Aroma: Code Recommendation via Structural Code Search

Sifei Luan
Facebook, Inc.
Menlo Park, CA, USA
lsf@fb.com

Di Yang*
UC Irvine
Irvine, CA, USA
diy4@uci.edu

Koushik Sen
UC Berkeley
Berkeley, CA, USA
ksen@cs.berkeley.edu

Satish Chandra
Facebook, Inc.
Menlo Park, CA, USA
satch@fb.com

Abstract—**Programmers often write code which have similarity to existing code written somewhere. A tool that could help programmers to search such similar code would be immensely useful. Such a tool could help programmers to extend partially written code snippets to completely implement necessary functionality, help to discover extensions to the partial code which are commonly done by other programmers, help to cross-check against similar code written by other programmers, or help to add extra code which would avoid common mistakes and errors. We propose Aroma, a tool and technique for code recommendation via structural code search. Aroma indexes a huge code corpus including thousands of open-source projects, takes a partial code snippet as input, searches the indexed method bodies which contain the partial code snippet, clusters and intersects the results of search to recommend a small set of succinct code snippets which contain the query snippet and which appears as part of several programs in the corpus. We evaluated Aroma on several randomly selected queries created from the corpus and as well as those derived from the code snippets obtained from Stack Overflow, a popular website for discussing code. We found that Aroma was able to retrieve and recommend most relevant code snippets efficiently.**

Index Terms—**code recommendation, structural code search**

I. Introduction

Suppose a programmer wants to write code to decode a bitmap without incurring a high memory overhead. The programmer is familiar with the libraries necessary to write the code, but they are not quite sure how to write the code completely with proper error handling and suitable configurations. They write the partial snippet shown in Table I, row A, left column. The programmer now wants to know how others have implemented this functionality fully and correctly in related projects. Specifically, they want to know what is the customary way to extend the code so that proper setup is done, common errors are handled, and appropriate library methods are called. It would be nice if a tool could return the code in Table I, row A, right column, which shows how to handle potential out-of-memory errors; it also shows how the configurable options for decoding the bitmap are commonly set. We call this the *code recommendation problem.*

There are a few existing techniques which could potentially be used to get code recommendations. For example, code-to-code search tools [1], [2], [40] could retrieve relevant code snippets from a corpus using a partial code snippet as query. However, such code-to-code search tools could return lots of relevant code snippets without removing or aggregating similar looking ones. Moreover, such tools do not make any effort to carve out common and concise code snippets from similar looking retrieved code snippets. Pattern-based code completion tools [48]–[50] mine common API usage patterns from a large corpus and use those patterns to recommend code completion for partially written programs as long as the partial program matches a prefix of a mined pattern. Such tools work well for the mined patterns; however, they cannot recommend any code outside the mined patterns—the number of mined patterns are usually limited to a few hundreds. Code clone detectors [34], [37], [38], [54] are another set of techniques that could potentially be used to retrieve recommended code snippets. However, code clone detection tools usually retrieve code snippets that are almost identical to a query snippet. Such retrieved code snippets may not always contain extra code which could be used to extend the query snippet.

We propose Aroma, a code recommendation engine. Given a code snippet as input query and a large corpus of code containing millions of methods, Aroma returns a set of recommended code snippets such that each recommended code snippet:

- contains the query snippet approximately, and
- is contained approximately in a non-empty set of method bodies in the corpus.

Furthermore, Aroma ensures that any two recommended code snippets are not quite similar to each other.

Aroma works by first indexing the given corpus of code. Then Aroma searches for a small set (e.g. 1000) of method bodies which contain the query code snippet *approximately*. We have designed this step such that the step can be completed in less than a second. For this search step, we cannot use a clone detection tool because clone detection tools are good at retrieving code snippets that are almost identical to the query snippet—Aroma's first step needs to retrieve code snippets that *contain* the query snippet in addition to extra code. Another challenge in designing this search step is that a query snippet, unlike a natural language query, has structure and such structure should be taken into account while searching for code. Once Aroma has retrieved a small set of code snippets which approximately contain the query snippet,

*This work was done in part while this author was an intern at Facebook.

AROMA prunes the retrieved snippets so that the resulting pruned snippets become similar to the query snippet. It then ranks the retrieved code snippets based on the similarity of the pruned snippets to the query snippet. This step helps to rank the retrieved snippets based on how well they contain the query snippet. The step is precise, but is relatively expensive; however, the step is only performed on a small set of code snippets, making it efficient in practice. After ranking the retrieved code snippets, AROMA clusters the snippets so that similar snippets fall under the same cluster. AROMA then intersects the snippets in each cluster to carve out a maximal code snippet which is common to all the snippets in the cluster and which contains the query snippet. The set of intersected code snippets are then returned as recommended code snippets. For the query shown in Table I, row A, left column, AROMA recommends the snippet shown in the right column of the same row.

To our best knowledge, AROMA is the first tool which could recommend relevant code snippets given a query code snippet. The advantages of AROMA are the following:

- A code snippet recommended by AROMA does not simply come from a single method body, but is generated from several similar looking code snippets via intersection. This increases the likelihood that AROMA's recommendation is idiomatic rather than one-off.
- AROMA does not require mining common coding patterns or idioms ahead of time. Therefore, AROMA is not limited to a set of mined patterns—it can retrieve new and interesting code snippets on-the-fly.
- AROMA is fast enough to use in real time. A key innovation in AROMA is that it first retrieves a small set of snippets based on approximate search, and then performs the heavy-duty pruning and clustering operations on this set. This enables AROMA to create recommended code snippets on a given query from a large corpus containing millions of methods within a couple of seconds.
- Although we developed AROMA for the purpose of code recommendation, it could be used to also perform efficient and precise code-to-code structural search.

We have implemented AROMA in C++ and have used it to index 5,417 GitHub Java Android projects. We evaluated AROMA using code snippets obtained from Stack Overflow. We manually analyzed and categorized the recommendations into several representative categories. We also evaluated AROMA recommendations on 50 partial code snippets, where we found that AROMA can recommend the exact code snippets for 37 queries, and in the remaining 13 cases AROMA recommends alternative recommendations that are still useful. On average, AROMA takes 1.6 seconds to create recommendations for a query code snippet. In our experimental evaluation, we also used a micro-benchmarking suite containing artificially created query snippets to evaluate the effectiveness of various design choices in AROMA.

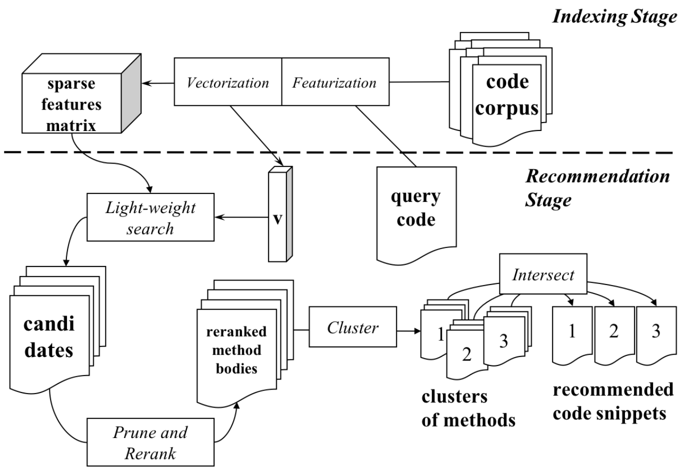

Fig. 1. AROMA Code Recommendation Pipeline.

II. ALGORITHM

Figure 1 illustrates overall architecture of AROMA. AROMA first indexes the code corpus as a sparse matrix in two steps:

- **Featurization**. AROMA parses each method in the corpus and creates its parse tree. Then it extracts a set of structural features from the parse tree of each method.
- **Vectorization**. The features of a method body are represented as a sparse binary vector whose i^{th} entry is 1 if feature i is present in the method body and 0 otherwise. The feature vectors of all method bodies are represented as a sparse matrix whose j^{th} row represents the feature vector of the j^{th} method body in the corpus.

In the recommendation stage, given a query code snippet, AROMA runs the following phases to create recommendations:

- **Light-weight Search.** AROMA first featurizes the query code's parse tree into a sparse vector. It then takes the dot product of this vector with the feature matrix. The top n_1 method bodies whose dot products are highest are retrieved as the candidate set for recommendation. Even though the code corpus could contain millions of methods, this retrieval is fast due to efficient implementations of dot products of sparse vectors and matrices. Due to the use of dot product, the ranking of the code snippets based on the dot product score may not necessarily rank the retrieved code snippets according to their closeness to the query code. Moreover, the retrieved method bodies may contain statements that are not relevant to the query code. Therefore, we first need to rerank the retrieved code snippets so that method bodies that contain more parts of the query snippet are ranked higher than the method bodies that contain fewer parts of the query snippet.
- **Prune and Rerank.** AROMA next ranks the retrieved code snippets from the previous phase based on their similarity to the query snippet. For this ranking, AROMA prunes each retrieved code snippet so that the resulting snippet becomes maximally similar to the query snippet. The similarity metric used for this pruning is based on

TABLE I
AROMA CODE RECOMMENDATION EXAMPLES

Query Code Snippet | AROMA Code Recommendation with Extra Lines Highlighted

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 8;
Bitmap preview_bitmap = BitmapFactory.decodeStream(is, null,
      options);
```

Example A: Error Handling and Additional Configurations.

- This code snippet [29] attempts to decode a bitmap without incurring high memory consumption.
- The recommended code suggests catching possible out-of-memory errors and setting additional options for decoding.
- **Intersected from a cluster of 4 methods** [3]–[6].

```
try {
    FileInputStream stream =
        new FileInputStream(new File(path+"test.jpg"));
    BitmapFactory.Options opts = new BitmapFactory.Options();
    opts.inSampleSize = 8;
    opts.inPurgeable = true;
    opts.inInputShareable = true;
    Bitmap bitmap = BitmapFactory.decodeStream(stream, null, opts);
    return bitmap;
} catch (OutOfMemoryError e) {
    return null;
} catch (Exception e) {
    return null;
}
```

```
TextView textView = (TextView) view.findViewById(R.id.textview);
SpannableString content = new SpannableString("Content");
content.setSpan(new UnderlineSpan(), 0, content.length(), 0);
textView.setText(content);
```

Example B: Configuring Objects.

- This code snippet [23] adds underline to a piece of text.
- The recommended code suggests adding a callback handler to pop up a dialog once the underlined text is touched upon.
- **Intersected from a cluster of 2 methods** [7], [8].

```
TextView licenseView = (TextView)
      findViewById(R.id.library_license_link);
SpannableString underlinedLicenseLink = new SpannableString(
    getString(R.string.library_license_link));
underlinedLicenseLink.setSpan(new UnderlineSpan(), 0,
      underlinedLicenseLink.length(), 0);
licenseView.setText(underlinedLicenseLink);
licenseView.setOnClickListener(v -> {
    FragmentManager fm = getSupportFragmentManager();
    LibraryLicenseDialog libraryLicenseDlg = new
          LibraryLicenseDialog();
    libraryLicenseDlg.show(fm, "fragment_license");
});
```

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
      R.drawable.image);
```

Example C: Post-Processing.

- This code snippet [27] again decodes a bitmap.
- The recommended code suggests applying Gaussian blur on the decoded image. While not obligatory, it shows a customary effect to be applied.
- **Intersected from a cluster of 4 methods** [9]–[12].

```
int radius = seekBar.getProgress();
if (radius < 1) {
 radius = 1;
}
Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
      R.drawable.image);
imageView.setImageBitmap(blur.gaussianBlur(radius, bitmap));
```

```
EditText et = (EditText)findViewById(R.id.inbox);
et.setSelection(et.getText().length());
```

Example D: Correlated Statements.

- This code snippet [28] moves the cursor to the end in a text area.
- The recommended code suggests also configuring the action bar to create a more focused view.
- **Intersected from a cluster of 2 methods** [13], [14].

```
super.onCreate(savedInstanceState);
setContentView(R.layout.material_edittext_activity_main);
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
getSupportActionBar().setDisplayShowTitleEnabled(false);
EditText singleLineEllipsisEt = (EditText)
      findViewById(R.id.singleLineEllipsisEt);
singleLineEllipsisEt.setSelection(
      singleLineEllipsisEt.getText().length());
```

```
PackageInfo pInfo =
      getPackageManager().getPackageInfo(getPackageName(), 0);
String version = pInfo.versionName;
```

Example E: Exact Recommendations.

- This partial code snippet (truncated from [24]) gets the current version of the application. The rest of the code snippet (not shown) catches and handles possible `NameNotFound` errors.
- The recommended code suggests the exact same error handling as in the original code snippet.
- **Intersected from a cluster of 2 methods** [15], [16].

```
try {
    PackageInfo pInfo =
          getPackageManager().getPackageInfo(getPackageName(), 0);
    String version = pInfo.versionName;
    TextView versionView = (TextView)
          findViewById(R.id.about_project_version);
    versionView.setText("v" + version);
} catch (PackageManager.NameNotFoundException ex) {
    Log.e(TAG, getString(R.string.about_error_version_not_found));
}
```

```
i.putExtra("parcelable_extra", (Parcelable) myParcelableObject);
```

Example F: Alternative Recommendations.

- This partial code snippet (adapted from [26]) demonstrates one way to attach an object to an `Intent`. The rest of the code snippet (not shown) shows a different way to serialize and attach an object.
- **Intersected from a cluster of 10 methods** including [17]–[20].

```
Intent intent = new Intent(this, BoardTopicActivity.class);
intent.putExtra(SMTHApplication.BOARD_OBJECT, (Parcelable) board);
startActivity(intent);
```

- The recommended code does not suggest the other way of serializing the object, but rather suggests a common way to complete the operation by starting an activity with an `Intent` containing a serialized object.

a modified formulation of Jaccard distance between the multi-set of features of the query and that of the pruned code snippet.

- **Cluster and Intersect.** In the final phase, AROMA clusters the code snippets and intersects the snippets in each cluster to come up with recommended code snippets. This approach of clustering and intersection helps to create succinct code snippets that have fewer irrelevant statements.

We next describe the details of each step using the code snippet shown in Listing 1 as the running example.

```
if (view instanceof ViewGroup) {
    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
        View innerView = ((ViewGroup) view).getChildAt(i);
    }
}
```

Listing 1. A code snippet adapted from [25]. We use this snippet as our running example through Section II.

A. Definitions

Definition 1 (Non-keyword tokens)**.** *This is the set of all tokens in a language whose values are not fixed. Non-keyword tokens include variable names, method names, field names, and literals. Examples are* `i`*,* `length`*, 0, 1, etc. The set of non-keyword tokens is non-finite for most languages.*

Definition 2 (Keyword tokens)**.** *This is the set of all tokens that are not non-keyword tokens. Keyword tokens include keywords such as* `while`*,* `if`*,* `else`*, and symbols such as* `{`*,* `}`*,* `.`*,* `+`*,* `*`*. The set of all keyword tokens is finite for a language.*

Definition 3 (Simplified Parse Tree)**.** *A simplified parse tree is a data structure we use to represent a program. It is recursively defined as a non-empty list whose elements could be any of the following:*

- *a non-keyword token,*
- *a keyword token, or*
- *a simplified parse tree.*

A simplified parse tree cannot be a list containing a single simplified parse tree.

We picked this particular representation of programs instead of conventional abstract syntax tree representation because the representation only consists of program tokens and does not use any special language-specific rule names such as `IfStatement`, `block` etc. As such the representation can be used uniformly across various programming languages. Moreover, one could perform an in-order traversal of a simplified parse tree and print the token names to obtain the original program albeit unformatted.

Definition 4 (Label of a Simplified Parse Tree)**.** *The label of a simplified parse tree is obtained by concatenating all the elements of the list representing the tree as follows:*

- *If an element is a keyword token, the value of the token is used for concatenation.*
- *If an element is a non-keyword token or a simplified parse tree, the special symbol # is used for concatenation.*

Figure 2 shows the simplified parse tree of the entire code snippet in Listing 1. In the tree, each internal node also represents a simplified parse tree whose elements are shown as its children. We show the label of each node in the tree, and add an unique index to each label as subscript to distinguish between any two similar labels.

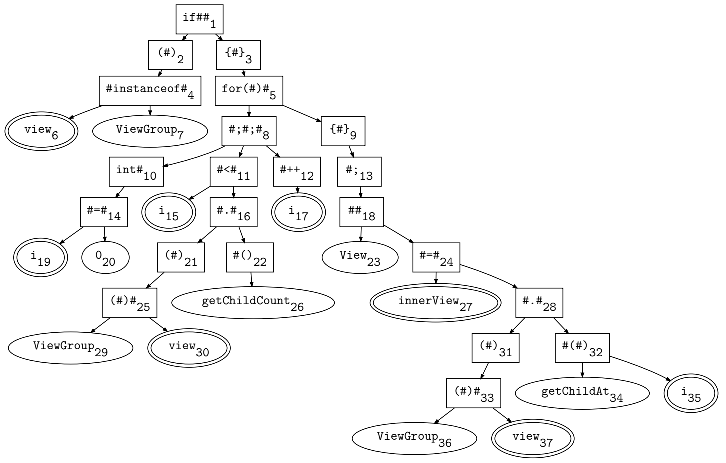

Fig. 2. The simplified parse tree representation of the code in Listing 1. Keyword tokens at the leaves are omitted to avoid clutter. Variable nodes are highlighted in double circle.

TABLE II
FEATURES FOR SELECTED TOKENS IN FIGURE 2

	Token Feature	Parent Features	Sibling Features	Variable Usage Features
`view`$_{30}$	`#VAR`	(`#VAR`, 4, `(#)#`) (`#VAR`, 2, `(#)`) (`#VAR`, 1, `#.#`)	(`ViewGroup`, `#VAR`) (`#VAR`, `getChildCount`)	((1, `#instanceof#`), (4, `(#)#`)) ((4, `(#)#`), (4, `(#)#`))
0_{20}	0	(0, 3, `#=#`) (0, 2, `int#`) (0, 1, `#;#;#;`)	(`#VAR`, 0) (0, `#VAR`)	-

Given a simplified parse tree t, we use the following notations. All examples refer to Figure 2.

- $L(t)$ denotes the label of the tree t. E.g. $L(\texttt{if\#\#}_1) = \texttt{if\#\#}$.
- $N(t)$ denotes the list of all non-keyword tokens present in t or in any of its sub-tree, in the same order as appearing in the source code. E.g. $N(\texttt{\#.\#}_{28}) = [\texttt{ViewGroup}_{36}, \texttt{view}_{37}, \texttt{getChildAt}_{34}, \texttt{i}_{35}]$.
- If n is a non-keyword direct child of t, then we use $P(n)$ to denote the parent of n which is t. E.g. $P(\texttt{view}_6) = \texttt{\#instanceof\#}_4$.
- If t' is a simplified parse tree and is a direct child of t, then we again use $P(t')$ to denote the parent of t' which is t. E.g. $P(\texttt{\#instanceof\#}_4) = \texttt{(\#)}_4$.
- If n_1 and n_2 are two non-keyword tokens in a program and if n_2 appears after n_1 in the program without any intervening non-keyword token, then we use $\texttt{Prev}(n_2)$ to denote n_1 and $\texttt{Next}(n_1)$ to denote n_2. E.g. $\texttt{Prev}(\texttt{view}_{30}) = \texttt{ViewGroup}_{29}, \texttt{Next}(\texttt{ViewGroup}_{29}) = \texttt{view}_{30}$.

- If n_1 and n_2 are two non-keyword tokens denoting the same local variable in a program and if n_1 and n_2 are the two consecutive usages of the variable in the program, then we use $\texttt{PrevVar}(n_2)$ to denote n_1 and $\texttt{NextVar}(n_1)$ to denote n_2. E.g. $\texttt{PrevVar}(\texttt{view}_{30}) = \texttt{view}_6, \texttt{NextVar}(\texttt{view}_{30}) = \texttt{view}_{37}$.
- If n is a non-keyword token denoting a local variable and it is the i^{th} child of its parent t, then the context of n, denoted by $C(n)$, is defined to be:
  - $(i, L(t))$, if $L(t) \neq \texttt{\#.\#}$. E.g. $C(\texttt{view}_{30}) = (2, \texttt{(\#)\#})$.
  - The first non-keyword token that is not a local variable in $N(t)$, otherwise. This is to accommodate for cases like `x.foo()`, where we want the context feature for `x` to be `foo` rather than $(1, \texttt{\#.\#})$, because the former better reflects its usage context.

B. Featurization and Vectorization

Given a simplified parse tree, we extract four kinds of features for each non-keyword token n in the program represented by the tree:

1) *Token Feature* of the form n. If n is a local variable, then we replace n with the special token `#VAR`. We want to ignore local variable names because we consider a query code snippet and a recommended code snippet to be similar even if they differ with respect to alpha renaming of variables. For example, for the query code in Figure 1, if we have a code snippet similar to the query code except that every occurrence of `i` is replaced with `j`, we want to recommend that code snippet.
2) *Parent Features* of the form $(n, i_1, L(t_1))$, $(n, i_2, L(t_2))$, and $(n, i_3, L(t_3))$. Here n is the i_1^{th} children of t_1, t_1 is the i_2^{th} children of t_2, and t_2 is the i_3^{th} children of t_3. As before, if n is a local variable, then we replace n with `#VAR`. These features help to capture some of the structural properties of a code snippet. Note that in each of these features, we do not specify if the third element in a feature is the parent, grand-parent, or the great-grand parent. This helps AROMA to tolerate some non-similarities in otherwise similar code snippets.
3) *Sibling Features* of the form $(n, \texttt{Next}(n))$ and $(\texttt{Prev}(n), n)$. As before, if any of $n, \texttt{Next}(n), \texttt{Prev}(n)$ is a local variable, it is replaced with `#VAR`.
4) *Variable Usage Features* of the form $(C(\texttt{PrevVar}(n)), C(n))$ and $(C(n), C(\texttt{NextVar}(n)))$. We only add these features if n is a local variable. Since we ignore the variable names, we need an approximate way to identify two variable usages in a code snippet which refer to the same local variable. Variable usage features help us to relate to two variable usages that refer to the same variable.

For a non-keyword token $n \in N(t)$, we use $F(n)$ to denote the multi-set of features extracted for n. We extend the definition of F to a set of non-keyword tokens Q as follows: $F(Q) = \uplus_{n \in Q} F(n)$ where $\uplus$ denotes multi-set union. For a simplified parse tree t, we use $F(t)$ to denote the multi-set of features of all non-keyword tokens in t, i.e. $F(t) = F(N(t))$. Let $\mathcal{F}$ be the set of all features that can extracted from a given corpus of code.

Table II illustrates the features extracted for tokens selected from the simplified parse tree in Figure 2.

C. Recommendation Algorithm

1) Phase I: Light-weight Search: We assume that we are given a large corpus of programs containing millions of methods. AROMA parses and creates a simplified parse tree for each method body. It then featurizes each simplified parse tree. Let M be the set of simplified parse trees of all method bodies in the corpus. AROMA also parses the query code snippet to create the simplified parse tree, say q, and extracts its features. AROMA's goal is to find a list of method bodies that approximately contains the query code snippet. To do so, AROMA first uses an imprecise, but light-weight technique, to compute a short list of method bodies that could potentially contain the query code snippet. For the simplified parse tree m of each method body in the corpus, we use the cardinality of the set $S(F(m)) \cap S(F(q))$ as an approximate score, called *containment score*, of how much of the query code snippet is contained in the method body. Here $S(X)$ denotes the set of elements of the multi-set X. AROMA computes a set of n_1 method bodies whose containment scores are highest with respect to the query code snippet. In our implementation n_1 is usually 1000.

The computation of the list can be reduced to a simple sparse dot product between a matrix and a vector as follows. The features of a code snippet can be represented as a sparse vector of length $|\mathcal{F}|$—the vector has an entry for each feature in $\mathcal{F}$. If a feature f_i is present in $F(m)$, the multi-set of features of the simplified parse tree m, then the i^{th} entry of the vector is 1 and 0 otherwise. Note that the elements of each vector can be either 0 or 1—we ignore the count of each feature in the vector. This is because if the count, say n, of a feature f is high for some method body m and if its count is 1 in the query snippet, then the final containment score will depend on n which is undesirable because more f in the feature multi-set of m does not imply that q is more contained in m. The sparse feature vectors of all method bodies can then be organized as a matrix, say D, of shape $|M| \times |\mathcal{F}|$. Let Q be the sparse feature vector of the query code snippet q. Then $D \cdot Q^T$ is a vector of size $|M|$ and gives the containment score of each method body with respect to the query snippet. AROMA picks the top n_1 method bodies with the highest containment scores. Let N_1 be the set of simplified parse trees of the method bodies picked by AROMA. Note that matrix multiplication described above can be done efficiently using a fast sparse matrix multiplication library. The correlation between the degree of containment of a query code snippet in a method body and the containment score is approximate, but it enables AROMA to isolate a small set of method bodies which are likely to contain the query code snippet. AROMA can then perform more precise containment analysis between q and the code snippets in N_1 to obtain the final ranked list of code snippets containing the query snippet.

2) Phase II: Prune and Rerank: In this phase, AROMA computes the *exact containment score* of a method body with respect to the query snippet. To do so, AROMA prunes the simplified parse tree m of the method body, so that the similarity between the simplified parse tree of the query and the pruned tree m', is maximized. The similarity score between q and m', denoted by $\mathit{Sim}(q, m')$, is defined as follows:

$$\mathit{Sim}(q, m') = \mathit{jaccardMod}(F(q), F(m')) = \frac{|F(q) \cap F(m')|}{|F(q)|}. \tag{1}$$

We find this modified Jaccard similarity metric to give better pruning results. The exact containment score between q and m, denoted by $\sigma(q, m)$ is then defined as $\mathit{Sim}(q, m')$. The computation of the optimal pruned simplified parse tree m' requires us to find a subset R of $N(m)$ such that $\mathit{jaccardMod}(F(q), F(R))$ is maximal. m' then consists of the nodes in R and any internal nodes in m which is along a path from any $n \in P$ to the root node in m. Computing the optimal m' requires us to enumerate all subsets of $N(m)$ which is exponential in the size of $N(m)$. In order to perform the pruning step efficiently, AROMA uses a quadratic time greedy algorithm to approximately compute m' as follows:

1) $R \leftarrow \emptyset$.
2) $F \leftarrow \emptyset$.
3) Find n such that

   $$n = \operatorname{argmax}_{n' \in N(m) - R} \mathit{jaccardMod}(F(q), F \uplus F(n'))$$

   and

   $$\mathit{jaccardMod}(F(q), F \uplus F(n)) > \mathit{jaccardMod}(F(q), F).$$
4) If such an n exists, then $R \leftarrow R \cup \{n\}$ and $F \leftarrow F \uplus F(n)$. Go back to Step 3.
5) Else return m' where m' is obtained from m by retaining all the non-keyword tokens in R and any internal node which appears in a path from a $n \in R$ and the root of m. The similarity score between q and m', which is also the exact containment score between q and m or $\sigma(q, m)$, is given by $\mathit{jaccardMod}(F(q), F)$. We use $\texttt{Prune}(F(q), m)$ to denote m'.

AROMA reranks the trees in N_1 in descending order using the exact containment scores between the trees in N_1 and q. Let N_2 be the list of the trees after reranking. We call this list the ***reranked search results***. Note that this pruning algorithm may not find the optimal parse tree because we are using a greedy algorithm. In the evaluation section, we show that in very rare cases the greedy pruning algorithm may not give us the best recommended code snippets.

Listing 2 shows a code snippet from the reranked search results for the query code snippet in Listing 1. In the code snippet, the highlighted tokens are selected by the pruning algorithm to maximize the similarity score to the query snippet. In this example, the exact containment score of this method is 1.0 (which is the max score) because the multi-set union of the features of the highlighted token matches the feature set of the query code snippet perfectly.

3) Phase III: Cluster and Intersect: From the list of method bodies N_2 (i.e. the reranked search results), AROMA picks the top n_2 methods with the highest exact containment scores that are also above a predefined threshold τ_1. In our implementation $n_2 = 100$ and $\tau_1 = 0.65$. AROMA then clusters the code snippets based on similarity and intersects the snippets in each cluster to get a concise code snippet for recommendation.

We use $N_2(i)$ to denote the tree at index i in the list N_2. A cluster is a tuple of indices of the form $(i_1, \ldots, i_k)$, where $i_j < i_{j+1}$ for all $1 \leq j < k$. A tuple $(i_1, \ldots, i_k)$ denotes that a cluster containing the code snippets $N_2(i_1), \ldots, N_2(i_k)$. We define the commonality score of the tuple $\tau = (i_1, \ldots, i_k)$ as

$$\texttt{cs}(\tau) = | \cap_{1 \leq j \leq k} F(N_2(i_j))|$$

Similarly, we define the commonality score of the tuple $\tau = (i_1, \ldots, i_k)$ with respect to the query q as

$$\texttt{csq}(\tau) = | \cap_{1 \leq j \leq k} F(\texttt{Prune}(F(q), N_2(i_j)))|$$

We say that a tuple $\tau = (i_1, \ldots, i_k)$ is a valid tuple or a valid cluster if

1) $\texttt{l}(\tau) = \texttt{cs}(\tau)/\texttt{csq}(\tau)$ is greater than some user-defined threshold τ_2 (which is 1.5 in our experiments). This ensures that after intersecting all the snippets in the cluster, we get a snippet that is at least τ_2 times bigger than the query code snippet.
2) $\texttt{s}(\tau) = \texttt{csq}(\tau)/|F(N_2(i_1))|$ is greater than some user-defined threshold τ_3 (which is 0.9 in our experiments). This requirement ensures that the trees in the cluster are quite similar to each other. Specifically, it says that the intersection of the pruned snippets in a cluster should be very similar to the first pruned snippet.

The set of valid tuples $\mathcal{C}$ is computed iteratively as follows:

1) $\mathcal{C}_1$ is the set $\{(i) \mid 1 \leq i \leq |N_2| \text{ and } (i) \text{ is a valid tuple}\}$.
2) $\mathcal{C}_{\ell+1} = \{(i_1, \ldots, i_\ell, i) \mid (i_1, \ldots, i_\ell) \in \mathcal{C}_\ell$ and $i_\ell < i \leq |N_2|$ and $(i_1, \ldots, i_\ell, i)$ is a valid tuple and $l(i_1, \ldots, i_\ell, i) = \operatorname{argmax}_{i_l < i_k \leq |N_2|} l((i_1, \ldots, i_\ell, i_k))\} \cup \mathcal{C}_\ell$

AROMA computes $\mathcal{C}_1, \mathcal{C}_2, \ldots$ iteratively until it finds an ℓ such that $\mathcal{C}_\ell = \mathcal{C}_{\ell+1}$. $\mathcal{C} = \mathcal{C}_\ell$ is then the set of all clusters.

After computing all valid tuples, AROMA sorts the tuples in ascending order first on the first index in each tuple and then on the negative length of each tuple. It also drops any tuple τ from the list if it is similar (i.e. has a Jaccard similarity more than 0.5) to any tuple appearing before τ in the sorted list. This ensures that the recommended code snippets are not quite similar to each other. Let N_3 be the sorted list of the remaining clusters.

Given a tuple $\tau = (i_1, \ldots, i_k)$, $\texttt{Intersect}(\tau, q)$ returns a code snippet that is the intersection of the code snippets $N_2(i_1), \ldots, N_2(i_k)$ while ensuring that we retain any code that is similar to q. $\texttt{Intersect}((i_1, \ldots, i_k), q)$ is defined recursively as follows:

- $\texttt{Intersect}((i_1), q) = \texttt{Prune}(F(q), N_2(i_1))$.

- $\texttt{Intersect}((i_1, i_2), q) = \texttt{Prune}(F(N_2(i_2)) \uplus F(q), N_2(i_1)).$
- $\texttt{Intersect}((i_1, \ldots, i_j, i_{j+1}), q) = \texttt{Prune}(F(N_2(i_{j+1})) \cup F(q), \texttt{Intersect}((i_1, \ldots, i_j), q)).$

AROMA picks the top K (where $K = 10$ in our implementation) tuples from N_3 and returns the intersection of each tuple with the query code snippet as recommendations.

In the running example, after finding all clusters, Listing 2 and Listing 3 will form a cluster. AROMA will prune Listing 2 with respect to the union set of features of the query code and Listing 3 as the intersection between Listing 2 and Listing 3. The intersection is taken as the final recommendation for the query code, which is shown in Listing 4.

```
if (!(view instanceof EditText)) {
    view.setOnTouchListener(new View.OnTouchListener() {
        public boolean onTouch(View v, MotionEvent event) {
            hideKeyBoard();
            return false;
        }
    });
}
if (view instanceof ViewGroup) {
    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
        View innerView = ((ViewGroup) view).getChildAt(i);
        setupUIToHideKeyBoardOnTouch(innerView);
    }
}
```

Listing 2. A method body [21] containing the query code snippet in Listing 1. The pruning algorithm selects the tokens as highlighted. The multi-set union of their features is identical to the feature set of the query code snippet.

```
if (!(view instanceof EditText)) {
    view.setOnTouchListener(new View.OnTouchListener() {
        public boolean onTouch(View v, MotionEvent event) {
            Utils.toggleSoftKeyBoard(LoginActivity.this, true);
            return false;
        }
    });
}
if (view instanceof ViewGroup) {
    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
        View innerView = ((ViewGroup) view).getChildAt(i);
        setupUI(innerView);
    }
}
```

Listing 3. Another method [22] containing the query code snippet in Listing 1.

```
if (!(view instanceof EditText)) {
    view.setOnTouchListener(new View.OnTouchListener() {
        public boolean onTouch(View v, MotionEvent event) {
            // your code...
            return false;
        }
    });
}
if (view instanceof ViewGroup) {
    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
        View innerView = ((ViewGroup) view).getChildAt(i);
        setupUIToHideKeyBoardOnTouch(innerView);
    }
}
```

Listing 4. A recommended code snippet created by intersecting code in Listing 2 and Listing 3. Extra lines are highlighted.

III. EVALUATION

A. Datasets

We evaluated the effectiveness of AROMA code recommendation by instantiating AROMA on 5,417 GitHub projects where Java is the main language and Android is the project topic. We ensured the quality of the corpus by picking projects that are not forked from other projects, and are with at least 5 stars. Previous study [42] shows that duplication exists pervasively on GitHub. To make sure AROMA recommendations are created from multiple different code snippets, rather than the same code snippet duplicated in multiple locations, we removed duplicates at project level, file level, and method level. We do this by taking hashes of these entities and by comparing these hashes. After removing duplicates, the corpus contains 2,417,125 methods.

For evaluation, we picked 500 most popular questions on Stack Overflow with the *android* tag. From these questions, we only considered the top voted answers. From each answer, we extracted all Java code snippets containing at least 3 tokens, a method call, and less than 20 lines, excluding comments. We randomly picked 64 from this set of Java code snippets. We then used these code snippets as queries to evaluate the quality of AROMA's recommendations.

On average, AROMA takes 1.6 seconds end-to-end to create recommendations. The median response time is 1.3s and 95% queries complete in 4 seconds. We believe this makes AROMA suitable for integration into the development environment as a code recommendation tool.

B. Recommendation Quality Evaluation

For each of the 64 query code snippets, we inspected the recommended code snippets, manually assessed their quality, and classified the recommended snippets into several categories. Two of the author did the manual inspection and categorization. The other authors verified the results. Note that a recommended code snippet can fall under multiple categories.

1) Configuring Objects: The recommended code suggests additional configurations on objects that are already appearing in the query code. Examples include adding additional callback handlers, and setting additional flags and properties of an existing object. Example B in Table I shows an example of this category.

2) Error Checking and Handling: The recommended code adds null checks and other checks before using an object, or adds a try-catch block that guards the original code snippet. Such additional statements are useful reminders to developers that the program might enter an erroneous state or even crash at runtime if exceptions and corner cases are not carefully handled. Example A in Table I shows an example of this category.

3) Post-processing: The recommended code extends the query code to perform some common operations on the objects or values computed by the query code. For examples recommended code can show API methods that are commonly called. Example C in Table I shows an example of this category.

4) Correlated Statements: The recommended code adds statements that do not affect the original functionalities of the query code, but rather suggests related statements that commonly appear alongside the query code. In Example D in Table I, the original code moves the cursor to the end of text in an editable text area, where the recommended code also configures the Android Support Action Bar to show the home button and hide the activity title in order to create a more focused view. These statements are not directly related to the text view, but are common in real-world.

5) Unclustered Recommendations: In rare cases, the query code snippet could match method bodies that are mostly different from each other. This results in clusters of size 1. In these cases, AROMA performs no intersection and recommends the full method bodies without any pruning.

The number of recommended code snippets for each category is listed in Table III. For recommendations that belong to multiple categories, we count them for each of the categories. In this manual inspection, we observed that AROMA provided useful recommendations for a majority of query code snippets from the Stack Overflow dataset.

TABLE III
CATEGORIES OF AROMA RECOMMENDATIONS

Configuring Objects	17
Error Checking and Handling	14
Post-processing	16
Correlated Statements	21
Unclustered Recommendations	5

C. Recommendation Quality on Partial Code Snippets

In this experiment, instead of using the full code snippets from Stack Overflow, we manually created partial code snippets by taking the first half of the statements of each snippet. Since each full code snippet from Stack Overflow represents a popular coding pattern, we wanted to check whether AROMA could recommend the missing statements in the code snippet given the partial query code snippet.

We could not extract partial query code snippets from 14 out of 64 code snippets because they contained a single statement. For the remaining 50 query code snippets, AROMA recommendations fall into the following two categories.

1) Exact recommendations.: In 37 cases, one of the AROMA recommendations matched the original code snippet. Example E in Table I shows a partial query snippet which included the first two statements in a try-catch block of a Stack Overflow code snippet, and AROMA recommended the same error handling code as in the original code snippet.

2) Alternative recommendations.: In the other 13 cases, none of the AROMA recommended code snippets matched the original snippets. While in each case the recommended snippets did not contain the original usage pattern, they still fall in some of the categories in Table III. Example F in Table I shows a partial code snippet which included one of two common ways to send an object with an `Intent`. Given the statement, AROMA did not recommend the other way to serialize an object in the original code snippet, but suggested a customary way to start an activity with an `Intent` containing a serialized object.

D. Comparison with Pattern-Oriented Code Completion

Pattern-oriented code completion tools [48]–[50] could also be used for code recommendation. For example, GRAPACC [49] proposed to use mined API usage patterns for code completion. We took the dataset of 15 Android API usage patterns manually curated from Stack Overflow posts and Android documentation by the authors of BIGGROUM [48]. We used this dataset because BIGGROUM has been shown to scale for large corpus and can mine more common patterns than existing work. Among these 15 snippets, 11 were found in BIGGROUM mining results. Therefore, if GRAPACC is instantiated on the patterns mined by BIGGROUM, 11 out of the 15 patterns could be recommended by GRAPACC.

In order to evaluate AROMA we followed the same methodology as in Section III-C to create a partial query snippet from each of the 15 full patterns, and checked if any of the AROMA recommended code snippets contained the full pattern. For 14 out of 15 patterns, AROMA recommended code containing the original usage patterns, i.e. they are *exact recommendations* as defined in Section III-C1. An advantage of AROMA is that it could recommend code snippets that do not correspond to any previously mined pattern by BIGGROUM. Moreover, AROMA could recommend code which may not contain any API usage.

IV. MICRO-BENCHMARKING

One of the most important and novel phases of the AROMA's recommendation algorithm is phase II: prune and rerank, which produces the *reranked search results*. The purpose of this phase is to rank the search results from phase I (i.e. the light-weight search phase) so that any method containing most part of the query code is ranked higher than a method body containing a smaller part of the query code. Therefore, if a method contains the entire query code snippet, it should be ranked top in the reranked search result list. However, in rare cases this property of AROMA may not hold due to two reasons: 1) AROMA's pruning algorithm is greedy and approximate due to efficiency reasons, and 2) the kinds of features that we extract may not be sufficient.

To evaluate the precision of the prune and rerank phase, we created a micro-benchmark dataset by extracting partial query code snippets from existing method bodies in the corpus. On each of these query snippets, AROMA should rank the original method body as number 1 in the reranked search result list. We created two kinds of query code snippets for this micro-benchmark:

- *Contiguous code snippets.* We randomly sampled 1000 method bodies with at least 12 lines of code. From each method body we take the first 5 lines to form a partial query code snippet.
- *Non-contiguous code snippets.* We again randomly sampled 1000 method bodies with at least 12 lines of code. From each method body we randomly sample 5 lines to form a partial query code snippet.

A. Recall in Prune and Rerank Phase

Recall@n is defined as the percentage of query code snippets for which the original method body is found in the top n methods in the reranked search result list. In addition to Recall@1, we considered Recall@100 because the first 100 methods in the reranked list are used in the clustering phase to create recommended code.

The result in the last row of Table IV shows that AROMA is always able to retrieve the original method in the top 100 methods in the reranked search result list. For 99.1% contiguous code queries and for 98.3% non-contiguous query code snippets, AROMA was able to retrieve the original method as the top-ranked result in the reranked search result list.

Listing 5 demonstrates a rare case where the original method was not retrieved as the top result. Since AROMA's pruning algorithm is greedy (Section II-C2), it erroneously decided to pick the statement at line 7, because the statement contains a lot of features which overlap with that of the query code despite the absence of that statement in the query code. This results in an imperfect similarity score of 0.984. Since there are other code snippets with similar structures which achieves a perfect similarity score of 1.0, the original method was not retrieved at rank 1. Fortunately, this scenario happens rarely enough that it does not affect overall recall.

```
1   int result = 0;
2   int cur;
3   int count = 0;
4   do {
5       cur = in.readByte() & 0xff;
6       result |= (cur & 0x7f) << (count * 7);
7       count++;
8   } while (((cur & 0x80) == 0x80) && count < 5);
9   if ((cur & 0x80) == 0x80) {
10      throw new DexException("invalid LEB128 sequence");
11  }
12  return result;
```

Listing 5. Lines 1–5 and 8 were used as query. The pruning algorithm selected the wrong token (as highlighted) that does not belong to the query, resulting in an imperfect similarity score of 0.984.

B. Evaluation of Design Choices

In this section, we compare the modified Jaccard similarity metric used in the pruning algorithm with two other similarity metrics in terms of recall on the same micro-benchmark dataset:

1) Traditional Jaccard Similarity:

$$jaccard(A, B) = \frac{|A \cap B|}{|A \uplus B|},$$

where A and B are multi-sets.

2) Cosine Similarity:

$$cosine(A, B) = \frac{\sum_{x \in S(A \uplus B)} \#_A(x) \cdot \#_B(x)}{\sqrt{\sum_{x \in A} \#_A^2(x)}\sqrt{\sum_{x \in B} \#_B^2(x)}}$$

where $\#_A(x)$ denotes the count of x in multi-set A.

We created two variants of AROMA by replacing *jaccardMod* in the pruning algorithm (see Section II-C2) with *jaccard* and *cosine*. We call the original AROMA as *jaccardMod*, and the two variants as *jaccard* and *cosine*, respectively.

Table IV shows the recall results. The results show that the modified Jaccard similarity yields the highest recall on both contiguous and non-contiguous code queries. Therefore, we used the modified Jaccard similarity metric in the pruning algorithm.

TABLE IV
COMPARISON OF DIFFERENT SIMILARITY METRICS

	Contiguous		Non-contiguous	
	Recall@1	Total Recall	Recall@1	Total Recall
cosine	95.1%	97.9%	90.0%	97.1%
jaccard	98.9%	**100%**	97.6%	**100%**
jaccardMod	**99.1%**	**100%**	**98.3%**	**100%**

C. Comparing with Clone Detectors and Conventional Search Techniques

AROMA's search and pruning phases are somewhat related to clone detection and traditional code search techniques. In principle, AROMA can use a clone detector or a traditional code search technique to first retrieve a list of methods that contain the query code snippet, and then cluster and intersect the methods to get recommendations.

1) Clone Detectors: We instantiated SOURCERERCC [54], a state-of-the-art clone detector that supports Type-3 clone detection, on the same corpus indexed by AROMA. We then used SOURCERERCC to find clones of the same 1000 contiguous and non-contiguous queries in the micro-benchmark suite. SOURCERERCC retrieved all similar methods above a certain similarity threshold which is 0.7 by default. However, it does not provide any similarity score between two code snippets, so we were unable to rank the retrieved results and report recall at a specific ranking.

SOURCERERCC's recall was 12.2% and 7.7% for contiguous and non-contiguous code queries, respectively. Since SOURCERERCC is designed to find almost similar code with slight variations, it could not retrieve large method bodies containing the query code snippet. We also found that in many cases SOURCERERCC found code snippets *shorter* than the query snippet, which by definition are Type-3 clones, but are not useful for AROMA for generating code recommendations. Therefore, we cannot use a clone detector to replace the search mechanism used in AROMA.

2) Conventional Search Using TF-IDF and Structural Features: We implemented a conventional code search technique using classic TF-IDF [55]. Specifically, instead of creating a binary vector in the vectorization stage, we created a normalized TF-IDF vector. We then created the sparse index matrix by combining the sparse vectors for every method body. The $(i, j)^{\text{th}}$ entry in the matrix is defined as:

$$tfidf(i, j) = (1 + \log tf(i, j)) \cdot \log \frac{J}{df(i)}$$

where $tf(i,j)$ is the count of occurrences of feature i in method j, and $df(i)$ is the number of methods in which feature i exists. J is the total number of methods. During retrieval, we created a normalized TF-IDF sparse vector from the query code snippet, and then took its dot product with the feature matrix. Since all vectors are normalized, the result contains the cosine similarity between the feature vectors of the query and of every method. We then returned the list of methods ranked by their cosine similarities.

3) Conventional Search Using TF-IDF and Keywords: We implemented another conventional code search technique by simply treating a method body as a bag of words and using the standard TF-IDF technique for retrieval. To do so, we extracted words instead of structural features from each token, and used the same vectorization technique as in Section IV-C2.

As shown in Table V, the recall rates of both conventional search techniques are considerably lower than AROMA. We observed that in many cases, though the original method was present in the top 100 results, it was not the top result because there are other methods with higher similarity scores due to more overlapping features or keywords. Without pruning, there is no way to determine how well a method contains the query code snippet. This experiment shows that pruning is essential in order to create a precise ranked list of search results.

TABLE V
COMPARISON WITH CLONE DETECTORS AND CONVENTIONAL SEARCH

	Contiguous		Non-contiguous	
	Recall@1	Recall@100	Recall@1	Recall@100
SCC	(12.2%)		(7.7%)	
Keywords Search	78.3%	96.9%	93.0%	99.9%
Features Search	78.3%	96.8%	88.1%	98.6%
AROMA	**99.1%**	**100%**	**98.3%**	**100%**

V. RELATED WORK

Code search engines. Code-to-code search tools like FaCoY [40], Searchcode [2], and Krugle [1] take a code snippet as query and retrieve relevant code snippets from the corpus. FaCoY aims at finding semantically similar results for input queries. Given a code query, it first searches in a Stack Overflow dataset to find natural language descriptions of the code, and then find related posts and similar code. While these code-to-code search tools retrieve similar code at different syntactic and semantic levels, they do not attempt to create concise recommendations from their search results. Most of these search engines cannot be instantiated on our code corpus, so we could not experimentally compare AROMA with these search engines. Instead, we compared AROMA with two conventional code search techniques based on featurization and TF-IDF in Section IV-C, and found that AROMA's pruning-based search technique in Phase II outperforms both techniques.

Many efforts have been made to improve keyword-based code search [30], [33], [41], [43]–[46], [53], [61]. For example, CodeGenie [41] uses test cases to search and reuse source code; SNIFF [33] works by inlining API documentation in its code corpus. It also intersects the search results to provide recommendations, but only targets at resolving natural language queries. MAPO [61] recommends code examples by mining and indexing associated API usage patterns. Portfolio [46] retrieves functions and visualizes their usage chains. CodeHow [43] augments the query with API calls which are retrieved from documentation to improve search results. CoCaBu [56] augments the query with structural code entities. These techniques aim at discovering API usage patterns given a keyword query, but do not support directly using code as query.

Clone detectors. Clone detectors are designed to detect syntactically identical or highly similar code. SOURCERERCC [54] is a token-based clone detector targeting Type 1,2,3 clones. Compared with other clone detectors that also support Type 3 clones including NiCad [34], Deckard [37], and CCFinder [38], SOURCERERCC has high precision and recall and also scales to large-size projects. One may repurpose a clone detector to find similar code, but since it is designed for finding highly-similar code rather than code that contain the query code snippet, as demonstrated in Section IV-C its results are not suitable for code recommendation.

Recent clone detection techniques explored other research directions from finding semantically similar clones [39], [40], [60] to finding gapped clones [58] and gapped clones with large number of edits (large-gapped clones) [59]. These techniques may excel in finding a particular type of clone, but they sacrifice the precision and recall for Type 1 to 3 clones.

Pattern mining and code completion. Code completion can be achieved by different approaches from extracting the structural context of the code to mining recent histories of editing [31], [35], [36], [52]. GraPacc [49] achieves pattern-oriented code completion in two steps: they first mine the idioms as graph-represented patterns using GrouMiner [50], then take input tokens as hint to search for these graphs and produce code completion suggestions. BigGroum [48] solves the scalability issue of GrouMiner by first clustering the groums into distinct itemsets. Pattern-oriented code completion techniques require elaborate effort to mine the usage patterns for code completion, and therefore cannot recommend any code outside of the mined patterns, while AROMA does not require any pattern mined ahead of time and retrieve relevant snippets on-the-fly.

Other techniques for discovering API usages [32], [47], [51], [57] either target natural language or keyword queries, or do not recommend code snippets with rich structures like error handling.

REFERENCES

[1] http://opensearch.krugle.org/. [Online; accessed in July 2018].
[2] https://searchcode.com/. [Online; accessed in July 2018].
[3] https://github.com/cheng2016/LoginSdk/blob/master/src/com/example/loginsdk/util/Util.java#L323, accessed in August 2018.
[4] https://github.com/zom/Zom-Android/blob/master/app/src/main/java/org/awesomeapp/messenger/util/SecureMediaStore.java#L125, accessed in August 2018.

[5] https://github.com/zom/Zom-Android/blob/master/app/src/main/java/org/awesomeapp/messenger/ui/widgets/SecureCameraActivity.java#L82, accessed in August 2018.

[6] https://github.com/junchenChow/checkin-mask-view/blob/master/app/src/main/java/me/checkin/android/helper/BitmapHelper.java#L162, accessed in August 2018.

[7] https://github.com/tonyvu2014/android-shoppingcart/blob/master/demo/src/main/java/com/android/tonyvu/sc/demo/ProductActivity.java#L58, accessed in August 2018.

[8] https://github.com/tonyvu2014/android-shoppingcart/blob/master/demo/src/main/java/com/android/tonyvu/sc/demo/MainActivity.java#L23, accessed in August 2018.

[9] https://github.com/TonnyL/GaussianBlur/blob/master/app/src/main/java/io/github/marktony/gaussianblur/MainActivity.java#L58, accessed in August 2018.

[10] https://github.com/TonnyL/GaussianBlur/blob/master/app/src/main/java/io/github/marktony/gaussianblur/MainActivity.java#L28, accessed in August 2018.

[11] https://github.com/irccloud/android/blob/master/src/com/irccloud/android/activity/ImageViewerActivity.java#L149, accessed in August 2018.

[12] https://github.com/REBOOTERS/My-MVP/blob/master/app/src/main/java/huyifei/mymvp/GlideActivity.java#L34, accessed in August 2018.

[13] https://github.com/cymcsg/UltimateAndroid/blob/master/deprecated/UltimateAndroidGradle/demoofui/src/main/java/com/marshalchen/common/demoofui/sampleModules/MaterialEditTextActivity.java#L14, accessed in August 2018.

[14] https://github.com/klinker24/Android-SlidingMessaging/blob/master/src/main/java/com/klinker/android/messaging_sliding/scheduled/NewScheduledSms.java#L105, accessed in August 2018.

[15] https://github.com/front-line-tech/background-service-lib/blob/master/SampleService/servicelib/src/main/java/com/flt/servicelib/AbstractPermissionExtensionAppCompatActivity.java#L53, accessed in August 2018.

[16] https://github.com/brarcher/video-transcoder/blob/master/app/src/main/java/protect/videotranscoder/activity/MainActivity.java#L1597, accessed in August 2018.

[17] https://github.com/zfdang/zSMTH-Android/blob/master/app/src/main/java/com/zfdang/zsmth_android/MainActivity.java#L736, accessed in August 2018.

[18] https://github.com/triline3/timecat/blob/master/app/src/main/java/com/time/cat/util/SharedIntentHelper.java#L148, accessed in August 2018.

[19] https://github.com/jrummyapps/BusyBox/blob/master/app/src/main/java/com/jrummyapps/busybox/fragments/InstallerFragment.java#L325, accessed in August 2018.

[20] https://github.com/ZeusWPI/hydra-android/blob/master/app/src/main/java/be/ugent/zeus/hydra/library/details/LibraryDetailActivity.java#L67, accessed in August 2018.

[21] https://github.com/arcbit/arcbit-android/blob/master/app/src/main/java/com/arcbit/arcbit/ui/SendFragment.java#L468, accessed in August 2018.

[22] https://github.com/AppLozic/Applozic-Android-Chat-Sample/blob/master/Applozic-Android-AV-Sample/app/src/main/java/com/applozic/mobicomkit/sample/LoginActivity.java#L171, accessed in August 2018.

[23] Can i underline text in an android layout? https://stackoverflow.com/questions/2394935/can-i-underline-text-in-an-android-layout/2394939#2394939, accessed in August 2018.

[24] How to get the build/version number of your android application? https://stackoverflow.com/questions/4616095/how-to-get-the-build-version-number-of-your-android-application/6593822#6593822, accessed in August 2018.

[25] How to hide soft keyboard on android after clicking outside edittext? https://stackoverflow.com/questions/4165414/how-to-hide-soft-keyboard-on-android-after-clicking-outside-edittext/11656129#11656129, accessed in August 2018.

[26] How to send an object from one android activity to another using intents? https://stackoverflow.com/questions/2139134/how-to-send-an-object-from-one-android-activity-to-another-using-intents/2141166#2141166, accessed in August 2018.

[27] How to set a bitmap from resource. https://stackoverflow.com/questions/4955268/how-to-set-a-bitmap-from-resource/4955305#4955305, accessed in August 2018.

[28] Place cursor at the end of text in edittext. https://stackoverflow.com/questions/6217378/place-cursor-at-the-end-of-text-in-edittext/6624186#6624186, accessed in August 2018.

[29] Strange out of memory issue while loading an image to a bitmap object. https://stackoverflow.com/questions/477572/strange-out-of-memory-issue-while-loading-an-image-to-a-bitmap-object/823966#823966, accessed in August 2018.

[30] S. Bajracharya, T. Ngo, E. Linstead, P. Rigor, Y. Dou, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *In Proc. IntâĂŹl Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLAâĂŹ06*, pages 25–26, 2006.

[31] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[32] W.-K. Chan, H. Cheng, and D. Lo. Searching connected api subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 10:1–10:11, New York, NY, USA, 2012. ACM.

[33] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, pages 385–400, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[34] J. R. Cordy and C. K. Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, June 2011.

[35] R. Hill and J. Rideout. Automatic method completion. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 228–235, Sept 2004.

[36] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 117–125, May 2005.

[37] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[38] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[39] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 301–310, May 2011.

[40] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon. Facoy: A code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 946–957, New York, NY, USA, 2018. ACM.

[41] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: Using test-cases to search and reuse source code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 525–526, New York, NY, USA, 2007. ACM.

[42] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, Oct. 2017.

[43] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270, Nov 2015.

[44] L. Martie, T. D. LaToza, and A. v. d. Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 24–35, Nov 2015.

[45] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, Sept 2012.

[46] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[47] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, Piscataway, NJ, USA, 2015. IEEE Press.

[48] S. Mover, S. Sankaranarayanan, R. B. Olsen, and B. E. Chang. Mining framework usage graphs from app corpora. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 277–289, March 2018.

[49] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 69–79, Piscataway, NJ, USA, 2012. IEEE Press.

[50] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[51] M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 357–367, New York, NY, USA, 2016. ACM.

[52] R. Robbes and M. Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326, Sept 2008.

[53] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra. Retrieval on source code: A neural code search. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 31–41, New York, NY, USA, 2018. ACM.

[54] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[55] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[56] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, 23(5):2622–2654, Oct 2018.

[57] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[58] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pages 327–336, Dec 2002.

[59] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. Ccaligner: A token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1066–1077, New York, NY, USA, 2018. ACM.

[60] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[61] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.