Scope-aware Code Completion with Discriminative Modeling

Sheng HU†, Chuan XIAO††, and Yoshiharu ISHIKAWA†

† Graduate School of Informatics, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464–8601 Japan
†† Institute for Advanced Research, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464–8601 Japan
E-mail: †hu@db.ss.is.nagoya-u.ac.jp, ††chuanx@nagoya-u.ac.jp, †††ishikawa@i.nagoya-u.ac.jp

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.

Key words 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 becomes 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] In Fig. 1, a user wants to input an API name "SwingUtilities". He types the first few characters such as "**swin**", and then the system automatically suggests

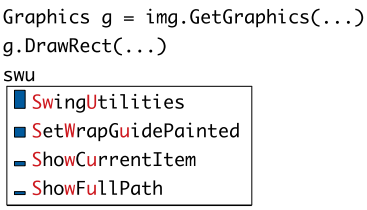

Figure 1 Code completion for acronym-like input

"**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 [11]. 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

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

Figure 2 Scope for graphics utility

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

Figure 3 Scope for video utility

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**w**W**ing**U**tilities", "**S**et**W**rapG**u**idePainted", "**S**how**C**u**r**rentItem" and "**S**ho**w**-F**u**llPath"(注1).

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(注2).

Existing solutions to code completion focus on using neural language models [6,7] or statistical language models [1,8–10] 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**W**ing**U**t**i**lities" than to match "**S**ho**w**F**u**llPa**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 [1,8–10], 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).

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 with the existing code suggestion methods [1,8–10] 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

(注1) : Without losing generality, we capitalize all the first characters of API names for the ease of illustration.

(注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.

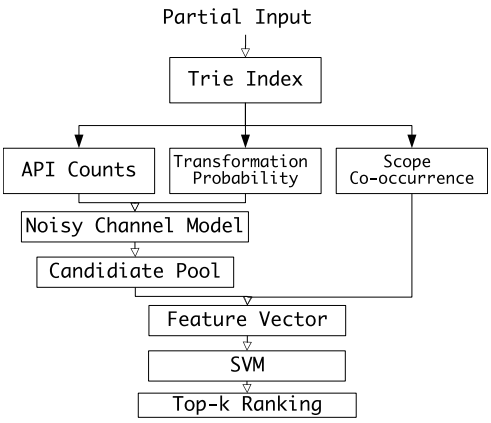

Figure 4 Overview of the process

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 with a unique scope ID and each API has a list of scope IDs such that this API appears in these 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 regarding to diversity of the sources. Moreover, noisy channel model cannot utilize additional useful features (E.g., scope context information) which 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.

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 [1,4].

The transformation probability $P(Q|C)$ is learned using a logistic regression model and the training details are the same with the work [2]. The logistic regression model is shown below:

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

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.

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`

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 nosiy 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 a 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.

2 4 An SVM-based Ranker

The feature vectors are passed to a support vector machine employing a simple radial basis function (RBF) kernel with $r = 1$ after all feature values are already normalized between 0 and 1. We employed Joachim SVM^{light} [5] implementation.

We train the SVM model on a training set that consists of ⟨`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. When a candidate is predicted, the trained SVM outputs a value between –1 and +1. Then we can use these values to rank the candidates.

3 Searching Algorithm

Table 2 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

In this section, we show the detailed algorithm and index implementation of the three steps showed in Section 2. Although 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.

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 usa ge counts) is given in Table 2 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 inverted list *scope inverted list*. With the ordered scope inverted list, we can look up for co-occurrences by simply doing a list intersection operation efficiently.

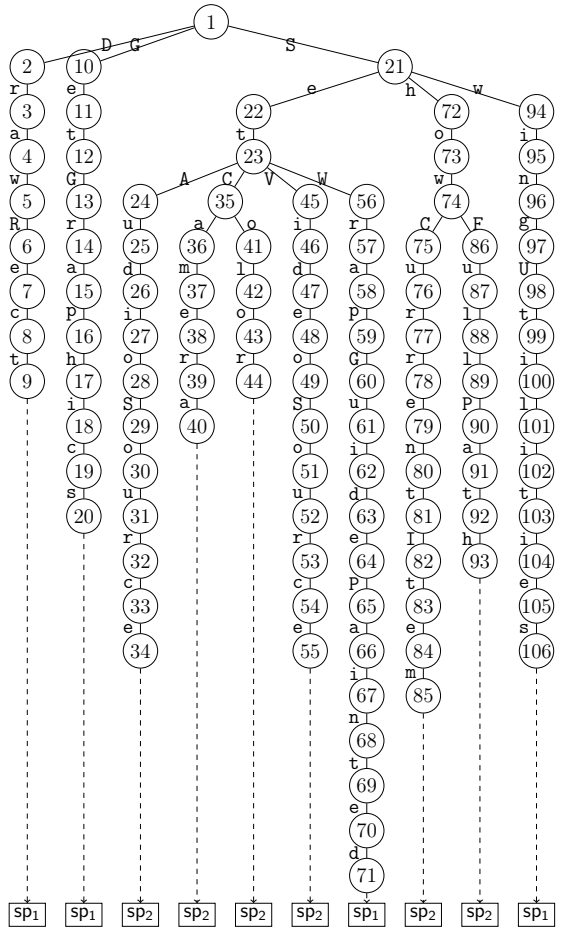

Figure 5 Trie with scope inverted lists

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.

Table 3 Dataset statistics

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

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 Equation 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 time complexity is $O(k|s| + \mathcal{N} log k)$, reducing from the naïve one's $O(\mathcal{O}|s| + \mathcal{O} log \mathcal{O})$, 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 $\mathcal{O}$ is the number of unique API names.

4 Experiments

In this section, we conducted extensive experiments to evaluate both effectiveness and efficiency of our proposed method against baseline methods.

4 1 Experiment Setup

Two datasets are collected for model training and testing tasks.

- **Java Corpus**(注3) is a large-scale code base collected from all Java projects on GitHub. We sample the large corpus into 1,000 projects **Java Corpus** 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 [2] collected from 12 popular Java projects. We use this dataset as a test set for evaluations.

Table 3 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.

We use the code base of **Java Corpus** as our corpus and

(注3) : http://groups.inf.ed.ac.uk/cup/javaGithub/

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.

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.

- POP is the popularity-based sorting method used in [4].
- 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 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 well balance the trade-off between processing time and 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 other explicit statement is made. The reason is explained in Section 4 2.

The experiments were carried out on a PC with an Intel i5 2.6GHz Processor and 32GB 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 [3, 7, 8]. 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.

Table 4 Accuracy comparison

Model	top-1	top-3	top-5	top-10	MRR
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

We first evaluate the top-k accuracy with the baseline methods. Table 4 shows the results. The last column shows the comparison of MRR. As seen, SDM achieves higher accuracy than any other baseline method. At top-1 accuracy, SDM has the largest improvements of 7.3%, 6.5%, over POP and NCM, 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 and NCM 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.

Fig. 6 shows an overall comparison for top-k accuracy by varying k from 1 to 30. We can observe that SDM outperforms other two baseline methods at any $k \in [1, 30]$. The advantage 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

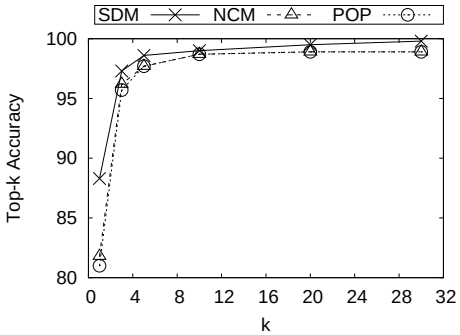

Figure 6 Top-k accuracy of different approaches

candidate are summed as the feature *co-occurrence counts*. Table 5 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 lead 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 properly but such techniques are beyond the scope of this paper.

Table 5 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

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 6. 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.

Table 6 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

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 clarified, 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 send all the matched API candidates to our SVM for ranking. We can directly observe that SDM is much faster than SDM-NoNCM, because SDM uses a noisy channel model to drop hopeless candidates with extreme 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 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.

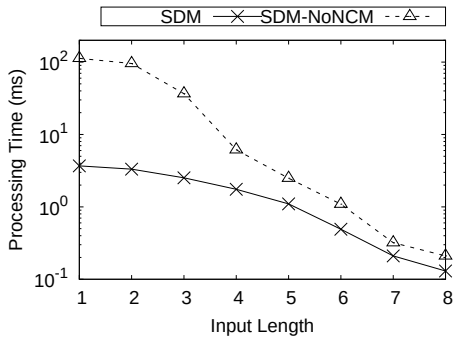

Figure 7 Processing time comparison

Varying Context Line Size. The processing times by varying scope context line size are compared. Fig. 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 6. 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.

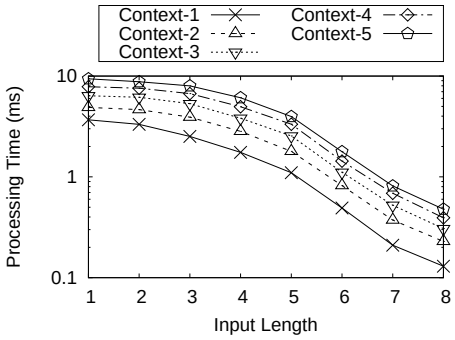

Figure 8 Processing time with different context size

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, thus a larger corpus might be always preferable for better accuracy performance.

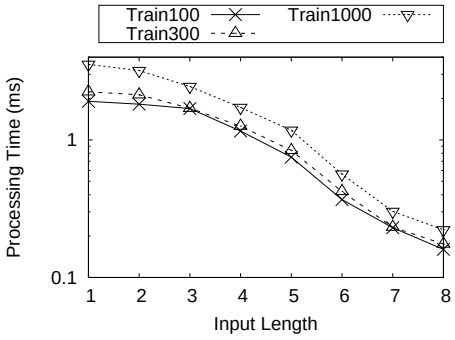

Figure 9 Processing time with different training set size

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.* [12] proposed an approach to expand some abbreviations into a sentence to save input efforts. After this work, Han *et al.* [2] used a hidden Markov Model learning from a code corpus to expand ordered abbreviated keywords into a valid code expression.

Code Suggestion. The code suggestion problem is to try using statistical language models [1] (LMs) to predict the next code line without any input from users. Nguyen *et al.* [10] 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 [8] to capture graph-based patterns from source code. Scope and context information has been proved to greatly improve the accuracy for predictions in these studies [8,10]. Recent trends feature a boom by applying the Deep Learning Network (DNN) instead of LMs.

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. The experimental results have shown that our proposed method outperforms the baseline methods in terms of both effectiveness and efficiency.

acknowledgment

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

References

[1] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: simple, efficient, context sensitive code completion. In *ICSME 2014, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 71–80, 2014.

[2] S. Han, D. R. Wallace, and R. C. Miller. Code completion of multiple keywords from abbreviated input. *ASE*, 18(3-4):363–398, 2011.

[3] V. J. Hellendoorn and P. T. Devanbu. Are deep neural networks the best choice for modeling source code? In *ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 763–773, 2017.

[4] D. Hou and D. M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 233–242, 2011.

[5] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11, pages 169–184. MIT Press, Cambridge, MA, 1999.

[6] J. Li, Y. Wang, M. R. Lyu, and I. King. Code completion with neural attention and pointer networks. In *IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 4159–4165, 2018.

[7] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen. A deep neural network language model with contexts for source code. In *SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 323–334, 2018.

[8] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 858–868, 2015.

[9] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 69–79, 2012.

[10] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 532–542, 2013.

[11] R. Robbes and M. Lanza. How program history can improve code completion. In *ASE 2008, 15-19 September 2008, L'Aquila, Italy*, pages 317–326, 2008.

[12] T. Willis, H. Pain, S. Trewin, and S. Clark. Informing flexible abbreviation expansion for users with motor disabilities. In *ICCHP 2002, Linz, Austria, July 15-20, Proceedings*, pages 251–258, 2002.