# Effective API Recommendation without Historical Software Repositories

Xiaoyu Liu
Department of Computer Science and Engineering, Southern Methodist University
Dallas, Texas, USA
xiaoyul@smu.edu

LiGuo Huang
Department of Computer Science and Engineering, Southern Methodist University
Dallas, Texas, USA
lghuang@smu.edu

Vincent Ng
Human Language Technology Research Institute, University of Texas at Dallas
Richardson, Texas, USA
vince@hlt.utdallas.edu

## ABSTRACT

It is time-consuming and labor-intensive to learn and locate the correct API for programming tasks. Thus, it is beneficial to perform API recommendation automatically. The graph-based statistical model has been shown to recommend top-10 API candidates effectively. It falls short, however, in accurately recommending an actual top-1 API. To address this weakness, we propose RecRank, an approach and tool that applies a novel ranking-based discriminative approach leveraging API usage path features to improve top-1 API recommendation. Empirical evaluation on a large corpus of (1385+8) open source projects shows that RecRank significantly improves top-1 API recommendation accuracy and mean reciprocal rank when compared to state-of-the-art API recommendation approaches.

## CCS CONCEPTS

• **Software and its engineering → API languages**; *Software maintenance tools*;

## KEYWORDS

API Recommendation, Machine Learning

**ACM Reference Format:**
Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective API Recommendation without Historical Software Repositories. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238216

## 1 INTRODUCTION

During daily software development, Application Programming Interfaces (APIs) are provided as functional building blocks to program software systems. APIs are classes, methods, and fields provided by the library's designers [25] to enable developers to access the functionality of a code library. However, developers need to spend a lot of effort on familiarizing themselves with the capabilities provided by a large number of APIs in the library and pick the correct API for development tasks. For instance, developers need to manually browse a long list of APIs to identify *BufferedWriter.write()*, the API that enables them to efficiently write to a file by buffering the characters in Java memory. As another example, developers have to choose from a list of 67 candidate member methods of *String* in the Java Development Kit (JDK) to identify the appropriate API for converting all of the corresponding characters to upper case (i.e., *String.toUpperCase()*). To address this challenge, many automated API recommendation approaches and tools have been proposed to relieve the burden of developers in understanding and locating APIs, either by taking advantage of API usage patterns [4, 10, 14, 33] or by using statistical learning to recommend the next token [6, 16, 23, 24]. For instance, *Gralan* uses a statistical language model for API recommendation that relies on features extracted from the preceding context (i.e., the code that has been written so far). The model was trained by collecting statistics on how often a candidate API co-occurs with the APIs in its preceding context [24]. Being a generative model, however, *Gralan* is sensitive to the presence of *overlapping* features and *irrelevant* features. Specifically, if two features encode overlapping information (e.g., two features are computed based on the same API in the preceding context), it will undesirably amplify the importance of this API in the prediction process, thus possibly harming model performance. Irrelevant features (i.e., features that are largely not predictive of the target API), too, could be harmful: while the statistics collected during the training process could to some extent indicate whether a feature is relevant, the multiplicative effect resulting from a large number of irrelevant features in a generative model could overwhelm the positive effect of the relevant features, again harming model performance. Hence, feature engineering is important when employing generative models. Unfortunately, as we will see in the next section, a number of features that *Gralan* employs are by design both overlapping and irrelevant.

More recently, *APIREC* [23], a state-of-the-art API recommendation approach, was proposed by *Gralan*'s authors. *APIREC* makes a key assumption: changes that serve the same higher-level intent of the developers will co-occur more frequently than non-related changes [23]. Hence, by leveraging the regularity and repetitiveness of software changes of a software system, *APIREC* can identify and focus on changes/features that are relevant to API recommendation, thereby reducing the impact of the feature irrelevance problem mentioned earlier. Nevertheless, the applicability of *APIREC* is severely limited by the large number of historical software change


*ASE '18, September 3–7, 2018, Montpellier, France*

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238216

repositories it requires. Specifically, not only does it need to be *trained* on 471,730 changed source code files and 113,103 change commits, but it can only be *applied* to files with a similarly long change history.

Our goal in this paper is to advance the state-of-the-art in API recommendation, specifically by improving the top-1 API recommendation accuracy. In view of the aforementioned limitations of *APIREC*, we desire an approach that does *not* rely on code change history. The design of our system, RecRank, is motivated by a key observation: while *Gralan* is unable to achieve a high top-1 recommendation accuracy, it achieves a reasonably high top-10 recommendation accuracy (73.4–80.6%). Given this observation, we take the top-10 API candidates identified by *Gralan* as our starting point and *re-rank* these candidates so that the correct API surfaces to the top of the list. The question, then, is: how should we re-rank? Recall that a key weakness of *Gralan* concerns the use of a generative model, which is sensitive to the presence of overlapping and irrelevant features. RecRank is specifically designed to address this weakness. First, RecRank employs a *discriminative* re-ranker that is trained to re-rank *Gralan*'s top-10 candidate APIs. The key advantage of a discriminative approach (over a generative approach) is that the former can automatically discriminate relevant from irrelevant features (by assigning high weights to the relevant ones and low weights to the irrelevant ones). Second, we propose a novel kind of features for use in conjunction with our discriminative re-ranker, *API usage path* based features. These features partially address the feature irrelevance problem and can arguably better capture the linguistic topic of the program expressing the intention of the developer.

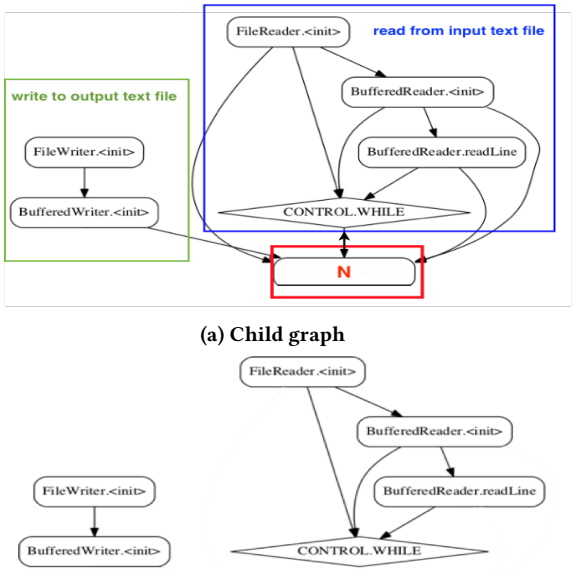
In sum, our contribution in this paper lies in the proposal of RecRank, a novel discriminative ranking approach that employs a novel kind of features based on usage paths to automatically recommend top-1 APIs based on the top-10 API candidates suggested by *Gralan*. In an evaluation on eight large-scale open source projects, RecRank outperforms *APIREC* with respect to two evaluation metrics, top-1 recommendation accuracy and mean reciprocal rank (MRR), a commonly used metric for evaluating ranking tasks in information retrieval, achieving state-of-the-art results.

## 2 PRELIMINARIES AND MOTIVATING EXAMPLES

### 2.1 Graph-based Generative API Recommendation (Gralan)

Since RecRank is built upon the top-10 API candidates suggested by *Gralan*, we will provide an overview of *Gralan* in this subsection.

As mentioned before, given a *recommendation point*, *Gralan* recommends an API using its *preceding context* (i.e., the code that has been written so far). [1] *Gralan* encodes the preceding context as a set of *API usage graphs*. In an API usage graph, each node is an API used in a method call, operator overloading, field access or branching (e.g., *if, while, for*, etc.). All API nodes are connected by directed edges. Each edge represents a data flow dependency (i.e., overloading operator, method calls, and field accesses) or a control flow dependency (i.e., condition and repetition) between two APIs.




**(a) Child graph**

**(b) Parent graph**

**Figure 1: Parent-child graph example**

An example of an API usage graph is shown in Figure 1(a), where node N is the recommendation point. The corresponding context graph (i.e., the graph that encodes the context in which N occurs) is shown in Figure 1(b). As can be seen, this context graph is created by removing node N as well as all of its incoming and outgoing edges. Throughout the paper, if two graphs (e.g., the ones shown in Figure 1) have a parent-child relationship, we will refer to the one without the recommendation point as the parent graph and the one with the recommendation point as the corresponding child graph.

As mentioned before, *Gralan* uses the parent graph for predicting the API at the recommendation point. One way to make use of the parent graph is to estimate the probability that a candidate API co-occurs with the parent graph in the training data. The higher the co-occurrence probability is, the more likely that the candidate API is the correct API. However, a parent graph (such as the one shown in Figure 1(b)) could be fairly complex. Complex parent graphs could yield a data sparsity problem: the more complex a parent graph is, the less likely it will be seen in the training data. To alleviate data sparsity, *Gralan* also makes use of all the (non-empty) subgraphs of the parent graph in the API prediction process. For instance, from the parent graph in Figure 1(b), we can extract subgraphs with one API (e.g., *CONTROL.WHILE*), subgraphs with two APIs (e.g., [*FileWriter.<init>*, *CONTROL.WHILE*]), subgraphs with three APIs (e.g., [*FileWriter.<init>*, *BufferedReader.<init>*, *CONTROL.WHILE*]), and subgraphs with four APIs (e.g., [*FileWriter.<init>*, *BufferedReader.<init>*, *BufferedReader.readLine*, *CONTROL.WHILE*]).

Specifically, given a parent graph $g$ and subgraphs $g_1, \ldots, g_n$ of $g$, *Gralan* computes the probability of a child graph, $C(g)$, which is

[1] The reason that only the *preceding* context is used is to mimic the realistic situation that when an API is to be recommended to a developer, only the code that has been written so far is available.

created by filling the recommendation point with a candidate API, using Bayesian statistical inference as follows:

$$\begin{aligned}
&\log(Pr(C(g)|g_1, g_2, ..., g_n, g)) \\
&\propto \log(Pr(g_1|C(g))...Pr(g_n|C(g))Pr(C(g))) \\
&= \sum_{j=1}^{n} \log(\#methods(g_j, C(g)) + \alpha) \\
&+ \log(\#methods(g, C(g))) \\
&- (n-1)\log(\#methods(C(g)) + \alpha\#methods) \\
&- \log(\#methods(g))
\end{aligned} \tag{1}$$

where the expression in the second line is obtained using Bayes rule, and the third line shows how the probabilities in the second line can be estimated. Specifically, $\#methods(g, C(g))$ is the number of times $g$ appears as the parent of $C(g)$ in the training data, $\#methods(g)$ is the number of times $g$ appears in the training data, and $\#methods$ is the total number of methods in the training data.[2] To avoid floating underflow, Logarithm (log) is applied to all the probabilities in the equation. To assign non-zero probabilities to events not seen in the training data, a smoothing factor (i.e., $\alpha$) is used. Note that each of the graphs being conditioned on in Equation 1 (i.e., $g, g_1, \ldots, g_n$) can be viewed as a feature used by *Gralan* in the recommendation process. Because the $g_i$'s are subgraphs of $g$, these features are by design *overlapping*, which could harm the performance of a generative model like *Gralan*, as noted in the introduction.

There is another caveat. Recall that Figure 1(a) only shows *one* of the many API usage graphs that *Gralan* generates for the recommendation point N. The exact number of API usage graphs that *Gralan* generates for a recommendation point depends on a parameter, $d$, which specifies the maximum distance between the recommendation point and any of the nodes in an API usage graph. For instance, if $d$=3, *Gralan* will generate *all* API usage graphs that can possibly be generated by including any subset of nodes whose distance is no larger than 3 from the recommendation point. For each of these API usage graphs, *Gralan* generates the corresponding parent graph. Given each parent graph $g$ (and its subgraphs $g_1, \ldots, g_n$), *Gralan* computes the probability of each child graph $C(g)$ using Equation (1). The candidate API that corresponds to the most probable child graph over all the parent graphs will be the API recommended by *Gralan* for a given recommendation point.

## 2.2 Motivating Example

We motivate the development of RecRank through the following example. A developer is developing a software function to read text from a .txt file ("input.txt") and write the processed text to another .txt file ("output.txt"). The code snippet is shown in Figure 2, in which the input text file "input.txt" is read using Java Development Kit (JDK) API *BufferedReader* (line 6) and written to "output.txt" using JDK API *BufferedWriter* (line 7). A while loop is used to iteratively read each line of the input text file (line 11). Now this developer needs to decide what API should be used in line 12 to write to the "output.txt" file. Modern Integrated Development Environment (IDE) tools, such as Eclipse, provide a list of

[2]Our training data is composed of the set of API usage graphs generated from all the methods in the source code collected from 1385 open source projects (see Section 4.1 for details).

```
5   public void readAndWrite() throws IOException{
6       BufferedReader in = new BufferedReader(new FileReader("Input.txt"));
7       BufferedWriter out = new BufferedWriter(new FileWriter("Output.txt"));
8
9       String i = null;
10      char c;
11      while((i = in.readLine())!=null){
12
13      }
14  }
```

**Figure 2: A code snippet**

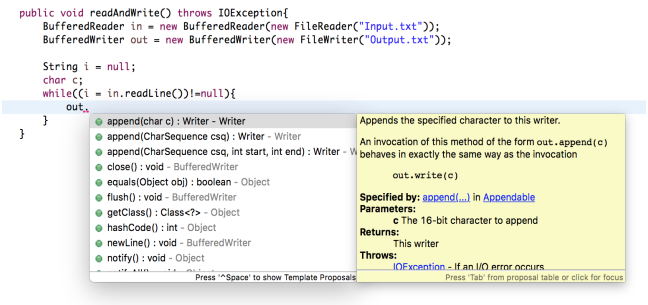



**Figure 3: An API recommendation example from Eclipse**

APIs for developers to choose. This list of methods and fields is usually ranked in alphabetical order since it simply shows all member methods/fields of the calling API. Figure 3 shows that Eclipse recommends 16 APIs for line 12 in Figure 2. Note that these member methods and fields of the calling API *BufferedWriter* are *not* prioritized based on relevance: they are simply listed in alphabetical order.

Since the developer still cannot decide which API to choose from the list recommended by IDE tools, she would like to ask for help from *Gralan*. As aforementioned, *Gralan* ranks candidate APIs by the probabilities of the corresponding child graphs given a parent graph and its subgraphs. Specifically, it starts by building a set of API usage graphs (such as the one shown in Figure 1-a) of the code snippet in Figure 2. For each of the API usage graphs, *Gralan* extracts the corresponding parent graph and its subgraphs. These context graphs are then used in calculating the probability of each candidate API using Equation (1). However, not all context graphs are relevant to the recommendation point. In other words, not all context graphs implement the same linguistic topics as that of the recommendation point. For example, the recommendation point N in Figure 1(a) implements the linguistic topic "write to output text file" with its context graph in the green rectangle, while its context graph in the blue rectangle implements the linguistic topic "read from input text file". However, based on Equation (1) this context graph is considered as important as other context graphs: like other generative models, the one employed by *Gralan* merely multiplies the probabilities associated with the parent graph and all of its subgraphs.

Table 1 shows a few examples of the parent graphs (i.e., $g$) and their corresponding child graphs (i.e., $C(g)$) for the code snippet in Figure 2 as well as the probability (i.e., score) of each child graph. The scores of the child graphs over all of the parent graphs are

**Table 1: Probability scores of candidate APIs**

| $g$ | C($g$) | Candidate API | Score |
|---|---|---|---|
| *FileReader.<init>, BufferedReader.-<init>, BufferedReader.readLine, CONTROL.WHILE* | *FileReader.<init>, ..., **BufferedReader.close*** | ***BufferedReader.-close*** | 0.33 |
| | *FileReader.<init>, ..., CONTROL.WHILE, **BufferedWriter.write*** | ***BufferedWriter.-write*** | 0.15 |
| | ... | ... | ... |
| *BufferedWriter.-<init>, CONTROL.WHILE* | *BufferedWriter.-<init>, CONTROL.WHILE, **BufferedWriter.write*** | ***BufferedWriter.-write*** | 0.25 |
| | *BufferedWriter.-<init>, CONTROL.WHILE, **BufferedReader.close*** | ***BufferedReader.-close*** | 0.02 |
| | ... | ... | ... |
| *CONTROL.WHILE* | *CONTROL.WHILE, **BufferedReader.close*** | ***BufferedReader.-close*** | 0.1 |
| | *CONTROL.WHILE, **BufferedWriter.write*** | ***BufferedWriter.-write*** | 0.05 |
| | ... | ... | ... |
| ... | ... | ... | ... |

calculated and sorted. As we can see in Table 1, even though API *BufferedWriter.write* is the correct API for the recommendation point N, *Gralan* recommended *BufferedReader.close* since it has the highest score (i.e., 0.33). The main reason behind this miss is that *BufferedReader.close* co-occurred more frequently with the irrelevant context graph in the blue rectangle in Figure 1, which implements the linguistic topic "read from input text file" rather than the topic that corresponds to the developer's intent, "write to output text file". Note that this is just *one* example of an *irrelevant* feature employed by *Gralan*: because of the way parent graphs are generated for a recommendation point, many of them (as well as the subgraphs generated from them) are irrelevant. Together with the overlapping features, these irrelevant features could harm *Gralan*'s performance.

Then the developer decides to try a state-of-the-art approach, *APIREC* [23]. The key idea behind *APIREC* is to leverage the regularity and repetitiveness of API usage patterns learned from software change history. It assumes that the changes that serve the same higher-level intent will co-occur more frequently than unrelated changes [23]. In other words, those APIs in the context graphs that have a higher frequency of source code change co-occurrence (and hence are assumed to have a higher predictive power in API recommendation) will be given more importance in the API recommendation process. For each candidate API, *APIREC* first computes a score based on the change history, and then adds the resulting score to the one computed by *Gralan* to form the final score.

Not all changes are applicable, however, since some of them could be specific to a historical project and could therefore incur noise in the change patterns. In the example in Figure 1(a), after analyzing a large number of historical fine-grained changes, *APIREC* learned that *BufferedWriter.write* changed with *BufferedReader.<init>* with a probability of 0.3 and that it changed with *FileReader.<init>* with a probability of 0.05. Meanwhile, it also learned that *BufferedReader.close* changed with *BufferedReader.<init>* with a probability of 0.7 and that it changed with *FileReader.-<init>* with a probability of 0.5. Hence, using only the code change history, *APIREC* will select the wrong API, *BufferedReader.close*, for the given recommendation point since its probability of change co-occurrence ($0.7 * 0.5 = 0.35$) is larger than that of *BufferedWriter.write* ($0.3 * 0.05 = 0.015$). In other words, using the code change history, *APIREC* cannot override *Gralan*'s erroneous recommendation for this recommendation point. In addition, *APIREC* requires a long source code change history of each subject project, which limits its applicability to scenarios where long code change history is unavailable or inaccessible.

To address the challenge of accurate API recommendation, we propose RecRank, which recommends APIs based on the API usage paths generated from API usage graphs. An API usage path (henceforth usage path) is generated to represent a data/control flow sequence of APIs that can arguably better encode the intention of the developer. Using discriminative learning in combination with usage paths as features, higher weights can be learned for usage paths that are more relevant and coherent to the given recommendation point, thereby reducing the noise possibly introduced by irrelevant or incoherent usage paths. For example, in Figure 1(a) we extract one usage path [*FileWriter.<init>* → *BufferedWriter.<init>* → *(recommendation point)*] from the code snippet in lines 7–12 in Figure 2. This usage path implements the linguistic topic "write to output text file", which is weighted higher than other usage paths extracted in Figure 1(a). Since RecRank seeks to improve the accuracy of recommending the top-1 API, it could save the developer's time and effort in manually selecting the correct API from multiple candidates. Note that RecRank seeks to achieve this goal *without* mining and using long fine-grained code change histories.

## 3 DISCRIMINATIVE RE-RANKING FOR API RECOMMENDATION (RECRANK)

### 3.1 Overview

In this section, we present a novel approach to API recommendation, RecRank, which operates by re-ranking the top-10 candidate APIs recommended by *Gralan* for each recommendation point using a learned discriminative re-ranker in combination with our usage path-based features. Before describing RecRank, we present two re-ranking systems that could help the reader better understand the power of discriminative re-ranking. The first re-ranking system is trained using the Naïve Bayes (NB) generative model on our usage path-based features. The second re-ranking system is a discriminative classifier trained using the support vector machine learner (henceforth SVC) on our usage path-based features.

The performance difference between the NB system and the SVC system can shed lights on the relative effectiveness of generative models, which are sensitive to the presence of overlapping and irrelevant features, and discriminative models, which are robust to such features. Note that the SVC system is one step closer to RecRank than the NB system in the sense that both SVC and RecRank are discriminative in nature: the primary difference between them lies in the fact that SVC recasts the API recommendation task as a *classification* task whereas RecRank recasts the task as a *ranking* task. The performance difference between them can therefore shed lights on the relative effectiveness of classification and ranking. We will discuss the differences between classification and ranking later in this section.

### 3.2 NB

In NB, we employ the Naïve Bayes learning algorithm implemented in scikit-learn Python library to train a binary classifier to classify whether a given recommended API is the correct API at the recommendation point (i.e., a "hit") or not (i.e., a "miss"). Recall that NB employs the following generative model:

$$P(c|\text{candidate API}) = P(c)\prod_{i=1}^{n} P(f_i|c)$$

where $c$ is the class (which in our case is either "hit" or "miss"), and each $f_i$ corresponds to a usage path-based feature extracted for the candidate API under consideration. As can be seen, the NB generative model assumes that the values of the usage path-based features are conditionally independent of each other given the class. Each of the probabilities in the generative model can be estimated using maximum likelihood estimation from the training data. Specifically, $P(c)$ is the fraction of instances in the training set that are labeled as $c$. $P(f_i|c)$ is the fraction of training instances labeled as $c$ that contain feature $f_i$.

We employ the trained NB model to re-rank the top-10 candidate APIs suggested by *Gralan* as follows. Since the model computes for each candidate API the probability that it is a "hit", we rank the candidate APIs using their associated probabilities, where higher probabilities correspond to higher ranks.

Next, we discuss how the training instances are created and how the usage path-based features are extracted for each training instance.

*Creating training instances.* For each API recommendation point in the training set, we create one training instance for each of the 10 API candidates recommended by *Gralan*, labeling an instance as "hit" or "miss" depending on whether the corresponding candidate API is the correct API for the recommendation point under consideration. Each instance is represented using a set of usage path-based features, each of which corresponds to an usage path. This set of usage paths is the union of the usage paths extracted from each of the API usage graphs created for the given recommendation point (see Section 2 on how these API usage graphs are created). Below we define usage paths.

Each usage path is extracted from an API usage graph and is defined by three constraints. First, a usage path is formed by a sequence of APIs connected by directed data and/or control flow edges. Second, the APIs in a usage path are sequentially connected/listed in API usage order with one entry API and one exit API. Finally, each usage path contains a candidate API (one of the 10 candidate APIs recommended by *Gralan*) that appears either at the end (where the directed flow ends) or at the beginning (where the directed flow starts) of the path. Usage paths of various lengths could be generated from an API usage graph. The length of a usage path is between 2 and the threshold parameter $d$, which determines the maximum distance between any node and the recommendation point in the graph (defined in Section 2.1). For example, 13 usage paths can be generated from the API usage graph in Figure 1(a), such as: [*FileReader.<init>* → *BufferedReader.<init>* → *BufferedReader.readLine* → *CONTROL.WHILE* → *(candidate API)*].

To model different data/control flow in usage paths, we have designed different types of usage path features, as described below.

A ***forward usage path feature*** is created from a usage path in which the APIs in the path are connected by edges in the point-forward direction with the candidate API appearing at the end of the path. A forward data/control flow towards the recommendation point usually implies that the API at the recommendation point "consumes" the data passed by data/control flow. As an example, consider the API usage graph in Figure 1. From this graph, we can create a *forward usage path feature* from the path [*FileReader.<init>* → *BufferedReader.<init>* → *BufferedReader.readLine* → *CONTROL.WHILE* → *(candidate API)*]. We create forward usage path features from paths of different lengths, where the length of a path is defined as the number of APIs involved in the path. For instance, the path [*CONTROL.WHILE* → *(candidate API)*] is of length 2. We consider all paths of up to length $d$.

A ***backward usage path feature*** is created from a usage path that starts with the candidate API, and in which the APIs are connected by edges with a point-backward direction. A backward data/control flow from the recommendation point usually implies that the API at the recommendation point "produces" or "returns" the data to be delivered to the APIs in the back track. In Figure 1(a), we can create a *backward usage path feature* from the path [*CONTROL.WHILE* ← *(candidate API)*]. Similar to *forward usage path features*, *backward usage path features* are generated from paths of different lengths.

In addition, we derive ***fuzzy usage path features*** from the *forward usage path features* and the *backward usage path features*. To motivate *fuzzy usage path features*, we note the correspondence between these usage paths and the word $n$-grams used in natural language processing (NLP). Specifically, the sequence of APIs in a forward/backward usage path is reminiscent of the sequence of words in a word $n$-gram. NLP researchers have noted a weakness of using word $n$-grams as features in natural language learning: if $n$ is large, the resulting $n$-grams will suffer from data sparsity; and if $n$ is small, the $n$-grams will fail to capture longer-distance dependencies. To address this weakness, they have proposed the use of *skipgrams*, in which they allow all but the first word and the last word in an $n$-gram to match any words. For instance, given the word $n$-gram "I am a boy", one can generate a skipgram "I * * boy", where each wildcard * can match any word. This provides generalization of the original $n$-gram (and therefore addresses data sparsity) but at the same time captures the relationship between non-adjacent words (in this case "I" and "boy").

*Fuzzy usage path features* are motivated by skipgrams. Specifically, a *fuzzy usage path feature* is created from a forward/backward usage path feature by replacing all but the entry API and the exit API in the corresponding path with wildcards. Returning to the example in Figure 1(a), [*FileReader.<init>* → * → * → *CONTROL.WHILE* → *(candidate API)*] is a fuzzy usage path feature with two "fuzzy" APIs in the path (represented as "*"). As in skipgrams, wildcards (i.e., fuzzy APIs) can only appear in the middle of a fuzzy usage path. As with skipgrams, the goal of these fuzzy path features is to provide generalizations of the forward/backward usage path features.

In comparison to the parent graphs and subgraphs that *Gralan* uses as features, our usage paths are arguably more relevant to API recommendation. First, since each usage path has to begin or end with a candidate API, it ensures that the path contains an API that is immediately adjacent to the candidate API, thereby increasing its relevance for API prediction. In contrast, a subgraph employed by *Gralan* may not contain any nodes that are adjacent to the recommendation point, thus possibly making it less relevant for API prediction. Second, from the example in Figure 1, each context graph can potentially contain more than one linguistic topic (e.g., both read and write to a file). On the other hand, a usage path can typically allow us to focus on just one linguistic topic. This is especially important when it comes to discriminative learning: a discriminative learner can assign high weights to those features that encode the intended linguistic topic and low weights to those features that do not. If context graphs encoding multiple linguistic topics were used as features, the learner could find it difficult to decide whether high or low weights should be assigned to such features. Note that the computation of these usage path features can be done offline (i.e., during training) with the resulting values stored in a database. During testing, their values can simply be retrieved from the database.

A final issue that we have eluded so far concerns how we obtain *Gralan*'s top-10 candidate APIs on the *training* set. Recall that we create one training instance from each of *Gralan*'s top-10 candidate APIs. This means that before we can create training instances, we need to *produce Gralan*'s top-10 candidate APIs on the training set. We do so using 5-fold cross validation on the training set: we partition the training set randomly into five folds of roughly equal sizes. In each fold experiment, we train *Gralan* on four folds and applying the trained *Gralan* to generate the top-10 candidate APIs on the remaining fold. We repeat this five times, each time generating top-10 candidates on a different fold.

*Applying the NB classifier.* Test instances are created in the same way as the training instances. Specifically, we create one test instance from each of *Gralan*'s top-10 candidate APIs. This means that before we create test instances, we need to produce *Gralan*'s top-10 candidate APIs for each recommendation point in the test set. To do so, we train *Gralan* on the entire training set and apply the trained *Gralan* to generate top-10 candidate APIs on the test set.

As mentioned before, the resulting NB classifier can be used to compute the probability that each candidate API is a "hit" for a recommendation point. These probabilities are then used to re-rank the 10 candidate APIs.

## 3.3 SVC

Our second re-ranking system, SVC, is a discriminative classifier trained using the SVM learning algorithm with a linear kernel, as implemented in the libSVM software package [9]. As in NB, we first use cross validation on the training set to produce *Gralan*'s top-10 candidate APIs on the training set, and then create one training instance from each of the 10 candidate APIs. Each training instance in SVC is represented using the same set of usage path-based features as in NB. The only difference lies in the value of each feature. As NB is generative, each feature is conditioned on the class. In contrast, SVC is discriminative, so we desire that the value of a feature provides some indication of how useful it is. Specifically, we desire that higher feature values imply more relevant features. To this end, we compute the value of a feature as follows. First, we count the number of times the corresponding usage path appears in the training set (call this number $a$). Second, we count the number of times the path appears in the training set *after* removing from it the candidate API (call this number $b$). Finally, we set the feature value to $\frac{b}{a}$. In other words, the more often the candidate API co-occurs with the rest of the path in the training set, the larger the feature value is. As in NB, the values of these usage path features can be computed and stored in a database during training, and they can simply be retrieved from the database during testing.

*Training the SVC classifier.* Given the training instances, the SVM learner learns a maximum margin hyperplane that minimizes the training error (i.e., the error of the hyperplane in classifying the training instances). A hyperplane is defined by a set of weights, each of which is associated with exactly one feature. In other words, the SVM learner learns a set of feature weights that minimizes training error, specifically by associating larger absolute weights with relevant features and lower absolute weights with irrelevant features. This distinguishes a discriminative learner from a generative model such as NB.

*Applying the SVC classifier.* After training, the resulting hyperplane can be used to classify the test instances, which are created in the same way as the training instances. As in NB, *Gralan*'s top-10 candidates on the test set are obtained by training *Gralan* on the entire training set and applying the trained *Gralan* on the test set. We re-rank the top-10 candidate APIs based on their distances from the hyperplane. Specifically, the candidate API on the "hit" side of the hyperplane that is farthest away from the hyperplane receives the highest rank, whereas the one on the "miss" side of the hyperplane that is farthest away from the hyperplane receives the lowest rank.

## 3.4 RecRank

Next, we describe *RecRank*, which differs from SVC in one respect: SVC *classifies* candidate APIs, whereas *RecRank ranks* candidate APIs. To understand the difference between classification and ranking, we first note that API recommendation is inherently a *ranking* task: its goal is to compare/rank candidate APIs and pick the best (i.e., highest-ranked) candidate API for a given recommendation point. When applying SVC, we essentially recast API recommendation as a classification task, where each candidate API is classified (as "hit" or "miss") *independently* of other candidate APIs. In other words, SVC does *not* compare candidate APIs against each other,

**Table 2: Dataset statistics**

| | Training | Test |
|---|---|---|
| Total projects | 1385 | 8 |
| Total classes | 138,791 | 8,621 |
| Total methods | 732,645 | 38,036 |
| Total distinctive JDK API elements | 18,166 | 7,272 |
| Total recommendation points | 70,377 | 11,872 |
| Average features per API candidate | 30 | 24 |

and without such comparisons, it fails to determine which candidate API is the best. In contrast, the goal of ranking is precisely to compare candidate APIs by imposing a ranking on them.

*Training RecRank. RecRank* trains an SVM ranker using the linear-kernel ranker learning algorithm implemented in $SVM^{rank}$ [15]. The training instances (and the features that represent each training instance) are created in the same way as in SVC. The resulting training instances are then grouped into different ranking problems. Specifically, each ranking problem corresponds to exactly one recommendation point and is composed of the 10 training instances corresponding to the top-10 candidate APIs for this recommendation point. The goal of the ranker training procedure is to learn a hyperplane (by adjusting the feature weights) to minimize the number of violations of pairwise ranking in the training set. Specifically, a violation occurs if a training instance labeled as "hit" is ranked below a training instance labeled as "miss" by the ranker.

*Applying RecRank.* After training, the ranker can be used to directly rank the top-10 candidate APIs for each recommendation point in the test set. Specifically, the ranker assigns each candidate API a value, based on which a ranking can be imposed on the candidate APIs. RecRank then recommends the candidate API that has the highest rank.

# 4 EMPIRICAL EVALUATION

## 4.1 Experiment Setup

**Datasets.** We collected a large dataset of 1385 Java projects from GitHub for training API recommendation systems and another eight for evaluation. Statistics of this dataset are shown in Table 2. In order to obtain high quality API usage graphs, we follow previous work [24]: we filter out the projects that are not parsable, experimental or toy programs. Also, we use only the latest snapshot of each project. For generalization purposes, we focus solely on Java Development Kit (JDK) APIs. To facilitate comparison with previous work, the eight projects in our evaluation set are the same as those used to evaluate *APIREC*, a state-of-the-art API recommendation system [23]. Training and test recommendation points are created from these projects in the same way as in previous work [23, 24]: except for the first two APIs in each method, we create one recommendation point for each API.

**Evaluation Measures.** We employ two evaluation measures, top-1 accuracy and mean reciprocal rank (MRR) [34]. Top-1 accuracy is a measure used in previous work on API recommendation [24]. For each API recommendation point in the test set, if the top-1 API candidate returned by a system is the correct API at the recommendation point, we count it as a "hit". The top-1 accuracy is the ratio of the total number of hits to the total number of recommendation points. MRR is an evaluation measure commonly used in information retrieval to evaluate search results. Like top-1 accuracy, a score of 1 is given to a recommendation point for which the top-1 candidate is the correct API. Unlike top-1 accuracy, where a system is not rewarded at all if the correct API is not the top-1 candidate API, MRR partially rewards a system as follows: a score of $\frac{1}{r}$ is given to a recommendation point if the correct API appears in rank $r$. In other words, the (partial) reward is inversely proportional to the rank of the correct API. MRR then averages the scores over the recommendation points in the test set. Thus, MRR can be viewed as a relaxed version of top-1 accuracy that partially rewards a system where the correct API is not the top-1 candidate. Since we are re-ranking *Gralan*'s top-10 candidate APIs, recommendation points where the correct API is not in *Gralan*'s top-10 will receive a score of 0.

**Table 3: Re-implemented and original *Gralan* results**

| Project | Top-1 Accuracy | | | Top-10 Accuracy | | |
|---|---|---|---|---|---|---|
| | ***Duplicated Gralan*** | ***Origin Gralan*** | ***Error*** | ***Duplicated Gralan*** | ***Origin Gralan*** | ***Error*** |
| antlr | 38.3 | 26.0 | +12.3 | 76.5 | 79.0 | -2.5 |
| Galaxy | 22.4 | 22.0 | +0.4 | 80.6 | 80.0 | +0.6 |
| Froyo-Email | 25.5 | 46.0 | -20.5 | 73.9 | 81.0 | -7.1 |
| Grid-Sphere | 31.2 | 26.0 | +5.2 | 76.9 | 85.0 | -8.1 |
| Itext | 24.7 | 33.0 | -8.3 | 80.5 | 76.0 | +4.5 |
| jGit | 33.6 | 20.0 | +11.6 | 77.1 | 74.0 | +3.1 |
| log4j | 28.0 | 29.0 | -1.0 | 75.2 | 74.0 | +1.2 |
| spring | 30.2 | 28.0 | +2.2 | 73.4 | 67.0 | +6.4 |

**Baseline Systems.** We employ two baseline systems, neither of which is publicly available. As our first baseline, we employ *APIREC*. The *APIREC* results reported in this paper are taken verbatim from the original *APIREC* paper [23].[3]

As our second baseline, we employ *Gralan*. Since NB, SVC, and RecRank are all built upon *Gralan*'s top-10 candidate APIs, we re-implement Gralan, following the steps mentioned in Section 2.1. Specifically, we first build the *API usage graphs* from the collected 1385 open source projects in the training set. Then, following Nguyen *et al.* [24], for each API usage graph we simulate the API recommendation process by predicting each API given its preceding context. We set the parameter $d$ to 3, meaning that only the context graphs involving the one, two or three APIs preceding a recommendation point are considered. The reason for setting $d$ to 3 is that according to Nguyen et al. [24], when $d$=3, the top-10 accuracy achieved by *Gralan* (86.0%) is close to the best accuracy (87.1%).

Table 3 compares the original *Gralan* results [23] with our duplicated/re-implemented *Gralan* results on the same eight subject projects.

[3]The reason we did not re-implement *APIREC* is that the significant large historical change repository dataset (i.e., 113,103 change commits and 471,730 changed source code files according to Nguyen et al. [23]) is hard to acquire.

As can be seen, duplicated *Gralan* achieves better or comparable top-1 and top-10 accuracies than original *Gralan* results across all projects except Froyo-Email (top-1). Note that a strict comparison is not possible owing to the fact that original *Gralan* and duplicated *Gralan* were trained on different projects.[4]

**Evaluation Settings.** For NB, SVC and RecRank, we use the 1385 projects in the training set for model training. We conduct an 8-fold cross validation on the 8 projects in the test set as follows. In each fold experiment, we hold out exactly one project for testing and use the remaining 7 projects for development (i.e., parameter tuning). We repeat this experiment 8 times, each time choosing a different project as our held-out test set. For parameter tuning, we tune NB's Laplace smoothing parameter $\alpha$ as well as libSVM and $SVM^{rank}$'s regularization parameter C to maximize the top-1 accuracy on the development set. We limit the length an usage path based feature to no more than 4.

## 4.2 Experimental Results

This section empirically answers our research questions.

***RQ1.*** *How accurate do RecRank, NB, and SVC recommend APIs in comparison to the two baselines?*

Results of NB, SVC, RecRank and the two baselines, *APIREC* and *duplicated Gralan (a.k.a., D-Gralan)*, expressed in terms of per-project and overall top-1 accuracy and MRR, are shown in Table 4. As we can see, our proposed approaches (rows 3–5) outperform *D-Gralan* (row 2) in both top-1 accuracy and MRR across all subject projects. In particular, RecRank is the best performer in terms of both measures, achieving better top-1 accuracy than *APIREC* in all eight projects.[5]

We further make several interesting observations. First, the proposed learning-based approaches (NB, SVC, and RecRank) achieve better top-1 accuracy and MRR than *D-Gralan*: top-1 accuracy improves by 0.7–50% and MRR improves by 0.13–0.49. Compared to *APIREC*'s top-1 accuracy (59.5%), SVC and RecRank achieve comparable or better results (i.e., 59.6 and 64.8% respectively). Encouragingly, RecRank improves the state-of-the-art top-1 accuracies across all eight subject projects by 1.7-23.7%.

To determine whether the improvements in overall top-1 accuracy and overall MRR between RecRank and other approaches are statistically significant or not, we conduct the Wilcoxon rank-sum test. Following Miller [20], the result of a significance test can be interpreted as follows. The performance difference between the two systems under comparison is (1) *highly significant* if the null hypothesis (i.e., there is no performance difference between the two systems) can be rejected at the 0.01 level (represented as "***" in the table); (2) *significant* if it can be rejected at the 0.05 level (represented as "**"); and (3) *moderately significant* if it can be rejected at the 0.1 level (represented as "*"). Otherwise, the difference is *statistically indistinguishable*. As can be seen in Table 4, RecRank is either highly or moderately significantly better than other systems.[6] To evaluate the amount of performance difference between RecRank and each of the other approaches, we compute Cliff's delta [7], a non-parametric effect size measure. Results show that in each case the delta value is greater than 0.474, which, according to Romano *et al.* [30], implies a large effect size.

***RQ2.*** *How effective are usage path features for API recommendation compared with context graphs?*

To compare the effectiveness of these two types of features, we employ them to train four approaches: RecRank, NB, SVC and *D-Gralan*. This results in the eight combinations shown in Table 5. For instance, RecRank+E is the variant of RecRank trained using the usage path features, whereas RecRank+C is the variant of RecRank trained using context graphs. Note that the two variants within each of the four approaches differ only with respect to the feature set. In particular, the value of a feature is computed in the same way in the two variants of an approach. For instance, the value of a feature in RecRank+C is computed in the same way as that in RecRank+E, which was described in Section 3.4.

As can be seen in Table 5, for NB, SVC, and RecRank, the E variant is highly significantly better than the C variant in terms of both top-1 accuracy and MRR with a large effect size. These results provide suggestive evidence that the usage path features are considerably more effective than the context graph-based features for both discriminative models (SVC and RecRank) and the NB generative model. The only exception is *Gralan*, where its C variant is highly significantly better than its E variant. We speculate that context graphs were specifically designed by their original authors so that they could work well when used in conjunction with *Gralan*'s generative model, but additional experiments are needed to determine the reason.

***RQ3.*** *How effective are different classes of usage path features for API recommendation?*

To answer this question, we divide our usage path features into 12 groups based on (1) whether the path is forward or backward; (2) whether the path contains fuzzy APIs or not; and (3) the length of the path, which could be 2, 3 or 4 (recall that we limit the length to no more than 4 in Section 4.1). To determine the contribution of each of these 12 groups of features to RecRank's performance, we conduct ablation experiments, where in each ablation experiment, we re-train RecRank by leaving out one or more of the 12 feature types and measure the performance of the re-trained RecRank on the test projects. Intuitively, the larger the drop in performance is in an ablation experiment, the more important the missing feature group(s) are as far as performance is concerned.

Ablation results are shown in Table 6. For ease of comparison, we show in row 1 the results of RecRank when all usage path features are used. The remaining rows show the results when one or more of the feature groups are removed. In comparison to the RecRank that uses all of the usage path features, performance drops highly significantly with respect to both top-1 accuracy and MRR in three cases: (1) when the length 2 forward features are removed; (2) when all forward features are removed; and (3) when all length 2 features are removed. Interestingly, removal of other feature groups does not result in significant drops in performance. In particular, removal of any of the length 3 and 4 features causes little and sometimes no change in performance. However, it is important to note that this by no means implies that features of lengths 3 and 4 are not useful: these experiments only suggest that the feature group that is being removed is not useful *in the presence of the remaining features*. In other words, if two feature groups encode redundant

[4] The list of projects used to train original *Gralan* is not revealed by the authors.
[5] MRR results are missing for *APIREC* because they are not reported in the original paper.
[6] Significance tests cannot be conducted on *APIREC* because we do not have its output.

**Table 4: Evaluation results of API recommendation systems**

| | System | Top-1 Accuracy | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Overall | antlr | Galaxy | Froyo-Email | Grid-Sphere | Itext | jGit | log4j | spring |
| 1 | APIREC | 59.5 | 57.0 | 56.0 | 60.0 | 58.0 | 44.0 | 54.0 | 52.0 | 57.0 |
| 2 | D-Gralan | 29.5*** | 38.3 | 22.4 | 25.5 | 31.2 | 24.7 | 33.6 | 28.0 | 30.2 |
| 3 | NB | 34.8*** | 45.1 | 37.3 | 35.3 | 38.3 | 29.4 | 38.2 | 35.6 | 30.9 |
| 4 | SVC | 59.6* | 60.1 | 61.2 | 51.1 | 58.9 | 57.3 | 56.1 | 51.4 | 47.4 |
| 5 | RecRank | **64.8** | **69.4** | **72.4** | **63.5** | **67.6** | **67.7** | **67.4** | **62.9** | **58.7** |
| | **System** | **MRR** | | | | | | | | |
| 1 | APIREC | – | – | – | – | – | – | – | – | – |
| 2 | D-Gralan | 0.27*** | 0.29 | 0.37 | 0.30 | 0.34 | 0.25 | 0.30 | 0.27 | 0.23 |
| 3 | NB | 0.60*** | 0.64 | 0.64 | 0.57 | 0.64 | 0.64 | 0.62 | 0.60 | 0.56 |
| 4 | SVC | 0.69* | 0.71 | 0.73 | 0.67 | 0.73 | 0.73 | 0.72 | 0.66 | 0.66 |
| 5 | RecRank | **0.70** | **0.73** | **0.69** | **0.69** | **0.73** | **0.74** | **0.73** | **0.69** | **0.66** |

**Table 5: Evaluation results for different model-feature combinations**

| | Combination | Overall Top-1 Accuracy | Overall MRR |
|---|---|---|---|
| 1 | RecRank+E | **64.8** | **0.70** |
| 2 | RecRank+C | 36.6*** | 0.58*** |
| 3 | SVC+E | 59.6 | 0.69 |
| 4 | SVC+C | 25.4*** | 0.58*** |
| 5 | NB+E | 34.8 | 0.60 |
| 6 | NB+C | 16.1*** | 0.47*** |
| 7 | D-Gralan+E | 24.1*** | 0.25*** |
| 8 | D-Gralan+C | 29.5 | 0.27 |

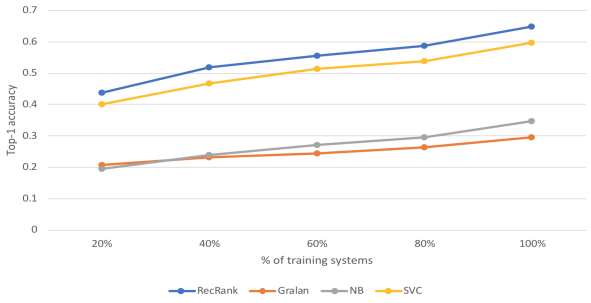



**Figure 4: Learning curves of API recommendation approaches on the entire test set**

**Table 6: Feature ablation results of RecRank**

| System | Overall Top-1 Acc | Overall MRR |
|---|---|---|
| All features | 64.8 | 0.70 |
| No length2 forward | 55.8*** | 0.64*** |
| No length3 forward fuzzy | 64.5 | 0.70 |
| No length3 forward no-fuzzy | 64.2 | 0.70 |
| No length4 forward fuzzy | 64.7 | 0.70 |
| No length4 forward no-fuzzy | 64.0 | 0.70 |
| No length2 backward | 64.5 | 0.68 |
| No length3 backward fuzzy | 64.0 | 0.70 |
| No length3 backward no-fuzzy | 64.7 | 0.70 |
| No length4 backward fuzzy | 64.1 | 0.70 |
| No length4 backward no-fuzzy | 64.8 | 0.70 |
| No backward | 60.1 | 0.65 |
| No forward | 39.9*** | 0.48*** |
| No length 3or4 no-fuzzy | 65.1 | 0.70 |
| No fuzzy | 64.4 | 0.70 |
| No length2 | 47.2*** | 0.53*** |

information, then removal of one of them will not cause large drops in performance. In fact, the usefulness of features of length 3 and 4 can be seen when comparing the "No length 2 forward" results and then "No forward" results: the performance differences between these two ablated systems can be attributed to the length 3 and 4 features. Specifically, top-1 accuracy drops by more than 15% points and MRR drops by 16% points when the length 3 and 4 features are removed. Similarly, the usefulness of the backward features can be seen by comparing the "No length 2 forward" results and the "No length 2" results: the performance differences between these two ablated systems can be attributed to the backward features. Specifically, top-1 accuracy drops by 8% points.

***RQ4.*** *What is the learning curve of each system?*

To answer this question, Figure 4 presents the learning curve for each of these four systems when measured in terms of top-1 accuracy. Each curve is plotted using five data points that correspond to using 20%, 40%, 60%, 80%, and 100% of the available training projects collected in Section 4.1. As we can see, in none of the systems does top-1 accuracy plateau even when we use all of the available training dataset. This implies that the performance of each API recommendation system will likely to improve further as additional training projects are made available, which is encouraging as additional projects can be easily obtained. In addition, we observe that SVC achieves consistently better overall top-1 accuracy than *D-Gralan* regardless of the amount of available training data. NB achieves better overall top-1 accuracy than *D-Gralan* when more

than 40% of training projects are available for training. The most effective learner, however, is RecRank.

## 5 THREATS TO VALIDITY

Our main threats to *internal validity* can occur to our training and test sets. To address this concern, we train all API recommendation systems on the same training set. Then each system is evaluated on each subject project with parameters tuned on the rest of the seven subject projects.

In addition, threats to *external validity* can occur during data collection. For generalization purposes, similar to previous works our experiments are performed on JDK APIs only. Meanwhile, for comparison purposes we run experiments on the same subject projects as the baseline API recommendation systems (i.e., *Gralan* and *APIREC*).

## 6 RELATED WORK

### 6.1 Code Suggestion based on Mined Software Repositories

In this subsection we summarize source code suggestion approaches based on mined software repositories. Bruch *et al.* [5] adapt the k-nearest-neighbor algorithm to find method calls to recommend for particular objects. Robbes *et al.* [29] use change history such as code insertion and modification to improve code completion. Hou *et al.* [14] present a way of grouping API proposals from historical data for better code completion. Hill *et al.* [12] build a tool to automatically complete a method by cloned code. Asaduzzaman *et al.* [3] and Zhang *et al.* [39] both use parameter filtering to do code recommendation. Omar *et al.* [26] uses an interactive way to generate code. Reiss *et al.* [28] and Stolee *et al.* [32] use semantic search to map retrieved code into what is asked for by users. Thung *et al.* [33] present an approach that learns from records of other changes made to software systems and compares the textual description of the requested feature with the textual descriptions of various API methods. Wang *et al.* [35] propose two quality metrics (succinctness and coverage) for mined usage patterns. Xie *et al.* [37] present MAPO to generate patterns by mining and ranking frequent sequences in each cluster according to the similarity heuristics of source code such as method names. Most of these approach rely on a large number of software historical repositories. This kind of approach is not applicable when such repository is not available. Different from the above approaches, RecRank does not rely on any software historical repository.

### 6.2 Code Suggestion Using Statistical Models

This subsection summarizes source code suggestion approaches using statistical models. Gu *et al.* [11] adapt a neural language model named RNN Encoder-Decoder, which encodes a word sequence (user query) into a fixed-length context vector, and generates an API sequence based on the context vector. White *et al.* [36] apply the RNN-LM model on lexically analyzed source code to code suggestion. Allamanis *et al.* [1] present NATURALIZE, which learns coding conventions to suggest natural identifier names and formatting conventions. They also apply the binomial model to retrieve source code snippets given a natural language query and retrieve natural language descriptions given a source code query [2]. Maddison *et al.* [17] use Probabilistic Context Free Grammar (PCFG)-based model to represent source code. McMillan *et al.* [18] propose a combination of association between queries and functions model and navigation behavior of programmers model to retrieve and visualize relevant functions and their usages. Chan *et al.* [6] perform API recommendation based on the textual similarity between code and query phrases. CodeHow [16] expands the query with the APIs and performs code retrieval by applying the Extended Boolean model, which considers the impact of both text similarity and potential APIs on code search. MULAPI [38] recommends feature related API from feature request documents. In this paper, we propose a novel ranking-based discriminative model to improve the state-of-the-art top-1 API recommendation accuracy.

### 6.3 Code Suggestion based on Code Structure

This subsection overviews source code structure based approaches. Asaduzzaman *et al.* [4] recommend API by ranking the similarities between code contexts and the context of the target API method call. Raychev *et al.* [27] extract indexed sequences of method calls and use a statistical language model to find the highest ranked sentences to synthesize a code completion. Mou *et al.* [22] propose a tree-based convolutional neural network (TBCNN) on AST tree structure to detect code snippets of certain patterns. Holmes *et al.* [13] present an approach for locating relevant code that is based on heuristically matching the structure of the code under development to the example code. Saul *et al.* [31] use a random walk approach on a subset of a callgraph in order to recommend source code. Ekoko *et al.* [8] propose an approach that leverages the structural relationships between APIs to discover inaccessible API methods or types. McMillan *et al.* [19] recommend source code examples by querying against API calls and documentations about code structural information. Moritz *et al.* [21] present an approach to recommend API usage by representing software as a Relational Topic Model. Fowkes *et al.* [10] propose a probabilistic algorithm to find the most informative and parameter-free API call patterns. In our approach, RecRank recommends API based on API usage graphs, which includes data flow dependencies and control flow dependencies among APIs. And compared with the state-of-the-art graph-based API recommendation approach *Gralan*, RecRank significantly improves the top-1 accuracy of API recommendation.

## 7 CONCLUSIONS AND FUTURE WORK

We proposed a novel discriminative re-ranking-based API recommendation system, RecRank, which uses usage path-based features to rank the top-10 API candidates generated by *Gralan*. In an evaluation on eight large scale open source projects, RecRank significantly improved top-1 accuracy by 28.5%–50.0% and MRR by 0.32–0.49 in comparison to *Gralan*. When compared to *APIREC*, RecRank improved top-1 accuracy by as much as 23.7%, yielding an overall improvement of 5.3% absolute. Perhaps even more encouragingly, we saw performance improvements in each of the eight projects. Importantly, RecRank does not require access to a large number of historical code changes for training and application. In future work, we will extend our approach on a wider spectrum of API types and experiment on additional projects.

## REFERENCES


[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 281–293.

[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning*. JMLR.org, 2123–2132.

[3] Muhammad Asaduzzaman, Chanchal K. Roy, Samiul Monir, and Kevin A. Schneider. 2015. Exploring API method parameter recommendations. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*. IEEE.

[4] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. 2016. A Simple, Efficient, Context-sensitive Approach for Code Completion. *Journal of Software: Evolution and Process* 28, 7 (2016), 512–541.

[5] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 213–222.

[6] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 10.

[7] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993).

[8] Ekwa Duala-Ekoko and Martin P. Robillard. 2011. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the 25th European Conference on Object-oriented Programming*. Springer, 79–104.

[9] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research* 9, Aug (2008), 1871–1874.

[10] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free probabilistic API mining at github scale. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. *ACM*.

[11] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 631–642.

[12] Rosco Hill and Joe Rideout. 2004. Automatic method completion. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 228–235.

[13] Reid Holmes and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th IEEE/ACM International Conference on Software Engineering*. IEEE, 117–125.

[14] Daqing Hou and David M. Pletcher. 2011. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*. IEEE, 233–242.

[15] Thorsten Joachims. 2006. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 217–226.

[16] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on API understanding and extended boolean model (e). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 260–270.

[17] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *Proceedings of the 31th International Conference on Machine Learning*. JMLR.org, 649–657.

[18] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering*. ACM, 111–120.

[19] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2010. Recommending source code examples via API call usages and documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 21–25.

[20] David A. Miller. 1966. Significant and highly significant. *Nature* 210, 5041 (1966).

[21] Evan Moritz, Mario Linares-Vásquez, Denys Poshyvanyk, Mark Grechanik, Collin McMillan, and Malcom Gethers. 2013. Export: Detecting and visualizing API usages in large source code repositories. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 646–651.

[22] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing.. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. 1287–1293.

[23] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 511–522.

[24] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th IEEE International Conference on Software Engineering*. IEEE, 858–868.

[25] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 438–449.

[26] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. In *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*. IEEE Press, 859–869.

[27] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 419–428.

[28] Steven P. Reiss. 2009. Semantics-based code search. In *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering*. IEEE Computer Society, 243–253.

[29] Romain Robbes and Michele Lanza. 2008. How program history can improve code completion. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 317–326.

[30] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *the Annual Meeting of the Florida Association of Institutional Research*. 1–33.

[31] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. 2007. Recommending random walks. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 15–24.

[32] Kathryn T. Stolee and Sebastian Elbaum. 2012. Toward semantic search via SMT solver. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 25.

[33] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. 2013. Automatic recommendation of API methods from feature requests. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 290–300.

[34] Ellen M. Voorhees. 1999. The TREC-8 Question Answering Track Report.. In *TREC*, Vol. 99. 77–82.

[35] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 319–328.

[36] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*. IEEE, 334–345.

[37] Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*. ACM, 54–57.

[38] Congying Xu, Xiaobing Sun, Bin Li, Xintong Lu, and Hongjing Guo. 2018. MULAPI: Improving API method recommendation with API usage location. *Journal of Systems and Software* 142 (2018), 195 – 205.

[39] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. 2012. Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*. IEEE Press, 826–836.