in which the developer is writing code. The instructions are matched to code snippets, providing a ranked list of patterns. These are used to navigate to the existing code snippets, which are displayed in a code viewer where the pattern appears highlighted. The user is required to go through the suggested snippets to investigate how the API is used and eventually learn how to achieve the desired goal. XSnippet [33] is a tool to assist code writing, that allows programmers to query repositories for relevant code snippets according to their context without having to explicitly formulate a query. Although this approach does not require programmers to formulate queries, they still need to know what to search for which implies some knowledge about the API. The Strathcona [34] is an example recommendation tool to help programmers locate source code examples by formulating a query based on the code's structural context.

7.3. Method chain recommenders

Another kind of tool consists of method chain recommenders that enable programmers to find a path from a source to a destination type. This kind of aid is useful in cases when a programmer has access to a certain type, and wants to know how to reach another type, if possible. The drawback of these tools is the requirement of having some knowledge about the API in order to be able to query the system, i.e., the developer has to know which object type to input and in some cases which one to expect, or otherwise the developer cannot benefit from the tool. In our approach, developers also need a starting point (instruction) to benefit from code completion, but there is no need to provide any clues regarding where one wants to reach, and hence, it provides a more explorative type of usage.

The API Explorer tool [8] aids discoverability by recommending methods and types that are not directly accessible from the type with which the developer is working with. The recommendations might consist of a single method call or a chain of object instantiations and calls to obtain a reference to a certain type. The Prospector tool [7] helps programmers by returning code snippets, referred to as *jungloids*, to obtain a certain type through a chain of method calls given another type that is present in the code. RECOS [35] consists of an object-instantiation and recommendation system capable of composing a chain of method calls that returns a certain type given an input type. These tools do not require a repository of sample code to mine snippets nor a source code search engine. The recommendations are only based on the context and structural relationships in the APIs. Not having to rely on source code corpora is a strong advantage of these approaches, in contrast to ours or others that rely on source code mining. As opposed to the previous method chain recommenders, the PARSEWeb tool [36] uses a code search engine to gather code sample and then analyse it statically to return the relevant method sequences, also through a query where source and destination object type are given.

8. Limitations

Although our approach scores high on predicting subsequent tokens of an API sentence, there are a number of limitations regarding the code completion proposals that we detail in this section.

Tokenization ignoring overloading. As explained, our tokenization of API sentences does not take into account method and constructor overloading. Our intuition was that abstracting overloaded methods and constructors into a single token would not have a significant impact on the results. However, we are aware that making the distinction in certain cases could yield value.

Long-distance relations among tokens. The ability to recommend useful API tokens is constrained by a relatively small number of previous of tokens (e.g., previous 4 tokens using a 5-gram model). This limitation is inherent to n-gram models [14]. The downside is that some recommendations could benefit from tokens written far back in the context, i.e. distance greater that the order of the n-gram.

Structural context. As opposed to other approaches (e.g., [34, 10, 9]), ours does not take into account the structural context in which the code is being written, for instance, if the programmer is overriding a certain method of a particular class. Using the structural context is relevant for handling the APIs of frameworks that are instantiated through inheritance or interface realization, given that certain API calls may occur predominantly when extending the framework at well-defined places. However, we argue that in the case of reusable libraries that are not frameworks, the structural context is not relevant given that the contexts where the library is being used are independent from the library.

Parameter values. Although our system may match parameters of the context whose type belongs to the API vocabulary, it is not able to provide example values for parameters of other types. We believe that one of the strengths of snippet matching recommenders (e.g., [34, 33, 10]) relies on this aspect, given that some parameter values might not be obvious to come up with.

9. Conclusions and Future Work

In this article we presented an approach that exploits *n*-gram probabilistic language models, widely used in natural language processing, in the context of API usage. Despite that *n*-grams have been used to analyze source code in previous approaches, our approach is novel given that it is targeted at learning a well-defined API. In the context of this work, we mainly aim at API usage assistance from the perspective of learning and exploration by unfamiliar users. Given the results we have obtained, we conclude that *n*-grams are a powerful tool for learning API usage patterns, given that they are effective in capturing their regularity. We also conclude that the *n*-gram models can effectively be used to feed proposals to a code completion system.

We realized the approach for Java and developed an extension for the Eclipse IDE as a proof of concept, demonstrating that the approach is feasible and adequate for being integrated in software development practices. We plan to conduct a user study through a controlled experiment to evaluate the usability of our code completion system. The effectiveness of the mechanism from a human-centered perspective may reveal other shortcomings that we did not anticipate, as well as give rise to new ways of improvement. For example, if the results of a user study would reveal that dealing with parameter values consists of a significant hurdle, the sentence extractor process could be enhanced to collect common parameter values, in order to use them in the proposals.

Acknowledgement

We would like to thank Fernando Batista and the anonymous reviewers for their valuable comments for improving earlier drafts of this article.

References

[1] B. A. Myers, J. Stylos, Improving API usability, Communications of the ACM (accepted for publication, 2016).

[2] B. Ellis, J. Stylos, B. Myers, The factory pattern in API design: A usability evaluation, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, 2007, pp. 302–312. `doi:10.1109/ICSE.2007.85`.

[3] J. Stylos, B. A. Myers, The implications of method placement on API learnability, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, 2008, pp. 105–112.

[4] M. P. Robillard, R. Deline, A field study of API learning obstacles, Empirical Software Engineering 16 (6) (2011) 703–732.

[5] M. Piccioni, C. A. Furia, B. Meyer, An empirical study of API usability, in: ESEM'13, 2013, pp. 5–14.

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

[7] D. Mandelin, L. Xu, R. Bodík, D. Kimelman, Jungloid mining: Helping to navigate the API jungle, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, 2005, pp. 48–61.

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

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

[10] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, MAPO: Mining and recommending API usage patterns, in: Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP '09, 2009, pp. 318–343.

[11] M. Mooty, A. Faulring, J. Stylos, B. A. Myers, Calcite: Completing code completion for constructors using crowds, in: Proceedings of the IEEE Symposium of Visual Languages and Human-Centric Computing, VL/HCC '10, 2010, pp. 15–22.

[12] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, A statistical semantic language model for source code, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 532–542.

[13] V. Raychev, M. Vechev, E. Yahav, Code completion with statistical language models, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2014, p. 44.

[14] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, J. C. Lai, Class-Based *n*-gram Models of Natural Language, Computational Linguistics 18 (4) (1992) 467–479.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 837–847.

[16] B. Roark, M. Saraclar, M. Collins, Discriminative n-gram language modeling, Computer Speech & Language 21 (2) (2007) 373 – 392.

[17] T. A. Pirinen, S. Hardwick, Effect of Language and Error Models on Efficiency of Finite-State Spell-Checking and Correction, in: Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing, Association for Computational Linguistics, 2012, pp. 1–9.

[18] T. Brants, A. C. Popat, P. Xu, F. J. Och, J. Dean, Large Language Models in Machine Translation, in: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, Association for Computational Linguistics, 2007, pp. 858–867.

[19] S. F. Chen, J. Goodman, An empirical study of smoothing techniques for language modeling, Computer Speech & Language 13 (4) (1999) 359–394.

[20] I. H. Witten, T. C. Bell, The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression, IEEE Transactions on Information Theory 37 (4) (1991) 1085–1094.

[21] R. Kneser, H. Ney, Improved backing-off for m-gram language modeling, in: Proceedings of 1995 International Conference on Acoustics, Speech, and Signal Processing, ICASSP'1995, IEEE, 1995, pp. 181–184.

[22] M. Sundermeyer, R. Schluter, H. Ney, On the Estimation of Discount Para meters for Language Model Smoothing, in: Proceedings of the 12th Annual Conference of the International Speech Communication Association (INTERSPEECH 2011), 2011, pp. 1444–1447.

[23] B. Roark, C. Allauzen, M. Riley, Smoothed marginal distribution constraints for language modeling, in: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 2013, pp. 43–52.

[24] H. Zhang, D. Chiang, Kneser-ney smoothing on expected counts, in: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Baltimore, Maryland, 2014, pp. 765–774.

[25] M. Allamanis, C. Sutton, Mining source code repositories at massive scale using language modeling, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 207–216.

[26] R. Kohavi, A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2, AAAI, 1995, pp. 1137–1143.

[27] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, T. Ratchford, Automated API property inference techniques, IEEE Transactions on Software Engineering 39 (5) (2013) 613–637.

[28] Z. Tu, Z. Su, P. Devanbu, On the localness of software, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 269–280.

[29] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, S. Khudanpur, Recurrent neural network based language model., in: T. Kobayashi, K. Hirose, S. Nakamura (Eds.), INTERSPEECH, ISCA, 2010, pp. 1045–1048.

[30] J. Bobadilla, F. Ortega, A. Hernando, A. Gutiérrez, Recommender systems survey, Knowledge-Based Systems 46 (2013) 109–132.

[31] D. Hou, D. M. Pletcher, An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion, in: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, IEEE, 2011, pp. 233–242.

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

[33] N. Sahavechaphan, K. Claypool, XSnippet: Mining for sample code, ACM Sigplan Notices 41 (10) (2006) 413–430.

[34] R. Holmes, G. C. Murphy, Using structural context to recommend source code examples, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, 2005, pp. 117–125.

[35] A. Alnusair, T. Zhao, E. Bodden, Effective API navigation and reuse, in: Information Reuse and Integration (IRI), 2010 IEEE International Conference on, IEEE, 2010, pp. 7–12.

[36] S. Thummalapenta, T. Xie, PARSEWeb: a programmer assistant for reusing open source code on the web, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, 2007, pp. 204–213.