Journal of Information Processing Vol.27 469–478 (Aug. 2019)

[DOI: 10.2197/ipsjjip.27.469]

Regular Paper

Scope-aware Code Completion with Discriminative Modeling

Sheng Hu[1,†1,a)] Chuan Xiao[1,†2,b)] Yoshiharu Ishikawa[1,c)]

Received: December 9, 2018, Accepted: April 9, 2019

Abstract: Code completion is a traditional popular feature for API access in integrated development environments (IDEs). It not only frees programmers from remembering specific details about an API but also saves keystrokes and corrects typographical errors. Existing methods for code completion usually suggest APIs based on statistics in code bases described by language models. However, they neglect the fact that the user's input is also very useful for ranking, as the underlying patterns can be used to improve the accuracy of predictions of intended APIs. In this paper, we propose a novel method to improve the quality of code completion by incorporating the users' acronym-like input conventions and the APIs' scope context into a discriminative model. The users' input conventions are learned using a logistic regression model by extracting features from collected training data. The weights in the discriminative model are learned using a support vector machine (SVM). To improve the real-time efficiency of code completion, we employ a trie to index and store the scope context information. An efficient top-k algorithm is developed. Experiments show that our proposed method outperforms the baseline methods in terms of both effectiveness and efficiency.

Keywords: code completion, discriminative model, top-k ranking

1. Introduction

Code completion is a very useful feature for programmers, especially beginners, when they input long API names in integrated development environments (IDEs). It aims to help formulate accurate predictions for users' intended input APIs to save keystrokes and avoid typographical errors. This feature has become popular in prevalent IDEs for the following three reasons: First, according to the receiver object type, code completion can provide meaningful API method calls appearing in this object's definition, hence avoiding low-level incorrect API invocations. Second, if developers are not familiar with the APIs that should be called in their current context, code completion is able to present all possible completions in a pop-up window, providing an overall view and documentation to help beginners to learn programming patterns. Third, with code completion, developers are encouraged to use longer and more descriptive method names to improve code readability. We show an example of code completion in Example 1.

Example 1 Suppose a user wants to input an API name "SwingUtilities". He types the first few characters such as "**swin**", and then the system automatically suggests "**Swin**gUtilities" and "**Swin**gWorker".

We call the above problem setting *code completion for prefix-like input*. It receives a prefix-like input and returns a candidate API if the method name begins with the given prefix. Such setting is adopted in Ref. [23]. However, there is a major drawback of such a problem setting which makes it unpractical in some cases: when the candidate API set becomes larger, completion becomes less effective, especially when some prefixes are found to be shared by many API names. E.g., more than a hundred methods in JButton, a class of Java, begin with the prefix "get". To narrow down the candidate list, a user has to additionally type a longer prefix which significantly compromises the benefit of code completion.

To solve this problem, we need to find the intended API name by requiring a short input from the user to reduce the typing efforts. In this paper, we adopt an acronym-like input paradigm instead of the prefix-like one. An illustrating example is shown below.

Example 2 In **Fig. 1**, suppose the user types in characters such as "**swu**". Then the system adopting acronym-like matching paradigm suggests "**S**wing**U**tilities", "**S**et**W**rapG**u**idePainted", "**S**ho**wCu**rrentItem" and "**S**ho**wFu**llPath"[*1].

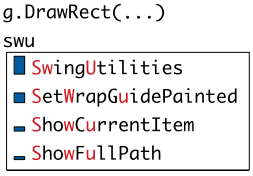

Fig. 1 Code completion for acronym-like input.

[1] Nagoya University, Nagoya, Aichi 464–8601 Japan
[†1] Presently with Kyoto University
[†2] Presently with Osaka University
[a)] hu@db.ss.is.nagoya-u.jp
[b)] chuanx@nagoya-u.jp
[c)] ishikawa@i.nagoya-u.ac.jp

[*1] Our method supports completions for both package names such as "SwingUtility" and API names such as "showFullPath". Without losing generality, we capitalize all the first characters of API names for the ease of illustration.

We call such a problem setting *code completion for acronym-like input*. It receives an acronym-like input and returns a candidate API if the method name contains all the characters of the input in a subsequence matching way [*2].

Existing solutions to code completion focus on using neural language models [4], [9], [13], [17], [22] or statistical language models [3], [16], [18], [19] learned from a large code base by modeling it into a natural language processing problem. However, they fail to utilize the user's input to narrow down the candidate list by proper relevance ranking. We argue that the acronym-like input from users will remarkably improve the completion accuracy when the underlying input patterns are taken into consideration. Intuitively, an acronym-like input from a user will be definitely affected by some acronym patterns due to human typing behavior. Take an example in Fig. 1. A user's input "swut" has a higher probability to match "**S**w**U**tilities" than to match "**S**ho**W**s**U**ll**P**a**t**h", because the acronym for the latter is hardly to be "swut" in common practice. In order to take advantage of the underlying patterns of human typing behavior, a transformation model needs to be learned for the purpose of providing high-quality candidates. In this work, we use a machine learning technique to learn the patterns to obtain a transformation model.

The scope context information [3], [16], [18], [19], which is described as the co-occurrences of APIs in a scope, is also found to be a helpful feature to improve the prediction accuracy. The reason is that there are many fixed API pattern flows in a scope for a specific utility. E.g., **Fig. 2** shows a typical API invoking pattern flow for graphics processing, while **Fig. 3** shows another case for video processing. Detecting such scope utility type can obviously improve the prediction accuracy when a new API invoking statement is inserted into the current scope.

To integrate different features, we propose a discriminative model which can assign different weights for each feature to achieve more accurate performance. This discriminative model consists of three features: 1) the API usage counts collected from the training corpus to reflect the popularity of each API, 2) the transformation probability that a user's input is transformed into the intended API, and 3) the scope context information represented by co-occurrence counts of APIs, for which a transformation model is learned by logistic regression. The above features are combined linearly in the discriminative model for the overall prediction, and their weights are learned by a support vector machine (SVM).

```
Graphics g = img.GetGraphics(...)
g.SetColor(...)
g.DrawRect(...)
g.FillRect(...)
g.DrawString(...)
SwingUtilities.InvokeAndWait(...)
painter.SetWrapGuidePainted(...)
```

Fig. 2 Scope for graphics utility.

```
VideoPlayer p;
p.SetCamera(...)
p.SetVideoSource(...)
p.SetAudioSource(...)
p.ShowFullPath(...)
p.ShowCurrentItem(...)
```

Fig. 3 Scope for video utility.

[*2] In our practical setting without losing generality, we assume that the first characters of an input and an API candidate must be exactly matched.

In addition to the prediction accuracy, we address the efficiency challenges when computing the top-k completions using our discriminative model. Specifically, we develop a candidate ranker framework to firstly generate the most possible k candidates and then use the ranker to re-rank the top-k results. We use a trie index to efficiently generate the possible candidates and then use inverted lists located on the trie's leaf nodes to store the scope context information for fast co-occurrence lookups.

Experiments are conducted with a large-scale training dataset collected from GitHub and a test set which covers 12 popular Java projects. The results demonstrate the effectiveness of our approach: it outperforms the baseline methods by up to 7.3% on top-1 accuracy. The experiments on efficiency show that our approach is faster than the baseline methods by up to 31 times.

To the best of our knowledge, this is the first work that focuses on improving ranking performance on the problem of code completion by utilizing the user's input. We also note that our method is orthogonal to the existing code suggestion methods [3], [16], [18], [19] because it can independently work as a standalone module after language model-based code suggestions.

Our main contributions are summarized as follows.

- We propose a novel method for code completion using acronym-like input.
- We propose a discriminative model that combines API counts, transformation probability, and scope context information for accurate code completion.
- We develop an efficient algorithm to compute top-k candidates from a trie index.
- Extensive experiments show the performance of both effectiveness and efficiency.

The rest of our paper is organized as follows: Section 2 presents the details of our discriminative model. Section 3 shows the index structure and candidate ranker framework. Section 4 reports the experimental results. Section 5 surveys related work. Section 6 concludes the paper.

2. A Ranker-based Model

In this section, we propose a discriminative model to rank the API candidates. The basic idea of our model is to combine three main features with a proper weighting for more accurate predictions. To account for efficiency challenges, we adopt a ranker-based model which consists of a candidate generator and an SVM-based ranker. First, the candidate generator uses a traditional noisy channel model to roughly pick up the most possible top-k API completions. Then, a carefully tuned SVM-based ranker will re-rank the top-k completions again and finally output a re-ranked completion list. An overview is showed in **Fig. 4**. Our process includes three steps:

(1) API names are indexed using a trie, with corresponding scope context information collected from our large training corpus. Each scope is assigned a unique scope ID and each API has a list of scope IDs such that this API appears in these

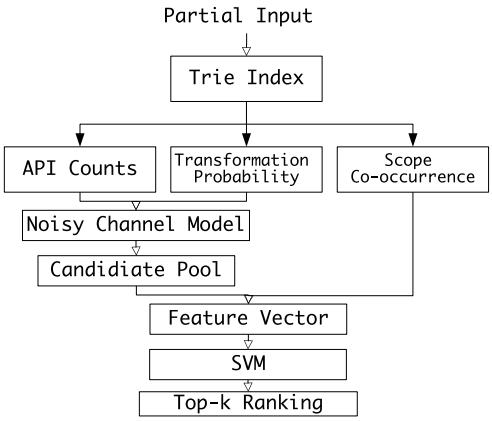

Fig. 4 Overview of the process.

scopes. Such lists of scope IDs are linked with the leaf nodes of our trie index for fast access.

(2) An SVM is trained using three features, which are API usage counts collected from our code base, transformation probability and scope co-occurrence counts. The transformation probability describes how likely the input from the user is the completion candidate and is trained by a logistic regression model.

(3) Search for API candidates from the trie by matching the user's input in a subsequence matching way. The previous s lines of context from current code position are taken as the scope context information. Then we use a traditional noisy channel model to roughly rank and output a top-k list as the candidate pool. Finally, we use the trained SVM to re-rank the top-k list to obtain an ultimate result list.

We first introduce a traditional noisy channel model and then give the definition of our discriminative model.

2.1 Noisy Channel Model

Noisy channel model is widely used in string transformation tasks, especially for spelling correction. Given an input $Q = q_1 \cdots q_{|Q|}$, we want to find the best transformed string $C = c_1 \cdots c_{|C|}$ among all candidates that match the input:

$$C* = \arg\max_C P(C|Q) \tag{1}$$

By applying Bayes' Rule and dropping the constant denominator, we have

$$C* = \arg\max_C P(Q|C)P(C) \tag{2}$$

where the transformation model $P(Q|C)$ models the transformation probability from C to Q, and the language model $P(C)$ models how likely C is the intended input. One problem with the noisy channel model is that there is no weighting for the two kinds of probabilities, and because they are often estimated from diverse sources, suboptimal performance might be incurred with regard to diversity of the sources [6], [7]. Moreover, noisy channel model cannot utilize additional useful features (e.g., scope context information), and this becomes a severe limitation in practice.

As our subsequence matching paradigm can be seen as a generalized case for string transformation, we can use noisy channel model directly to model our problem.

Table 1 Transformation model features *Sim*.

ID	Description
Sim_1	number of consonant letter matches
Sim_2	number of vowel letter matches
Sim_3	number of capital letter matches
Sim_4	number of letter skips
Sim_5	number of skipping occurs
Sim_6	percentage of letter matches

The language model $P(C)$ can be trained by simply counting the frequency of the API names in the code base, in line with many previous works [3], [5], [10], [27].

The transformation probability $P(Q|C)$ is learned using a logistic regression model and the training details are the same with the work [8]. The logistic regression model is shown below:

$$\begin{aligned} &P(Q|C) = g(\beta_0 + \beta_1 \cdot Sim_1(Q,C) + \cdots + \beta_6 \cdot Sim_6(Q,C)) \\ &\text{where} \quad g(z) = \frac{1}{1+e^{-z}} \end{aligned} \tag{3}$$

where β_i represents the regression coefficients, $Sim_i(Q,C)$ is the similarity feature showed in **Table 1** and $g(z)$ is the logistic function.

2.2 Discriminative Model

A discriminative model may overcome the shortcomings of noisy channel model by adding additional features and applying proper weightings. A general discriminative formulation of the problem is of the following form:

$$C* = \arg\max_C [\mathbf{w} \cdot \mathbf{F}(Q,C)] \tag{4}$$

where $\mathbf{F}(Q,C)$ is a vector of features and $\mathbf{w}$ is the model parameter which is a vector of weights. Compared with the noisy channel model, this discriminative formulation is more general. We can deem the noisy channel model as a special case of the discriminative form where only two features, the language model estimates and the transformation probability are used and uniform weightings are applied. In this work, the weightings $\mathbf{w}$ are derived by training an SVM showed in Section 2.4.

2.3 Scope-awareness

Scope context information has been proved to be very helpful in code suggestions, as it can describe which API methods are often invoked before the intended API method is called. We add the scope context variable in our discriminative model as the scope co-occurrence counts. It describes how often the candidate API appears with its previous API names in the same scope by collecting the statistics of the large training corpus. After adding in this feature, our discriminative model can be extended as:

$$\begin{aligned} C* = \arg\max_C [&w_0 + w_1 \cdot F_{lang}(Q,C) \\ &+ w_2 \cdot F_{trans}(Q,C) \\ &+ w_3 \cdot F_{scope}(Q,C)] \end{aligned} \tag{5}$$

where Q is the input, C is the candidate, $F_{lang}(Q,C)$ represents the unigram language model probability, which is calculated by a normalized usage counts, $F_{trans}(Q,C)$ represents the transformation probability from C to Q and $F_{scope}(Q,C)$ represents the scope co-occurrence counts of C and Q.

Table 2 Example candidates for input "swu".

Candidates	F_{lang}	F_{trans}	F_{scope}
SwingUtilities	0.6	0.7	0.5
SetWrapGuidePainted	0.2	0.2	0.1
ShowCurrentItem	0.2	0.1	0.1
ShowFullPath	0.1	0.1	0.1

2.4 An SVM-based Ranker

As only one candidate API is relevant to code completion (such setting was also adopted by many previous studies [9], [16], [17]), which is different from the traditional document retrieval problem, we only need to consider the possibility of an API as "hit" or "not hit". This can be essentially handled by a classification model such as a support vector machine. The one with higher possibility to be classified as "hit" will be ranked higher, and vice versa. We do not employ RankSVM [12] because usually detailed human-judged ranking data such as clickthrough data or log is not available. Our training SVM examples are generated from the noisy channel model, whose detailed rankings do not really matter. Hence they cannot be used as pair-wise training data in RankSVM.

The settings of our SVM are similar with Ref. [26]. The feature vectors are passed to a support vector machine employing a simple radial basis function (RBF) kernel with $\gamma = 1$ after all feature values are normalized between [0, 1]. We employed Joachim SVM^{light} [11], [26] implementation.

Our SVM model is trained on a training set comprising ⟨`API-candidate, feature-vector, class`⟩ triples. The `class` has two values: +1 and −1. +1 will be assigned if `API-candidate` is the intended API, otherwise −1 will be assigned. To describe how likely a candidate API is a "hit", the trained SVM will output a value between −1 and +1 for each candidate it generates. Then we can use these values to rank these candidates. This works because the value output by the SVM represents the distance to the maximum-margin hyperplane [26].

Table 2 gives an example to illustrate the features of candidates for an input "swu" from the user. The scope context information is extracted from Fig. 2 and Fig. 3. Note that each feature is a normalized value between [0, 1]. Suppose a user types an input "swu", then all the matched candidates are listed in Table 2 according to the subsequence matching paradigm. For example, "SwingUtilities" has an F_{lang} which is its usage counts in the code base, an F_{trans} which is $P(swu|SwingUtilities)$ value computed from the transformation model and an F_{scope} which is the scope co-occurrence value with its previous context "DrawRect" and "GetGraphics" showed in Fig. 1. These feature vectors are then passed into our trained SVM to give a final ranking list.

3. Searching Algorithm

In this section, we show the detailed algorithm and index implementation of the three steps showed in Section 2. There is a straightforward way to generate candidates: traverse the trie for all possible matching strings and then calculate all the probabilities for them and sort them in descending order. However, the number of all matching candidates might be prohibitive. Thus we adopt a ranker-based method which consists of a candidate generator and an SVM-based ranker. The candidate generator is responsible for only picking up the most possible top-k candidates and then SVM-based ranker will carefully rank these results for higher accuracy. Moreover, in the generator step, early termination techniques and a threshold-based algorithm are applied to efficiently compute top-k candidates.

Table 3 Example dataset S.

ID	String	Popularity
API_1	DrawRect	0.6
API_2	GetGraphics	0.8
API_3	SetAudioSource	0.2
API_4	SetCamera	0.1
API_5	SetColor	0.6
API_6	SetVideoSource	0.1
API_7	SetWrapGuidePainted	0.2
API_8	ShowCurrentItem	0.2
API_9	ShowFullPath	0.1
API_{10}	SwingUtilities	0.6

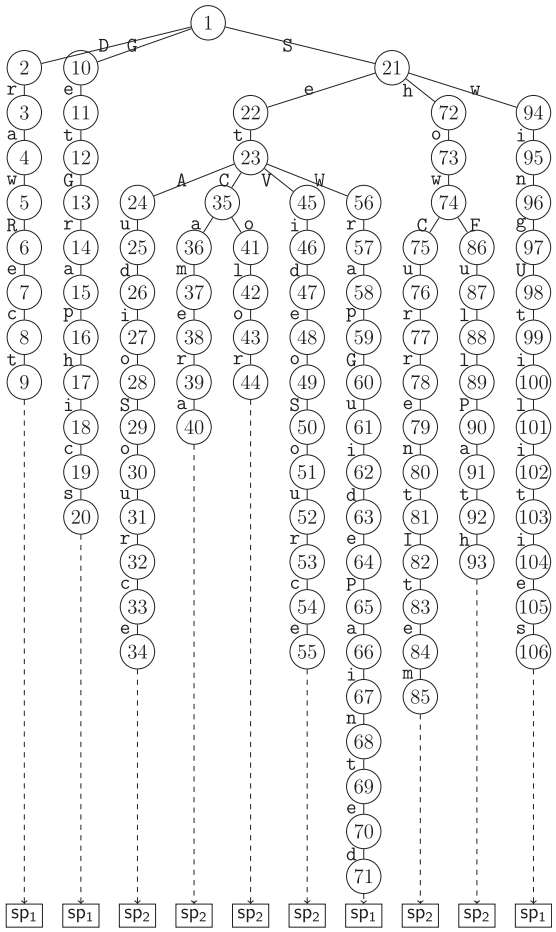

Fig. 5 A trie with scope inverted lists.

3.1 Candidate Generator of a Trie

The basic index structure of the candidate generator is a trie built on top of all the API names. Trie is a tree structure where each path from the root to a leaf node corresponds to a API name. Given an input, we can find its corresponding node, and traverse all its leaf nodes to obtain the corresponding API candidates. An example API set with popularities (normalized usage counts) is given in **Table 3** and the corresponding trie is showed in **Fig. 5**. The API "SwingUtilities" has a trie node 106. On each leaf node, we store the API usage counts in it and link it with an inverted

list of the ordered scope IDs. We call such an inverted list a *scope inverted list*. With the ordered scope inverted list, we can look for co-occurrences by simply doing a list intersection operation efficiently.

Our search strategy is based on two assumptions: First, the first characters of an input and an API candidate must be exactly matched. E.g., it is very unlikely that a user will input "sw" to look for "Ba**s**h**W**rite". Second, the user will not skip too many keywords in an acronym-like input. Such an assumption is made after the observation of the collected acronyms from Amazon Mechanical Turk. Thus, we only match the nodes whose distances to the current one are within next two keywords in the trie. E.g., "snt" can match **S**howCurre**nt**Item but cannot match **S**etWrapGuidePai**nt**ed, because in the trie, the next two keywords for `S` are `CurrentItem` and `WrapGuide`. Thus the keyword `Painted` cannot be matched.

The algorithm is illustrated in Algorithm 1. It takes as input the user's input, the scope context and the trie. The first characters of user input and candidate API are exactly matched (Line 5). Then it begins to traverse the trie to match each character of the input for each node's descendant nodes iteratively (Lines 1–10). This ensures it searches for all the API names in a subsequence matching way. Line 8 makes sure that the distance between the current node and next matching node will not be too long. We set the distance threshold τ as the distance to the end of next two keywords w.r.t the specific API names. After fetching all the matched API candidates, it intersects each of the candidates with the context API(s) to calculate co-occurrences, respectively (Lines 11–15). Especially when the scope context s consists of multiple lines of APIs, we take the union of the scope-lists for all APIs in context s without removing repeated scope IDs as s.scope-list. Finally, all the generated candidates will be sent to SVM for further ranking (Line 16).

Algorithm 1: Generator-Trie (q, s, T)

Input : q is the user input, s is the scope context, T is a trie built on S.
Output : $\{API_i\}$, such that $API_i \in S$ and q is a subsequence of API_i.

1 $A \leftarrow \{$ the root of $T\}$; /* node set */
2 **foreach** character $q[i]$ **do**
3 $A' \leftarrow \emptyset$;
4 **foreach** $n \in A$ **do**
5 **if** $q[i]$ is the first character in q AND n has a child n' through $q[i]$ **then**
6 $A' \leftarrow A' \cup \{n'\}$; /* first character */
7 **continue**;
8 **if** n has any descendant n' through character $q[i]$ within a distance threshold τ **then**
9 $A' \leftarrow A' \cup \{n'\}$;
10 $A \leftarrow A'$;
11 $R \leftarrow \emptyset$;
12 **foreach** $n \in A$ **do**
13 $R \leftarrow R \cup$ API candidates stored in the subtree rooted at n;
14 **foreach** API $\in R$ **do**
15 API.co-occurrence $\leftarrow$ API.scope-list Intersects s.scope-list;
16 **return** SVM-Ranking(R);

Take an example in Fig. 1, for an input "swu" and its scope context "DrawRect". One node that matches "swu" is node 98. One node that matches "DrawRect" is node 9. Node 9 is already a leaf node and node 98 has an only descendant leaf node 106. Both nodes have scope inverted lists of $\langle\, \mathsf{sp}_1 \,\rangle$ and $\langle\, \mathsf{sp}_1 \,\rangle$, respectively. After intersecting these two lists, we obtain the co-occurrence scope and the co-occurence count for "swu" and "DrawRect" is $\langle\, \mathsf{sp}_1 \,\rangle$ and 1, respectively.

3.2 Efficient Ranking Algorithm

We observe that with the increase of the scope context lines, the co-occurrence computation will need prohibitive intersection operations for the scope inverted lists of all the possible candidates. This cost can be unbearable for a real-time code completion system. To solve this issue, we first select the most possible candidates by using a noisy channel model showed in Section 2. Recall that this model needs to compute a product of $P(C)\cdot P(Q|C)$. $P(C)$ is stored at each leaf node in the trie. Hence we can materialize the maximum $P(C)$ at each intermediate node for a threshold algorithm (TA). $P(Q|C)$ is computed on-the-fly using a logistic function showed in Section 2. Note that as the logistic function is a monotonically increasing function, we can also obtain a maximum upper value of $P(Q|C)$ by only computing the longest prefix matched with the input. Moreover, as the input length of users mainly lies in the range of $[1, 6]$ (see Table 5), we do not consider off-line probability pre-computation methods, which practically do not improve the overall runtime by much but will result in large memory consumption.

In detail, when a user issues an input q, we send the q and previous scope context line(s) s to the search algorithm. We first search for s to obtain the scope inverted lists of the API results. Then we search q according to a subsequence matching manner in the trie. For each matched path in the trie, we compute the transformation probability by the logistic function in Eq. (3). Then we can use $P(C) \cdot P(Q|C)$ to compute a upper bound score UB. By using the UB, we can compute the rough top-k candidates efficiently by using a priority queue for early termination. After that, intersection operations are only done between scope context and top-k candidates. Feature vectors will be sent to our SVM for a final ranking.

The detailed algorithm is showed in Algorithm 2. To apply the pruning techniques, we simply use it to replace Lines 11–16

Algorithm 2: TopK-Pruning (q, A, k)

1 $R \leftarrow \emptyset$; /* a priority queue of size k */
2 **foreach** $n \in A$ **do**
3 **if** n.UB $\leq R[k].score$ **then**
4 **continue**;
5 **foreach** API as n's descendant leaf **do**
6 **if** $|R| < k$ **or** $score(API, q) > R[k].score$ **then**
7 $R.insert(API)$;
8 **foreach** API $\in R$ **do**
9 API.co-occurrence $\leftarrow$ API.scope-list Intersects s.scope-list;
10 **return** SVM-Ranking(R);

in Algorithm 1. A priority queue is initialized for early termination (Line 1). Then for each node, it iterates through the corresponding API candidates of each node (Line 5), compute the $P(API) \cdot P(q|API)$ as an overall score by $score(API, q)$ (Line 6). If the number of candidates in the priority queue is less than k or the candidate's score is greater than the k-th temporary API, we insert this API candidate into the queue (Lines 6–7). If the node's upper bound UB is less than the score of k-th temporary API, we skip this node for pruning (Lines 3–4). Eventually, we only need to do the intersection operations between k API candidates and the scope context, then send the k candidates to our SVM for further ranking (Lines 8–10).

The time complexity is $O(k|s| + \mathcal{N} \log k)$, reducing from the naïve one's $O(O|s| + O \log O)$, where k is the candidate pool size, $|s|$ is the context size, i.e., the number of context lines, $\mathcal{N}$ is the number of candidates scanned until the process terminated and O is the number of unique API names.

We show an example for the top-2 ranking process according to Table 2. Given the scope context "DrawRect" and an input "swu", we first search for "swu" in the trie and find node 61, 76, 87, 98. The maximum usage counts for them are 0.6, 0.2, 0.2, 0.1. The transformation probability is computed as $P(swu|SwingU) = 0.7$, $P(swu|SetWrapGu) = 0.2$, $P(swu|ShowCu) = 0.1$ and $P(swu|ShowFu) = 0.1$. Finally the noisy channel model probability is computed as $\mathsf{UB}_{\mathsf{SwingU}} = 0.6 \times 0.7 = 0.42$, $\mathsf{UB}_{\mathsf{SetWrapGu}} = 0.2 \times 0.2 = 0.04$, $\mathsf{UB}_{\mathsf{ShowCu}} = 0.2 \times 0.1 = 0.02$, $\mathsf{UB}_{\mathsf{ShowFu}} = 0.1 \times 0.1 = 0.01$. We only keep the top-2 API names, "SwingUtilities" and "SetWrapGuidePainted" remain. The intersections between scope inverted lists are done for both API names to compute co-occurrences. After that, both API names are sent to our SVM for the final ranking.

4. Experiments

We conducted extensive experiments to evaluate both effectiveness and efficiency of our proposed method against baseline methods. In this section, we report the experimental results and analyses.

4.1 Experiment Setup

Two datasets are collected for model training and testing tasks.

- **Java Corpus** [*3] is a large-scale code base collected from all Java projects on GitHub. We sampled the large corpus into the 1,000 projects **Java Corpus** to more clearly show the impact of training set size, in line with the previous works [15], [16]. We use this dataset as a training set in our evaluations. We use the API usage counts and scope context extracted from this code base.
- **Java Test** is a dataset used in Ref. [14] collected from 12 popular Java projects. We use this dataset as a test set for evaluations.

Table 4 shows the statistics of the two datasets, where LOCs is the line of codes, Files is the number of files and Total Projects is the number of projects included.

We only use APIs in Java Development Kit (JDK) as the dictionary to build the trie index. In total 17,116 APIs appearing in JDK 8 are collected to build the trie.

[*3] http://groups.inf.ed.ac.uk/cup/javaGithub/

Table 4 Dataset statistics.

Dataset	LOCs	Files	Total Projects
Java Corpus	10,753,168	86,158	1,000
Java Test	2,788,955	11,371	12

Table 5 Input length distribution.

length of input	1	2	3	4	5	6
average length of candidate API names	5.3	7.4	9.0	12.0	14.0	16.1
# of API names	17	62	259	354	139	52

We use the code base of **Java Corpus** as our corpus and use Eclipse's Java parser to parse the code for scope extractions. We tried different scope granularity such as class scope, method scope and block scope. Finally we choose to use class scope for the best balance between efficiency and accuracy. API usage counts are also extracted from **Java Corpus**.

To train the logistic regression model, we extract 5,000 API names, 4,000 from **Java Corpus** and 1,000 from **Java Test**. Then we use these API names to collect 5,000 acronym-like $\langle input, API \rangle$ pairs. The acronym-like input is collected from volunteers in Amazon Mechanical Turk, by telling them to intuitively give an input when they see an original API name. Our transformation model is trained on the complete training set using logistic regression model showed in Section 2 and well tuned by enough iterations.

Although the acronym-like input collected from Amazon Mechanical Turk has lengths ranging from 1 to 8, most of the input does not need to be fully typed to obtain its corresponding top-1 completions. E.g., the acronym-like input for "SwingUtility" we collected is "swut", but "SwingUtility" will be ranked as top-1 in our approach when "swu" is typed.

To quantitatively demonstrate that the completion is useful, we show in **Table 5** the statistics about the input length distribution for how many characters are needed when the corresponding API completion is ranked as top-1 in our approach. In this table, **length of input** means the necessary characters for its completion to be ranked as top-1, **average length of candidate API names** means the average length of such API names. **# of API names** means how many API names, out of the 1,000 collected ones, will be ranked as top-1 for the given length of input. E.g., the first column means that a total of 17 API names will be ranked as top-1 when the first letter is input, and their average length is 5.3. It can be seen that the length of input is significantly shorter than the average length of API names.

To train the SVM model, for each API name in the training set, we find an appropriate scope in **Java Corpus** for the API to fit in and extract its previous API names located in the scope context. Similarly, for each API in the test set, we also find an appropriate scope in **Java Test** and extract its previous API names as context for prediction purpose. Then we generate the candidates' feature vectors as $\langle candidate, input, context \rangle$, where *candidate* represents the candidate API name, *input* is the user input and *context* is the scope context API(s).

The following algorithms are compared.

- APIREC is a statistical model based approach in Ref. [15].

We carefully implement the method and use our own test set **Java Test** for evaluations.

- POP is the popularity-based sorting method used in Ref. [10].
- NCM is the noisy channel model method described in Section 2.
- SDM is our proposed scope-aware ranker-based method with discriminative modeling.

Note that APIREC does not utilize any input API name but provide suggestions when the user presses the "." button. In NCM and SDM, the candidate pool size is set to 50. That is, in SDM, the top-50 results are first selected and then passed to our SVM for further re-ranking. We have tried different values for the candidate pool size and 50 is proved to have a considerable processing time and does not lose any accuracy.

In addition, we extract the nearest *one* line prior to the current calling method as the scope context information in default for better performance unless explicitly stated otherwise. The reason is explained in Section 4.2.

The experiments were carried out on a PC with an Intel i5 2.6 GHz Processor and 8 GB RAM, running Ubuntu 14.04.3. The algorithms were implemented in C++ and in a main memory fashion.

4.2 Evaluation of Effectiveness

We adopt the same evaluation metrics used in existing studies [9], [16], [17]. We evaluate: (1) top-k accuracy, which indicates the fraction of times the correct API appears in the top-k candidates, where $k \in 1, 3, 5, 10$. (2) Mean Reciprocal Ranking (MRR), which is calculated as the average reciprocal of the correct API's rank in the top-k candidates. MRR can give an overall evaluation of the model. The closer to 1 the MRR value, the better the ranking accuracy.

We first evaluate the top-k accuracy with the baseline methods. **Table 6** shows the results. The last column shows the comparison of MRR. As seen, SDM achieves higher accuracy than any other baseline method. Without utilizing the user's input, APIREC can only reply on predictions by statistical models, thus causing low accuracy compared with the other methods. At top-1 accuracy, SDM has the largest improvements of 7.3%, 6.5%, 58.7%, over POP, NCM and APIREC, respectively. Along with the increase of k, the advantage becomes not that obvious because the correct API will be more easily included in a larger top-k list. Nonetheless, SDM still outperforms POP, NCM and APIREC on all the values of k. NCM is better than naïve POP approach, which suggests that the input from user is useful to predict the intended API names. However, the observed improvements over POP are very limited due to the lack of weights applied in NCM. We also observe that, SDM achieves the highest MRR of 0.928, meaning that on average in 10 cases, it can almost correctly rank the API on top of its list among 9 cases. The relative improvements on MRR is 3.3% and 1.9% over POP and NCM, respectively. These experimental results verified the significant improvements on accuracy of our proposed scope-aware discriminative model. We also observe that SDM has higher accuracy than APIREC on the column top-10, indicating that some rare API names which would hardly appear in statistical models can be retrieved in SDM by more strictly input matching.

Figure 6 shows an overall comparison for top-k accuracy by varying k from 1 to 30. The top-20 and top-30 accuracy for APIREC is 74.2% and 76.4%, respectively. Hence we do not show APIREC anymore because the conclusions are almost the same as Table 6. We can observe that SDM outperforms other two baseline methods at any $k \in [1, 30]$. The gap becomes close in $[20, 30]$ because for a large k, even the baseline method can include the correct candidate easily. Another valuable observation is that when $k = 30$, SDM is still higher than POP and NCM. This suggests that there are some API names especially with low frequency in the code base can never be retrieved by POP or NCM. As our SDM can consider the scope context and apply a proper weighting on it, these rare API names can be easily retrieved in our model.

Varying Context Line Size. Scope context size may have large impacts on the accuracy of our model. Here, we explain the scope context size as the *number of lines* prior to the current calling method within the same scope. The co-occurrence counts between each context line and current API candidate are summed as the feature *co-occurrence counts*. **Table 7** shows the results by varying the scope context size from 1 to 5. Interestingly, we can observe that with the increase of the context size, the accuracy slightly drops, the same as the MRR. We examined the test examples and found that summing up co-occurrence counts of multiple previous lines might over-weigh the co-occurrence feature in our model thus leading to inaccurate predictions. This fact reminds us that excessive context information might be not beneficial to accuracy but lead to deteriorative performances. There might be a way to take use of the multiple lines of context information more

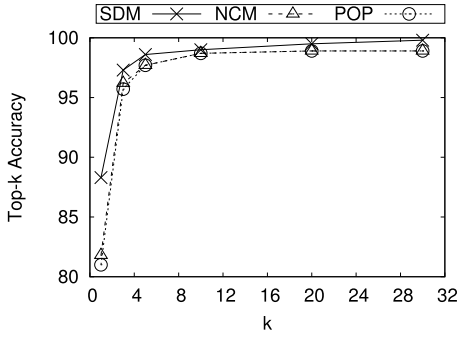

Fig. 6 Top-k accuracy of different approaches.

Table 6 Accuracy comparison.

Model	top-1	top-3	top-5	top-10	MRR
APIREC	29.6	43.4	58.2	70.2	0.407
POP	81.0	95.7	97.7	98.7	0.884
NCM	81.8	96.2	97.7	98.7	0.891
SDM	88.3	97.3	98.6	99.0	0.928

Table 7 Accuracy with different context size.

Context Size	top-1	top-3	top-5	top-10	MRR
1	88.3	97.3	98.6	99.0	0.928
2	85.5	97.3	98.6	99.0	0.913
3	85.5	97.1	98.6	99.0	0.912
4	85.5	97.1	98.4	99.0	0.912
5	85.5	97.1	98.4	99.0	0.912

Table 8 Accuracy of different training set size.

Dataset	top-1	top-3	top-5	top-10	MRR
Train100	83.5	96.1	97.6	98.7	0.898
Train300	85.7	96.4	98.1	98.7	0.911
Train1000	88.3	97.3	98.6	99.0	0.928

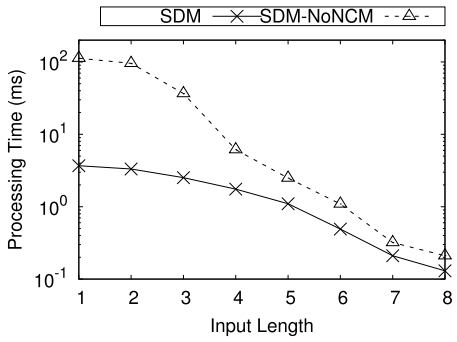

Fig. 7 Processing time comparison.

properly but such techniques are beyond the scope of this paper.

Varying Training Set Size. We also want to analyze the impacts on accuracy by varying the size of our training set. For our large-scale dataset **Java Corpus**, we randomly sample the 1,000 projects into two subsets, one with 300 projects and one with 100 projects, denoted by Train300 and Train100. Then we evaluate the accuracy by varying these three training sets. We show the results in **Table 8**. We can obviously see that Train1000 always outperforms Train100 and Train300. The largest improvements occur at top-1, are 4.8% and 2.6% over Train100 and Train300, respectively. Train300 is always better than Train100. This verified our intuition that a larger training set will contribute to better prediction accuracy.

4.3 Evaluation of Efficiency

A practical code completion system must be efficient enough to work in real-time to avoid interrupting a developer's flow of coding. Thus, for efficiency, we evaluate the overall processing times in **Fig. 7**. If not otherwise noted, the scope context size and training set will be set to 1 and Train1000 in default. In Fig. 7, we vary the input length and plot the average runtime of the code completion system as SDM. We also plot the results of the straightforward method mentioned in Section 3, denoted by SDM-NoNCM, meaning without the noisy channel model but sending all the matched API candidates to our SVM for ranking. For SDM, a larger k will cause better accuracy but longer processing time. By trying different k values, we choose $k = 50$ since it has the same accuracy with SDM-NoNCM but also runs efficiently. We can directly observe that SDM is much faster than SDM-NoNCM, because SDM uses a noisy channel model to drop hopeless candidates with extremely low probabilities to avoid prohibitive additional computations. The maximum speedup is 31 times, at length of 1. SDM-NoNCM is very slow due to the numerous matched API candidates which are needed to be passed to SVM given a short input while our SDM only keeps the most possible k API candidates for re-ranking. SDM is around 1 ms for all the lengths, and thus can be applied to Web settings, such as online IDEs. The times begin to decrease when we use a longer input because longer input is more selective thus less candidates are processed for both SDM and SDM-NoNCM.

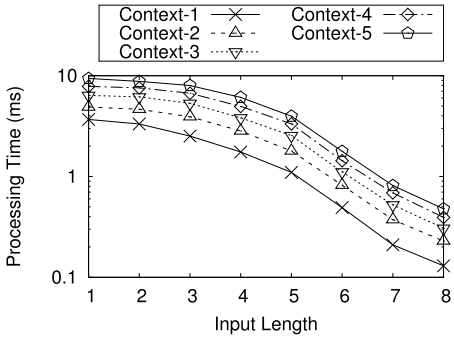

Fig. 8 Processing time with different context size.

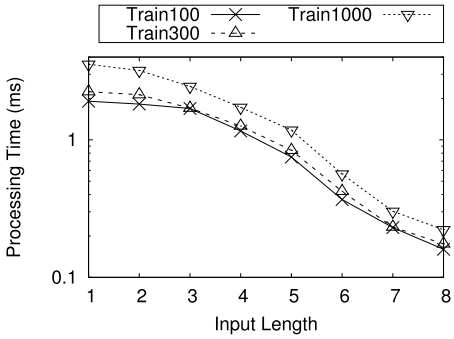

Fig. 9 Processing time with different training set size.

Varying Context Line Size. The processing times by varying scope context line size are compared. **Figure 8** shows the results by varying the context size from 1 to 5. As seen, a large context size might incur considerable overhead that causes sensible system delays. Context-5 is almost 3–4 times slower than Context-1. This suggests that if multiple lines of context information are used, further optimization on processing might be required.

Varying Training Set Size. The processing times by varying training set size are evaluated. For our large-scale dataset **Java Corpus**, we show the results in **Fig. 9**. The evaluated training set is the same with that in Table 8. Train1000 is the slowest, 1.8 and 1.5 times slower than Train100 and Train300, at the length of 1 while its corpus size is 10 and 3.3 times larger than Train100 and Train300. Intuitively, the larger the training corpus, the slower the processing time. This is mainly because larger training corpus will contain more scope context information such that the scope inverted list will become longer and slower for lookup operations. Nonetheless, the growth rate on processing time is much lower than the corpus size, and thus a larger corpus might be always preferable for better accuracy performance.

5. Related Work

Code Completion. With the birth of text editors, the research on code completion has received much attention in the past several decades. In one early study, Willis et al. [28] proposed an approach to expand some abbreviations into a sentence to save input efforts. In the domain of programming IDEs, Little and Miller [14] proposed to translate a small number of unordered keywords provided by the user into a valid expression in order to reduce the need to remember syntax and API names in Java. After this work, Han et al. [8] used a hidden Markov Model learn-

ing from a code corpus to expand ordered abbreviated keywords into a valid code expression. Their work was followed by Pini et al. [21] to additionally deal with the keyword missing problem. They used Support Vector Machine (SVM) to create classifiers to judge whether a keyword is an abbreviation or not. A recent study by Sandnes [24] developed a system to predict word input by only using a simple longest common subsequence algorithm for practical use.

Code Suggestion. The code suggestion problem is to try using statistical language models [3] (LMs) to predict the next code line without any input from users. We refer readers to two latest studies [1], [2] about source code naturalness. Nguyen et al. [19] proposed to use a semantic model to capture the patterns of source code, by incorporating a local semantic *n*-gram model with a global *n*-gram topic model. Graph-based LMs are proposed in Refs. [18] and [16] to capture graph-based patterns from source code. After that, Savchenko and Vokkov [25] also propose a probabilistic model with *n*-gram models to calculate a sorted list of all possible functions. Scope and context information has been proved to greatly improve the accuracy for predictions in these studies [3], [16], [18], [19]. Recent trends feature a boom by applying the Deep Learning Network (DNN) instead of LMs. However, while many studies [4], [13], [17], [22] have been developed to accommodate DNN in their code suggestion systems, a study [9] from Hellendoorn and Devanbu showed that carefully adapting *n*-gram models for source code can yield better performance than deep-learning models.

String Transformation. The string transformation problem is to map a source string *s* into another desirable form *t*. This problem has been extensively studied in the natural language processing community. A specific case for this problem is spelling correction. Okazaki et al. [20] proposed to use substring substitution rules as features in their discriminative models to generate transformed string candidates. Duan et al. [6] proposed a discriminative model based on latent structural SVM to model the alignment of words in the spelling correction process.

6. Conclusion

In this paper, we have studied the problem of code completion using a scope-aware ranker-based discriminative ranking model. We use an acronym-like input setting to avoid the fatal drawback of existing code completion systems. To improve the accuracy, we utilize API usage counts, transformation probability and scope context information as the features to pass to our trained SVM as a discriminative model. To solve the efficiency challenge, we adopt a ranker-based model to use noisy channel model as a filter to eliminate hopeless candidates. We have examined our approach with a training corpus and a test set. The experimental results have shown that our proposed method outperforms the existing methods in terms of both effectiveness and efficiency.

Acknowledgments This work was partly supported by JSPS KAKENHI Grant Number JP16H01722 and JP19K11979.

References

[1] Allamanis, M., Barr, E.T., Bird, C. and Sutton, C.A.: Learning natural coding conventions, *FSE-22*, pp.281–293 (online), DOI: 10.1145/2635868.2635883 (2014).

[2] Allamanis, M., Barr, E.T., Devanbu, P.T. and Sutton, C.A.: A Survey of Machine Learning for Big Code and Naturalness, *ACM Comput. Surv.*, Vol.51, No.4, pp.81:1–81:37 (online), DOI: 10.1145/3212695 (2018).

[3] Asaduzzaman, M., Roy, C.K., Schneider, K.A. and Hou, D.: CSCC: Simple, Efficient, Context Sensitive Code Completion, *ICSME 2014*, pp.71–80 (online), DOI: 10.1109/ICSME.2014.29 (2014).

[4] Bhoopchand, A., Rocktäschel, T., Barr, E.T. and Riedel, S.: Learning Python Code Suggestion with a Sparse Pointer Network, *CoRR*, Vol.abs/1611.08307 (2016) (online), available from ⟨http://arxiv.org/abs/1611.08307⟩.

[5] Duan, H. and Hsu, B.P.: Online spelling correction for query completion, *WWW 2011*, pp.117–126 (online), DOI: 10.1145/1963405.1963425 (2011).

[6] Duan, H., Li, Y., Zhai, C. and Roth, D.: A Discriminative Model for Query Spelling Correction with Latent Structural SVM, *EMNLP-CoNLL 2012*, pp.1511–1521 (2012) (online), available from ⟨http://www.aclweb.org/anthology/D12-1138⟩.

[7] Gao, J., Li, X., Micol, D., Quirk, C. and Sun, X.: A Large Scale Ranker-Based System for Search Query Spelling Correction, *COLING 2010*, pp.358–366 (2010) (online), available from ⟨http://aclweb.org/anthology/C10-1041⟩.

[8] Han, S., Wallace, D.R. and Miller, R.C.: Code completion of multiple keywords from abbreviated input, *ASE*, Vol.18, No.3-4, pp.363–398 (online), DOI: 10.1007/s10515-011-0083-2 (2011).

[9] Hellendoorn, V.J. and Devanbu, P.T.: Are deep neural networks the best choice for modeling source code?, *ESEC/FSE 2017*, pp.763–773 (online), DOI: 10.1145/3106237.3106290 (2017).

[10] Hou, D. and Pletcher, D.M.: An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion, *ICSM 2011*, pp.233–242 (online), DOI: 10.1109/ICSM.2011.6080790 (2011).

[11] Joachims, T.: Making large-Scale SVM Learning Practical, *Advances in Kernel Methods - Support Vector Learning*, Schölkopf, B., Burges, C. and Smola, A. (Eds.), MIT Press, Cambridge, MA, chapter 11, pp.169–184 (1999).

[12] Joachims, T.: Optimizing search engines using clickthrough data, *ACM SIGKDD 2002*, pp.133–142 (online), DOI: 10.1145/775047.775067 (2002).

[13] Li, J., Wang, Y., Lyu, M.R. and King, I.: Code Completion with Neural Attention and Pointer Networks, *IJCAI 2018*, pp.4159–4165 (online), DOI: 10.24963/ijcai.2018/578 (2018).

[14] Little, G. and Miller, R.C.: Keyword programming in Java, *ASE*, Vol.16, No.1, pp.37–71 (online), DOI: 10.1007/s10515-008-0041-9 (2009).

[15] Nguyen, A.T., Hilton, M., Codoban, M., Nguyen, H.A., Mast, L., Rademacher, E., Nguyen, T.N. and Dig, D.: API code recommendation using statistical learning from fine-grained changes, *FSE 2016*, pp.511–522 (online), DOI: 10.1145/2950290.2950333 (2016).

[16] Nguyen, A.T. and Nguyen, T.N.: Graph-Based Statistical Language Model for Code, *ICSE 2015*, pp.858–868 (online), DOI: 10.1109/ICSE.2015.336 (2015).

[17] Nguyen, A.T., Nguyen, T.D., Phan, H.D. and Nguyen, T.N.: A deep neural network language model with contexts for source code, *SANER 2018*, pp.323–334 (online), DOI: 10.1109/SANER.2018.8330220 (2018).

[18] Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J.M. and Nguyen, T.N.: Graph-based pattern-oriented, context-sensitive source code completion, *ICSE 2012*, pp.69–79 (online), DOI: 10.1109/ICSE.2012.6227205 (2012).

[19] Nguyen, T.T., Nguyen, A.T., Nguyen, H.A. and Nguyen, T.N.: A statistical semantic language model for source code, *ESEC/FSE '13*, pp.532–542 (online), DOI: 10.1145/2491411.2491458 (2013).

[20] Okazaki, N., Tsuruoka, Y., Ananiadou, S. and Tsujii, J.: A Discriminative Candidate Generator for String Transformations, *EMNLP 2008*, pp.447–456 (online), available from ⟨http://www.aclweb.org/anthology/D08-1047⟩ (2008).

[21] Pini, S., Han, S. and Wallace, D.R.: Text entry for mobile devices using ad-hoc abbreviation, *AVI 2010*, pp.181–188 (online), DOI: 10.1145/1842993.1843026 (2010).

[22] Raychev, V., Vechev, M.T. and Yahav, E.: Code completion with statistical language models, *PLDI '14*, pp.419–428 (online), DOI: 10.1145/2594291.2594321 (2014).

[23] Robbes, R. and Lanza, M.: How Program History Can Improve Code Completion, *ASE 2008*, pp.317–326 (online), DOI: 10.1109/ASE.2008.42 (2008).

[24] Sandnes, F.E.: Reflective Text Entry: A Simple Low Effort Predictive Input Method Based on Flexible Abbreviations, *DSAI 2015*, pp.105–112 (online), DOI: 10.1016/j.procs.2015.09.254 (2015).

[25] Savchenko, V. and Volkov, A.: Statistical Approach to Increase Source

Code Completion Accuracy, *PSI 2017*, pp.352–363 (online), DOI: 10.1007/978-3-319-74313-4_25 (2017).
[26] Schaback, J. and Li, F.: Multi-level feature extraction for spelling correction, *IJCAI-2007 Workshop on Analytics for Noisy Unstructured Text Data*, pp.79–86 (2007).
[27] Wang, Z., Xu, G., Li, H. and Zhang, M.: A Probabilistic Approach to String Transformation, *TKDE*, Vol.26, No.5, pp.1063–1075 (online), DOI: 10.1109/TKDE.2013.11 (2014).
[28] Willis, T., Pain, H., Trewin, S. and Clark, S.: Informing Flexible Abbreviation Expansion for Users with Motor Disabilities, *ICCHP 2002*, pp.251–258 (online), DOI: 10.1007/3-540-45491-8_52 (2002).

Sheng Hu is a research associate at the Graduate School of Informatics, Kyoto University and a Ph.D. candidate at the Graduate School of Information Science, Nagoya University. He received his B.E. degree from North China Electric Power University in 2013. His research interests include textual databases and spatio-temporal databases.

Chuan Xiao is an associate professor at the Graduate School of Information Science and Technology, Osaka University and a guest associate professor in Nagoya University. He received his B.E. degree from Northeastern University, China in 2005, and Ph.D. degree from The University of New South Wales in 2010. His research interests include data clearning, data integration, textual databases, and graph databases. He is a member of DBSJ.

Yoshiharu Ishikawa is a professor at the Graduate School of Informatics, Nagoya University. He received B.S., M.E., and Dr. Eng. degrees from University of Tsukuba in 1989, 1991, and 1995, respectively His research interests include spatio-temporal databases, mobile databases, sensor databases, data mining, information retrieval, and e-science. He is a member of ACM, DBSJ, IEEE, IEICE, IPSJ, and JSAI.