Aroma: Code Recommendation via Structural Code Search

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

Di Yang
University of California, Irvine
Irvine, CA, USA
diy4@uci.edu

Celeste Barnaby
Facebook
Menlo Park, CA, USA
celestebarnaby@fb.com

Koushik Sen
University of California, Berkeley
Berkeley, CA, USA
ksen@cs.berkeley.edu

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

Abstract

Programmers often write code that has 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 included by other programmers, help to cross-check against similar code written by other programmers, or help to add extra code which would fix 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 corpus for method bodies containing the partial code snippet, and clusters and intersects the results of the search to recommend a small set of succinct code snippets which both contain the query snippet and appear as part of several methods in the corpus. We evaluated Aroma on 2000 randomly selected queries created from the corpus, as well as 64 queries derived from code snippets obtained from Stack Overflow, a popular website for discussing code. We implemented Aroma for 4 different languages, and developed an IDE plugin for Aroma. Furthermore, we conducted a study where we asked 12 programmers to complete programming tasks using Aroma, and collected their feedback. Our results indicate that Aroma is capable of retrieving and recommending relevant code snippets efficiently.

1 Introduction

Suppose an Android programmer wants to write code to decode a bitmap. 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 code snippet shown in Listing 1 as a first attempt. 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 a few code snippets shown in Listings 2, 3, which demonstrate how to configure the decoder to use less memory, and how to handle potential runtime exceptions, respectively. We call this the *code recommendation problem.*

```
InputStream input = manager.open(fileName);
Bitmap image = BitmapFactory.decodeStream(input);
```

Listing 1. Suppose an Android programmer writes this code to decode a bitmap.

```
final BitmapFactory.Options options = new
      BitmapFactory.Options();
options.inSampleSize = 2;
Bitmap bmp = BitmapFactory.decodeStream(is, null, options);
```

Listing 2. A recommended code snippet that shows how to configure the decoder to use less memory. Recommended lines are highlighted. [1]

```
try {
    InputStream is = am.open(fileName);
    image = BitmapFactory.decodeStream(is);
    is.close();
} catch (IOException e) {
    // ...
}
```

Listing 3. Another recommended code snippet that shows how to properly close the input stream and handle any potential `IOException`. Recommended lines are highlighted.[2]

There are a few existing techniques which could potentially be used to get code recommendations. For example, code-to-code search tools [Kim et al. 2018; Krugler 2013] could retrieve relevant code snippets from a corpus using a partial code snippet as query. However, such code-to-code search tools 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 [Mover et al. 2018; Nguyen et al. 2012, 2009] 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. We emphasize that the meaning of the phrase "code recommendation" in Aroma is different from the term "API code recommendation" [Nguyen et al. 2016a,b]. The latter is a recommendation engine for the next API method to invoke given a code change, whereas Aroma aims to recommend code snippets, as shown in Listings 2, 3, for programmers to learn common usages and integrate those usages with their own code. Code clone detectors [Cordy and Roy 2011; Jiang et al. 2007; Kamiya et al. 2002; Sajnani et al. 2016] 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*. A challenge in designing this search step is that a query snippet, unlike a natural language query, has structure, which 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, 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. Figure 3 shows an outline of the algorithm. For the query shown in Listing 1, Aroma recommends the code snippets shown in Listings 2, 3. The right column of Table 1 shows more examples of code snippets recommended by Aroma for the code queries shown on the left column of the table.

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 on a multi-core server machine.
- Aroma is easy to deploy for different programming languages because its core algorithm works on generic parse trees. We have implemented Aroma for Hack, Java, JavaScript and Python.
- 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++ for four programming languages: Hack [Verlaguet and Menghrajani 2014], Java, JavaScript and Python. We have also implemented IDE plugins for all of these four languages. We report our experimental evaluation of Aroma for the Java programming language. We have used Aroma to index 5,417 GitHub Java Android projects. We performed our experiments for Android Java because we initially developed Aroma for Android based on internal developers' need. 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 on a 24-core CPU. In our large-scale automated evaluation, we used a micro-benchmarking suite containing artificially created query snippets to evaluate the effectiveness of various design choices in Aroma.

Table 1. AROMA code recommendation examples

Query Code Snippet	AROMA Code Recommendation with Extra Lines Highlighted

`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 A: Configuring Objects.
• This code snippet adds underline to a piece of text.[3]
• 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**.[4]
	`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 B: Post-Processing.
• This code snippet decodes a bitmap.[5]
• 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**.[6]
	`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 C: Correlated Statements.
• This code snippet moves the cursor to the end in a text area.[7]
• The recommended code suggests also configuring the action bar to create a more focused view.
• **Intersected from a cluster of 2 methods**.[8]
	`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 D: Exact Recommendations.
• This partial code snippet gets the current version of the application. The rest of the code snippet (not shown) catches and handles possible `NameNotFound` errors.[9]
• The recommended code suggests the exact same error handling as in the original code snippet.
• **Intersected from a cluster of 2 methods**.[10]
	`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 E: Alternative Recommendations.
• This partial code snippet 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.[11]
• **Intersected from a cluster of 10 methods**.[12]
	`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.

Finally, we conducted a user study of AROMA by observing 12 Hack programmers interacting with the IDE on 4 short programming tasks, and found that AROMA is a helpful addition to the existing coding assistant tools.

The rest of the paper is organized as follows: Section 2 presents a case study that reveals the opportunity for a code recommendation tool like AROMA. In Section 3, we describe the algorithm AROMA uses to create code recommendations. In Section 4 we manually assess how useful AROMA code recommendations are. Since code search is a key component of creating recommendations, in Section 5 we measure the search recall of AROMA and compare it with other techniques. Section 6 introduces AROMA for additional languages. In Section 7, we report the initial developer experience with using the AROMA tool. Section 8 presents related work. Finally, Section 9 concludes the paper.

2 The Opportunity for AROMA

AROMA is based on the idea that new code often resembles code that has already been written— therefore, programmers can benefit from recommendations from existing code. To substantiate this claim, we conducted an experiment to measure the similarity of new code to existing code. This experiment was conducted on a large codebase in the Hack language.

We first collected all code commits submitted in a two-day period. From these commits, we extracted a set of changesets.

A changeset is defined as a set of contiguous added or modified lines in a code commit. We filtered out changesets that were shorter than two lines or longer than seven lines. We decided to use this filter because longer changesets are more likely to span multiple methods, and we wanted to limit our dataset to code added or modified within a single method.

For each of the first 1000 changesets in this set, we used Aroma to perform a code-to-code search, taking the snippet as input and returning a list of methods in the repository that contain structurally similar code. Aroma was used because it was already implemented for Hack—but for the purpose of this experiment, any code-to-code search tool or clone detector can work. The results are ranked by similarity score: the percentage of features in the search query that are also found in the search result. For each changeset, we took the top-ranked method and its similarity score. 71 changesets did not yield any search result, because they contained only comments or variable lists, which Aroma disregards in search (see Section 3.2).

To interpret the results, we first needed to assess the correlation between the similarity score (i.e. a measure of the syntactic similarity) and the semantic similarity between the changeset and the result. Two authors manually looked over a random sample of 50 pairs of changesets and result methods, and decided whether this method contained code similar enough to the changeset that a programmer could adopt the existing code (by copy-pasting or refactoring) with minimal changes. Using this criteria, each pair was deemed "similar" or "not similar". Conflicting judgments were cross-checked and re-assessed. As shown in the box plot in Figure 1, there is a clear distinction in similarity scores between the manually-labeled "similar" and "not similar" pairs. Note that in this figure, the top and bottom of the box represents the third and first quartile of the data. The lines extending above and below the box represent the maximum and minimum, and the line running through the box represents the median value.

We chose the first quartile of the similarity scores in the manually-labeled similar pairs—0.71—as the threshold similarity score to decide whether a retrieved code snippet contains meaningful semantic similarities to new code in the commit. We found that for 35.3% of changesets, the most similar result had a score of at least 0.71, meaning that in these cases it would be easy for a programmer to adapt the existing code with minimal efforts, should the code be provided to them.

These results indicate that a considerable amount of new code contains similarities to code that already exists in a large code repository. With Aroma, we aim to utilize this similar code to offer concise, helpful code recommendations to programmers. The amount of similar code in new commits suggests that Aroma has the potential to save a lot of programmers' time and effort.

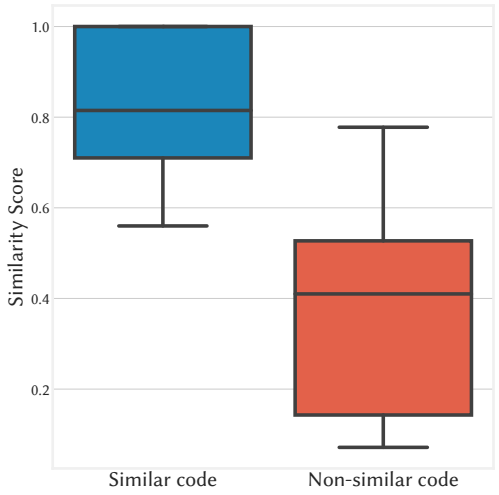

Figure 1. Distribution of similarity scores used to obtain threshold.

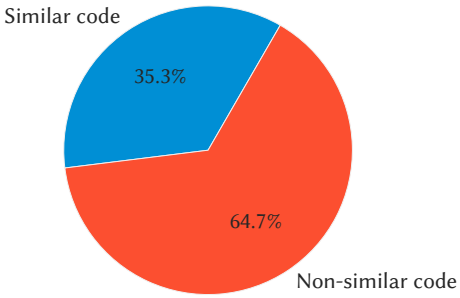

Figure 2. Proportion of new code similar to existing code.

3 Algorithm

Figure 3 illustrates the overall architecture of Aroma. Aroma first featurizes the code corpus. To do so, Aroma parses the body of each method in the corpus and creates its parse tree. Then it extracts a set of structural features from each parse tree.

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. It then selects a set of method bodies from the corpus such that query code's feature set has the most overlap with the feature sets of the selected method bodies over that of the non-selected method bodies. The selection is done by taking dot

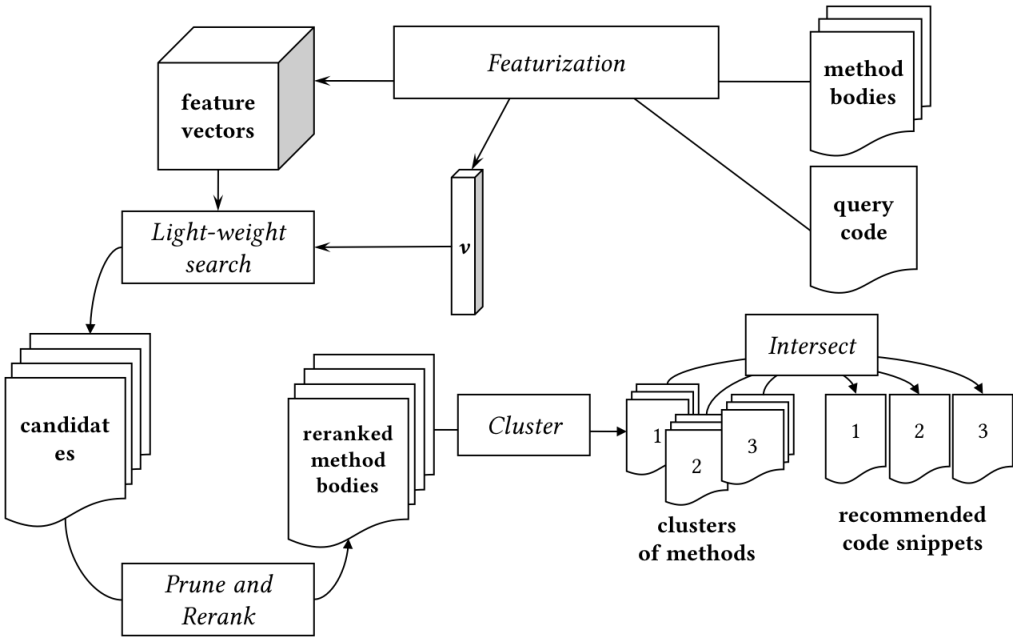

Figure 3. AROMA Code Recommendation Pipeline.

product of query code's feature vector with the feature vector of each method body in the corpus. (The technique for creating these feature vectors, which are sparse, is described in Section 3.3.) The top η_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. Since featurization of a code snippet loses information about the code snippet, the ranking of the code snippets based on feature set overlap 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 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 a modified formulation of the Jaccard distance between the features of the query and that of the pruned code snippet.
- **Cluster and Intersect.** In the final phase, AROMA clusters the code snippets based on their similarity 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 4 as the running example.

3.1 Definitions

We next introduce several notations and definitions which are used to compute the features of a code snippet.

Definition 1 (Keyword tokens)**.** *This is the set of all tokens in a language whose values are fixed as part of the language. 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 2 (Non-keyword tokens)**.** *This is the set of all tokens that are not keyword tokens. Non-keyword tokens include variable names, method names, field names, and literals.*

Examples of non-keyword tokens are `i`, `length`, 0, 1, etc. The set of non-keyword tokens is non-finite for most languages.

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. We use this feature to show the recommended code snippets.

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 4 shows the simplified parse tree of the entire code snippet in Listing 4. 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 a unique index to each label as subscript to distinguish between any two similar labels.

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

Listing 4. A code snippet adapted from a Stack Overflow post.[13] We use this snippet as our running example through Section 3.

Given a simplified parse tree t, we use the following notations. All examples refer to Figure 4.

- $L(t)$ 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-trees, 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. $P(\texttt{\#instanceof\#}_4) = \texttt{(\#)}_2$.
- 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 source code, then we use $\texttt{PrevUse}(n_2)$ to denote n_1 and $\texttt{NextUse}(n_1)$ to denote n_2. E.g. $\texttt{PrevUse}(\texttt{view}_{30}) = \texttt{view}_6, \texttt{NextUse}(\texttt{view}_{30}) = \texttt{view}_{37}$.
- 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.

3.2 Featurization

Code search, clustering, and intersection operations are performed using structural features obtained from code snippets. We next describe how a code is featurized by Aroma. 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 4, 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. Note that we do not replace global variables and method names with `#VAR`. This is because such identifiers are often part of some library API and cannot be alpha-renamed to obtain equivalent programs.
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} child of t_1, t_1 is the i_2^{th} child of t_2, and t_2 is the i_3^{th} child 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 $(C(\texttt{PrevUse}(n)), C(n))$ and $(C(n), C(\texttt{NextUse}(n)))$.

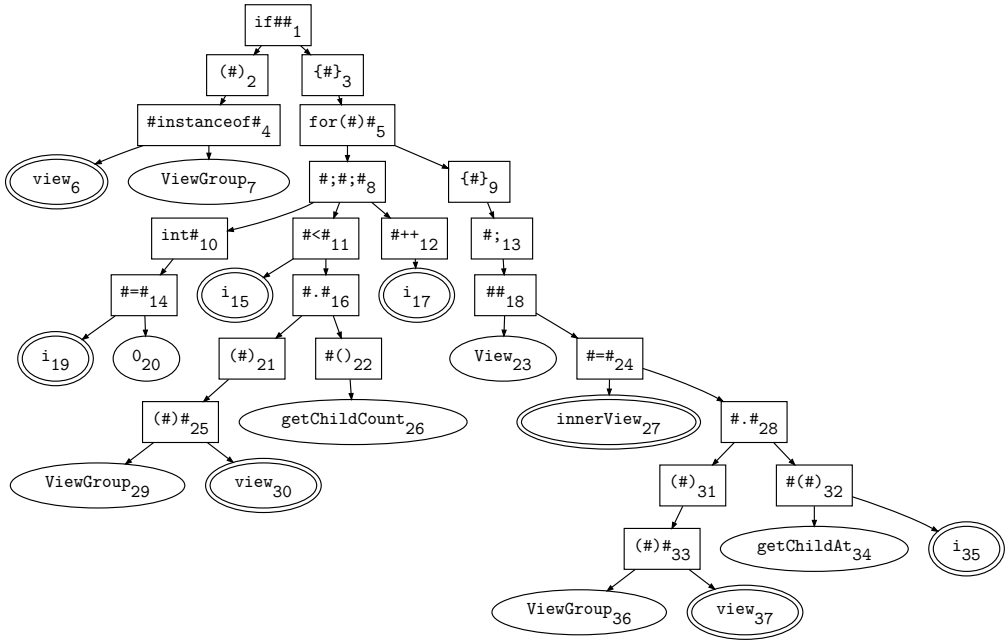

Figure 4. The simplified parse tree representation of the code in Listing 4. Keyword tokens at the leaves are omitted to avoid clutter. Variable nodes are highlighted in double circles.

We only add these features if n is a local variable. Since we ignore the local 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 usages of 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 2 illustrates the features extracted for tokens selected from the simplified parse tree in Figure 4.

3.3 Recommendation Algorithm

3.3.1 Phase I: Light-weight Search

In this phase, Aroma finds a list of method bodies that overlap with the query code snippet. To do so, Aroma uses an imprecise, but light-weight technique. Aroma assumes that it is 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. 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 *overlap score*, of how much of the query code snippet overlaps with the method body. Here $S(X)$ denotes the set of elements of the multi-set X where we ignore the count of each element in the multi-set. Aroma computes a set of η_1 method bodies whose overlap scores are highest with respect to the query code snippet. In our implementation η_1 is usually 1000.

The computation of the list can be reduced to a simple multiplication between a matrix and a sparse 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 overlap 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 overlapping with 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 v_q be the sparse feature vector of the query code snippet q. Then $D \cdot v_q$ is a vector of size $|M|$ and gives the overlap score of each method body with respect to the query snippet. Aroma picks the top η_1 method bodies with the highest overlap scores. Let N_1 be the set of simplified

Table 2. Features for selected tokens in Figure 4

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

parse trees of the method bodies picked by Aroma. This matrix multiplication described above can be done efficiently using a fast sparse matrix multiplication library.

The overlap score is designed to be fast to compute for millions of methods, and enables Aroma to isolate a small set of method bodies which are likely to contain the query code snippet. Because the count of each feature is ignored, the correlation between the overlap score and the degree of containment of a query code snippet in a method body is approximate. In the next phase, Aroma performs 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.

3.3.2 Phase II: Prune and Rerank

In this phase, Aroma reranks the code snippets from the previous phase using a *containment score*, which tells to what degree a query code snippet is contained in a code snippet. To compute the *containment score* of a method body with respect to the query snippet, 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 $Sim(q, m')$, is defined as follows:

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

We find this modified Jaccard similarity metric to give better pruning results compared to traditional Jaccard similarity and cosine similarity metrics. (In Appendix Section A.1, we compare the effectiveness of these similarity metrics.) The containment score between q and m, denoted by $\sigma(q, m)$ is then defined as $Sim(q, m')$. The computation of the optimal pruned simplified parse tree m' requires us to find a subset R of $N(m)$ (where $N(m)$ is the set of non-keyword tokens in the parse tree m) such that $jaccardMod(F(q), F(R))$ is maximal. m' then consists of the nodes in R and any internal nodes in m which is along a path from any $n \in R$ 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

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

   and

   $$jaccardMod(F(q), F \uplus F(n)) > jaccardMod(F(q), F).$$

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 containment score between q and m or $\sigma(q, m)$, is given by $jaccardMod(F(q), F)$. We use $\texttt{Prune}(F(q), m)$ to denote m'.

Aroma reranks the trees in N_1 in descending order using the 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 5 shows a code snippet from the reranked search results for the query code snippet in Listing 4. 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 containment score of this method is 1.0 (the maximum possible score) because the multi-set union of the features of the highlighted token matches the feature set of the query code snippet perfectly.

3.3.3 Phase III: Cluster and Intersect

From the list of method bodies N_2 (i.e. the reranked search results), Aroma picks the top η_2 methods with the highest containment scores that are also above a predefined threshold τ_1. In our implementation $\eta_2 = 100$ 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. The clustering step is necessary to avoid showing recommendations that are similar to each other—for each cluster, only one code snippet is shown as recommendation. The code snippets in a cluster may contain

unnecessary statements. An intersection of the code snippets in a cluster helps to create a concise recommendation by getting rid of these unnecessary statements.

We use $N_2(i)$ to denote the tree at index i in the list N_2. A cluster of code snippets contains method bodies that are similar to each other. Formally, 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

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

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

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

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

1. $\mathsf{l}(\tau) = \mathsf{cs}(\tau)/\mathsf{csq}(\tau)$ 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. $\mathsf{s}(\tau) = \mathsf{csq}(\tau)/|F(N_2(i_1))|$ 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 C is computed iteratively as follows:

1. C_1 is the set $\{(i) \mid 1 \leq i \leq |N_2| \text{ and } (i) \text{ is a valid tuple}\}$.
2. $C_{\ell+1} = \{(i_1, \ldots, i_\ell, i) \mid (i_1, \ldots, i_\ell) \in C_\ell \text{ and } i_\ell < i \leq |N_2| \text{ and } (i_1, \ldots, i_\ell, i) \text{ is a valid tuple and } l(i_1, \ldots, i_\ell, i) = \text{argmax}_{i_l < i_k \leq |N_2|} l((i_1, \ldots, i_\ell, i_k))\} \cup C_\ell$

Aroma computes $C_1, C_2, \ldots$ iteratively until it finds an ℓ such that $C_\ell = C_{\ell+1}$. $C = C_\ell$ is then the set of all clusters. We had to develop this custom clustering algorithm because existing popular clustering algorithms such as k-means, DBSCAN and Affinity Propagation all gave poor results.

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 $\tau = (i_1, \ldots, i_k)$, $\mathtt{Intersect}(\tau, q)$ 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. $\mathtt{Intersect}((i_1, \ldots, i_k), q)$ is defined recursively as follows:

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

In the running example, Listing 5 and Listing 6 form a cluster. Aroma prunes Listing 5 with respect to the union set of features of the query code and Listing 6 as the intersection between Listing 5 and Listing 6. The result of the intersection is shown in Listing 7, which is returned as the recommended code snippet from this cluster.

Finally, Aroma picks the top K (where $K = 5$ in our implementation) tuples from N_3 and returns the intersection of each tuple with the query code snippet as recommendations.

```
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 5. A method body containing the query code snippet in Listing 4.[14]

```
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 6. Another method containing the query code snippet in Listing 4.[15]

```
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 7. A recommended code snippet created by intersecting code in Listing 5 and Listing 6. Extra lines are highlighted.

4 Evaluation of Aroma's Code Recommendation Capabilities

Our goal in this section is to assess how Aroma code recommendation can be useful to programmers. To do so, we collected real-world code snippets from Stack Overflow, used them as query snippets, and inspected the code recommendations provided by Aroma to understand how they can add value to programmers in various ways.

4.1 Datasets

We instantiated 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 have at least 5 stars. A previous study [Lopes et al. 2017] 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 the 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 to carry out the experimental evaluations in the following two sections. In these experiments we found that 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.

4.2 Recommendation Performance on Partial Code Snippets

In this experiment, we manually created partial code snippets by taking the first half of the statements from each of the 64 code snippets. 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.

4.2.1 Exact recommendations.

In 37 cases (74%), one of the top 5 Aroma recommendations matched the original code snippet. Example D in Table 1 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.

4.2.2 Alternative recommendations.

In the other 13 cases (26%), 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 3 which we discuss in the next section. Example E in Table 1 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.

4.3 Recommendation Quality on Full Code Snippets

In this experiment, we used each of the 64 code snippets as queries to evaluate the quality of Aroma's recommendations. We manually inspected the recommended code snippets and determined whether they are useful. We considered a recommended code snippet to be "useful" if in a programming scenario a programmer writes the query code, they would benefit from seeing the recommended code snippets by learning about related methods or common usage patterns. We classified the recommended snippets into several categories by how the recommended code relates to the query snippet. The classification is subjective because there is no "ground truth" on what the recommended code should be, and the actual usefulness depends on how familiar the programmer is with the language and framework. Nevertheless, we present the categories and some examples in Table 1 to demonstrate the variety of code recommendations Aroma can provide. Two of the authors did the manual inspection and categorization, and two other authors verified the results.

4.3.1 Configuring Objects

In this category, the recommended code suggests additional configurations on objects that are already appearing in the query code. Examples include adding callback handlers, and setting additional flags and properties of an existing object. Listings 1, 2 in the introduction, as well as Example A in Table 1 shows examples of this category. These recommendations can be helpful to programmers who are unfamiliar with the idiomatic usages of library methods.

4.3.2 Error Checking and Handling

In this category, 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 programmers that the program might enter an erroneous state or even crash at runtime if exceptions and corner cases are not carefully handled. Listings 1, 3 in the introduction show an example of this category.

4.3.3 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 example, recommended code can show API methods that are commonly called. Example B in Table 1 shows an example of this category, where the recommendation applies Gaussian blurring on the decoded bitmap image. This pattern is not obligatory but demonstrates a possible effect that can be applied on the original object. This category of recommendations can help programmers discover related methods for achieving certain tasks.

4.3.4 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 C in Table 1, 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 code.

4.3.5 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 3. For recommendations that belong to multiple categories, we counted them for each of the categories. We believed the first four categories all can be useful to programmers in different ways, where the unclustered recommendations may not be. For 59 out of the 64 query code snippets (92%), Aroma generated at least one useful recommended snippet that falls in the first four categories.

Table 3. Categories of Aroma code recommendations

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

4.4 Comparison with Pattern-Oriented Code Completion

Pattern-oriented code completion tools [Mover et al. 2018; Nguyen et al. 2012, 2009] could also be used for code recommendation. For example, GraPacc [Nguyen et al. 2012] 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 [Mover et al. 2018]. 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 4.2 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 4.2.1. 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.

5 Evaluation of Search Recall

One of the most important and novel phases of the Aroma's code 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 recall 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, or the original method body should be 100% similar to the first code snippet in the ranked results. 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.

We first evaluated Aroma's search recall on this dataset, then compared it with alternative setups using clone detectors and conventional search techniques.

5.1 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 4 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% of contiguous code queries and for 98.3% of 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 8 demonstrates a rare case where the original method was not retrieved as the top result. Since Aroma's pruning algorithm is greedy (Section 3.3.2), 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 8. A method body in the GitHub corpus.[16] 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.

5.2 Comparison with Clone Detectors and Conventional Search Techniques

Aroma's search and pruning phases are somewhat related to clone detection and conventional code search. In principle, Aroma can use a clone detector or a conventional 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. We tested these alternative setups for search recall on the same micro-benchmark dataset.

5.2.1 Clone Detectors

We instantiated SourcererCC [Sajnani et al. 2016], 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.

5.2.2 Conventional Search Using TF-IDF and Structural Features

We implemented a conventional code search technique using classic TF-IDF [Salton and McGill 1986]. Specifically, instead of creating a binary vector in the featurization 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

Table 4. Comparison of recall between a clone detector, conventional search techniques, and Aroma

	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%**

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.

5.2.3 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 5.2.2.

As shown in Table 4, 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.

6 Aroma for Additional Languages

We have implemented Aroma for additional languages: Hack, JavaScript and Python. One advantage of Aroma is its language-agnostic nature: accommodating a new language requires only implementing a parser that parses code from the target language into a simplified parse tree (as defined in Section 3). The rest of the algorithm, including pruning, clustering and intersecting, all work on the generic form of simplified parse trees. This makes Aroma suitable for real-world deployment on a codebase that consists of many different programming languages.

The code recommendations for these languages are similar to those for Java, as shown in Table 1. We have also conducted the same recall experiment as described in Section 5. This ensures that Aroma instantiations on different languages all have high retrieval recall rates and thus are capable of creating code recommendations pertinent to a query. The results are shown in Table 5.

The recall rates are on par with the original Aroma version on the open-source Java corpus. Non-contiguous test samples were not generated for JavaScript or Python for practical reasons: JavaScript code is often embedded with HTML tags, and Python code structure is dependent on indentation levels. Both language features made generating non-contiguous code queries that resemble real-world code queries particularly difficult. Nevertheless, the high recall rates suggest Aroma's algorithm works well across different languages, and Aroma is capable of creating useful code recommendations for each language.

We have implemented Aroma as an IDE plugin for all four languages (Hack, Java, JavaScript and Python). A screenshot of the integrated development environment is shown in Figure 5 in the Appendix.

7 Initial Developer Experience

We asked 12 Hack programmers to each complete 4 simple programming tasks. For 2 randomly selected tasks, they were allowed to use Aroma; for the other 2, they were not. After each programmer completed the tasks, we gave them a brief follow-up survey. For each task, we provided a description of the functionality to achieve, and some incomplete code to start with. The participants were requested to write between 4 to 10 lines of code to implement the desired functionality. Measuring the time taken to complete these tasks with versus without Aroma was also initially of interest to us; however, we found that the time taken varied greatly, depending mostly on the experience level of the participant and how familiar they were with the particular frameworks being used in the tasks. It had little correlation with the choice of tools.

We focused on getting some initial feedback on developers' experiences using Aroma. The survey began with 2 yes/no questions regarding Aroma's usefulness:

- *Did you find Aroma useful in completing the programming tasks?* 6 participants answered "always useful"; 6 other participants answered "sometimes useful".

Table 5. Aroma Recall Performance on Different Languages

	Contiguous		Non-contiguous	
	Recall@1	Recall@100	Recall@1	Recall@100
Aroma for Hack	98.5%	100%	98.3%	99.9%
Aroma for JavaScript	93.9%	99.6%	not applicable	
Aroma for Python	97.5%	99.4%	not applicable	

- *Did you wish Aroma were available in the programming tasks where you were not permitted to use it?* 6 participants answered "yes"; 4 other participants answered "sometimes"; 2 participant answered "no".

We also asked for more detailed feedback, and found that Aroma adds value to existing tools as follows:

- *Aroma is convenient for discovering usage patterns.* Four participants stated they found Aroma is "convenient" and they were able to "quickly find answers". One participant stated "being able to see general patterns is nice." Another participant commented that when they did not have Aroma, they "used BigGrep* Code Search to achieve the same goal, but it took longer."
- *Aroma is more capable.* One participant commented that Aroma is "more capable at finding results" with multi-line queries, or queries that do not have exact string matches.
- *Aroma is as useful as documentation.* Two participants commented that the Aroma code recommendations helped them write correct code in the same way as manually curated examples seen in documentations. One said "it would be a nice backup if there were no documentation," another said "otherwise I would have to read wikis; that would be more tedious."

We also found some common reasons why participants did not feel Aroma added any additional value.

- *Familiarity with the libraries.* Three participants said they did not find Aroma code recommendations to be useful because they either "had just been working on something like that," or "already knew what to call." In these cases they finished the complete code without the help of Aroma.
- *Simple code search sufficed.* One participant claimed they can get the same amount of information using BigGrep code search. When asked whether clustered results provided additional value, they answered "the code search results were sufficient for completing the tasks at hand." One other participant stated they "got lucky to find a relevant example using BigGrep, but that might not always be the case."

*BigGrep refers to a version of `grep` that searches the entire codebase.

Based on this initial developer survey, we found that sentiment towards Aroma is generally positive, as the participants found Aroma useful for conveniently identifying common coding patterns and integrating them into their own code.

8 Related Work

Code search engines. Code-to-code search tools like FaCoY [Kim et al. 2018]and Krugle [Krugler 2013] 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 finds 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 5.2, 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 [Bajracharya et al. 2006; Chan et al. 2012; Martie et al. 2015; McMillan et al. 2012; Sachdev et al. 2018]. CodeGenie [Lemos et al. 2007] uses test cases to search and reuse source code; SNIFF [Chatterjee et al. 2009] 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 [Zhong et al. 2009] recommends code examples by mining and indexing associated API usage patterns. Portfolio [McMillan et al. 2011] retrieves functions and visualizes their usage chains. CodeHow [Lv et al. 2015] augments the query with API calls which are retrieved from documentation to improve search results. CoCaBu [Sirres et al. 2018] augments the query with structural code entities. A developer survey [Sadowski et al. 2015] reports the top reason for code search is to find code examples or related APIs, and tools have been created for this need. While these code search techniques focus on creating code examples based on keyword queries, they do not support code-to-code search and recommendation.

Clone detectors. Clone detectors are designed to detect syntactically identical or highly similar code. SourcererCC [Sajnani et al. 2016] 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 [Cordy and Roy 2011], Deckard [Jiang et al. 2007], and CCFinder [Kamiya et al. 2002], 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 5.2 its results are not suitable for code recommendation.

Recent clone detection techniques explored other research directions from finding semantically similar clones [Kim et al. 2011, 2018; Saini et al. 2018; White et al. 2016] to finding gapped clones [Ueda et al. 2002] and gapped clones with large number of edits (large-gapped clones) [Wang et al. 2018]. 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 [Bruch et al. 2009; Hill and Rideout 2004; Holmes and Murphy 2005; Robbes and Lanza 2008]. GraPacc [Nguyen et al. 2012] achieves pattern-oriented code completion by first mining graph-represented coding patterns using GrouMiner [Nguyen et al. 2009], then searching for input code to produce code completion suggestions. More recent work [Nguyen et al. 2016a, 2018, 2016b] improves code completion by predicting the next API call given a code change. Pattern-oriented code completion requires mining usage patterns ahead-of-time, and cannot recommend any code outside of the mined patterns, while Aroma does not require pattern mining and recommends snippets on-the-fly.

API Documentation Tools. More techniques exist for improving API documentations and examples. The work by Buse and Weimer [2012] synthesizes API usage examples through data flow analysis, clustering and pattern abstraction. The work by Subramanian et al. [2014] augments API documentations with up-to-date source code examples. MUSE [Moreno et al. 2015] generates code examples for a specific method using static slicing. SWIM [Raghothaman et al. 2016] synthesizes structured call sequences based on a natural language query. The work by Treude and Robillard [2016] augments API documentation with insights from stack overflow. These tools are limited to API usages and do not generalize to structured code queries.

9 Conclusion

We presented Aroma, a new tool for code recommendation via structural code search. Aroma works by first indexing a large code corpus. It takes a code snippet as input, assembles a list of method bodies from the corpus that contain the snippet, and clusters and intersects those method bodies to offer several succinct code recommendations.

To evaluate Aroma, we indexed a code corpus with over 2 million Java methods, and performed Aroma searches with code snippets chosen from the 500 most popular Stack Overflow questions with the *android* tag. We observed that Aroma provided useful recommendations for a majority of these snippets. Moreover, when we used half of the snippet as the query, Aroma exactly recommended the second half of the code snippet in 37 out of 50 cases.

Further, we performed a large-scale automated evaluation to test the accuracy of Aroma search results. We extracted partial code snippets from existing method bodies in the corpus and performed Aroma searches with those snippets as the queries. We found that for 99.1% of contiguous queries and 98.3% of non-contiguous queries, Aroma retrieved the original method as the top-ranked result. We also showed that Aroma's search and pruning algorithms are decidedly better at finding methods *containing* a code snippet than conventional code search techniques.

Finally, we conducted a case study to investigate how programmers interact with Aroma, wherein participants completed two short programming tasks with Aroma and two without Aroma. We found that many participants successfully used Aroma to identify common patterns for libraries they were unfamiliar with. In a follow-up survey, a majority of participants stated that they found Aroma useful for completing the tasks.

Our ongoing work shows that Aroma has the potential to be a powerful developer tool. Though new code is frequently similar to existing code in a repository, currently-available code search tools do not leverage this similar code to help programmers add to or improve their code. Aroma addresses this problem by identifying common additions or modifications to an input code snippet and presenting them to the programmer in a concise, convenient way.

References

Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 681–682. https://doi.org/10.1145/1176617.1176671

Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. 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)*. ACM, New York, NY, USA, 213–222. https://doi.org/10.1145/1595696.1595728

Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 782–792. http://dl.acm.org/citation.cfm?id=2337223.2337316

Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching Connected API Subgraph via Text Phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/2393596.2393606

Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering*, Marsha Chechik and Martin Wirsing (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–400.

J. R. Cordy and C. K. Roy. 2011. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*. 219–220. https://doi.org/10.1109/ICPC.2011.26

R. Hill and J. Rideout. 2004. Automatic method completion. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* 228–235. https://doi.org/10.1109/ASE.2004.1342740

R. Holmes and G. C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 117–125. https://doi.org/10.1109/ICSE.2005.1553554

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

H. Kim, Y. Jung, S. Kim, and K. Yi. 2011. MeCC: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering (ICSE)*. 301–310. https://doi.org/10.1145/1985793.1985835

Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: A Code-to-code Search Engine. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 946–957. https://doi.org/10.1145/3180155.3180187

Ken Krugler. 2013. *Krugle Code Search Architecture*. Springer New York, New York, NY, 103–120. https://doi.org/10.1007/978-1-4614-6596-6_6

Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. 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)*. ACM, New York, NY, USA, 525–526. https://doi.org/10.1145/1321631.1321726

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133908

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 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 260–270. https://doi.org/10.1109/ASE.2015.42

L. Martie, T. D. LaToza, and A. v. d. Hoek. 2015. CodeExchange: Supporting Reformulation of Internet-Scale Code Queries in Context (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 24–35. https://doi.org/10.1109/ASE.2015.51

C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. 2012. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering* 38, 5 (Sept 2012), 1069–1087. https://doi.org/10.1109/TSE.2011.84

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1985793.1985809

Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 880–890. http://dl.acm.org/citation.cfm?id=2818754.2818860

S. Mover, S. Sankaranarayanan, R. B. Olsen, and B. E. Chang. 2018. Mining framework usage graphs from app corpora. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Vol. 00. 277–289. https://doi.org/10.1109/SANER.2018.8330216

Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016a. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 69–79. http://dl.acm.org/citation.cfm?id=2337223.2337232

Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2018. Complementing Global and Local Contexts in Representing API Descriptions to Improve API Retrieval Tasks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 551–562. https://doi.org/10.1145/3236024.3236036

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. 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)*. ACM, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016b. Learning API Usages from Bytecode: A Statistical Approach. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 416–427. https://doi.org/10.1145/2884781.2884873

Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 357–367. https://doi.org/10.1145/2884781.2884808

R. Robbes and M. Lanza. 2008. How Program History Can Improve Code Completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 317–326. https://doi.org/10.1109/ASE.2008.42

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. 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)*. ACM, New York, NY, USA, 31–41. https://doi.org/10.1145/3211346.3211353

Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855

Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software*

Engineering (ESEC/FSE 2018). ACM, New York, NY, USA, 354–365. https://doi.org/10.1145/3236024.3236026

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

Gerard Salton and Michael J. McGill. 1986. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA.

Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering* 23, 5 (01 Oct 2018), 2622–2654. https://doi.org/10.1007/s10664-017-9544-y

Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 643–652. https://doi.org/10.1145/2568225.2568313

Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 392–403. https://doi.org/10.1145/2884781.2884800

Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. 2002. On detection of gapped code clones using gap locations. In *Ninth Asia-Pacific Software Engineering Conference, 2002*. 327–336. https://doi.org/10.1109/APSEC.2002.1183002

Julien Verlaguet and Alok Menghrajani. 2014. Hack: a new programming language for HHVM. https://code.fb.com/developer-tools/hack-a-new-programming-language-for-hhvm/.

Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A Token Based Large-gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1066–1077. https://doi.org/10.1145/3180155.3180179

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 318–343.

Online references

[1]This code snippet is adapted from https://github.com/zom/Zom-Android/blob/master/app/src/main/java/org/awesomeapp/messenger/ui/stickers/StickerGridAdapter.java#L67. Accessed in August 2018.

[2]This code snippet is adapted from https://github.com/yuyuyu123/ZCommon/blob/master/zcommon/src/main/java/com/cc/android/zcommon/utils/android/AssetUtils.java#L37. Accessed in August 2018.

[3]This code snippet is adapted from the Stack Overflow post "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.

[4]This code snippet is adapted from https://github.com/tonyvu2014/android-shoppingcart/blob/master/demo/src/main/java/com/android/tonyvu/sc/demo/ProductActivity.java#L58. Accessed in August 2018.

[5]This code snippet is adapted from the Stack Overflow post "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.

[6]This code snippet is adapted from https://github.com/TonnyL/GaussianBlur/blob/master/app/src/main/java/io/github/marktony/gaussianblur/MainActivity.java#L58. Accessed in August 2018.

[7]This code snippet is adapted from the Stack Overflow post "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.

[8]This code snippet is adapted from 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.

[9]This code snippet is adapted from the Stack Overflow post "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.

[10]This code snippet is adapted from 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.

[11]This code snippet is adapted from the Stack Overflow post "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.

[12]This code snippet is adapted from https://github.com/zfdang/zSMTH-Android/blob/master/app/src/main/java/com/zfdang/zsmth_android/MainActivity.java#L736. Accessed in August 2018.

[13]This code snippet is adapted from the Stack Overflow post "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.

[14]This code snippet is adapted from https://github.com/arcbit/arcbit-android/blob/master/app/src/main/java/com/arcbit/arcbit/ui/SendFragment.java#L468. Accessed in August 2018.

[15]This code snippet is adapted from 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.

[16]This code snippet is adapted from https://github.com/iqiyi/dexSplitter/blob/master/extra/hack_dx/src/com/android/dex/Leb128.java#L82. Accessed in August 2018.

A Appendix

A.1 Comparison of similarity metrics

In this section, we compare three different similarity metrics that can be used in the pruning algorithm in terms of recall on a micro-benchmark dataset. The dataset is described in Section 5. Specifically, we compare the modified Jaccard similarity metric with the following:

- Traditional Jaccard similarity, defined as

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

where A and B are multi-sets.

- Cosine similarity, defined as

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

  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 3.3.2) with *jaccard* and *cosine*. We call the original Aroma as *jaccardMod*, and the two variants as *jaccard* and *cosine*, respectively.

Table 6 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 6. Comparison of different similarity metrics

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

A.2 Aroma IDE plugin

A screenshot of the Aroma Code Recommendation plugin is shown in Figure 5.

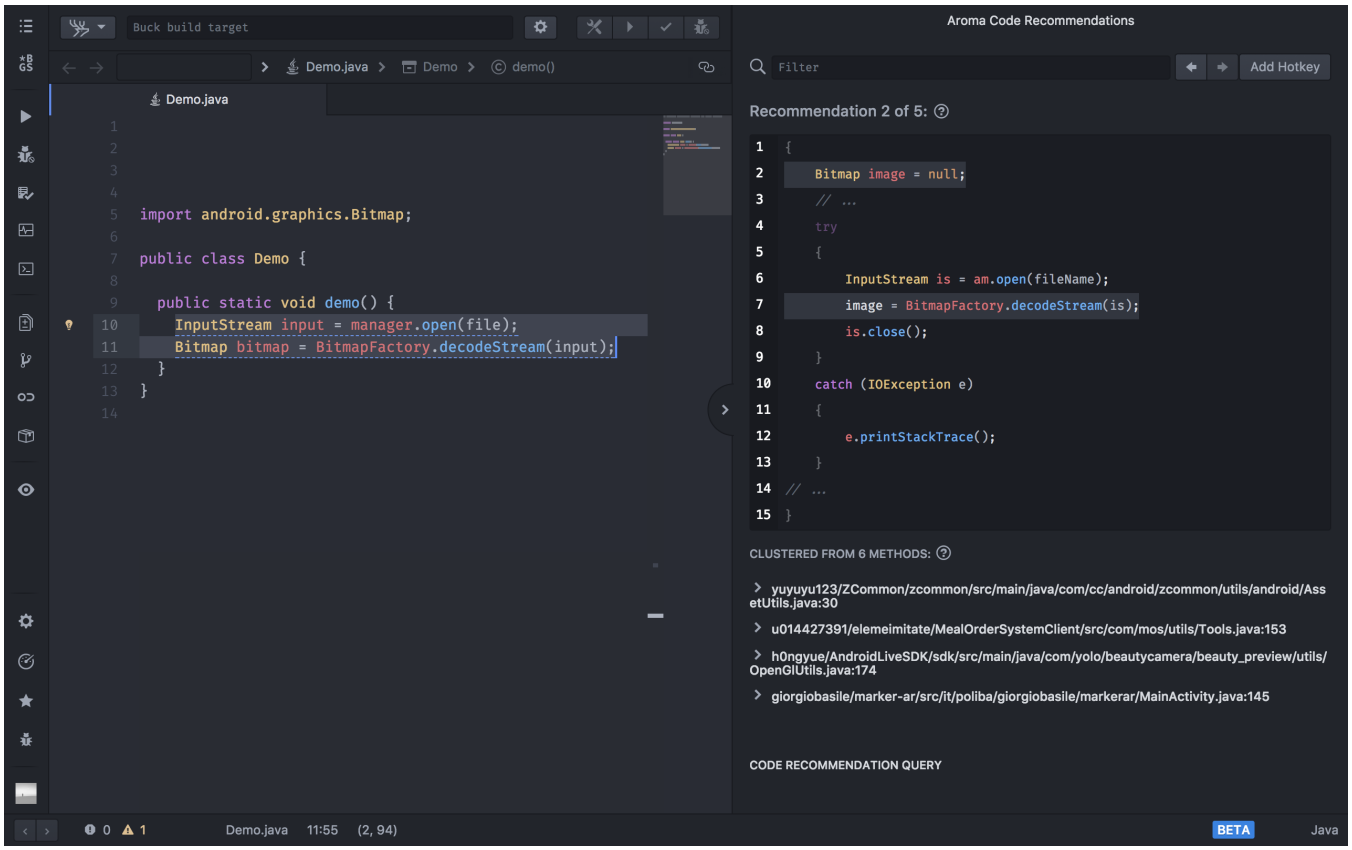

Figure 5. AROMA Code Recommendation Plugin in an IDE. Recommended code snippets are shown in the side pane for code selected in the main editor.