Effective API Recommendation without Historical Software Repositories

Xiaoyu Liu\textsuperscript{1}, LiGuo Huang\textsuperscript{1}, Vincent Ng\textsuperscript{2}

\textsuperscript{1} Dept. of Computer Science and Engineering, Southern Methodist University, Dallas, TX
\textsuperscript{2} Human Language Technology Research Institute, University of Texas and Dallas, TX

Emails: \{xiaoyul, lghuang\}@smu.edu \quad vince@hlt.utdallas.edu
Automated API Recommendation, Why?

equals(Object anObject) : boolean - String
length() : int - String
equalsIgnoreCase(String anotherString) : boolean - String
getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) : void - String
concat(String str) : String - String
substring(int beginIndex, int endIndex) : String - String
charAt(int index) : char - String
chars() : IntStream - String
codePointAt(int index) : int - String
codePointBefore(int index) : int - String
codePointCount(int beginIndex, int endIndex) : int - String
codePoints() : IntStream - String
compareTo(String anotherString) : int - String
compareToIgnoreCase(String str) : int - String
contains(CharSequence s) : boolean - String
contentEquals(CharSequence cs) : boolean - String
contentEquals(StringBuffer sb) : boolean - String
endsWith(String suffix) : boolean - String
getBytes() : byte[] - String
getBytes(Charset charset) : byte[] - String
getBytes(String charsetName) : byte[] - String
getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin) : void - String
getClass() : Class<?> - Object
hashCode() : int - String
indexOf(int ch) : int - String
Existing API Recommendation Methods

Statistical Learning: irrelevant features (noises) and overlapping features


Code Structure: missing certain semantics of source code


Mining Historical Software Repositories: maintain tremendous amount of historical data


State-of-the Art


How does *Gralan* Recommend APIs?

```java
public void readAndWrite() throws IOException{
    BufferedReader in = new BufferedReader(new FileReader("Input.txt"));
    BufferedWriter out = new BufferedWriter(new FileWriter("Output.txt"));

    String i = null;
    char c;
    while((i = in.readLine())!=null){
        // ?
    }
}
```

**Proceeding Context:**

Generate API Usage Graph

1. Extract Child Graph, Parent Graph and Subgraphs
2. Compute the probability of candidate APIs
3. Predict API with the highest probability
How does *Gralan* Recommend APIs?

Proceeding Context:
- Generate API Usage Graph
- Extract Child Graph, Parent Graph and Subgraphs
- Compute the probability of candidate APIs
- Predict API with the highest probability
How does *Gralan* Recommend APIs?

1. Generate API Usage Graph
2. Extract Child Graph, Parent Graph and Subgraphs
3. Compute the probability of candidate APIs
4. Predict API with the highest probability
How does **Gralan** Recommend APIs?

1. Proceeding Context: Generate API Usage Graph
2. Extract Child Graph, Parent Graph and Subgraphs
3. Compute the probability of candidate APIs
4. Predict API with the highest probability
How does *Gralan* Recommend APIs?

Proceeding Context:
- Generate API Usage Graph
- Extract Child Graph, Parent Graph and Subgraphs
- Compute the probability of candidate APIs
- Predict API with the highest probability
How does **Gralan** Recommend APIs?

Proceeding Context:
- Generate API Usage Graph
- Extract Child Graph, Parent Graph and Subgraphs

\[ \log(Pr(C(g)|g_1, g_2, \ldots, g_n, g)) \]
\[ \propto \log(Pr(g_1|C(g)) \ldots Pr(g_n|C(g))Pr(C(g)) \]
\[ = \sum_{j=1}^{n} \log(#\text{methods}(g_j, C(g)) + \alpha) \]
\[ + \log(#\text{methods}(g, C(g))) \]
\[ - (n - 1) \log(#\text{methods}(C(g)) + \alpha\#\text{methods}(g)) \]

Predict API with the highest probability
# How does Gralan Recommend APIs?

<table>
<thead>
<tr>
<th>$g$</th>
<th>$C(g)$</th>
<th>Candidate API</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileReader.&lt;init&gt;, BufferedReader.&lt;init&gt;, BufferedReader.readLine, CONTROL.WHILE</td>
<td>FileReader.&lt;init&gt;, ..., BufferedReader.close</td>
<td>BufferedReader.close <em>(incorrect)</em></td>
<td>0.33</td>
</tr>
<tr>
<td>FileReader.&lt;init&gt;, ..., CONTROL.WHILE, BufferedReader.readLine</td>
<td>BufferedReader.readLine</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BufferedWriter.&lt;init&gt;, CONTROL.WHILE</td>
<td>BufferedWriter.&lt;init&gt;, CONTROL.WHILE, BufferedWriter.write</td>
<td>BufferedWriter.write</td>
<td>0.25</td>
</tr>
<tr>
<td>BufferedWriter.&lt;init&gt;, CONTROL.WHILE, BufferedWriter.write</td>
<td>BufferedWriter.write</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>BufferedWriter.&lt;init&gt;, CONTROL.WHILE, BufferedReader.close</td>
<td>BufferedReader.close</td>
<td>0.02</td>
<td></td>
</tr>
</tbody>
</table>

**Proceeding Context:**
Generate API Usage Graph

1. Generate API Usage Graph
2. Extract Child Graph, Parent Graph and Subgraphs
3. Compute the probability of candidate APIs
4. Predict API with the highest probability

@SMU-CSE
How does **APIREC** recommend APIs?

**APIREC** requires a long code change history of each project, which limits its applicability to scenarios where long code change history is unavailable or inaccessible.

**APIREC** uses all changes. BUT some of them could be specific to a historical project and could therefore incur noise in the change patterns.
Objective and Major Contributions

**Objective:** To improve the top-1 accuracy of API recommendation.

**RecRank:** a novel discriminative ranking approach to automatically recommend top-1 APIs based on the top-10 API candidates suggested by *Gralan*.

**Usage path features:** a novel kind of features based on API usage paths
Discriminative Re-Ranking for API Recommendation (RecRa)

1. Create Training Instances
2. Extract API Usage Path Features
3. Train RecRank Re-ranker
4. Re-rank API Candidates in Test Set
Discriminative Re-Ranking for API Recommendation (RecRa)

1. **Create Training Instances**
   - For each API recommendation point in the training set, training instances are created.
   - Create one training instance for each of the 10 API candidates recommended by *Gralan*.
   - Correct API labeled as “hit”, incorrect API labeled as “miss”
   - Each instance is represented using a set of usage path features.

2. **Extract API Usage Path Features**
   - A sequence of APIs sequentially connected/listed in API usage order with one entry and one exit API.
   - Each usage path contains a candidate API (one of 10 candidate APIs recommended by *Gralan*) that appears either at the end or beginning of the path.
   - Represent a data/control flow sequence of APIs
   - Compared to context API usage graphs, it is better including many irrelevant APIs
Create Training Instances

For each API recommendation point in the training set, training instances are created.

Create one training instance for each of the 10 API candidates recommended by Gralan.

Correct API labeled as “hit”, incorrect API labeled as “miss”

Each instance is represented using a set of usage path features.
Discriminative Re-Ranking for API Recommendation (RecRan)

1. Create Training Instances

For each API recommendation point in the training set, training instances are created.

Create one training instance for each of the 10 API candidates recommended by Gralan.

Correct API labeled as “hit”, incorrect API labeled as “miss”

Each instance is represented using a set of usage path features.

2. Extract API Usage Path Features

backward usage path feature

[(candidate API) → CONTROL.WHILE]
For each API recommendation point in the training set, training instances are created. Create one training instance for each of the 10 API candidates recommended by Gralan. Correct API labeled as “hit”, incorrect API labeled as “miss”. Each instance is represented using a set of usage path features.
Generative Naïve Bayes Classifier

To investigate how generative model performs compared to discriminative model in API recommendation.

Generative model assumes that the values of the usage path-based features are conditionally independent of each other given the class.

Candidate APIs are ranked using their associated probabilities, where higher probabilities correspond to higher ranks.
Discriminative Re-Ranking for API Recommendation
Discriminative Re-Ranking for API Recommendation (SVC)

Discriminative SVC

Adjust feature weights based on their relevance to the recommendation point.
✓ The more often the candidate API co-occurs with the rest of the path in the training set, the higher the feature value (more relevant features).

Re-rank Top-10 candidate APIs based on their distances from the hyperplane

to investigate how discriminative classification model performs compared to discriminative ranking model in API recommendation
3. Discriminative RecRank
• **SVM** does not compare candidate APIs.

• **Train SVM Ranker:** Each ranking problem is composed of the 10 training instances corresponding to the top-10 candidate APIs for the recommendation point.

• **RecRank** learns a hyperplane (by adjusting the feature weights) to minimize the number of violations of pairwise ranking in the training set:
  - A violation occurs if a training instance labeled as “hit” is ranked below a training instance labeled as “miss” by the ranker.
Discriminative Re-Ranking for API Recommendation (RecRank)

- Create Training Instances
- Train RecRank Re-ranker
- Re-rank API Candidates in Test Set

RecRank recommends the candidate API that has the highest rank.
Empirical Evaluation

- Datasets: Open Source Java Projects from GitHub
  - 1385 Projects (Training)
  - 8 Projects (Evaluation)

- Metrics:
  - Top-1 Accuracy = \( \frac{\text{#of correct API Recommendations}}{\text{Total # of Recommendation Points}} \)
  - Mean Reciprocal Rank (MRR): Partial reward is inversely proportional to the rank of the correct API

- Two Baseline Systems
  - Gralan: \( d = 3 \) for context diagram generation
  - APIREC
How Well Does **RecRank** Perform?

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

![Bar chart showing top-1 accuracy for different methods](chart.jpg)

- **APIREC**
- **D-Gralan**
- **NB**
- **SVC**
- **RecRank**

Overall accuracy comparison for different methods.
How Well Does *RecRank* Perform?

1. How accurate do *RecRank*, NB, and SVC recommend APIs in comparison to the two baselines?
How Well Does RecRank Perform?

Q2. How effective are usage path features for API recommendation compared with context graphs?

![Overall Top-1 Accuracy Chart]

- E: API Usage Path Features
- C: API Context Graph Features
How Well Does *RecRank* Perform?

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

- Performance drops highly significantly in three cases:
  - when the length 2 forward features are removed
  - when all forward features are removed
  - when all length 2 features are removed

- 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
Conclusions and Future Work

Compared with Gralan, discriminative re-ranking-based API recommendation system, RecRank, uses usage path-based features to significantly improved top-1 accuracy by 28.5%–50.0% and MRR by 0.32–0.49.

Compared with APIREC, top-1 accuracy is improved by as much as 23.7%.

Extend RecRank to a wider spectrum of API types and additional project domains.