Aroma: Code Recommendation via Structural Code Search

SIFEI LUAN, Facebook, USA
DI YANG*, University of California, Irvine, USA
CELESTE BARNABY, Facebook, USA
KOUSHIK SEN†, University of California, Berkeley, USA
SATISH CHANDRA, Facebook, USA

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.

CCS Concepts: • **Information systems** → *Near-duplicate and plagiarism detection*; • **Software and its engineering** → *Development frameworks and environments*; *Software post-development issues*.

Additional Key Words and Phrases: code recommendation, structural code search, clone detection, feature-based code representation, clustering

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

---

*This work was done in part while this author was an intern at Facebook.
†This work was done in part while this author was a visiting scientist at Facebook.

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 152. Publication date: October 2019.

152

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. AROMA's recommendations contain more syntactic variety than just API usages; for instance, the recommended code snippet in Listing 3 includes a `try-catch` block, and Example B in Table 1 recommends adding an `if` statement that modifies a variable. 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.

---

[1]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]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.

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

Table 1. Aroma code recommendation examples

Query Code Snippet	Aroma Code Recommendation with Extra Lines Highlighted
```\nTextView textView = (TextView)\n view.findViewById(R.id.textview);\nSpannableString content = new SpannableString("Content");\ncontent.setSpan(new UnderlineSpan(), 0, content.length(), 0);\ntextView.setText(content);\n```	```\nTextView licenseView = (TextView)\n findViewById(R.id.library_license_link);\nSpannableString underlinedLicenseLink = new SpannableString(\n getString(R.string.library_license_link));\nunderlinedLicenseLink.setSpan(new UnderlineSpan(), 0,\n underlinedLicenseLink.length(), 0);\nlicenseView.setText(underlinedLicenseLink);\nlicenseView.setOnClickListener(v -> {\n FragmentManager fm = getSupportFragmentManager();\n LibraryLicenseDialog libraryLicenseDlg = new\n LibraryLicenseDialog();\n libraryLicenseDlg.show(fm, "fragment_license"); });\n```
Example A: Configuring Objects. • This code snippet adds underline to a piece of text.[1] • 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.**[2]	
```\nBitmap bitmap = BitmapFactory.decodeResource(getResources(),\n R.drawable.image);\n```	```\nint radius = seekBar.getProgress();\nif (radius < 1) {\n radius = 1;\n}\nBitmap bitmap = BitmapFactory.decodeResource(getResources(),\n R.drawable.image);\nimageView.setImageBitmap(blur.gaussianBlur(radius, bitmap));\n```
Example B: Post-Processing. • This code snippet decodes a bitmap.[3] • The recommended code suggests applying Gaussian blur on the decoded image, a customary effect to be applied. • **Intersected from a cluster of 4 methods.**[4]	
```\nEditText et = (EditText)findViewById(R.id.inbox);\net.setSelection(et.getText().length());\n```	```\nsuper.onCreate(savedInstanceState);\nsetContentView(R.layout.material_edittext_activity_main);\ngetSupportActionBar().setDisplayHomeAsUpEnabled(true);\ngetSupportActionBar().setDisplayShowTitleEnabled(false);\nEditText singleLineEllipsisEt = (EditText)\n findViewById(R.id.singleLineEllipsisEt);\nsingleLineEllipsisEt.setSelection(\n singleLineEllipsisEt.getText().length());\n```
Example C: Correlated Statements. • This code snippet moves the cursor to the end in a text area.[5] • The recommended code suggests also configuring the action bar to create a more focused view. • **Intersected from a cluster of 2 methods.**[6]	
```\nPackageInfo pInfo =\n getPackageManager().getPackageInfo(getPackageName(),\n 0);\nString version = pInfo.versionName;\n```	```\ntry {\n PackageInfo pInfo =\n getPackageManager().getPackageInfo(getPackageName(),\n 0);\n String version = pInfo.versionName;\n TextView versionView = (TextView)\n findViewById(R.id.about_project_version);\n versionView.setText("v" + version);\n} catch (PackageManager.NameNotFoundException ex) {\n Log.e(...);\n}\n```
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.[7] • The recommended code suggests the exact same error handling as in the original code snippet. • **Intersected from a cluster of 2 methods.**[8]	
```\ni.putExtra("parcelable_extra", (Parcelable)\n myParcelableObject);\n```	```\nIntent intent = new Intent(this, BoardTopicActivity.class);\nintent.putExtra(SMTHApplication.BOARD_OBJECT, (Parcelable)\n board);\nstartActivity(intent);\n```
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.[9] • **Intersected from a cluster of 10 methods.**[10]	• 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.

[1] Adapted from the Stack Overflow post "Can I underline text in an android layout?" [https://stackoverflow.com/questions/2394939], by Anthony Forloney [https://stackoverflow.com/users/166712].

[2] Adapted from https://github.com/tonyvu2014/android-shoppingcart/blob/master/demo/src/main/java/com/android/tonyvu/sc/demo/ProductActivity.java.

[3] Adapted from the Stack Overflow post "How to set a bitmap from resource" [https://stackoverflow.com/questions/4955305], by xandy [https://stackoverflow.com/users/109112].

[4] Adapted from https://github.com/TonnyL/GaussianBlur/blob/master/app/src/main/java/io/github/marktony/gaussianblur/MainActivity.java.

[5] Adapted from the Stack Overflow post "Place cursor at the end of text in EditText" [https://stackoverflow.com/questions/6624186], by Marqs [https://stackoverflow.com/users/400493].

[6] Adapted from https://github.com/cymcsg/UltimateAndroid/blob/master/deprecated/UltimateAndroidGradle/demoofui/src/main/java/com/marshalchen/common/demoofui/sampleModules/MaterialEditTextActivity.java.

[7] Adapted from the Stack Overflow post "How to get the build/version number of your Android application?" [https://stackoverflow.com/questions/6593822], by plus- [https://stackoverflow.com/users/709635].

[8] Adapted from https://github.com/front-line-tech/background-service-lib/blob/master/SampleService/servicelib/src/main/java/com/flt/servicelib/AbstractPermissionExtensionAppCompatActivity.java.

[9] Adapted from the Stack Overflow post "How to send an object from one Android Activity to another using Intents?" [https://stackoverflow.com/questions/2141166], by Jeremy Logan [https://stackoverflow.com/users/76835].

[10] Adapted from https://github.com/zfdang/zSMTH-Android/blob/master/app/src/main/java/com/zfdang/zsmth_android/MainActivity.java.
All Stack Overflow content is licensed under CC-BY-SA 3.0. All URLs are accessed in August 2018.

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. 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 the real-world deployment of Aroma. 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. Alternatively, we could have taken portions of changesets found within a single method; however, since changesets are raw text, finding the method boundaries involves additional parsing. We stuck to the simple solution of taking short changesets.

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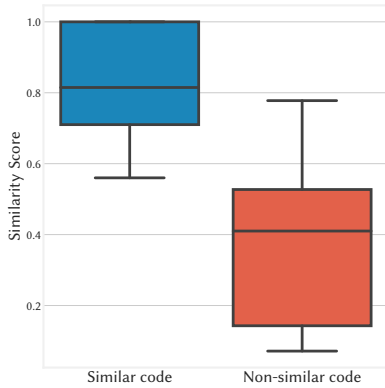
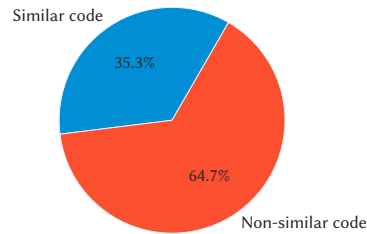Fig. 1. Distribution of similarity scores used to obtain threshold

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

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.

3 ALGORITHM

Figure 3 illustrates the overall architecture of AROMA. In order to generate code recommendations, AROMA must first featurize the code corpus. To do so, AROMA parses the body of each method in the corpus and creates its parse tree. It extracts a set of structural features from each parse tree. Then, given a query code snippet, AROMA runs the following phases to create recommendations:

- **Light-weight Search.** In this phase, AROMA takes a query code snippet, and outputs a list of the top few (e.g. 1000) methods that have the most overlap with the query. To do so, AROMA extracts custom features from the query and each method in the corpus. AROMA intersects the set of features of the query and each method body, and uses the cardinality of the intersection to compute the degree of overlap between the query and the method body. To make this computation efficient, AROMA represents the set of features of a code snippet as a sparse vector and performs matrix multiplication to compute the degree of overlap of all methods with the query code.
- **Prune and Rerank.** In this phase, AROMA reranks the list of method bodies retrieved from the previous phase using a more precise, but expensive algorithm for computing similarity.
- **Cluster and Intersect.** In the final phase, AROMA clusters the reranked list of code snippets from the previous phase. Clustering is based on the similarity of the method bodies. Clustering
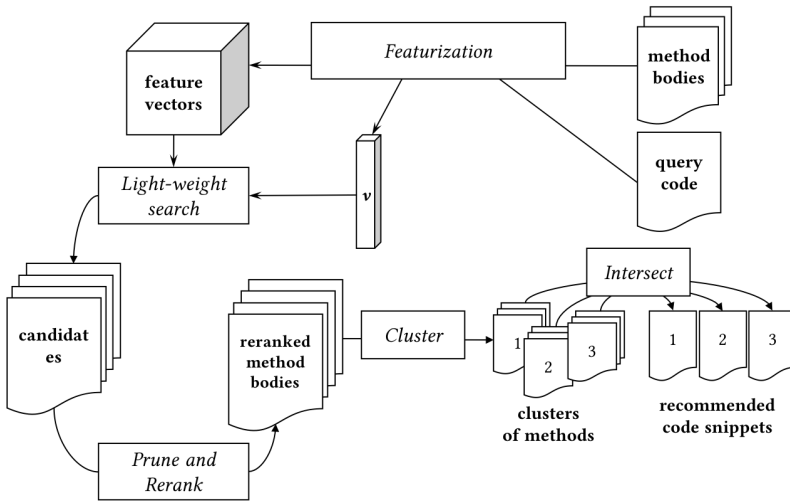
Fig. 3. AROMA code recommendation pipeline

also needs to satisfy constraints which ensure that recommendations are of high quality. Therefore, we have devised a custom clustering algorithm which takes the constraints into account. AROMA then intersects the snippets in each cluster to come up with recommended code snippets. This approach of clustering and intersection helps to create a succinct, yet diverse set of recommendations.

We next describe the details of each step using the code snippet shown in Listing 4 as the running example.

```java
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.[1] This snippet is used as the running example through Section 3.

3.1 Definitions

In this section, we introduce several notations and definitions used to compute the features of a code snippet. The terminologies and notations are also used to describe AROMA formally.

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.

---

[1] Adapted from the Stack Overflow post "How to hide soft keyboard on android after clicking outside EditText?" [https://stackoverflow.com/questions/11656129], by Navneeth G [https://stackoverflow.com/users/1135909]. CC-BY-SA 3.0 License. Accessed in August 2018.

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.*

Moreover, a simplified parse tree cannot be a list containing a single simplified parse tree.

We picked this particular representation of programs instead of a 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 of a simplified parse tree 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.*

For example, the label of the simplified parse tree `["x", ">", ["y", ".", "f"]]` is `"#>#"`.
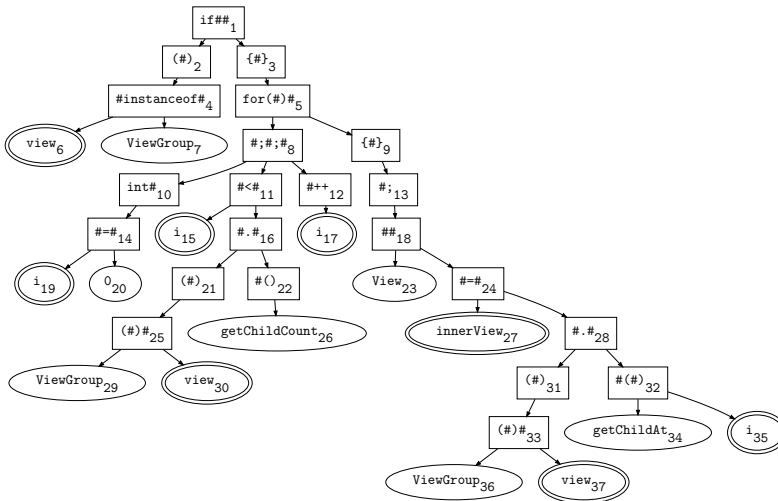

Fig. 4. The simplified parse tree representation of the code in Listing 4. Keyword tokens at the leaves are omitted to avoid clutter. Variable nodes are highlighted in double circles.

Figure 4 visualizes the simplified parse tree of the code snippet in Listing 4. In the figure, each internal node represents a simplified parse tree, and is labeled using the tree's label as defined above. Since keyword tokens in a simplified parse tree become part of the label of the tree, we do not create leaf nodes for keyword tokens in the tree diagram—we only add leaf nodes for non-keyword tokens. 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.

To obtain the simplified parse tree of a code snippet, AROMA relies on a language-specific parser. For example, AROMA utilizes the `ANTLR4` [Parr 2013] Java parser to produce the parse tree for a Java program. AROMA traverses the parse tree produced by the parser to collect at each internal node the tokens and subtrees that are immediate descendants of each internal node of the parse tree. The collected elements at each node form a list, which is a simplified parse tree. AROMA uses the list at each internal node to create the label for the node. The distinction between keyword and non-keyword tokens is done using the language's lexical specification. AROMA performs a second traversal of the tree and uses the static scoping rules of the language to identify the global variables. AROMA uses the knowledge of the global variables in the featurization phase which we describe later. Note that this process of creating a simplified parse tree from a code snippet is language-dependent and requires knowledge about the grammar and static scoping rules of the language. Once the simplified parse tree of a code snippet has been created, the rest of AROMA's algorithm is language-agnostic.

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 $\mathsf{Prev}(n_2)$ to denote n_1 and $\mathsf{Next}(n_1)$ to denote n_2. E.g. $\mathsf{Prev}(\texttt{view}_{30}) = \texttt{ViewGroup}_{29}, \mathsf{Next}(\texttt{ViewGroup}_{29}) = \texttt{view}_{30}$.
- 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 $\mathsf{PrevUse}(n_2)$ to denote n_1 and $\mathsf{NextUse}(n_1)$ to denote n_2. E.g. $\mathsf{PrevUse}(\texttt{view}_{30}) = \texttt{view}_6, \mathsf{NextUse}(\texttt{view}_{30}) = \texttt{view}_{37}$.
- If n is a non-keyword token denoting a local variable and it is the ith 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

The high-level goal of this step is to take a simplified parse tree for a code snippet, and extract a set of structural features from that parse tree. A key requirement of the features is that if two code snippets are similar, they should have the same collection of features.

A simple way to featurize a code snippet is to treat the labels of all nodes in the simplified parse tree as features. This simple approach creates problem if we have two code snippets 1) which only differ in local variable names, and 2) where one code snippet can be obtained from the other by alpha renaming the local variables. The two code snippets should be considered as similar, but the collection of features will differ in the name of some of the variables. Therefore, we replace each token that denotes a local variable by a special token `#VAR`. We do not perform similar replacements

for global variables and method names. This is because such identifiers are often part of some library API and cannot be alpha-renamed to obtain similar programs.

Treating the labels of parse tree nodes as the only features does not help to capture the relation between the nodes. Such relations are necessary to identify the structural features of a code snippet. For example, without such a relation Aroma will treat the snippets `if (x > 0) z = 3;` and `if (z > 3) x = 0;` as similar since they have the exact same collection of node labels (i.e. `{if##, (#), #>#, #;, #=#, 0, 3, #VAR, #VAR}`). If we can somehow create a feature encapsulating the fact that 3 belongs to the body of the first `if` statement, Aroma will distinguish between the two snippets. Therefore, Aroma also creates features which represent some relations between certain pairs of nodes in the parse tree. Examples of some such features involving the token 3 are `(if##,2,3)`, `(#=#,2,3)`, and `(#VAR,3)`. The first feature, which is denoted as a triplet, states that the 2nd child of a node labeled `if##` has a descendant leaf node with label 3. Similarly, the second feature asserts that the 2nd child of a node labeled `#=#` has a descendant leaf node with label 3. We call these two features *parent features*, as they help capture the relation of a leaf node with its parent, grand-parent, and great-grand parent. The third feature relays the fact that a variable leaf node appears before 3. We call such features *sibling features*. In summary, the parent features and sibling features capture some local relations between the nodes in a parse tree. However, these features are not exhaustive enough to recreate the parse tree from the features. These non-exhaustiveness of features helps Aroma tolerate some non-similarities in otherwise similar code snippets, and helps Aroma to retrieve some closely related, but nonidentical code snippets during search.

Since we replace all local variable names with `#VAR`, we also need to relate two variable usages in a code snippet which refer to the same local variable. For example, in the code snippet `if (y < 0) x = -x;`, we will have three features of the form `#VAR` corresponding to the two occurrences of `x` and one occurrence of `y`. However, the collection of features described so far does not express the fact that two `#VAR` features refer to the same variable. There is no direct way to state two variables are related since we have gotten rid of variable names. Rather, we capture features about the fact that the *consecutive usage context* of the same local variables are related. We call such features *variable usage features*. For example, the context of the two usages of `x` are `(1,#=#)` and `(1,-#)`, respectively. The first context corresponds to the first usage of `x` and denotes that there is a variable which is the first child of the node labeled `#=#`. The index and the parent node label together forms the context of this particular variable usage. Similarly, the second context denotes the second usage of `x`. We create a feature of the form `((1,#=#),(1,-#))` which captures the relation between the context of two consecutive usage of the same variable.

We now describe formally how a code snippet 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) A *Token Feature* of the form n. If n is a local variable, we replace n with `#VAR`.
(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`. 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 $(n, \mathrm{Next}(n))$ and $(\mathrm{Prev}(n), n)$. As before, if any of $n, \mathrm{Next}(n), \mathrm{Prev}(n)$ is a local variable, it is replaced with `#VAR`.
(4) *Variable Usage Features* of the form $(C(\mathrm{PrevUse}(n)), C(n))$ and $(C(n), C(\mathrm{NextUse}(n)))$. We only add these features if n is a local 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 two non-keyword tokens from the simplified parse tree in Figure 4. In the interest of space, we do not show the features extracted by Aroma for all non-keyword tokens.

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)	-

3.3 Recommendation Algorithm

3.3.1 Phase I: Light-weight Search. In this phase, Aroma takes a query code snippet, and outputs a list of the top few (e.g. 1000) methods that contain the most overlap with the query. To compute the top methods, we need to compute the degree of overlap between the query and each method body in the corpus. Because our corpus has millions of methods, we need to make sure that the degree of overlap can be computed fast at query time. We use the feature sets of code snippets to compute the degree of overlap, which we call the **overlap score**. Specifically, Aroma intersects the set of features of the query and the method body, and uses the cardinality of the intersection as the overlap score. Computing intersection and its cardinality could be computationally expensive. For efficient computation, we represent the set of features of a code snippet as a sparse vector and perform matrix multiplication to compute the overlap score of all methods with the query code. We next describe the formal details of the phase.

Given a large code corpus 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 its 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 list 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 this 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 ith 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. To understand this decision, consider a method m that contains a feature f numerous times (say n). Then, say we give Aroma a query q that contains f once. The overlap score between m and q will be increased by n, even though the multiple instances of this feature do

not actually indicate greater overlap between m and q. 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 $|M|$ that 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 parse trees of the method bodies picked by Aroma.

The corpus we used for evaluation has over 37 million unique features. But each method has an average of 63 methods, so the feature vectors are very sparse. Thus, the matrix multiplication described above can be done efficiently using a fast sparse matrix multiplication library—for our corpus, this phase finishes in less than a second.

3.3.2 Phase II: Prune and Rerank. In the following phases, we need a sub-algorithm to compute a maximal code snippet that is common to two given code snippets. For example, given the code snippets `x = 1; y = 2;` and `y = 2; z = 3;`, we need an algorithm that computes `y = 2;` as the intersection of the two code snippets. This algorithm could be implemented using a longest-common subsequence (LCS) [Porter 1997] computation algorithm on strings by treating the two code snippets as strings. Such an algorithm was used in SNIFF [Chatterjee et al. 2009] (which performs natural language small code snippet search). However, LCS does not work well for Aroma because often the common parts between two code snippets may not be exactly similar. To illustrate this point, suppose we are given the two code snippets `x = 1; if (y > 1) if (z < 0) w = 4;` and `if (z < 0) if (y > 1) w = 4; v = 10;`, where we have swapped the nesting of the two `if` statements. LCS will retrieve either `if (y > 1) w = 4;` or `if (z < 0) w = 4;` as the intersection, i.e. LCS drops one of the `if` statements along with the non-common assignment statements. Ideally, we should have both `if` statements in the intersection, i.e. the intersection algorithm should compute either `if (y > 1) if (z < 0) w = 4;` or `if (z < 0) if (y > 1) w = 4;` as the intersection.

The example also shows that we can have at most two intersected code snippets when fuzzy similarity exists between the given code snippets—a snippet will either be most similar to the first snippet, or the second snippet. We resolve this ambiguity by picking the intersected snippet that is close to the second snippet. Thus, we can think of the intersection as a code snippet obtained by taking the second snippet, and dropping its fragments which have no resemblance to the first snippet. That is, the algorithm is *pruning* the second snippet while retaining the parts common with the first one.

A simple way to *prune* the second snippet is to look at its parse tree and find a subtree which is most similar to the first snippet. However, such an algorithm will be expensive because there are exponentially many subtrees in a given tree. Instead, Aroma uses a greedy algorithm which gives us a maximal subtree of the second snippet's parse tree. We have also observed that if we can identify all the leaf nodes in the second snippet's parse tree that need to be present in the intersection, we can get a maximal subtree by simply retaining all the nodes and edges in the tree that lie in a path from the root to the identified leaf nodes. We next formally describe the pruning algorithm.

Let us assume we are given two code snippets, say m_1 and m_2, in the form of their parse trees. The computation of the optimal pruned simplified parse tree, say m_p, requires us to find a subset, say R, of the leaf nodes of m_2. Recall that the set of leaf nodes of m is denoted by $N(m)$ and contains exactly the non-keyword tokens in the parse tree. The set R should be such that the similarity between m_p and m_1 is maximal. We will use the cardinality of the multi-set intersection of the features of two code snippets as their similarity score. That is, the similarity score between two snippets given as parse trees, say m_1 and m_2, is $|F(m_1) \cap F(m_2)|$. Let us denote it by $SimScore(m_1, m_2)$. Once we

have computed the set of leaf nodes (i.e. R) that need to be present in the intersection, m_p is the subtree consisting of the nodes in R, and any internal nodes and edges in m_2 which are along a path from any $n \in R$ to the root node in m_2. The greedy algorithm for computing R is described in Algorithm 1.

Algorithm 1 The Pruning Algorithm

1: **procedure** PRUNE($F(m_1), m_2$)
2: ▷ The first argument takes the features of m_1 instead of m_1 itself to simplify the description of Phase III.
3:     $R \leftarrow \emptyset$
4:     $F \leftarrow \emptyset$
5:     **repeat**
6:        $n \leftarrow \text{argmax}_{n' \in N(m_2) \backslash R} SimScore(F(m_1), (F \uplus F(n')))$
7:        **if** n exists and $SimScore(F(m_1), F \uplus F(n)) > SimScore(F(m_1), F)$ **then**
8:           $R \leftarrow R \cup \{n\}$
9:           $F \leftarrow F \uplus F(n)$
10:        **end if**
11:     **until** R does not change anymore
12:     **return** m_p where m_p is obtained from m_2 by retaining all the non-keyword tokens in R and
13:     any internal node or edge which appears in a path from a $n \in R$ to the root of m_2
14: **end procedure**

In the algorithm, AROMA maintains the collection of the features of the intersected snippet in the variable F. The variable R maintains the set of leaf nodes in the intersected code. Initially, the algorithm starts with an empty set of leaf nodes. It then iteratively adds more leaf nodes to the set from the parse tree of the second method (i.e. m_2). A node n is added if it increases the similarity between the first method and the tree that can be obtained from R. Since F maintains the features of the tree that can be constructed from R, we can get the features of $R \cup \{n\}$ by simply adding the features associated with n (i.e. $F(n)$) to F. If such a node cannot be found, the algorithm constructs the intersected tree from R and returns it.

We are next going to show how AROMA uses the pruning algorithm to rerank the snippets retrieved in Phase 1. Given a query, say q, and a method body, say m, pruning of the method with respect to the query (i.e. $\text{Prune}(F(q), m)$) gives a code snippet that is common to both the query and method. If we consider the similarity score between the query and the pruned code snippet, the score should be an alternative way to quantify the overlap between the query and the method. We found empirically that if we use this alternative score to rerank the methods retrieved in Phase 1 (i.e. N_1), then our ranking of search results improves slightly. AROMA uses the reranked list, which we call N_2, in the next phase for clustering and intersection. Note that the pruning algorithm is greedy, so we may not find the best intersection between two code snippets. In Section 5, 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.

3.3.3 Phase III: Cluster and Intersect. In the final phase, AROMA prepares recommendations by clustering and intersecting the reranked search results from the previous phase. Clustering and intersection are computationally expensive. Therefore, we pick from the list of search results the

top $\eta_2 = 100$ methods whose overlap score with the query is above a threshold $\tau_1 = 0.65$, and run the last phase on them. In the discussion below, we assume that N_2 has been modified to contain the top η_2 search results.

Clustering. Aroma clusters together method bodies that are similar to each other. The clustering step is necessary to avoid creating redundant recommendations—for each cluster, only one recommendation is generated. Furthermore, the methods in a cluster may contain unnecessary, extraneous code fragments. An intersection of the code snippets in a cluster helps to create a concise recommendation by getting rid of these unnecessary code fragments.

A cluster contains method bodies that are similar to each other. Specifically, a cluster must satisfy the following two constraints:

(1) If we intersect the snippets in a cluster, we should get a code snippet that has more code fragments than the query. This ensures that Aroma's recommendation (which is obtained by intersecting the snippets in the cluster) is an extension to the query snippet.

(2) The pruned code snippets in a cluster are similar to each other. This is because Aroma has been designed to perform search that can tolerate some degree of differences between the query and the results. As such, two code snippets may overlap with different parts of the query. If two such code snippets are part of a cluster, then their intersection will not contain the query snippet. Therefore, the recommendation, which is obtained by intersecting all the snippets in a cluster, will not contain any part of the query. This is undesirable because we want a recommendation that contains the query and some extra new code.

Moreover, Aroma does not require the clusters to be disjoint.

Because of these constraints on a cluster, we cannot simply use a textbook clustering algorithm such as k-means, DBSCAN, or Affinity Propagation. We tried using those clustering algorithms initially (ignoring the constraints) and got poor results. Therefore, we developed a custom clustering algorithm that takes the constraints into account. At a high level, the clustering algorithm starts by treating each method body as a separate cluster. Then, it iteratively merges a cluster with another cluster with single snippet provided that the merged cluster satisfies the cluster constraints and the size of the recommended code snippet from the merged cluster is minimally reduced. We next formally describe the clustering algorithm.

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 $1 \le j < k$. A tuple $(i_1, \ldots, i_k)$ denotes 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

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

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

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

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 not too 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 $\{(i) \mid 1 \leq i \leq |N_2|$ and (i) is a valid tuple$\}$.
(2) $C_{\ell+1} = C_\ell \cup \{(i_1, \ldots, i_k, i) \mid (i_1, \ldots, i_k) \in C_\ell$ and $i_k < i \leq |N_2|$ and $(i_1, \ldots, i_k, i)$ is a valid tuple and $\forall j$ if $i_k < j \leq |N_2|$ then $\mathrm{l}((i_1, \ldots, i_k, i)) \geq \mathrm{l}((i_1, \ldots, i_k, j))\}$

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 developed this custom clustering algorithm because existing popular clustering algorithms such as k-means, DBSCAN and Affinity Propagation all gave poor recommendations. Our clustering algorithm makes use of several similarity metrics (the containment score, the Jaccard similarity of various feature sets), whereas standard clustering algorithms usually depend on a single notion of distance. We found the current best similarity metric and clustering algorithm through trial and error.

After computing all valid tuples, AROMA sorts the tuples in ascending order on the first index in each tuple and then in descending order on the 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 too similar to each other. Let N_3 be the sorted list of the remaining clusters.

Intersection. AROMA creates a recommendation by intersecting all the snippets in each cluster. The intersection algorithm uses the Prune function described in Algorithm 1 and ensures that the intersection does not discard any code fragment that is part of the query. Formally, 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. The highlighted text represents tokens selected in the pruning step.[1]

---

[1] 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.

```
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. The highlighted text represents tokens selected in the pruning step.[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 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

---

[1]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.

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 on a 24-core CPU. The median response time is 1.3s and 95% queries complete in 4 seconds. A 24-core server was not necessary to achieve reasonable response time: We reran our experiments on a 4-core desktop machine, and the average response time is 2.9 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 always selected the first half of each code snippet to avoid subjective bias. Since we know how the tool works, we would be inclined to pick the lines that we think will produce better results. On average, the query code snippets were 1 to 5 lines and contained 10 to 100 features.

We could not extract partial query code snippets from 14 out of 64 code snippets because they contained a single statement. Single-statement snippets do get recommendations, but since we do not have a ground truth, we cannot judge their quality objectively. 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 the 64 code snippets as queries to evaluate the quality of Aroma's recommendations. While the experiment in the previous section used *partial* snippets extracted from each of the 64 code snippets, here we used the *full* code snippets. This meant that we could use all 64 snippets instead of just the 50 used in Section 4.2, as we did not have to filter out single-statement code snippets.

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 where a programmer writes the query code, they would benefit from seeing the related methods or common usage patterns in the code recommendations. 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 using mined API usage patterns for code completion. To compare GraPacc's code recommendation capabilities to Aroma's, 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 BigGroum's dataset because this tool extends the pattern-mining tool Groum to scale to large corpora with over 1000 repos. While there are more recent ML-based code completion tools, they focus on completing the next token or predicting the correct API method to invoke, which does not directly compare to Aroma. Among the 15 snippets in this dataset, 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 parts 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. We employed statistical bootstrapping to minimize sampling bias from the dataset. Then, we compared it with alternative setups using clone detectors and conventional search techniques. The results are reported in Table 4.

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

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%**

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.[1] 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.

We employed the statistical bootstrapping method and randomly sampled the dataset 30 times, each time from the entire code corpus (i.e. with replacement), in order to test the robustness of our approach and to reduce sampling bias. We ran the recall test on each of the 30 samples, and calculated a bootstrapped confidence interval using $\hat{\theta} \pm Z_{\alpha/2}\hat{\sigma}/\sqrt{B}$, where $\hat{\theta}$ is the average recall rate, $\hat{\sigma}$ the standard deviation of the recall rates, α the confidence level set to 0.05, and B the number of resampling 30. For contiguous code snippets, the bootstrapped confidence interval is 99.34% ± 0.25% for Recall@1, and 99.97% ± 0.03% for Recall@100. For non-contiguous code snippets, the bootstrapped confidence interval is 98.71% ± 0.49% for Recall@1, and 99.80% ± 0.26%

---

[1]Adapted from https://github.com/iqiyi/dexSplitter/blob/master/extra/hack_dx/src/com/android/dex/Leb128.java#L82. Accessed in August 2018.

for Recall@100. These confidence intervals suggest that AROMA is able to robustly recall the original method bodies given partial code snippets.

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. SOURCERERCC [Sajnani et al. 2016] is a state-of-the-art clone detector that supports Type-3 clone detection. We wanted to compare AROMA with SOURCERERCC to examine whether a current-generation clone detector can be used as the light-weight search phase in AROMA.

We instantiated SOURCERERCC 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. We could modify SOURCERERCC to return the similarity scores, but we do not expect the results to change.

SOURCERERCC's recall was 12.2% and 7.7% for contiguous and non-contiguous code queries, respectively. SOURCERERCC indexes at method-level granularity, and only returns methods whose entire body matches the query code. We also found that in many cases SOURCERERCC found code snippets *shorter* than the query snippet. While these are Type-3 clones by definition, they are not useful for generating code recommendations in AROMA. Extending SOURCERERCC to return the methods enclosing the clone snippets found would not work, because it does not consider the methods enclosing the target snippets as "clones" in the first place. We worked closely with a member of the SOURCERERCC team, and found that making SOURCERERCC find all occurrences of an arbitrary code snippet, contiguous and non-contiguous, would require significant reengineering. Therefore, we conclude that current-generation clone detectors may not suit Aroma's requirements for light-weight search.

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 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 IN DEPLOYMENT

For deployment, we have implemented AROMA for three 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.

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	

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 our setup, we deployed AROMA on a dedicated set of servers to respond to queries from all developers. Compared to a distributed setup—where AROMA runs on individuals' laptops—this setup allowed for easier control and delivery of new search models. Our search server has 24 cores, and on average took 1.6 seconds end-to-end to create a code recommendation. However, a 24-core server was not necessary to achieve reasonable response time: We reran our experiments in Section 4.2 on a 4-core desktop machine, and the average response time is 2.9 seconds.

The indexing stage (i.e. the creation of the feature vectors, as described in the beginning Section 3.3) was also designed for scalability. In our setup, we rebuilt the feature vectors for all methods every day. The size of the codebase was comparable to the evaluation dataset. On a 24-core server,
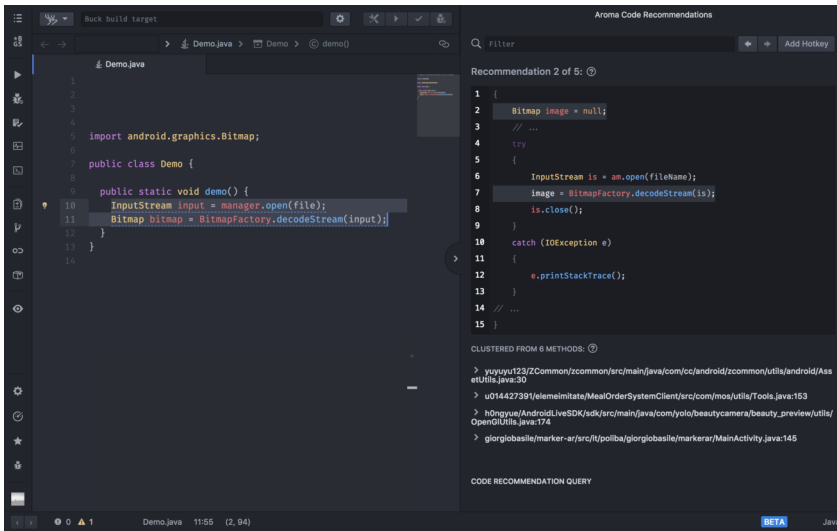
Fig. 5. Aroma code recommendation plugin in an IDE. Recommended code snippets are shown in the side pane for code selected in the main editor.

building the feature vectors took 20 minutes on average. If Aroma is being deployed on a larger-scale codebase, it is possible to implement incremental indexing for only the changed files, without rebuilding the entire feature matrix.

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".
- *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[1] 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."

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 to find 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. Further, most of these search engines cannot be instantiated on our code corpus, so we could not experimentally compare Aroma with these search engines. For instance, the code search engine FaCoY only provides a VM-based demo that is instantiated on a fixed corpus which is not available publicly. We were also unable to instantiate FaCoY on our corpus for a direct comparison. Most other open-source code search tools, including Krugle and searchcode.com, suffer from the same problem. 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. SNIFF also intersects the search results to provide recommendations, but only targets at resolving natural language queries. The clustering algorithm in SNIFF is limited and does not take structure into account. Two statements are considered similar if they are syntactically similar after replacing variable names with types. The intersection of two

---

[1]BigGrep refers to a version of `grep` that searches the entire codebase.

code snippets is the set of statements that appear in both snippets. Due to the strict definition of similarity, SNIFF cannot find large clusters that contain approximately similar code snippets. Also, SNIFF uses the longest common subsequence algorithm, whose limitations we discuss in Section 3.3.2. 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, and 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 contains 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 a 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

Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2 ed.). Pragmatic Bookshelf.

M. F. Porter. 1997. Readings in Information Retrieval. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter An Algorithm for Suffix Stripping, 313–316. http://dl.acm.org/citation.cfm?id=275537.275705

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.