



FACER: An API usage-based code-example recommender for opportunistic reuse

Shamsa Abid¹ · Shafay Shamail¹ · Hamid Abdul Basit² · Sarah Nadi³

Accepted: 9 June 2021 / Published online: 18 August 2021

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

To save time, developers often search for code examples that implement their desired software features. Existing code search techniques typically focus on finding code snippets for a single given query, which means that developers need to perform a separate search for each desired functionality. In this paper, we propose FACER (Feature-driven API usage-based Code Examples Recommender), a technique that avoids repeated searches through opportunistic reuse. Specifically, given the selected code snippet that matches the initial search query, FACER finds and suggests related code snippets that represent features that the developer may want to implement next. FACER first constructs a code fact repository by parsing the source code of open-source Java projects to obtain methods' textual information, call graphs, and Application Programming Interface (API) usages. It then detects unique features by clustering methods based on similar API usages, where each cluster represents a feature or functionality. Finally, it detects frequently co-occurring features across projects using frequent pattern mining and recommends related methods from the mined patterns. To evaluate FACER, we run it on 120 Java Android apps from GitHub. We first manually validate that the detected method clusters represent methods with similar functionality. We then perform an automated evaluation to determine the best parameters (e.g., similarity threshold) for FACER. We recruit 10 professional developers along with 39 experienced students to judge FACER's recommendation of related methods. Our results show that, on average, FACER's recommendations are 80% precise. We also survey a total of 20 professional Android and Java developers to understand their code search and reuse experiences, and also to obtain their feedback on the usability and usefulness of FACER. The survey results show that 95% of our surveyed professional developers find the idea of related method recommendations useful during code reuse.

Keywords Code recommendation · Code search engine · Software features · API usage · Code clones

Communicated by: Ali Ouni, David Lo, Xin Xia, Alexander Serebrenik and Christoph Treude

This article belongs to the Topical Collection: *Recommendation Systems for Software Engineering*

This work is funded by Lahore University of Management Sciences (LUMS), Ignite National Technology Fund Pakistan (SRG-257), and Prince Sultan University Faculty Research Fund.

✉ Shamsa Abid
shamsa.abid@lums.edu.pk

Extended author information available on the last page of the article.

1 Introduction

When developers are implementing a given system, they usually have a set of *features* (i.e., units of functionality) they need to implement. For example, a Bluetooth chat application can have features like setting up Bluetooth, scanning for other Bluetooth devices, connecting to a remote device, and transferring data over Bluetooth. A feature may be implemented in a single method or across a group of methods that call each other.

To speed up development, developers often resort to code search to find code that they can reuse for certain features in their application (Sadowski et al. 2015; Xia et al. 2017). For example, if Alice is developing a music application, one of the features her application must support is playing a given media file. Thus, Alice might search for “play media file” and use the returned code snippet in her code. Our premise is that Alice might then need to implement “pause media file”, “get current track progress”, “check if media playing”, and “handle touch event” features. With the availability of very large codebases, additional functionality related to Alice’s query is likely to already exist somewhere in those codebases. However, most of the existing code search systems focus on providing code corresponding to a single query related to the current feature the developer needs to implement (Keivanloo et al. 2014; McMillan et al. 2011; Bajracharya et al. 2010; Chatterjee et al. 2009; Ishihara et al. 2013; Gu et al. 2018; Lv et al. 2015; Sachdev et al. 2018). As a result, developers may have to conduct a new search for every next feature they need to implement and later integrate the obtained code. Existing code search and recommendation systems do not support the need to find code for additional features related to a developer’s query. On the other hand, existing feature recommendation systems enable the exploration of text-based related features and either support domain analysis for the requirements gathering phase (He et al. 2019; Chen et al. 2018; Hong et al. 2016; Yu et al. 2013; Dumitru et al. 2011) or enable rapid prototyping by recommending related code modules (McMillan et al. 2012). However, they do not provide code examples at a fine-grained method-level for reuse. Hence, we identify two gaps in existing systems: one is the lack of support for providing related code recommendations in existing code search systems and the other is the inability of existing feature recommendation systems to provide code for features at the method-level granularity.

Based on this perspective, in this paper, we fill in the above gaps by proposing a recommendation system that provides developers with related method recommendations that have functionality relevant to their application under development. Studies of rapid prototype development have shown that programmers iteratively add features by reusing source code examples (Brandt et al. 2008, 2009). This iterative process is known as *opportunistic programming* (Brandt et al. 2008). To this end, we propose a system that provides code recommendations for these related features to support *opportunistic code reuse* (Jansen et al. 2008); such support enables rapid application development without the need to conduct multiple searches and thus enhances developer productivity and saves time (Brandt et al. 2008; Abid et al. 2017; Jansen et al. 2008; Hartmann et al. 2008).

Our proposed recommendation system is called FACER, Feature-driven API usage-based Code Examples Recommender, and works at the granularity of methods, where the recommended code snippet is a full method itself. We use a combination of static code analysis, information retrieval, and data mining techniques to build FACER. More precisely, we add another layer on top of traditional code search techniques in order to additionally propose code snippets corresponding to features related to the original search query.

FACER generates related method recommendations in two stages. The first stage corresponds to traditional code search where any existing code search technique (Sachdev et al.

2018; Gu et al. 2018; Chatterjee et al. 2009; Bajracharya et al. 2010; Keivanloo et al. 2014) can be used. We use Lucene (2020) to implement the code search engine behind FACER. Given a developer's feature query in the form of natural language description, the search stage of FACER recommends a set of methods that implement the desired feature. Upon selection of one of these recommended methods by the developer, the second stage of FACER starts. In this second stage, which is the main contribution of this paper, FACER provides subsequent recommendation of related methods for reuse. FACER recommends these related methods based on patterns of frequently co-occurring features which we identify as frequently co-occurring method clones. Since methods with similar uses of Application Programming Interfaces (APIs) are semantically related (Bajracharya et al. 2010), we identify Method Clone Groups (MCG) based on API usages. Thus, a *method clone group* contains code examples for a common feature. To find semantically related features, we then identify frequently co-occurring Method Clone Groups (which we refer to as *Method Clone Structures*), leveraging the idea of market basket analysis (Han et al. 2011). *Market basket analysis* attempts to identify associations, or patterns, between the various items that have been chosen by a particular shopper and placed in their basket (Market-basket analysis 2019). Items that frequently co-occur are related to each other. In our context, one or more methods of a particular project may be cloned across other projects. Such a pattern of co-occurring method clones identifies related functionality and forms the basis of suggesting relevant related methods.

While the concepts behind FACER are not tied to a particular programming language or type of application, we focus on Java Android apps for building and evaluating the first version of FACER. According to the latest Stack Overflow developer survey 2020, 57.1% of 65,000 developers surveyed are developing Android apps (Stack Overflow developer survey 2020). A recent exploratory study focusing on code reuse from StackOverflow in the context of mobile apps found that feature additions and enhancements in apps are the main reasons for code reuse from StackOverflow (Abdalkareem et al. 2017). Furthermore, findings from a large-scale empirical study on software reuse in mobile apps indicate a high percentage of code reuse across applications (Mojica et al. 2013). Since Android development involves rapid release cycles (McIlroy et al. 2016), there is a need to facilitate opportunistic reuse to enable rapid application development. An empirical study involving the manual analysis of 5,000 commit messages from 8,280 Android apps found that application enhancement is the most frequent self-reported activity of Android developers (Pascarella et al. 2018). In the same study, self-reported activities of Android developers have been categorized and each development category is seen to be composed of a related set of activities. For example, activities related to using the device camera include taking a picture when using the app, when and how to show the preview of a taken picture, usage of the flash light, switching between front and rear camera. The challenge is whether code for these related activities can be made available to a developer without the need for the user to explicitly perform a search for each desired activity, which is what we address in this paper.

We previously proposed the idea of FACER in a short research abstract (Abid 2019) (FSE student research competition). This manuscript extends that abstract by providing an elaborate description of an enhanced system design and a full evaluation of the system's performance. Thus, this paper makes the following main contributions:

- We present a recommendation approach named FACER that recommends methods to implement additional features related to the developers' currently searched feature. These recommended additional methods are based on API usage-based Method Clone Groups and Method Clone Structures.

- We develop a procedure for the construction of FACER’s code fact repository, (henceforth referred to as the *FACER repository*) that contains methods’ search index, call graphs, API usages, and API usage-based Method Clone Structures mined from source code.
- We apply this procedure to 120 Java Android apps to populate our FACER repository.
- We perform a manual validation of a sample of the clone groups that FACER detects as methods implementing the same feature. We find that 91% of the analyzed clone groups are valid.
- We perform an automated evaluation of the above 120 apps to determine the best configuration for parameters that affect the precision of our FACER’s related method recommendations.
- We engage 10 professional Android developers and 39 experienced students to manually evaluate the performance of FACER for recommending related methods. Our results show that FACER achieves 80% precision, on average.
- We survey 20 professional Android and Java developers to investigate their code search and reuse requirements. We find that 70% of the developers face the need to search for related features which supports the motivation of our work.
- We survey 20 professional Android and Java developers to capture their feedback on the usability and usefulness of FACER. The survey results show that 90% of the professional developers perceive that FACER is effective for their development activities and 95% of the developers find the related method recommendations useful. We also survey 39 experienced Masters students to provide their feedback on the usefulness of FACER, where 85% of the students find the related method recommendations useful.

The rest of this paper is organized as follows: Section 2 includes a problem scenario as a motivating example for our proposed solution. Section 3 gives an overview of code search and recommendations systems, discusses related work from the literature, and highlights current limitations. Section 4 describes our proposed approach. Section 5 includes the research questions and details of our dataset.

In Section 6, we discuss the evaluation of clone groups’ validity. In Section 7, we describe the evaluation which measures the precision of related method recommendations using automated and manual methods. In Section 8, we describe the user survey. We discuss threats to validity in Section 9. In Section 10, we discuss aspects of FACER that can be improved as future work. Finally, Section 11 concludes this article.

2 Motivating Example

We use an example from a real Stack Overflow user question (2020) to demonstrate the problem of developers spending a lot of time searching for related functionality. The title of the question is “android:select image from gallery then crop that and show in an imageview” and the description of the question includes the following:

“I really need this code and I searched for 3 hours on internet but i couldn’t find a complete and simple code and I tested many codes but some didn’t work and others [weren’t] good, please help me with a full and simple code ... edit:I have this code for select image but please give me a full code for all [the] things that i said in title ...”

This question is viewed 50K times on Stack Overflow. The user is looking for code that allows her to perform three functionalities: first, selecting an image from gallery, second,

cropping the image and third, showing the image in an image view. She has spent a lot of time searching for these related functionalities. StackOverflow lists questions that are linked to this user question in a “Linked” sidebar. These linked questions include “How to pick an image from gallery and save within the app after cropping?” and “crop image by taking photo from camera”. We see that a number of functionalities are frequently desired together; users who select an image from a gallery need to crop it and then show it in an image view or save it after cropping. A user may need to crop an image after selecting it from a gallery or capturing it from the camera. There are numerous Android applications from a diverse range of categories including photo sharing applications, photo editing applications, virtual try-on applications for eye glasses, etc. in which all of these functionalities related to manipulating images are present. The current gap in existing code recommendation systems is that they do not cater to the user need for finding related functionality. McMillan et al. highlight programmers’ need of accomplishing a whole task quickly, rather than obtaining multiple examples for different components of the task (McMillan et al. 2011).

We now explain how our approach could have helped in this situation. To illustrate this, we collect a set of 30 photo sharing applications from GitHub using the search string “android photo sharing app”, sort the results by GitHub Stars and choose the top 30 most relevant apps to populate a sample FACER repository. We then enter a search query to our system “select image from gallery” and from the recommended methods, we select one that contains the desired functionality (shown in Fig. 1a). Next, we use FACER to retrieve related method recommendations against the selected method. The related methods FACER recommended included methods that implement the above Stack Overflow user’s desired features of cropping an image and also showing an image in an *ImageView* as shown in Fig. 1b and c respectively. FACER also recommended additional related methods, which include functionality for resizing a bitmap, getting a URI to save the cropped image, getting the URI to an image received from a capture by camera, and decoding an image from a URI. By receiving such related method recommendations, the developer can obtain information about related features, in the form of concrete methods, to enhance her application. Furthermore, this reduces the need to perform repeated searches.

3 Related Work

There are various types of techniques and support systems proposed in the literature to enable code search, code reuse, and feature exploration. In this section, we discuss these systems under three major categories; namely, code search, code recommendation, and feature recommendation systems. We first define each category and describe the purpose of systems belonging to that category. We then discuss existing systems from each category in relation to our approach.

Finally, we highlight the limitations of code search and feature recommendation systems that lead to the inception of our approach.

3.1 Code Search Systems

Code search systems are mainly used to retrieve code samples and reusable open source code from the web (Bielik et al. 2015; Vechev et al. 2016). To understand why programmers search for code, Umarji et al. (2008) conducted a web-based survey and categorized code search purposes along two orthogonal dimensions: motivation (reuse vs. reference example) and size of search target. The targets of these searches ranged in size from a block

```

public static Intent getPickImageChooserIntent(
    @NonNull Context context, CharSequence title, boolean includeDocuments) {
    List<Intent> allIntents = new ArrayList<>();
    PackageManager packageManager = context.getPackageManager();
    if (!isExplicitCameraPermissionRequired(context)) {
        allIntents.addAll(getCameraIntents(context, packageManager));
    }
    List<Intent> galleryIntents = getGalleryIntents(
        packageManager, Intent.ACTION_GET_CONTENT, includeDocuments);
    if (galleryIntents.size() == 0) {
        galleryIntents = getGalleryIntents(packageManager,
            Intent.ACTION_PICK, includeDocuments);
    }
    allIntents.addAll(galleryIntents);
    Intent target;
    if (allIntents.isEmpty()) {
        target = new Intent();
    } else {
        target = allIntents.get(allIntents.size() - 1);
        allIntents.remove(allIntents.size() - 1);
    }
    Intent chooserIntent = Intent.createChooser(target, title);
    chooserIntent.putExtra(Intent.EXTRA_INITIAL_INTENTS,
        allIntents.toArray(new Parcelable[allIntents.size()]));
    return chooserIntent;
}

```

(a) Selected code snippet

```

private static Bitmap cropBitmapObjectWithScale(Bitmap bitmap, float[] points,
    int degreesRotated, boolean fixAspectRatio, int aspectRatioX, int aspectRatioY,
    float scale) {
    Rect rect = getRectFromPoints(points, bitmap.getWidth(), bitmap.getHeight(),
        fixAspectRatio, aspectRatioX, aspectRatioY);
    Matrix matrix = new Matrix();
    matrix.setScale(scale, scale);
    matrix.postRotate(degreesRotated, bitmap.getWidth() / 2, bitmap.getHeight() / 2);
    Bitmap result = Bitmap.createBitmap(bitmap, rect.left, rect.top,
        rect.width(), rect.height(), matrix, true);
    if (result == bitmap) {
        result = bitmap.copy(bitmap.getConfig(), false);
    }
    if (degreesRotated % 90 != 0) {
        result = cropForRotatedImage(result, points, rect, degreesRotated,
            fixAspectRatio, aspectRatioX, aspectRatioY);
    }
    return result;
}

```

(b) Crop image

```

@Override
public View getView(int i, View view, ViewGroup viewGroup)
{
    ImageView imageView;
    if (view == null) {
        int gridWidth = fragment.getScreenWidth();
        imageView = new ImageView(mContext);
        imageView.setLayoutParams(
            new GridView.LayoutParams(gridWidth/5 - 30, gridWidth/5 - 30));
        imageView.setScaleType(ImageView.ScaleType.FIT_CENTER);
        imageView.setPadding(5, 5, 5, 5);
    } else {imageView = (ImageView) view;}
    Bitmap bmp = getResizedBitmap(loadImage(imageFileNames.get(i)), 200);
    imageView.setImageBitmap(bmp);
    return imageView;
}

```

(c) Show image in ImageView

Fig. 1 Motivating example for code recommendations related to “select image from gallery”. **a** shows the selected code snippet based on the initial search query and **b** and **c** show code snippets corresponding to two related features, as recommended by FACER

(a few lines of code) to a subsystem (e.g. library or API), to an entire system. A study on developers’ code search behavior finds that most searches are related to searching for code examples, discovering a library for some task, or discovering the usage of some API (Sadowski et al. 2015). In this section, we discuss code search systems that retrieve code against

a user query, because they are related to the first stage of FACER. Chatterjee et al. (2009), Bajracharya et al. (2010), McMillan et al. (2011), Keivanloo et al. (2014), Ishihara et al. (2013), Gu et al. (2018), and Sachdev et al. (2018).

Sniff (Chatterjee et al. 2009) is one such system which helps users discover code snippets involving library usage. It uses the documentation of the library methods to annotate code with plain English for the purpose of free-form query search. It then takes an intersection of the candidate code snippets obtained from a query search to generate a set of relevant code snippets. The drawback of this technique is the dependency on the availability of library documentation. Our approach is free from this dependency.

The Structural Semantic Indexing (SSI) technique (Bajracharya et al. 2010) finds API usage examples corresponding to standard keyword-based queries. The authors create a baseline retrieval system that uses a Lucene-based (2017) search index of code entities based on their simple name, Fully Qualified Name (FQN), and full method bodies. We implement a similar technique for the first search stage of FACER, but we additionally support free-form queries and tokenize API usages for better matching of queries to source code. We opt for using this technique because of its simplicity and good performance (Linstead et al. 2009; lucenecore 2020).

Keivanloo et al. (2014) propose a system that retrieves code examples from a corpus of code snippets based on free-form querying (composed of keywords). They create a *p-string* (Baker 1993) for each line of a code snippet in the corpus and encode matching *p-strings* as a pattern. By applying identifier splitting techniques on all strings that belong to an encoded pattern, they extract a set of associated keywords. The retrieval involves matching keywords of a user query with keywords associated with patterns and getting the most popular code examples containing the matched patterns.

Portfolio retrieves and ranks relevant functions against query terms that are also connected on a call-graph. They output results as a list of function names and a visualization showing dependencies between retrieved functions. While *Portfolio* can provide related functions for opportunistic reuse, the functions are limited to call-graph dependencies and therefore, do not cover the scope of an entire project. As such, it might not be able to retrieve related functions that are not necessarily found on call chains.

Ishihara et al. (2013) use source-code clone detection to find instances of copy-paste reuse scenarios. Keywords are extracted from the clones and saved in a database. Code is then retrieved against a user query using keyword matching. This search technique is effective in organizations where similar projects are frequently developed and a local source code repository is maintained.

Several of the latest code search techniques that find code given a natural language query rely on machine learning techniques (e.g., *NCS* (Sachdev et al. 2018), *DeepCS* (Gu et al. 2018), *UNIF* (Cambronero et al. 2019), *MMAN* (Wan et al. 2019), *TBCAA* (Chen et al. 2019), and *CoaCor* (Yao et al. 2019)). *NCS* proposes an enhanced word embedding for a natural language query (Sachdev et al. 2018). The *NCS* model captures the co-occurrence frequency of word pairs from Stack Overflow questions and their respective code solutions. Using its model, *NCS* finds synonyms of query words to enhance the query and improve the quality of recommendations. *DeepCS* (Gu et al. 2018) introduces the use of a unified vector representation of code and natural language descriptions. This unified representation bridges the lexical gap between queries and source code resulting in relevant code fragments that do not necessarily contain query words.

UNIF (Cambronero et al. 2019) is an extension of *NCS* that adds supervision to modify embeddings during training with the overall effect of improving the performance for code

search. *MMAN* (Wan et al. 2019) is a Multi-Modal Attention Network for semantic source code retrieval. It generates a code representation that covers both unstructured and structured features of source code including code tokens, abstract syntax trees, and control flow graphs to form a single hybrid representation. This has been shown to outperform *DeepCS*. *TBCAA* (Chen et al. 2019) employs tree-based convolution over API-enhanced ASTs for semantics-based code search. This technique aims to capture semantics by incorporating API call information into ASTs which is otherwise abstracted as the same AST node type.

CoaCor (Yao et al. 2019) uses reinforcement learning to build a code annotation framework for effective code retrieval. By generating detailed code annotations using multiple keywords, *CoaCor* improves the performance of existing code retrieval models.

For our purposes of locating code against a user query in the first stage of FACER, any of the above code search methods would work. We currently use Lucene (2020) to build the code search engine behind FACER. Lucene is a popular search library for the development of various information retrieval solutions because of its scalability, high-performance and efficient search algorithms (Yang et al. 2017). It is shown to answer the highest number of queries as compared to other code search approaches (Yan et al. 2020).

3.2 Code Recommendation Systems

In general, *recommendation systems* aid people to find relevant information and to make decisions when performing particular tasks. Different recommendation systems use different user inputs to provide the output code. The input may be a free form textual query or it may include components of a user's currently active code environment like method signatures, keywords, or structural information. We focus on *source code-based recommendation systems* (SCoReS) (Mens and Lozano 2014), that is, recommendation systems that produce their recommendations by essentially analyzing the source code of a software system. Given our scope, we review only a subset of the research that provides code-snippet based recommendations as output.

Some of the earliest code recommendation systems for methods are *CodeBroker* (Ye and Fischer 2002) and *Strathcona* (Holmes et al. 2005a, b, 2006). *CodeBroker* uses comments and method signatures of a yet-to-be-written method to retrieve relevant code, whereas in *Strathcona*, the search query is either the structural information of some class or method (signature, object instantiations) that the developer needs help for. Others include: *A-Score* (Shimada et al. 2009), which recommends a list of classes against user code based on cosine similarity of code characteristics; *Selene* (Takuya and Masuhara 2011), which forms a search query from the code around the user's cursor in an IDE and provides code examples from files containing those lines; and *ROSF* (Jiang et al. 2016), which recommends code snippets against a free-form query by first generating a candidate set of snippets using information retrieval followed by re-ranking the code snippets using a learned prediction model that is trained on a set of user queries and code-snippet features such as text, topic, and structure.

Among existing code recommendation techniques, Ichii et al. (2009) allow opportunistic reuse by using collaborative filtering to help developers find components suitable for their needs. This system extracts a developer's browsing history when the developer starts navigating through the search results provided by a SPARS-J (Inoue et al. 2005) search engine. It recommends components to the developer using browsing session similarities based on the assumption that two developers having similar browsing history require similar components. However, it is effective only if developers' browsing profiles are available. *Rascal*

(Mccarey et al. 2005) is a collaborative filtering-based recommendation system that predicts the next method that a developer could use by analyzing classes similar to the one currently being developed. It tracks usage histories of developers for recommending components to an individual developer. Here, a component refers to a method call made on a class instance. We rely on a similar notion of collaborative filtering but instead of relying on method usage profiles of classes or browsing session profiles, we rely on feature co-occurrence profiles for projects, where a feature represents a collection of API usages. In the context of FACER, recommended components are complete methods pertaining to a feature.

In previous work, we developed CodeEase (Abid et al. 2017), which provides method completion recommendations against a partial method as well as related method recommendations for the completed method. CodeEase mines association patterns over a source code collection of Java projects by first detecting type-2 clones and then finding frequently co-occurring inter-project clones (Ishihara et al. 2012). First, CodeEase uses a type-2 clone search to suggest method completions. Then, for a selected method completion, CodeEase looks up methods that occur alongside the selected method in its collection of association patterns. Our internal experiments on mining Method Clone Structures based on type-2 clones proved that patterns detected using traditional clone detection were very rare. This led us to move beyond the notion of detecting similar methods on the basis of type-2 or type-3 clones and to experiment with the notion of functional similarity based on common API usages among methods (Bajracharya et al. 2010). Shifting our focus from syntactic matching in conventional clone detection to API calls matching allows us to identify clones as a set of methods having similar behavior irrespective of syntactic differences. As a result, new co-occurrence patterns emerge, offering more possibilities of opportunistic reuse.

3.2.1 API Recommendation Systems

API recommendation systems are a type of code recommendation systems focusing particularly on helping developers use library APIs. Some of these systems recommend code on the basis of mining API usage patterns (Xie and Pei 2006; Wang et al. 2013; Niu et al. 2017; Nguyen et al. 2019); however, none of these use the notion of opportunistic reuse of related API usage patterns. There are systems that suggest complete code snippets or usage sequences that demonstrate how to use a given API (Mandelin et al. 2005; Thummalapenta and Xie 2007; Wang et al. 2011; Mishne et al. 2012; Lv et al. 2014; Subramanian et al. 2014; Moreno et al. 2015; Gu et al. 2016; Zhao and Liu 2017). API class recommendation systems (Zhang et al. 2018; Rahman et al. 2016; Thung et al. 2017; Tsunoda et al. 2005) output only the name of a relevant API class against a query.

Thung et al. recommend additional libraries based on the ones currently used by an application or project (Thung et al. 2013). Similarly, FACER recommends additional methods based on the one currently selected. Thung et al. find libraries that are commonly used together with the currently used libraries. They also find libraries that are used by the n most similar projects, then rate a library based on how many of the top- n projects use it. Their technique is a combination of association rule mining and collaborative filtering to find the top- n libraries. The notion of recommending additional items based on the market-basket principle of frequent co-occurrence is seen in their systems' *LibReCRULE* component. FACER bases its recommendation of additional methods on the same principle; however, the goal (library vs method) and code analysis techniques used in both cases are different.

3.2.2 Code Completion Systems

Code completion systems (Eclipse code recommender 2018; Hill and Rideout 2004; Bruch et al. 2009; Raychev et al. 2014; Nguyen et al. 2012; Asaduzzaman et al. 2016) suggest completions based on the context of the code being currently edited. Completions may simply be method calls for a given object (Eclipse code recommender 2018; Bruch et al. 2009; Raychev et al. 2014; Nguyen et al. 2012; Asaduzzaman et al. 2016) or can be complete method code for a given partial code snippet (Hill and Rideout 2004). Code completion is an integral feature of modern IDEs (Asaduzzaman et al. 2016). Most of the proposed techniques enable the integration of the code completion recommendations into the active user context, typically within their Integrated Development Environment (IDE). A code recommendation technique is typically at the back-end of code completion systems and so we consider code completion systems to be a sub-type of code recommender systems.

Code completion helps to avoid remembering every detail of the available API methods, write error-free code, speed up typing, and enables the completion of partial method bodies (Asaduzzaman et al. 2016). Hill and Rideout (Hill and Rideout 2004) propose automatic method completion based on the idea of atomic clones. Atomic clones are usually small units of implementation of 5-10 lines each, such as implementing a listener interface, or handling a keyboard event. By looking at these atomic clones and comparing them with the current code, a programmer is able to identify any critical points that she should remember to address. *Lancer* (Zhou et al. 2019), a context-aware tool, also assists method completion by analyzing partial method code to recommend relevant code samples. *Lancer* predicts and appends tokens to the current tokens within the context of a partially written method in order to produce a more complete token sequence for code retrieval. *Lancer* trains a Library-Sensitive Language Model (LSLM) on source code files to capture code patterns for each library separately. Using tokens from the original context, *Lancer* finds relevant libraries and predicts more tokens from these libraries. The final set of tokens is used to retrieve code samples which are further filtered and ranked based on similarity of the original context's tokens with tokens of the retrieved code samples. *Aroma* (Luan et al. 2019) is another tool that takes partial code as input and recommends code snippets containing the partial code in order to help developers write additional code and complete programming tasks effectively.

Bruch et al. (2009) make context sensitive method call recommendations against object instances of a particular framework. Their technique is based on a variant of the K-nearest neighbors algorithm, called Best Matching Neighbors (BMN). The context of the variable is extracted and variables used in similar situations are searched in an example codebase, then method recommendations are synthesized out of these nearest snippets. *CSCC* (Asaduzzaman et al. 2016) also performs API method call completion. To recommend completion proposals, *CSCC* ranks candidate methods by the similarities between their contexts and the context of the target call. *SLANG* (Raychev et al. 2014) is a code completion tool for Java that synthesizes complete method invocation sequences, including the arguments for each invocation. It inputs partial code snippet with holes, specified using a special construct and outputs API method calls with parameters as completions for these holes.

GraPacc (Nguyen et al. 2012) is a graph-based, pattern-oriented, context-sensitive code completion approach that is based on a database of API usage patterns. *GraPacc* extracts the context-sensitive features from the code being edited and uses these features to search and rank the patterns that best match the current code. When a pattern is selected, the current code is completed via a graph-based code completion algorithm.

MACs (Hsu and Lin 2011) is another system aimed at providing code completions for reuse and rapid application development. It recommends code against an input statement for completing an API usage sequence inside a method declaration or for completing code within a class declaration. While *MACs* facilitates code completions by recommending statements at the class or method scope, *FACER* facilitates feature completions at the project scope. Whereas *MACs* mines co-occurring associations between individual statements found across code files, *FACER* mines co-occurring associations between features found across projects. Our proposed approach goes beyond the completion of a developer's current statement or partial method. *FACER*'s scope of providing completions consists of the current project being developed and the completion proposals are methods containing code for the features relevant to the application.

3.3 Feature Recommendation Systems

Feature recommendation systems are meant to help software requirements engineers or developers with the discovery of new software features for their product by providing a list of relevant software feature descriptions (He et al. 2019; Chen et al. 2018; Hong et al. 2016; Yu et al. 2013; Dumitru et al. 2011).

Due to the popularity of mobile applications, recent work proposes solutions for recommending software features for mobile applications (He et al. 2019; Chen et al. 2018; Hong et al. 2016). For these recommender systems, a feature is recommended as a textual description. He et al. (2019) recommend features from applications that are similar to the developer's application. Recommended features are those that frequently co-occur with a developer's feature across highly similar projects. Chen et al. recommend features against a given User Interface (UI) based on user interface comparison of mobile applications (Chen et al. 2018). Their idea is based on the intuition that mobile applications with similar UIs may have both shared and unique features. They leverage the similarity of a given UI's components to other similar UIs in order to recommend unique features from the text of similar UIs. Yu et al. propose a hybrid feature recommendation approach that processes both textual descriptions and code information of mobile applications (Hong et al. 2016). They detect the most relevant applications against the query and recommend the main features of the relevant applications.

There are feature recommendation approaches not specific to mobile applications. One is proposed by Yu et al. in which a list of related textual feature descriptions are offered to users against an input textual query (Yu et al. 2013). The features are taken from marketing-like summaries, release notes and feature descriptions on the online profile pages of products. They perform feature pattern mining from a co-occurrence matrix of software projects and features. Their approach works for applications hosted on web-based repositories with rich profiles for effective topic modeling. Another approach facilitates domain analysis by recommending features derived from mining product descriptions (Dumitru et al. 2011). In both of these approaches, the recommended features are textual descriptions and do not map to actual code, whereas *FACER*'s feature recommendations are methods with API usages.

The closest work to ours in terms of goals is that by McMillan et al. (2012). Given a natural language query representing a description of their desired product, their system first uses cosine similarity with existing descriptions in software documentation to find related features. After the user confirms the desired features, the system does a feature to module mapping to recommend associated source code modules, specifically Java packages. While their goals are similar in terms of allowing a developer to quickly locate code for multiple

related features, there are fundamental differences in terms of code granularity level, techniques used, and user workflow. First, they recommend code at the level of a Java package while we recommend a single atomic method that encapsulates the desired functionality. This allows users to narrow down to relevant code without having to look at an entire package which may have irrelevant code. Second, while they rely on textual cosine similarity, FACER identifies features at the code level based on API usages and looks for co-occurring features in the code. Finally, their system first finds related features and then performs the feature to module mapping, whereas our system performs the query to method mapping first through code search and then finds the related methods based on co-occurrence across different projects.

3.4 Limitations of Code Search and Feature Recommendation Systems

We now summarize the current limitations of code search and recommendation systems that we have observed in the literature and which we are focusing on in our research work.

1. Existing code search engines do not support opportunistic reuse. They are effective for locating code against a single feature, but they are not designed to provide code for additional relevant features (McMillan et al. 2012). As a result, developers might have to conduct a new search for every next feature that they need to implement and later integrate the obtained code.
2. Existing feature recommender systems do not provide associated code against recommended features. Only one system allows the location of Java packages relevant to related features (McMillan et al. 2012). However, packages have a lot of unrelated code and may be grouping together classes implementing unrelated responsibilities with low cohesion (Bavota et al. 2014; Ishihara et al. 2013). On the other hand, code suggestions at the method-level granularity provide more concrete reusable code (Ohtani et al. 2015), which the existing feature recommender systems do not provide.

4 Proposed Approach: FACER

To address the limitations discussed in Section 3.4, we propose a system for opportunistic reuse of related code which allows developers to receive code examples for functionality they may like to implement next. Allowing developers to receive code for related features can enhance productivity and save search time (Abid et al. 2017). In this section, we discuss the components of our proposed Feature-driven API usage-based Code Examples Recommendation (FACER) system. Figure 2 provides an overview of the various components in FACER and its workflow.

FACER has two main workflows: (1) the *offline FACER repository building workflow* which builds facts through mining information from source code repositories and (2) the *online recommendation workflow* which uses this information to make recommendations. From a user perspective, the user provides a feature query as a natural language description; in other words, this is the task or feature they want to implement. The FACER search engine then returns a list of matching methods. After the user selects a method from that list, the FACER recommender then returns a list of related methods that correspond to additional features the developer may want to implement. We now discuss these two workflows in more detail.

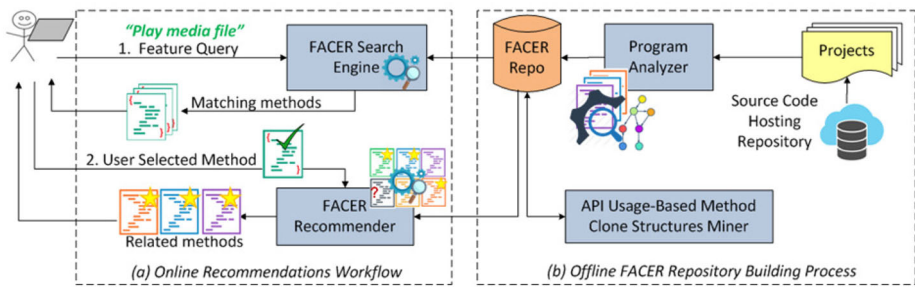


Fig. 2 FACER system components and workflow

4.1 Offline FACER Repository Building Workflow

In order to provide its recommendations, FACER first has an “offline” phase where it populates its repository (a MySQL database) with source code information from open-source Java applications hosted on GitHub (2020). We discuss the details of the data we select to populate this repository for our evaluation in Section 5.2. Figure 3a shows the three types of information we extract from each application’s methods using the Eclipse JDT parser (Eclipse Java development tools 2020): keywords, method calls, and API usages.

4.1.1 Extracting Keywords for Search Index

To implement a simple retrieval scheme (Bajracharya et al. 2010) for code search, FACER’s program analyzer builds a search index. Any code search technique can be used to retrieve code. For the purposes of this work, we implement a simple Lucene-based search index. Lucene is a high-performance, full-featured text search engine library suitable for full-text search over documents (lucenecore 2020). We use Lucene to build a search index over methods and store them as a collection of documents. A document is a set of fields. Each field has a name and a textual value. A field may be stored with the document, in which case it is returned with search hits on the document. Thus each document should typically contain one or more stored fields that uniquely identify it (lucenedoc 2020). The program analyzer extracts all the terms from the simple name, Fully Qualified Name (FQN), and full text of a method, and tokenizes each set of terms using camel-case and special characters. It then creates a separate Lucene field to store the extracted terms from the method name,

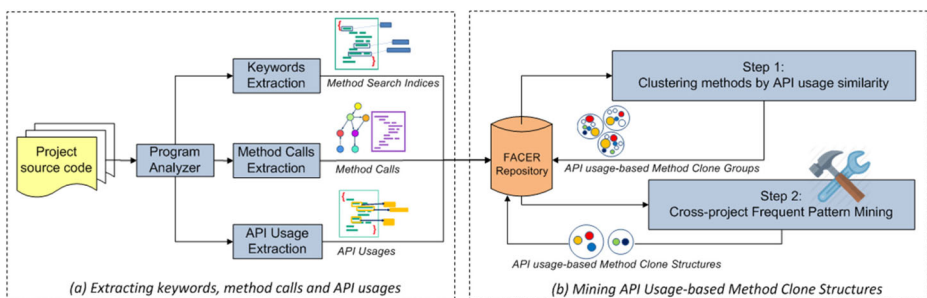


Fig. 3 Offline FACER repository building components

Fully Qualified Name (FQN), and the full text of the method body respectively. To give more significance to matches with the method name during code search, it assigns the *methodName* field a higher boost value (Score boosting 2020) than the other fields. Finally, it creates a Lucene document against every method to build the search index.

4.1.2 Extracting Method Calls, API Calls, and API Call Density

An *API usage* is a set of API calls found in a method. The underlying premise of FACER is that these API calls together represent the implementation of a feature. FACER detects repeatedly co-occurring features on the basis of repeatedly co-occurring API usages. Thus, in its repository, FACER needs information about API usages. A software application interacts with external libraries or system libraries/packages through various API classes to implement desired features. For example, building the connection to a Bluetooth device requires the use of the Bluetooth API package and different methods of the API to setup the connection. When analyzing a source code project, we parse the class declarations in Java files and save them as user-defined classes. The Eclipse JDT (Eclipse Java development tools 2020) parser is able to trace the objects to their respective types. While parsing method invocations, if a type identified by the parser does not match any user-defined class, then we consider this type as an API class. Thus, we refer to any method call from an API class as an *API call*. For example, `BluetoothAdapter.getDefaultAdapter()` is an API call of class `BluetoothAdapter` from the `android.bluetooth` API package.

Using the Abstract Syntax Tree (AST) (Abstract Syntax Trees 2020) provided by Eclipse JDT (Eclipse Java development tools 2020), the Program Analyzer module visits all method declarations and parses their content to identify calls. For each detected call, we record the call site, which is the location of the call in the method body and is identified by a line number. In the FACER repository, we differentiate between *user-defined method calls* (or *method calls* for short), which are invocations of methods that have been defined in the current project, and *API calls* which are invocations of API methods. To differentiate the types of calls, we check the receiver type of the call. API types may be classes from the Java Class libraries (JCL) (Java Class Libraries 2020) in JDK (Java Development Kit 2020) or Android classes (Android SDK Classes 2020) in Android SDK (Android Studio SDK 2020) or any other third-party library imported by the user. FACER stores API calls for every new API instance created (i.e., constructor calls) and for every API method called, including static calls. Listing 1 shows an example method where the API calls that FACER extracts are underlined.

FACER stores API calls that it mines from all the methods in its repository. We encode the API calls occurring across the entire FACER repository with unique identifiers which we call *API Call IDs*. Table 1 shows an example of the information we store.

At this point, the program analyzer also calculates an API call density for each method it analyzes. We define *API call density* as the fraction of statements containing API calls over the total number of statements in the method as shown in (1) as follows:

$$APICallDensity(M) = \frac{|StatementsContainingAPICalls(M)|}{|Statements(M)|} \quad (1)$$

The API call density value of a method indicates the concentration of statements containing API calls with respect to other statements in a method. For example, in Listing 1, the method contains 8 statements out of which 5 contain API calls. This results in an API call density score of 0.6. If there are no API calls in a method then its API call density score

```

protected void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_video);

    //for play video by net

    //id
    VideoView videoView = findViewById(R.id.videoView);
    //set url
    videoView.setVideoURI(Uri.parse(LINK));
    // Open lib MediaController for stop and play and set time play
    MediaController controller = new MediaController(this);

    //set controllerto video view
    controller.setAnchorView(videoView);
    //set videoView to controller
    videoView.setMediaController(controller);

    //start this put
    videoView.start();
}

```

Listing 1 Example of extracted API calls (underlined) from a given method

will be 0 and if each statement in the method body contains one or more API calls, then its API call density will be 1.

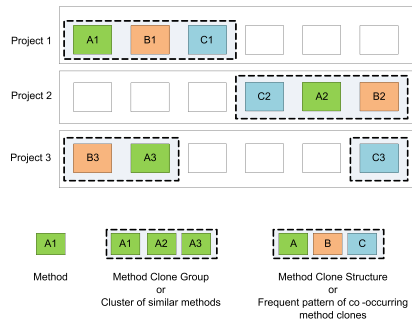
4.1.3 Mining API Usage-based Method Clone Structures

To find related methods that implement related features, FACER's high-level idea is to find similar methods based on their API usages and cluster them together, where a cluster represents a particular feature. Then, we can find commonly co-occurring method clusters, where commonly co-occurring method clusters represent related features since they frequently appear together.

We use the term *Method Clone Group* to refer to such a method cluster. Traditionally, a clone group is a set of code snippets in which token sequence similarity exists between any pair of code snippets (Kamiya et al. 2002). Given our purposes, we specifically look at similar API usages to identify members of a clone group. Thus, we define a *Method Clone Group* (or *clone group* for short) as a set of methods in which API usage similarity exists between any pair of methods. These methods may implement the same feature or functionality and could be instances of a particular feature. For high-level illustration, Fig. 4a shows methods found across three projects where methods of the same color have similar API usages and thus are functionally similar. Thus, methods A_1 , A_2 , and A_3 belong to the same clone group A.

Table 1 Assigning API Call IDs to methods

	Method ID	API name	API method	API Call ID
	6	VideoView	setVideoURI	31
	6	Uri	parse	11
	6	MediaController	new	32
	6	MediaController	setAnchorView	12
	6	VideoView	setMediaController	13
Example based on code shown in Listing 1	6	VideoView	start	33



(a) Abstract Method Clone Structure across projects

```

public void startMeasureResult() {
    btAdapter = BluetoothAdapter.getDefaultAdapter();
    BluetoothDevice device =
        btAdapter.getRemoteDevice(address);
    try {
        btSocket = createBluetoothSocket(device);
    } catch (IOException e) {
        errorExit("Fatal Error",
            "In onMessage() and socket create failed: " +
            e.getMessage() +
            ".");
    }
    btAdapter.cancelDiscovery();
    Log.d("bluetooth", "...Connecting...");
    try {
        btSocket.connect();
    } catch (IOException e) {
        btSocket.close();
    } catch (IOException e2) {
        errorExit("Fatal Error",
            "In onMessage() and unable to close " +
            "socket ..." +
            e2.getMessage() +
            ".");
    }
    Log.d("bluetooth", "...Create Socket...");
    mConnectedThread = new ConnectedThread(btSocket);
    mConnectedThread.start();
}
    
```

(b) Method A1 from Project 1

```

public void run() {
    Log.i(TAG, "BEGIN mConnectThread SocketType:"
        + mSocketType);
    setName("ConnectThread" + mSocketType);
    // Always cancel discovery because
    // it will slow down a connection
    mAdapter.cancelDiscovery();
    // Make a connection to the BluetoothSocket
    try {
        // This is a blocking call and will
        // only return on a successful
        // connection or an exception
        mMSocket.connect();
    } catch (IOException e) {
        // Close the socket
        try {
            mMSocket.close();
        } catch (IOException e2) {
            Log.e(TAG, "unable to close() " +
                mSocketType +
                " socket during connection failure", e2);
        }
        connectionFailed();
        return;
    }
    // Reset the ConnectThread because we're done
    synchronized (BluetoothChatService.this) {
        mConnectThread = null;
    }
    connected(mMSocket, mMDevice, mSocketType);
}
    
```

(c) Method A2 from Project 2

```

public ConnectedThread(BluetoothSocket socket) {
    InputStream tmpIn = null;
    OutputStream tmpOut = null;
    // Get the BluetoothSocket input and output streams
    try {
        tmpIn = socket.getInputStream();
        tmpOut = socket.getOutputStream();
    } catch (IOException e) { }
    mMInStream = tmpIn;
    mMOutStream = tmpOut;
}
    
```

(d) Method B1 from Project 1

```

public ConnectedThread(BluetoothSocket socket,
    String socketType) {
    Log.d(TAG, "create ConnectedThread: " +
        socketType);
    mMSocket = socket;
    InputStream tmpIn = null;
    OutputStream tmpOut = null;
    // Get the BluetoothSocket input and
    // output streams
    try {
        tmpIn = socket.getInputStream();
        tmpOut = socket.getOutputStream();
    } catch (IOException e) {
        Log.e(TAG, "temp sockets not created", e);
    }
    mMInStream = tmpIn;
    mMOutStream = tmpOut;
    mState = STATE_CONNECTED;
}
    
```

(e) Method B2 from Project 2

Fig. 4 A real example of a API Usage-based Method Clone Structure taken from Bluetooth chat projects. Highlighting shows common API usages

Since our goal is to find related functionality, we want to find clone groups that frequently occur together. For example, if methods that implement a *connect to Bluetooth* functionality often occur with methods that implement a *send file over Bluetooth* functionality, then we know that these two functionalities are related. Accordingly, we use the term *Method Clone Structure (MCS)* (Kanwal et al. 2019) to refer to a set of methods that are frequently cloned together across different projects. In other words, a recurring pattern of method clones is a Method Clone Structure. The participating methods in a method clone structure all relate

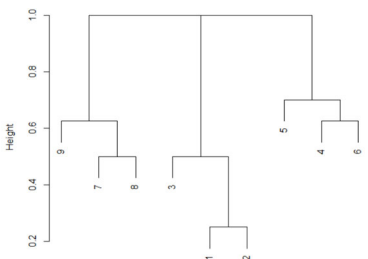
to each other. Since our method clones are based on API usages, we call these structures *API usage-based Method Clone Structure*. In Fig. 4a, clone groups A, B and C together form a frequent pattern across the three analyzed projects. Hence, they form a method clone structure. Based on the heuristic of frequent co-occurrence, members of the clone structure are all related to each other. Figure 4a and b show some of the corresponding methods taken from a real clone structure, which we mine from projects implementing Bluetooth chat functionality. The green highlighting represents clone group A and the pink highlighting represents the clone group B. The highlighted API usages are the basis of similarity between members of a clone group.

Figure 3b shows the two steps we take to mine clone structures. We now explain these steps in detail.

Step 1: Cluster methods by API usage similarity In this step, we group all the methods in the repository into clusters on the basis of similar API usages and API call densities between them. We use Figure 5 to explain this process. As explained in Section 4.1.2, we already record unique API call IDs for all API usages that we analyze across all projects. Assume that our repository consists of the nine methods listed in the table in Fig. 5a. The sequence of numbers shown in the second column represents the IDs of the API calls that appear in each method. For the sake of simplicity, we only demonstrate the effect of clustering on the basis of API calls similarity without considering the effect of API call density. So, we assume that all methods have an API call density equal to 1.

Method ID	API Call IDs
1	1 2 3 4
2	1 2 3
3	7 8 1 2 3
4	11 12 13 24 25
5	26 27 11 28 12 29 13
6	31 11 32 12 13 33
7	8 35 9 10
8	8 9 10 15 16
9	41 42 8 43 9 10

(a) Example methods and API Call IDs



(b) Dendrogram obtained by clustering methods 1 - 9

Method ID	Clone Group ID
1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	3

(c) Resulting clone group for each method

Fig. 5 Step 1: Cluster methods by API usage similarity. After this step, each method in our repository has a clone group ID

Distance matrix computation We define the similarity of two methods based on the intuition that if two methods share a high percentage of API calls (represented by their corresponding set of API call IDs), then they perform the same functionality and implement the same feature. We also factor in the API call density similarity to favor the methods with similar and high API call density to be clustered together. Otherwise, a method that contains a single statement with API call $f_{\circ\circ}()$ might be clustered together with a method that contains 20 statements, only one of which contains the same API call $f_{\circ\circ}()$.

Let $M_1 = \{u_1, u_2, \dots, u_n\}$ and $M_2 = \{u_1, u_2, \dots, u_m\}$ be the sets of API call IDs of two methods M_1 and M_2 . We compute the API usage similarity of two methods M_1 and M_2 using the Jaccard index (Jaccard 1901) as follows:

$$api_sim(M_1, M_2) = \frac{|M_1 \cap M_2|}{|M_1 \cup M_2|} \quad (2)$$

The similarity score has a value between 0 and 1, where 0 means completely dissimilar and 1 means completely similar. Let d_1 and d_2 be the API call densities (calculated using (1)) of methods M_1 and M_2 respectively. We define the API call density similarity of the two methods as follows:

$$density_sim(M_1, M_2) = \frac{d_1 + d_2 + (1 - |d_1 - d_2|)}{3} \quad (3)$$

In (3), $1 - |d_1 - d_2|$ indicates the similarity of density values between the two methods. We factor in individual density values of methods together with density similarity, because we want to give a higher score to two methods with similar high density than to two methods with similar low-density values. The final similarity score is calculated as follows:

$$Sim(M_1, M_2) = api_sim(M_1, M_2) \times density_sim(M_1, M_2) \quad (4)$$

In preparation for clustering, we calculate the distance between the two methods as follows and store all pairwise method distances in a distance matrix:

$$Dist(M_1, M_2) = 1 - Sim(M_1, M_2) \quad (5)$$

Cluster Identification To identify method clusters, we pass the calculated distance matrix as input to the standard average linkage hierarchical clustering algorithm (Defays 1977). This algorithm performs a hierarchical cluster analysis using a set of dissimilarities for the n methods being clustered. Initially, the algorithm assigns each method to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. The result is a tree-based representation of the methods being clustered which is called a dendrogram (Heirarchical clustering 2019). Figure 5b shows the dendrogram obtained after clustering the nine methods from Fig. 5a. Each leaf of the dendrogram corresponds to one method. As we move up the tree, methods that are similar to each other get linked into branches, which are themselves fused at a higher height. The height of the fusion, provided on the vertical axis, indicates the dissimilarity between two methods. In order to identify sub-groups (i.e. clusters), we can cut the dendrogram at a certain height. A cut is a demarcation line at a certain height of a dendrogram which results in the intersection of dendrogram branches with the cut. All nodes of the branch that intersects the cut end up in one cluster. The branches and nodes above the cut form independent clusters. Choosing the optimal cut-point on a dendrogram is an NP-complete problem. In our case, we experiment with a number of cut-point values at different heights corresponding to a similarity threshold α . The height of the cut to the dendrogram controls the similarity threshold, and thus the number of clusters obtained. The greater the height of the cut, the looser the similarity threshold and the fewer the number of clusters

formed. The smaller the height of the cut, the stricter the similarity threshold and the greater the number of clusters formed. If we specify a height of 0.7, this means that the final clusters at that height would have at least $1 - 0.7 = 0.3$ similarity score between their members. Methods joined at height 0 are exactly similar. We obtain a vector containing the clone group ID (i.e., cluster ID) of each method after cutting at a certain height and store this information in our repository. In our evaluation in Section 7.1, we evaluate the effect of varying similarity thresholds by obtaining clusters against various values of height $=\{0.1, 0.3, 0.5, 0.7\}$, corresponding to similarity thresholds $\alpha = \{0.9, 0.7, 0.5, 0.3\}$ respectively.

The results of clustering the nine methods from our example are shown in the table of Fig. 5c. The clone group IDs are obtained using a similarity threshold of 0.3, which implies a height of 0.7. We can see that the first three methods are assigned to the first cluster, methods 4, 5 and 6 are assigned to the second cluster, and the last three methods are assigned to the third cluster. The output of Step 1 is now a mapping of method IDs against the unique clone group IDs of each cluster.

Step 2: Mining frequent patterns of method clones across projects The idea of frequent association pattern mining is to find recurring sets of items among transactions. The concept of transactions originates from sales transactions where one or more items are purchased in a single sales transaction. In our context, items are clone groups of a project that make up a transaction in the FACER repository R. We are interested in mining recurrent patterns of clone groups. The strength of a frequent pattern is measured by a support count. Support count is the number of transactions in R containing a unique pattern of clone groups. A frequent association pattern describes a set of items that has support greater than a predetermined threshold called a minimum support threshold which we identify as β .

We have so far identified the clone group IDs of all methods in our repository. To mine API usage-based method clone structures across projects, we first create a transaction table where each row of the table contains all the clone group IDs assigned to methods of a project. Assume that we have a repository of five projects with the transaction table shown in Fig. 6a.

Given this table, we perform frequent item set mining to get frequently co-occurring clone groups that repeat across projects. Such repeating item sets represent the API usage-based Method Clone Structures. We can say that for a given clone structure and its constituent clone groups, the methods mapped to those clone groups are all related to each other.

This is based on the market basket intuition (Han et al. 2011). Market basket analysis attempts to identify associations, or patterns, between the various items that have been chosen by a particular shopper and placed in their basket (Market-basket analysis 2019). Items that frequently co-occur are related to each other. In our context, a particular project may use a group of methods which may be cloned across other projects. Such a pattern of co-

Project ID	Clone Group IDs
1	1 2 3 11 19
2	1 9 2 4 3
3	5 6 15 18 19
4	21 5 22 6
5	26 1 2 3

(a) Example clone group IDs recorded for each project

Clone Structure ID	Clone Group IDs	Support
C1	5 6	2
C2	1 2 3	3

(b) Resulting Method Clone Structures across projects

Fig. 6 Step2: Mining frequent patterns of method clones across projects

occurring method clones identifies related functionality and forms the basis of suggesting relevant methods.

We use the frequent closed itemsets mining algorithm *FPClose* (Grahne and Zhu 2005; FPClose 2019) to get frequent items. *FPClose* is an algorithm of the FPGrowth family of algorithms, designed for mining frequent closed itemsets and is claimed to be one of the fastest closed itemset mining algorithm. The input to the algorithm is a transaction table and a support/frequency threshold β . We evaluate the sensitivity of recommendation results against varying thresholds of $\beta=(3, 5, 10, 15)$ in Section 7.1. The result of the execution of FP mining on our example transaction table with $\beta = 2$ is shown in the right table in Fig. 6b. The clone structure C1 indicates that clone groups 5 and 6 are frequently found together. Similarly, C2 indicates that the clone groups 1, 2 and 3 are frequently found together. This provides the basis of FACER's recommendation which we explain next in Section 4.2.

To summarize, Algorithm 1 shows all the steps discussed above which are involved in mining Method Clone Structures (MCS) in the FACER repository R given a similarity threshold α and a minimum support threshold β . First, we obtain API calls and API call densities of all methods in the FACER repository R (Lines 4-7). Then, we obtain pairwise similarities for all methods (Lines 8-10). The distance matrix is obtained from the similarity matrix and used by the clustering algorithm to detect and label clusters with respect to α (Lines 11-13). The resulting clusters are saved as clone groups in FACER (Line 14). Next, in order to mine frequently co-occurring features across all projects, we create a transaction table where each row contains clone group IDs for a project in the FACER repository R (Lines 16-19). The resulting table is used to perform frequent item set mining with respect to a certain threshold β to obtain frequently co-occurring sets of clone groups which are then saved as Method Clone Structures in the repository R (Lines 20-21).

Algorithm 1 Mining API usage-based method clone structures.

```

1: procedure MINEMCS( $R, \alpha, \beta$ )
2:    $APICallsMap \leftarrow \{\}$ 
3:    $APICallDensityMap \leftarrow \{\}$ 
4:   for all  $m \in methods(R)$  do
5:      $APICallsMap.add(m.ID, m.APICalls)$ 
6:      $APICallDensityMap.add(m.ID, m.APICallDensity)$ 
7:   end for
8:    $simMatrix1 \leftarrow getPairwiseJaccardSim(APICallsMap)$ 
9:    $simMatrix2 \leftarrow getPairwiseDensityBasedSim(APICallDensityMap)$ 
10:   $simMatrix3 \leftarrow simMatrix1 \times simMatrix2$ 
11:   $distMatrix \leftarrow 1 - simMatrix3$ 
12:   $dendrogram \leftarrow cluster(distMatrix)$ 
13:   $methodClusterIDMap \leftarrow getClusters(dendrogram, \alpha)$ 
14:   $saveCloneGroups(methodClusterIDMap, R)$ 
15:   $transactionTable \leftarrow \phi$ 
16:  for  $p \in projects(R)$  do
17:     $transaction \leftarrow getCloneGroupIDs(p)$ 
18:     $transactionTable.add(transaction)$ 
19:  end for
20:   $CloneStructures \leftarrow getFreqPatterns(transactionTable, \beta)$ 
21:   $save(CloneStructures, R)$ 
22: end procedure

```

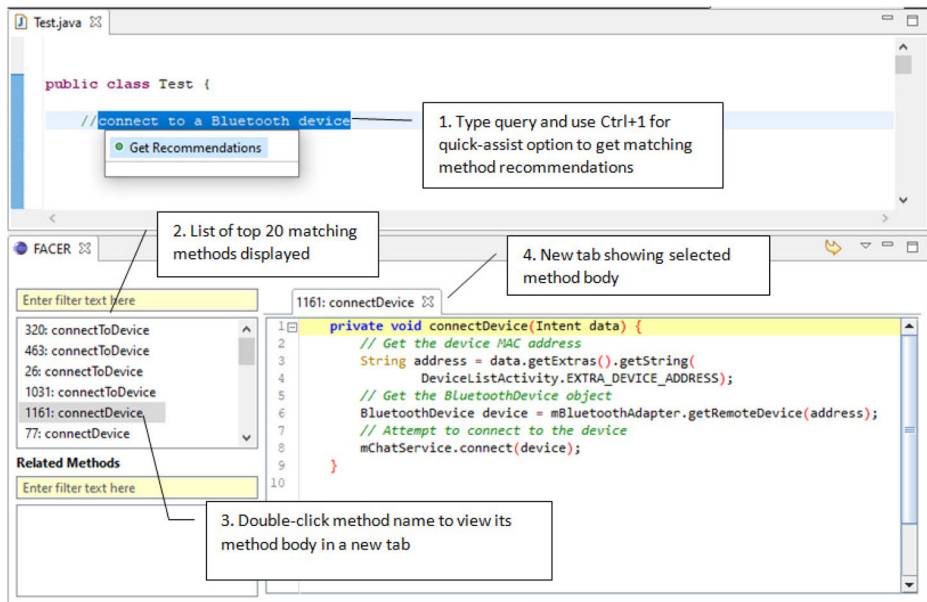


Fig. 7 Stage 1: Method Search

4.2 Online FACER Recommendation Workflow

We implement FACER as an Eclipse IDE plugin. The “online” workflow is what developers experience when they interact with FACER. This interaction process is comprised of two stages. Stage 1 performs retrieval against a user’s feature query to provide a ranked list of top methods that implement the requested feature. Upon selection of a method by the user, Stage 2 recommends related methods for opportunistic reuse. The FACER system components for online recommendation workflow are shown in Fig. 2a and are discussed below.

4.2.1 FACER Stage 1: Method Search

This module gets triggered whenever the user types a comment and presses the *CTRL+I* key combination afterwards. The comment should describe the feature they wish to implement. We show an example in Fig. 7 where the developer types the feature query “Connect to a Bluetooth device”, uses *Ctrl+I* and selects “Get Recommendations” from a quick-assist popup. FACER then processes the input comment (query string) and initiates a search to retrieve top-N matching methods from the FACER repository. Note that we are not contributing a novel code search engine. Any code search technique (Luan et al. 2018; Sachdev et al. 2018; Gu et al. 2018; Umarji et al. 2008; Chatterjee et al. 2009; Bajracharya et al. 2010; Keivanloo et al. 2014) can be used here. However, to implement the whole workflow, we develop a simple code search engine (B1 (Bajracharya et al. 2010)) using Lucene (lucene 2017). The performance of such a search engine has been shown to significantly improve software retrieval performance, increasing the area under the curve (AUC) retrieval metric to 0.92 – roughly 10–30% better than previous approaches based on text alone (Linstead et al. 2009). Lucene uses the BM25 (Best Matching) textual similarity ranking method implemented in Okapi (Stephen and et al 1995). The output of this FACER search stage is

a ranked list of top matching methods against the input comment. We currently show the developer the top 20 matching methods. The bottom left of Fig. 7 shows the list of methods retrieved against the example query. Developers can click on any of these methods to view their content in the right pane. Once decided, they can get related method recommendations against the currently selected method by clicking the arrow button on the top right corner of FACER's view panel as shown in Fig. 8.

4.2.2 FACER Stage 2: Related Method Recommendations

To obtain a list of related method recommendations, we use the user-selected method (m_u) from the previous step along with a minimum support threshold (β) as input. The clone structures of co-occurring API usage-based Method Clone Groups mined from Step 2 (Section 4.1.3) are the basis for FACER's recommendations.

Algorithm 2 summarizes the steps for recommending related methods against an input method (m_u). We first identify which clone group (i.e., cluster) m_u belongs to (Line 5), and use it to get related method recommendations (Line 5). The procedure GETRELATEDMETHODS for getting recommendations against a clone group ID is shown on Line 14.

In this procedure, FACER retrieves only those Method Clone Structures (MCS) that satisfy the threshold β and performs highest-support-first ordering of the MCS (Line 15). After obtaining a list of MCS, we gather all distinct clone groups found in each MCS as co-occurring features against our input feature (Line 17). Next, for each of the clone groups, we get representative methods and add those methods to the list of recommended related methods (Lines 19–23). To select a representative method from each clone group, we follow a simple rule: if a clone group contains a method that belongs to the same project as the user's already selected m_u , then we choose that method as the representative method. Otherwise, we choose the method with the highest API call density within the clone group. This is to ensure that the recommended method has the least amount of noise in the form of statements without API calls.

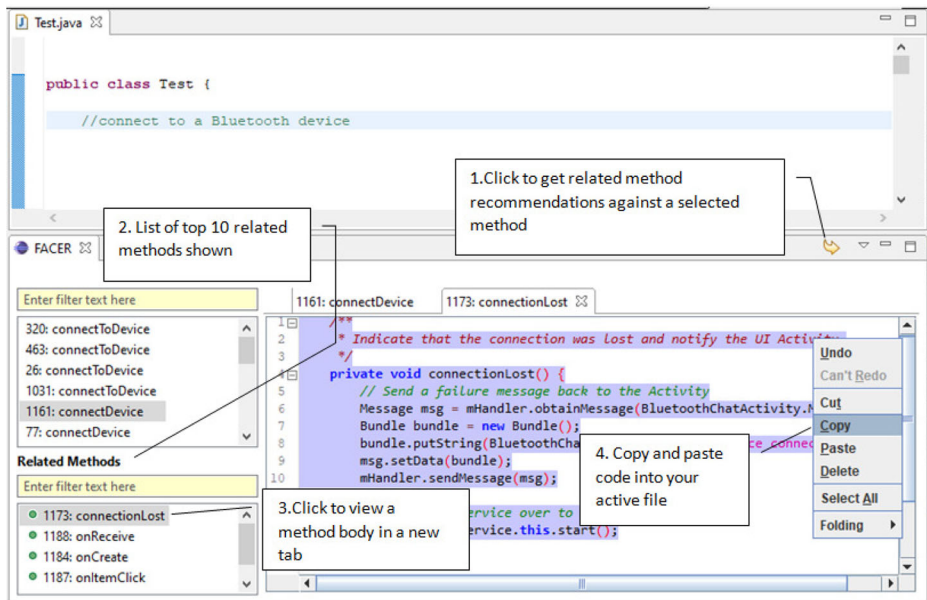


Fig. 8 Stage 2: Related Method Recommendations

Algorithm 2 FACER stage 2: getting related method recommendations.

```

1: procedure MAIN( $m_u, \beta$ )
2:    $relatedMethods \leftarrow \{\}$ 
3:    $cloneGroupID \leftarrow getCloneGroupID(m_u.ID)$ 
4:   if  $cloneGroupID \neq \emptyset$  then
5:      $relatedMethods \leftarrow GETRELATEDMETHODS(cloneGroupID, \beta)$ 
6:     if  $relatedMethods = \emptyset$  then
7:        $relatedMethods \leftarrow GETFROMNEIGHBORHOOD(m_u.ID, cloneGroupID, \beta)$ 
8:     end if
9:   else
10:     $relatedMethods \leftarrow GETFROMNEIGHBORHOOD(m_u.ID, cloneGroupID, \beta)$ 
11:  end if
12:  return  $relatedMethods$ 
13: end procedure

14: procedure GETRELATEDMETHODS( $cloneGroupID, \beta$ )
15:    $cloneStructsList \leftarrow getMCS(cloneGroupID, \beta)$  ▷ highest support first
16:   if  $cloneStructsList \neq \emptyset$  then
17:      $cloneGroupsList \leftarrow getUniqueCloneGroups(cloneStructsList)$ 
18:     for all  $CID \in cloneGroupsList$  do
19:        $m \leftarrow getRepresentativeMethod(CID)$ 
20:        $relatedMethods.add(m)$ 
21:     end for
22:   end if
23:   return  $relatedMethods$ 
24: end procedure

25: procedure GETFROMNEIGHBORHOOD( $m.ID, cloneGroupID, \beta$ )
26:    $neighborSource \leftarrow CallGraph(m.ID)$ 
27:    $relatedMethods \leftarrow NEIGHBORHOODRETRIEVAL(neighborSource, m.ID,$ 
28:      $cloneGroupID, \beta)$ 
29:   if  $relatedMethods = \emptyset$  then
30:      $neighborSource \leftarrow HostFile(m.ID)$ 
31:      $relatedMethods \leftarrow NEIGHBORHOODRETRIEVAL(neighborSource, m.ID,$ 
32:        $cloneGroupID, \beta)$ 
33:   end if
34:   return  $relatedMethods$ 
35: end procedure

36: procedure NEIGHBORHOODRETRIEVAL( $neighborSource, m.ID, cloneGroupID, \beta$ )
37:    $NeighborCloneGroupsList \leftarrow getCloneGroups(neighborSource, m.ID)$ 
38:   for  $CID \in NeighborCloneGroupsList$  do
39:      $MCS\_Support\_Map.add(getMCS(CID, \beta))$ 
40:   end for
41:   if  $MCS\_Support\_Map \neq \emptyset$  then
42:      $sortedMCS\_Support\_Map \leftarrow sortMCSByDecreasingSupport$ 
43:      $(MCS\_Support\_Map)$ 
44:     for all  $cloneStructs \in sortedMCS\_Support\_Map$  do
45:        $cloneGroupsList \leftarrow getUniqueCloneGroups(cloneStructs)$ 
46:       for all  $CID \in cloneGroupsList$  do
47:          $m \leftarrow getRepresentativeMethod(CID)$ 
48:          $relatedMethods.add(m)$ 
49:       end for
50:     end for
51:   end if
52:   return  $relatedMethods$ 
53: end procedure

```

In case m_u 's clone group does not belong to any clone structure (Lines 6-8) or if m_u does not belong to a clone group (Lines 9-11), we scan the neighboring methods of m_u to perform neighborhood-based retrieval. The procedure for getting recommendations based on neighboring methods is shown on Line 25. We first use the call graph of m_u as the source of neighboring methods to obtain recommendations (Lines 26-27). Specifically, we use the caller and callee methods of m_u as its neighborhood and thus input methods. If this returns an empty set of related methods, we then use the host file of m_u as the source of neighboring methods to obtain recommendations (Lines 29-30). In this case, all the methods of the host file containing m_u form its neighborhood and are used as input.

Line 34 shows the procedure to perform neighborhood retrieval, regardless of the neighborhood source used. It involves getting the clone groups of all methods in the neighborhood of m_u . For each of these clone groups, we get all the MCS in which they occur (Lines 36-38). We then sort these MCS in order of highest-support-first and build a list of distinct clone groups that occur in those MCS (Lines 40-42). Finally, we get representative methods against the clone groups as before (Lines 43-45) and return the list of related method recommendations for m_u .

5 Research Questions and Experimental Setup

The ultimate goal of our related feature recommendation system is to support opportunistic reuse through recommending relevant related methods. Our approach is designed to minimize the number of irrelevant related features recommended, while maximizing the success rate of obtaining code examples for relevant features. We now discuss our research questions and the evaluation setup we use.

5.1 Research Questions

We aim to answer the following four research questions (RQs):

- **RQ 1:** Are the clone groups detected by FACER valid?
- **RQ 2:** How precise is FACER in terms of recommending related features?
- **RQ 3:** Do developers need to search for related features?
- **RQ 4:** What are developers' perceptions regarding the usefulness and usability of FACER?

The underlying premise of FACER is that methods with similar API usages are semantically related and can represent methods implementing the same feature. If this is not true in practice or if our clone groups are meaningless, then the rest of FACER's workflow will not be useful. Thus, in RQ1 (Section 6), we manually validate a sample of the clone groups detected by FACER. The goal is to make sure that methods belonging to the same clone group implement the same functionality and that different clone groups represent different functionality. The aim of RQ2 is to evaluate whether the methods recommended in FACER's Stage 2 actually implement features that relate to the user's selected feature/method. To evaluate this, we perform two types of evaluation in Section 7. The first is an automated evaluation that compares the recommendations against ground truth data to determine the best threshold values and the second is a manual evaluation that involves human validation. The aim of RQ3 is to understand the code search and reuse practices of developers and find out whether they need to search for related features. In RQ4, we determine how developers perceive the usability and usefulness of the current FACER tool and its recommendations.

To answer RQ3 and RQ4, we conduct a user survey (Section 8) which includes assessing developers’ code search and reuse practices, presenting the developers with recommendation scenarios for reviewing FACER’s related method recommendations and then getting their feedback on the FACER tool’s interface and recommendations.

5.2 Dataset

We collect applications from four different categories of Java-based Android applications: (1) music player, (2) Bluetooth chat, (3) weather, and (4) file management (FACER Artifacts 2020). We choose these categories because of their use in previous research on feature recommendations (Dumitru et al. 2011) and API usage pattern recommendations (Niu et al. 2017). We include 30 applications from each category, resulting in a total of 120 applications. We intentionally choose multiple applications from each category to allow the discovery of frequently co-occurring features across similar category applications.

To collect the applications forming the dataset, we use GitHub’s search where we use each category name prefixed with *android* and postfixed with *app* as search queries. Then, we filter the search results by choosing Java as the language and sort them using *relevance* option. We then select the top 30 relevant GitHub repositories against each search query. We manually judge the relevance to a category by analyzing the description of each application on GitHub. If an application is not deemed relevant, we skip it. Figure 9 shows the distribution of star ratings for the selected applications across the four categories. We can see that the weather category has the highest starred applications, followed by music, file manager and Bluetooth categories.

Overall, our dataset for the evaluation consists of 120 Java-based Android applications which we analyze in order to populate the FACER repository.

5.3 Constructing the FACER Repository

To populate the FACER repository in offline mode, we analyze the source code of the collected 120 applications. The time to execute the program analyzer on this dataset is almost

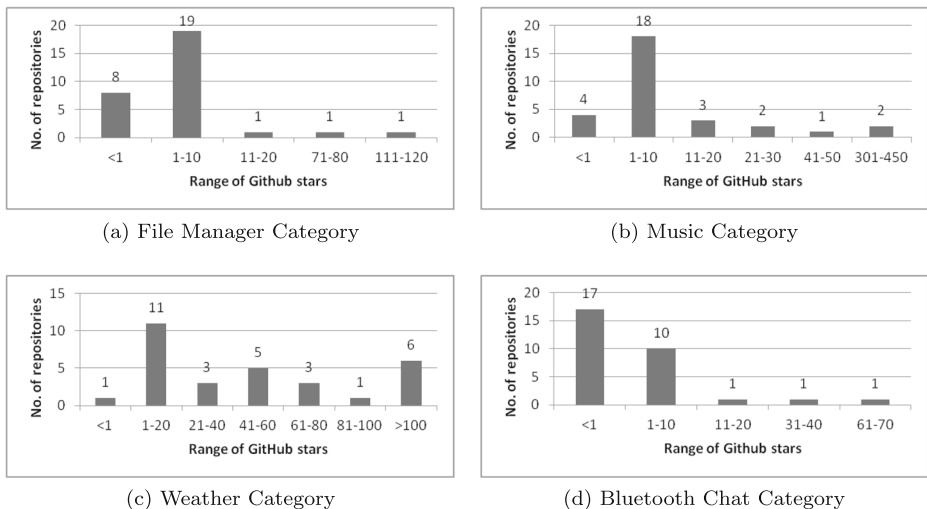


Fig. 9 The number of GitHub repositories from the four categories across different ranges of the number of stars

55 minutes on a Core i7 2.2 GHz machine with 8GB memory running Windows 10. Table 2 summarizes some of the key statistics of the FACER repository that we built from the 120 applications, and which we use to answer our research questions.

During the detection of clone groups, we consider only methods having a minimum of three unique API calls to ensure that we have meaningful clusters (Han et al. 2012; Yun and Leggett 2005). We also ignore API calls involving the usage of *Log*, *Intent* and *Toast* API classes, because want to filter out common API calls which appear in almost every application and do not contribute towards a particular feature of an application. Thus, out of the 37,303 methods in the repository, we mine clusters from 7,922 methods. Overall, these 7,922 methods have 7,028 unique API calls.

We input a 7922×7028 binary matrix whose rows represent methods and columns represent all unique API calls found across all the methods. A value of 1 in the matrix means that the API call exists in the method. One of the challenges of clustering methods on the basis of API calls is the storage and computation required to process large matrix sizes when the number of methods and the number of API calls increase. To efficiently calculate pair-wise API usage similarity between methods, we make use of a third-party function (binaryDist 2020; Sparse matrix clustering 2019) that performs rapid calculation of the Jaccard distance of a matrix by making use of raw vectors with the binary data packed efficiently. For calculating pair-wise API call density-based distances between methods, we use another third-party library function (parallelDist 2020) to perform distance matrix computation in parallel using multiple threads. It supports predefined distance measures and user-defined distance functions. For our purpose, we specify our own distance function based on our similarity formula shown in (3). Table 3 shows the number of clone groups and Method Clone Structures that we obtain as a result of clustering and frequent pattern mining under various similarity thresholds. Increasing the threshold results in fewer clone groups and Method Clone Structures because of the stricter clustering criteria.

6 RQ1: Method Clone Group Evaluation

In RQ1, we evaluate the Method Clone Groups detected by FACER to determine whether the methods that FACER clusters into the same clone group actually implement the same functionality/feature. This is *intra-clone group similarity validation*. We also evaluate *inter-clone group dissimilarity* to verify that the Method Clone Groups do not share any functionality with each other.

Table 2 FACER code fact repository statistics

Metric	Value
No. of applications	120
No. of files	4,369
Lines of comments	175,000
Lines of code (LOC)	498,261
No. of methods	37,303
No. of method calls	150,341
No. of API classes	2,209
No. of unique API calls	7,607
Total no. of API calls	85,386

Table 3 Method Clone Groups (MCG) and Method Clone Structures (MCS) detected with varying similarity threshold α

α	No. of MCG	No. of MCS
0.3	1445	536
0.5	1397	107
0.7	812	37
0.9	347	11

6.1 Validation Method

We manually evaluate the clone groups which FACER detects with a minimum similarity score threshold $\alpha = 0.5$. This relieves us from evaluating clone groups obtained with larger threshold values of α since they will always be better due to a higher similarity between clone group members. We also observe from our automated evaluation in Section 7.1.2 that this alpha value gives us the optimal precision and success rate. We use this same alpha value for all our evaluations in this paper. The authors of this paper as well as one professional senior Android developer conduct the manual validation. We first explain how we select the clone groups that we evaluate and then explain the manual validation process we follow for inter and intra clone group validation.

6.1.1 Clone Group Sampling

Since manually evaluating all 1,397 clone groups where each clone group has several methods is not practically feasible, we perform multi-stage sampling to select the clone group and methods for our evaluation.

We first need to select clone groups to evaluate. We want to make sure we manually validate a diverse set of clone groups. Thus, we take into account the following clone-group characteristics during our sampling:

- *size*: We calculate the size of a clone group as the number of methods in a clone group. The higher the number of methods in a clone group, the more common the feature represented by the clone group is. Sampling by size allows us to choose from a spectrum of less common features as well as widespread features. Figure 10 shows that the *size* of clone groups in our data set varies from 2 to 52. We observe that, not surprisingly, there are a larger number of small-sized clone groups when compared to larger clone groups. This observation is also reported in previous code clone detection studies where small clone groups are overwhelmingly the most common of all clone groups (Venkatasubramanyam et al. 2013; Svajlenko et al. 2013).
- *API call size diversity*: Each clone group can have methods with a varying number of unique API calls. We calculate the *API call size diversity score* of a clone group as the difference between the minimum and the maximum number of unique API calls found across all methods of a clone group. For example, if a clone group of size 2 has a method with API calls A, A, B, and C and the other method has API calls A, B, C, D, E, and F, then the diversity score of this clone group will be $6-3=3$. A higher diversity value is a proxy for more diverse functionality in the clone group. Sampling clone groups by diversity enables us to sample methods having various API call sizes in the next sampling stage.

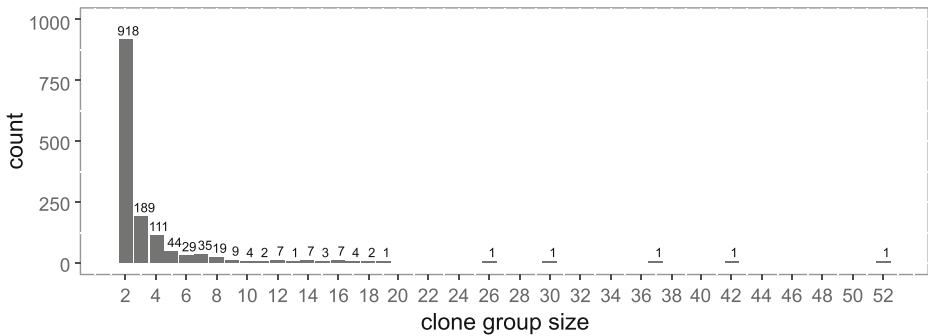
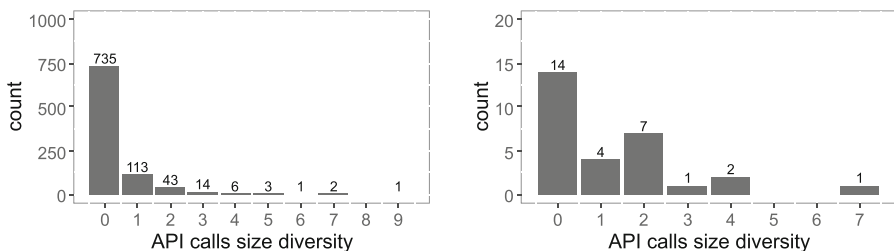


Fig. 10 Frequencies of clone groups of varying sizes with similarity threshold $\alpha = 0.5$

Given the above two criteria, we select clone groups using a two-stage sampling. In the first stage, we perform systematic cluster sampling to select clone group sizes. For the number of different clone group sizes n , we systematically select every alternate clone group size which results in selecting 50% of the available sizes. In the second sampling stage, we select clone groups from low, median and high API call size diversity strata found within each sampled clone group size. For example, from Fig. 11a we can see that for clone groups of size 2, the API call size diversity values range from 0 to 9 on the x-axis. The median of these values is 4, and we form strata using the median value as a reference. Values close to the median value (3 and 5) fall in the *median stratum*, whereas values (0, 1, 2) lower than those of the median stratum values fall in the *low stratum* and values (6, 7, 9) higher than median stratum values fall in the *high stratum*. Having determined the strata, we now randomly select a diversity value from each stratum. From Fig. 11a, we choose the API diversity values 0, 3, and 5. We then continue randomly selecting one clone group from the sampled diversity values until we sample at least 10% of the total number of clone groups of a particular clone group size. Table 4 shows the results of our sampling criteria until this step. This sample size of 126 clone groups gives us an 8% margin of error at a 95% level of confidence.

6.1.2 Method Sampling

Note that we have so far identified clone groups to evaluate but not the particular methods that we will manually validate from those clone groups. Thus, we now discuss how we identify the particular methods for validation.



(a) Frequency of API call size diversity for clone groups of size 2 (b) Frequency of API call size diversity for clone groups of size 6

Fig. 11 Example API call size diversity for clone groups of size 2 and 6

Table 4 Two-stage sampling of 126 clone groups from a total of 1,397 available clone groups

Sampled clone group sizes (Stage 1 sampling)	No. of clone groups in each size	No. of sampled clone groups of each size (Stage 2 sampling)
2	918	92
4	111	12
6	29	3
8	19	3
10	4	3
12	7	3
14	7	3
16	7	3
18	2	1
26	1	1
37	1	1
52	1	1

For sampled clone groups of size 2 which only contain two methods, we include both methods in our sample. However, it is a big manual overhead to manually validate each method for large-sized clone groups. Thus, we sample representative methods from the selected clone groups for manual validation. To sample these methods, we take into account the following method characteristics:

- *API-call size*: We select methods with the smallest, median and largest number of unique API calls within a sampled clone group. This allows us to sample methods of various API call sizes. Figure 12 shows the distribution of API-call sizes for all methods in our selected sample of 126 clone groups from Table 4. We use this distribution to sample methods on the basis of API-call size. The number of unique API calls ranges from a minimum of 3 (due to our clustering criteria explained in Section 5.3) to a maximum of 46.
- *API-call density*: In addition to selecting methods by API-call size, we select methods with the highest and lowest API-call densities to add more methods to the sample. Note that the previous sampling using API-call size may already contain a high and low density method from each clone group, in which case we do not need to add more methods from this step. Figure 13 shows the distribution of all the methods from our sampled clone groups across different API-call density ranges. We observe that almost 99% of methods in our sampled clone groups have API-call density values greater than 50%. This implies that API calls form a major part of the code of these methods and also supports our technique of clustering methods on the basis of API calls to detect common functionality. We sample methods on the basis of API-call density from this distribution.

Based on the above sampling criteria, our final sample consists of 126 clone groups with a total of 305 methods.

6.1.3 Intra-clone group similarity validation

Setup All four authors of this paper and one professional Android developer perform the manual intra-clone group validation where our goal is to check whether methods of a clone

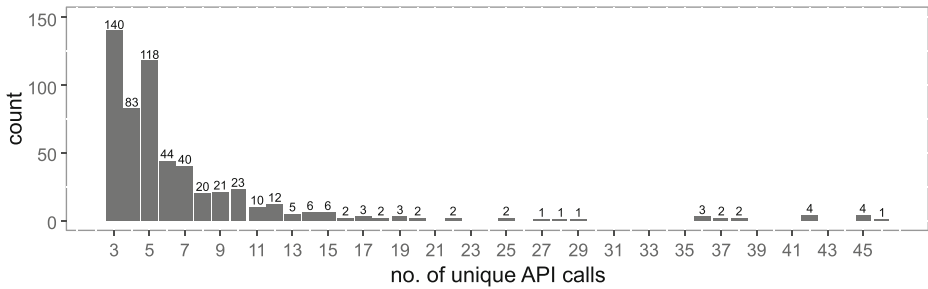


Fig. 12 Distribution of API call size for all the methods from our sampled clone groups in Table 4

group are functionally similar. We first explain the evaluation procedure and then explain how the clone groups were distributed among the evaluators.

We follow the following evaluation procedure for all 126 clone groups. For each method of a clone group, the evaluator writes a feature description describing what the method is doing. This description is based on using the code in the method body (including method invocations and API calls), any Javadoc comments, the method name, and any inline comments. Once the evaluator writes feature descriptions for all methods in the clone group, they then write a feature description for the clone group which represents the functionality shared by all the methods in the clone group. For methods having functionality that does not match with the core functionality of the clone group, this functionality is noted by the evaluator as a *divergent* feature. Finally, the evaluator gives a decision regarding the validity of the clone group. A clone group is *valid* if all of its member methods (or the analyzed sub sample) implement similar functionality. A clone group is *invalid* if the evaluator is able to identify one or more of its member methods having several divergent functionalities. Having too many divergent features in a method indicates that the methods do not implement the same functionality, which affects the validity of the clone group. In other words, the decision to assign a valid label to a clone group is based on the following observations:

- The resulting feature description of a clone group describes the common functionality in the clone group.
- There are no major divergent features inside any member method or the number of divergent features is few or minor in comparison to the clone group’s feature description.

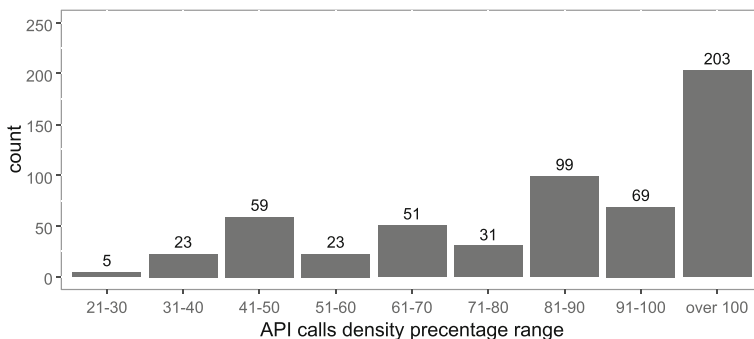


Fig. 13 Method distribution from sampled clone groups based on API call density

We first assign the evaluation of each sampled clone group to two authors such that we get two unique author evaluations per clone group. Each author independently evaluates their assigned clone groups and labels each clone group as *valid* or *invalid*.

To ensure external validation and reduce author subjectivity, we also recruit a professional Android developer to evaluate and label all 126 clone groups. We recruit the professional Android developer using a freelance website Fiverr (Fiverr - freelance services marketplace for businesses 2021). The developer has six years of experience developing Android applications such as online food ordering, shopping, personalization, video players, live streaming, VPN, utility apps, and customized apps for several businesses.

We provide the professional developer with the evaluation data accompanied by instructions about the evaluation procedure we described above. This means that each of the 126 clone groups are evaluated by two authors and one external professional developer. For all agreements between authors, we compare the professional developers' evaluation with the author evaluations to see whether their rating matches ours. We obtain the final labels of all clone groups using a majority vote of the three ratings.

Intra-clone group similarity results We evaluate 126 clone groups containing a total of 305 methods. For the author ratings, we obtain an 84% agreement rate and a Cohen's kappa score (Cohen 1960; Richard Landis and Koch 1977) of 0.38, which indicates a fair agreement. There were a total of 20 disagreements between the author ratings, which we resolve using the majority vote of all three ratings.

There were 106 clone groups for which both authors agreed on the label. We compare these 106 ratings to the corresponding ratings of the professional developer to check whether the authors' perception matches that of an impartial third party. The authors and the professional developer agreed on 97 clone groups being valid and five being invalid. Overall, we find that the authors had a 96% agreement rate with the professional developer and a Cohen's kappa of 0.69, indicating a substantial agreement (Cohen 1960; Richard Landis and Koch 1977). Overall, after resolving all disagreements across the 126 clone groups using majority vote, we confirm that 115/126 (91%) clone groups are valid.

In Fig. 14a-e, we provide examples of two valid clone groups, one with size 10 and one with size 37. We can see that the member methods of each clone group do share common functionality. For the first clone group in Fig. 14a and b, the shared functionality checks for the availability and connectivity of network, and for the second clone group shown in Fig. 14c-e, the shared functionality sends a failure message to some activity and restarts a service. We also present an example of clone groups with longer methods in our online artifact page. Additionally, all the data and labels from this manual evaluation are provided in our online artifact page (FACER Artifacts 2020).

6.1.4 Inter-clone group dissimilarity validation

Setup We also verify whether the clone groups detected by FACER share any functionality with each other. The idea is to look at clone group descriptions and decide whether any two clone groups are semantically similar. Ideally, there should be minimal functionality overlap between clone groups. This evaluation is performed by the first author of this paper and the same professional Android developer who performed the intra-clone group similarity validation.

```

public boolean isNetworkAvailableAndConnected() {
    ConnectivityManager connectivityManager
        = (ConnectivityManager)
            mContext.getSystemService(Context.CONNECTIVITY_SERVICE);

    NetworkInfo networkInfo = connectivityManager.getActiveNetworkInfo();
    return networkInfo != null && networkInfo.isConnected();
}

```

(a) Clone Group 1 Method 1

```

public boolean isNetworkAvailable() {
    ConnectivityManager connectivityManager
        = (ConnectivityManager)
            mContext.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo activeNetworkInfo = connectivityManager.getActiveNetworkInfo();
    return activeNetworkInfo != null && activeNetworkInfo.isConnected();
}

```

(b) Clone Group 1 Method 2

```

/**
 * Indicate that the connection was lost and notify the UI Activity.
 */
private void connectionLost() {
    // Send a failure message back to the Activity
    Message msg = mHandler.obtainMessage(LanlyActivity.MESSAGE_TOAST);
    Bundle bundle = new Bundle();
    bundle.putString(LanlyActivity.TOAST, "?????????");
    msg.setData(bundle);
    mHandler.sendMessage(msg);

    // Start the service over to restart listening mode
    LanlyService.this.start();
}

```

(c) Clone Group 2 Method 1

```

private void connectionLost() {
    Message msg = handler.obtainMessage(MainActivity.MESSAGE_TOAST);
    Bundle bundle = new Bundle();
    bundle.putString("toast", "Conexion perdida con el dispositivo");
    msg.setData(bundle);
    handler.sendMessage(msg);

    // Start the service over to restart listening mode
    ChatController.this.start();
}

```

(d) Clone Group 2 Method 2

```

private void connectionFailed() {
    Message msg = handler.obtainMessage(MainActivity.MESSAGE_TOAST);
    Bundle bundle = new Bundle();
    bundle.putString("toast", "Unable to connect device");
    msg.setData(bundle);
    handler.sendMessage(msg);

    // Start the service over to restart listening mode
    ChatController.this.start();
}

```

(e) Clone Group 2 Method 3

Fig. 14 Examples of evaluated clone groups. **a** and **b** show two methods from a clone group of size = 10. **c-e** show three methods from a clone group of size = 37

Each evaluator uses their own previously written feature descriptions so that it is easier for them to perform the task and because that reflects how they perceive the clone group's functionality. For each evaluator, we first collect the feature descriptions of all 115 clone groups that we resolve as valid in the intra-clone group validation phase. To reduce manual effort and chances of incurring human error while analyzing all $\binom{115}{2} = 6555$ combinations of feature descriptions, we form a subset of the clone group descriptions of each evaluator based on TF-IDF (Sammur and Webb 2010) similarity. After stemming all words in the

descriptions, we calculate pair-wise similarity between all feature descriptions of an evaluator using a TF-IDF similarity score. We filter out the clone group pairs with a similarity score less than or equal to 0.5 and assume that these are dissimilar. The clone group pairs that have a similarity score greater than 0.5 are the ones we need the evaluators to manually validate. We then ask the evaluator to analyze the similar pairs of clone group descriptions (i.e., those with a TF-IDF score of > 0.5) and their associated code to determine whether the clone group pairs are similar or distinct. The two evaluators discuss any disagreements on labels for commonly evaluated clone group pairs until they reach a resolution.

Inter-clone group results We first execute the TF-IDF similarity calculation on the clone group descriptions (of valid clone groups) written by the professional developer. As a result, we obtain 23 clone group pairs having > 0.5 clone group description similarity. We then execute the TF-IDF similarity calculation on the clone group descriptions (of valid clone groups) written by the first author. As a result, we obtain 27 clone group pairs having > 0.5 clone group description similarity. Both the author and the professional developer then evaluate the clone group pairs obtained from their own respective descriptions and having a TF-IDF similarity > 0.5 . The professional developer manually evaluates each of their 23 clone group pairs by looking at the corresponding code for the sampled methods of a clone group and concludes that 16 of these pairs are semantically similar to each other. The first author manually evaluates each of their 27 clone group pairs by looking at the corresponding code for sampled methods of a clone group and concludes that only two pairs of clone groups are semantically similar to each other. We note that there are seven clone group pairs in common between the 23 clone group pairs obtained from the developer's descriptions and the 27 clone group pairs obtained from the author's descriptions. Thus, based on the descriptions from both evaluators, there are a total of 43 unique clone group pairs that are potentially similar based on a TF-IDF similarity threshold > 0.5 .

For the seven clone group pairs evaluated by both the author and the developer, there was an agreement on only one clone group pair being similar. After resolving the six disagreements through discussion, we find that out of the 43 unique clone group pairs across both evaluators, 31 are dissimilar, and 12 are similar. Thus overall, out of 6,555 clone group pairs formed from 115 unique clone groups, only 12 clone group pairs are semantically similar. This means that 99.8% of the clone group pairs are dissimilar, which means that our clustering based on API usages works well.

Overall, our manual evaluation results for both intra- and inter- clone group validation give us confidence that common API usages can indeed be used as a proxy for similar functionality, and that the clone groups detected by FACER are meaningful. With that, we can proceed to evaluate FACER's recommendations of related functionality.

RQ1: Based on a manually validated sample, 91% of the clone groups detected on the basis of API usage similarity consist of methods having similar functionality. This demonstrates that the similarity of API calls is a valid indicator of functional and feature similarity between methods. Furthermore, we observe that 99.8% of clone group pairs in our sample are functionally dissimilar which indicates that the clone groups detected are unique and rarely overlap with another clone group in terms of functional similarity.

7 RQ2: Recommending Related Features

In this section, we discuss the evaluation of the main contribution of FACER, recommending related features. This is the functionality for FACER's Stage 2 described in Section 4. We have two evaluation goals. One is to determine the optimal threshold parameters for similarity (α) and minimum support (β) used in providing recommendations. We determine these thresholds using an automated evaluation setup. The second goal is to determine the precision of the related methods that FACER recommends as judged by a human. Specifically, given a query and a selected method matching this query, we recruit participants to evaluate whether the related methods recommended by FACER indeed represent additional functionality related to the initial query, considering the application being developed.

7.1 Automated Evaluation for Determining Thresholds

7.1.1 Evaluation Methodology

In a real recommendation scenario, a developer inputs a feature query and gets matching methods against the query from the FACER repository. Then, the developer selects one of those methods for reuse and based on her selection, gets additional related method recommendations. If she finds them relevant, she can reuse them as part of her application.

In an automated evaluation scenario, we need to verify that the related method recommendations are relevant by measuring their precision against a ground truth. In other words, we need a criteria for automatically specifying that a recommended method is indeed related to the input method since using human validation for different recommendations at different thresholds is infeasible due to overloading our human participants. We thus create a proxy ground truth for automated evaluation as follows. Given an input method m from a project p in FACER's repository, we consider any method in p as the ground truth for related method recommendations for m . Thus, we consider the project to which the input method m belongs to as the ground truth project. The ground truth we use in our automated evaluation is a proxy for a subjective decision that should be made by the developer. Methods appearing in the same project typically represent related functionality, and thus conceptually match FACER's intended purpose.

As part of the recommendation process, FACER maps an input method to a clone group, gets related clone groups through examining the Method Clone Structures, and finally returns representative methods from the related clone groups. For our automated evaluation, we consider that a true positive recommendation occurs whenever FACER returns a representative method for related clone groups that happens to be from the ground truth project. Inversely, a false positive occurs when the representative method for the related clone group happens to be from a different project.

We now discuss how we select test input methods to evaluate FACER, as well as the metrics we use for the evaluation. In total, we evaluate 20 recommendation scenarios corresponding to 20 test input methods, which we also use for the manual evaluation in the next section.

To obtain related method recommendations from FACER, we need to start with a feature query and then select a relevant method from FACER's Stage 1 recommendations. Methods recommended in Stage 1 may or may not yield related method recommendations. While getting related method recommendations from Stage 2 is optional for a user, we only consider recommendation scenarios that include related method recommendations for the purpose of our evaluation. The feature query, selected method, and related method recommendations together make up a recommendation scenario. Since we have four categories of Android

applications, we want to evaluate a few recommendation scenarios for each category. Thus, we evaluate a total of 20 recommendation scenarios with five for each of the four categories of applications. We need to simulate these recommendation scenarios by issuing feature queries and then selecting a method that corresponds to Stage 1 recommendations, against which FACER Stage 2 can then return related method recommendations for evaluation.

To make our evaluation as realistic as possible, we create feature queries by manually gathering a list of feature descriptions from the README files of all applications in the FACER repository. These feature descriptions are mainly short phrases that begin with an action verb. We then short-list the feature descriptions that are common across multiple applications within a particular category. This results in a set of 10 queries. To collect an additional 10 queries, we manually locate methods from the back-end FACER repository using SQL queries that look for certain domain-specific keywords in the API calls of methods belonging to Method Clone Structures. One of the authors then assigns a feature description to the method using its name, comments and body. This feature description is then input to FACER Stage 1 to retrieve a set of matching methods. Thus, the 20 queries are a mix of feature descriptions we get from README files and some that we manually create.

We use each feature description of the 20 queries as the feature query to FACER Stage 1 and manually examine the list of returned recommendations. We select a method from the set of Stage 1 recommendations based on the following criteria: (1) the method name, comments and variable names indicate that it implements the desired functionality, (2) it is capable of generating related method recommendations. We consider criterion 2, because we want to evaluate the quality of related methods FACER recommends, which we cannot do for scenarios where FACER makes no related recommendations. Such a scenario would occur when the input method does not belong to any Method Clone Structure, which can be solved with considering more input repositories for the mining stage. Thus, given our evaluation goals, we focus on the case when FACER can find *any* related methods and evaluate the quality of these related recommendations.

One of the authors having professional experience with Android development first selects methods against queries using the FACER tool in a way that a real developer would do by looking through the complete list of retrieved methods, then clicking on a few methods which look relevant by name, scanning method bodies to check for desired functionality, and finally checking whether these methods have any related method recommendations (without evaluating the related recommendations). This way, we obtain a set of 20 test input methods corresponding to the 20 feature queries. We then use the ID of the selected method as input to FACER Stage 2 to get related method recommendations. These related method recommendations are what we evaluate. The 20 queries, the method identifiers of the selected relevant methods for the queries, and their application category names are shown in Table 5.

Evaluation Metrics We use the following metrics to evaluate the related method recommendations:

Precision: Precision measures FACER's ability to correctly recommend related methods. The precision of recommendations is calculated as the fraction of recommended methods that are relevant i.e., belonging to the ground truth project of the input method, as shown in (6). If all the recommended methods occur at least once in the test project, we have 100% precision.

$$\text{Precision}@N = \frac{|recommended\ methods \cap test\ project\ methods|}{|recommended\ methods|} \quad (6)$$

Table 5 Feature queries

No.	Feature description	Method ID	Category
1	receive paired devices name and address	33	Bluetooth chat
2	update list of paired Bluetooth devices	80	Bluetooth chat
3	do discovery of Bluetooth devices	423	Bluetooth chat
4	send message over Bluetooth	1066	Bluetooth chat
5	connect to a Bluetooth device	1161	Bluetooth chat
6	create new folder	2250	File Manager
7	browse to file or directory	2616	File Manager
8	move file	2642	File Manager
9	put file to cache	2971	File Manager
10	draw bitmap	3017	File Manager
11	set data source for media player	14435	Music Player
12	receive key press to start stop pause media	14490	Music Player
13	search for song	15214	Music Player
14	download music	22968	Music Player
15	play music	24068	Music Player
16	save forecast in database	28669	Weather
17	send Http request to get weather	29298	Weather
18	check if network connection available	29947	Weather
19	check and add permissions for location access	31838	Weather
20	create new memory cache to store weather icons	33549	Weather

Success Rate: This metric measures the rate at which the recommender can return at least one relevant recommendation against an input method. The success rate is defined as shown in (7).

$$SuccessRate = \frac{|queries\ answered|}{|queries|} \quad (7)$$

where *queries* represents the set of test input methods FACER receives and *queries answered* represents the number of times FACER successfully retrieves at least one correct recommendation against a test input method.

Mean Reciprocal Rank: The mean reciprocal rank is the average of the reciprocal ranks of the results for the number of test input methods *M*. It is defined in (8).

$$MRR = \frac{1}{|M|} \sum_{i=1}^{|M|} \frac{1}{rank_i} \quad (8)$$

where *rank_i* refers to the rank position of the first relevant result for the *i*-th test input method.

7.1.2 Automated Evaluation Results

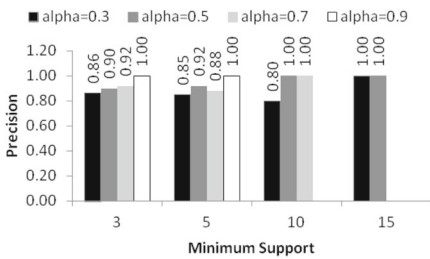
Using the 20 test input methods we obtain, we evaluate related method recommendations from FACER using different configurations. We obtain top N recommendation sets by varying the similarity threshold α and minimum support threshold β across a range of values. Table 6 shows the precision (P), success rate (SR), and mean reciprocal rank (MRR) @N

Table 6 Automated evaluation results using various thresholds of similarity α and minimum support β

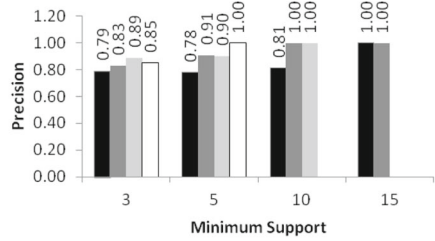
α	β	P@5	P@10	P@15	SR	MRR
0.3	3	0.86	0.79	0.75	1.00	0.97
	5	0.85	0.78	0.79	0.85	0.97
	10	0.80	0.81	0.81	0.45	1.00
	15	1.00	1.00	1.00	0.15	1.00
0.5	3	0.90	0.83	0.80	1.00	1.00
	5	0.92	0.91	0.91	0.60	1.00
	10	1.00	1.00	1.00	0.15	1.00
	15	1.00	1.00	1.00	0.15	1.00
0.7	3	0.92	0.89	0.89	0.65	1.00
	5	0.88	0.90	0.90	0.35	1.00
	10	1.00	1.00	1.00	0.10	1.00
0.9	3	1.00	0.85	0.85	0.30	1.00
	5	1.00	1.00	1.00	0.20	1.00

The success rate (SR) and mean reciprocal rank (MRR) values are for all $N=\{5,10,15\}$

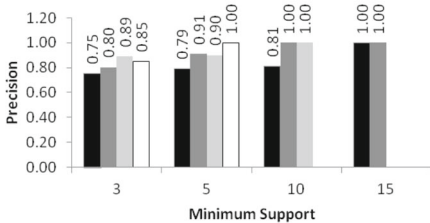
$= \{5, 10, 15\}$ for the recommendations obtained using different configurations of similarity threshold $\alpha=\{0.3, 0.5, 0.7, 0.9\}$ and minimum support thresholds $\beta =\{3, 5, 10, 15\}$. α indicates the strength of intra-clone group similarity between methods and β indicates the



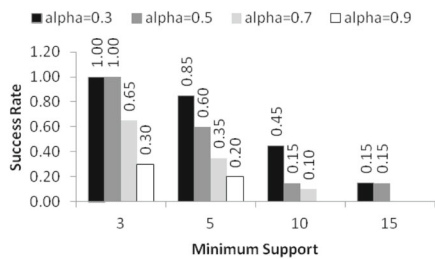
(a) Precision@5



(b) Precision@10



(c) Precision@15



(d) SuccessRate

Fig. 15 Precision and success rate of recommendations across varying similarity threshold (alpha) and minimum support (beta)

minimum support a Method Clone Structure should have to be considered a source for recommending methods.

Figure 15 is a visual summary of the same results, showing the precision@N for the recommendations obtained using different configurations of α and β . The success rate is same across all values of $N = \{5, 10, 15\}$ and is shown in Fig. 15d. Our objective is to determine the best combination of α and β that gives us a good precision without compromising success rate. From Fig. 15a, we notice that for the same $\alpha=0.3$, the precision decreases as β increases from 3 to 10, but then increases when β is further increased to 15. This is counter-intuitive but there is a reason behind the decreasing precision with increasing minimum support. In case increasing the minimum support does not yield any recommendations from the test input method itself, the recommendation algorithm switches strategy to obtain recommendations from the neighborhood of an input method. This results in a new pool of recommendations and thus a different precision value. We also observe a general trend of increasing precision as the similarity threshold α increases; however, it can have an opposite effect on success rate which decreases as α increases. Intuitively, increasing α makes the criteria for clustering of methods into clone groups more strict and results in fewer but more precise clone groups and also fewer but more precise recommendations. This intuition is also reflected in the graphs. According to this sensitivity analysis, we choose the following configuration for our manual user evaluation: we fix N at top 5, $\alpha = 0.5$, and $\beta = 3$.

7.2 Manual Evaluation of FACER's Precision

For our second evaluation, we recruit professionals and students to manually evaluate the recommended related methods and to determine the relevance of FACER's recommendations. We use the same feature queries and test methods used in our automated evaluation in Section 7.1.1. For each test method, we ask FACER to generate the top 5 related method recommendations from the optimal similarity thresholds $\alpha=0.5$ and $\beta = 3$. We then ask the participants to evaluate the relevance of these recommended methods to the input query and test method considering the category of the test method's project. A recommendation against a test input method is deemed relevant by a participant if she is able to identify the functionality of the method as being relevant to the application domain of the test method.

7.2.1 Manual Evaluation Setup

We recruit 10 industry professionals and 39 Master's students to participate in the manual evaluation. The industry professionals have both Android and Java experience and the 39 students have some experience in Java and/or Android. This number already excludes the evaluations of two students who had no Android/Java experience. To identify relevant industry professionals, we use LinkedIn's search option which allows one to search for professionals based on their job titles. We search for Android developers and send direct messages to multiple professionals to invite them to participate in our evaluation. We also use our own professional/academic contacts to recruit additional professional participants. The students were required to participate in the evaluation as a graded instrument for the Software Development: Tools and Processes graduate course at the Lahore University of Management and Sciences. The detailed demographics of our subjects for manual evaluation are shown in Table 7. The students have varying levels of experience with Android and Java. From the table, we see that all 39 students have Java experience and 36 of them also have Android experience.

We ask participants to imagine themselves as being in the process of developing an Android application of a certain category and that they have just written a method to implement a certain feature of the application. That method is the test method we have selected for each query. Based on that method, FACER provides top 5 related method recommendations which they need to evaluate and see whether they are relevant for the application they are developing. We assign two recommendation scenarios to each student. We assign at least 4 recommendation scenarios to each of the 10 professional developers. This guarantees that we have at least three evaluations for each scenario.

We create a set of 20 files containing the recommendation scenarios to be evaluated. Each file consists of an evaluation ID (method ID), the application category name, the code for the test method with its feature description, followed by the code for top 5 related methods retrieved by FACER. In our evaluation instructions, we ask the subjects to understand the features being implemented by the recommended methods by looking at the name of the method, its comments (if any), its API calls, other variables, and the overall semantics. We also ask the subjects to rate the relevance of the recommended methods on a three-point Likert scale with an integer range from zero to two, where two is *relevant*, one is *maybe relevant* and zero means *irrelevant*. We also require the students to write a feature description for each method that they evaluate to make sure that they understand the functionality before rating it. The students input their evaluation in a Google form. Five professionals provide their evaluations through a Google form and the other five provide their evaluations through e-mail.

Measuring precision We calculate the relevance of a recommended method m as the median of the relevance ratings of the participants who evaluate m . We obtain separate relevance ratings for student and professional ratings and obtain a median relevance over all ratings for each query.

We calculate the precision of recommendations of a query as a fraction of relevant methods over the total methods recommended. For each query, we consider a recommendation as relevant only if its median relevance is greater than or equal to 1. We calculate FACER's overall precision as the mean precision of all queries Q as shown in the (9):

$$Precision = \frac{\sum_{q=1}^{|Q|} |relevantMethods(q)/recMethods(q)|}{|Q|} \quad (9)$$

Table 7 Participant demographics for the manual evaluation of FACER's related methods recommendation (Precision)

Type of participant	Range of experience	No. of participants
Professional developers	1-2 years	6
	3-4 years	4
Students with Android experience	<1 year	34
	1-2 years	2
Students with Java experience	<1 year	26
	1-2 years	10
	3-4 years	3

7.2.2 Manual Evaluation Results

We first obtain separate precision values for student and professional ratings for 14 of the 20 test queries. The remaining six test queries are evaluated by professionals only and two are evaluated by students only. We then perform a paired samples Wilcoxon test (Wilcoxon test 2020) on the ratings of the 14 queries to test our null hypothesis which asserts that the medians of the precision values for students and professionals are identical. A p-value of 0.33 means that we cannot reject the null hypothesis, and accordingly there is no statistically significant difference between both groups. We therefore combine the relevance ratings of the 20 test queries of all student and professional participants to obtain the median value for a recommendation. We calculate the number of relevant recommendations for a query considering those median values that are greater than or equal to one. Table 8 shows the precision of related method recommendations for each query. Recall that we use FACER's top 5 recommendations. For 18 of the scenarios, FACER returns 5 related recommendations, while for the two remaining scenarios, FACER produces only 2 related recommendations. Overall, the evaluation contains 94 related method recommendations over the 20 scenarios. We obtain an average precision of 79.5% over the 20 recommendation scenarios.

RQ2: Based on a user evaluation involving 10 professional developers and 39 students, we find that FACER's precision for recommending related method recommendations is approximately 80%.

8 RQ3 and RQ4: User Survey

The research questions RQ1 and RQ2 allow us to evaluate FACER's effectiveness in terms of its clustering of similar functionality and its precision for related method recommendations, respectively. In RQ3 and R4, we want to additionally understand professional developers' code search and reuse practices to make sure that FACER can serve real needs, and to assess the usability and usefulness of our FACER tool and its recommendations. To investigate these points, we survey 20 professional developers including 15 Android developers and 5 Java developers. The detailed demographics of our professional survey participants are shown in Table 9.

As part of our survey, we first capture the developer's profile which includes the number of years of experience, and also the types of applications they have previously developed. We then ask them about their current code search and reuse practices. Next, we ask the developers to review recommendations from FACER. At this point, our survey breaks into three parts based on whether the developer has Android or Java expertise and whether they opt for a short review of recommendations or a longer evaluation of recommendations. The longer evaluation includes evaluating the 20 recommendation scenarios discussed in Section 7.1.1 which we distribute across the evaluators assigning four scenarios per evaluator. Eight Android developers opt for this longer evaluation and the evaluation of five of these developers is included in the manual evaluation results we reported in Section 7, whereas for the remaining three developers, the evaluation results were incomplete and were not included in the manual evaluation. Table 10 summarizes the information of the number of professional developers reviewing recommendations from various recommendation scenarios. Overall, eight Android developers are part of the longer evaluation, seven Android developers opt for a shorter evaluation where they review recommendations from

Table 8 Manual Evaluation of FACER’s related method recommendations (Relevant = no. of recommendations that are relevant, Recommended = total no. of system generated recommendations, Precision = Relevant/Recommended)

Method ID	Recommended	Relevant	Precision
33	5	5	1.0
80	5	5	1.0
423	5	5	1.0
1066	5	5	1.0
1161	5	5	1.0
2250	5	3	0.6
2616	5	4	0.8
2642	5	4	0.8
2971	5	5	1.0
3017	5	0	0.0
14435	5	5	1.0
14490	5	5	1.0
15214	5	4	0.8
22968	5	4	0.8
24068	5	4	0.8
28664	5	3	0.6
29298	2	2	1.0
29947	5	5	1.0
31838	5	1	0.2
33549	2	1	0.5
Average			0.795

our motivating example discussed in Section 2, and five Java developers review Java recommendation scenarios. We discuss the setup for reviewing the recommendation scenario from the motivating example and the recommendation scenarios generated from Java information management systems (IMS) in Section 8.1.2. Having developers review real recommendations from FACER allows us to ask them about their perceptions of its usefulness in the next section.

Finally, we get feedback from the professional developers on the tool’s interface, its potential to speed up development, their interest in future adoption of this tool, their perception of reduced time-to-search, and an overall satisfaction with the quality of recommendations. We then ask them to give their comments on our approach. Finally, we ask them

Table 9 Demographics of the professional participants who participated in our user survey

Type of participant	Range of experience	No. of participants
Professional Android developers	1-2 years	3
	3-4 years	8
	5-6 years	3
	7-10 years	1
Professional Java developers	1-2 years	1
	3-4 years	2
	7-10 years	2

to give ratings on the perceived overall usefulness and usability of FACER. Note that we also ask the 39 students who perform the manual evaluation of the 20 Android scenarios from Section 7.2 to provide an overall rating on the usefulness of FACER's recommendations. We also ask these students to provide comments on our recommendation approach.

8.1 Survey Design

8.1.1 Section 1: Developer Code Search and Reuse Practices

In this part of the survey, we focus on understanding developer practices while searching for (related) features.

We ask about those practices before they review specific recommendations from FACER to avoid biasing their opinion in any way. Specifically, we capture the frequency of performing the following 7 activities on a scale of 1 - 5 (where 1=never, 2= rarely, 3= sometimes, 4= often, 5= always).

1. Whenever I need to implement a new feature for the application I am developing, I start by searching for code examples.
2. When I search for a code example to help me implement a feature, I find what I am looking for in the results of the first search query.
3. If I get the desired code after a successful online search, I need to search again for related functionality to proceed with development.
4. While implementing the features of my application, I need to perform repeated online searches to find code for various features.
5. I reuse code for various functionalities from my previously developed applications.
6. While writing code for some feature, I recall that I have written similar code in the past and want to search for it again.
7. When writing a new application, I find myself reusing multiple methods which implement different functionality from a single application I have developed before.

8.1.2 Section 2: Recommendation Scenarios

In this section of the survey, we present participants with code recommendations to give them a demonstration of the capabilities of FACER. Eight Android developers opting for a longer evaluation evaluate the recommendations from the 20 scenarios discussed previously. Since the manual evaluation results are discussed in the previous section, we do not discuss them here again.

Table 10 Professional developers involved in reviewing recommendation scenarios for user survey

Recommendation scenarios	Dataset	No. of professionals	Professional expertise	Manual evaluation participants?
Assigned from 20 scenarios	120 Android apps	5	Android	Yes
Assigned from 20 scenarios	120 Android apps	3	Android	No
Motivating example scenario	30 Android apps	7	Android	No
Java IMS scenarios	53 Java IMS	5	Java	No

We present recommendations from the motivating example (discussed in Section 2) to seven of the Android developers. The recommended methods evaluated by the developers include those for cropping an image, showing it in `ImageView`, as well as for resizing image and getting the uniform resource identifier (URI) of a captured image. After understanding the scenario and reviewing the recommendations, the developers provide responses against two questions. The first question asks whether the recommended methods are related to the given method (select image) and system (photo sharing application) being developed. The second question asks whether the recommended methods are useful and can be reused in the context of their method and system being developed.

We present recommendations from Java projects implementing information management systems to the developers with Java experience. We create a separate FACER repository of Java projects related to information management systems. Our choice of selecting the information management systems domain is based on our premise that most of the professional Java developers would be familiar with information management systems. Thus, they would easily be able to understand recommendation scenarios and review recommendations for information management systems. This was also evident from the profile of all our survey participants who are Java professionals (we explicitly ask the Java developers about their experience with developing information management systems to confirm that our assumption is true). We collect Java projects from GitHub using the search query “Java information management systems” and get 187 results which we sort by star ratings and select the top 53 projects. The remaining projects had no stars. We create five recommendation scenarios from the Method Clone Structures detected by FACER on these projects. These recommendation scenarios are available in our online artifact page (FACER Artifacts 2020).

8.1.3 Section 3: Feedback on FACER Tool's Interface and its Usefulness

Finally, in the last section, we use Figs. 7 and 8 to show them how the interface of FACER looks like and present them with five statements to get feedback on the tools interface, its capacity to speed up development, their interest in future adoption of this tool, their perception of reduced time-to-search, and an overall satisfaction with the quality of recommendations. We capture all responses to these five statements on a five-point Likert scale, which allows us to measure the strength of their agreement.

1. The organization of information on the tool screens is clear.
2. I perceive that this tool can speed up my development.
3. I would be interested in using this tool.
4. This tool can reduce the need to perform repeated online searches to find code for various features of an applications.
5. Based on my evaluation of the various recommendation scenarios, on average, the recommender was successfully able to predict related functionality or set of functionalities.

We also ask the developers to provide open-ended feedback on the advantages or disadvantages of our approach and any other comments they might have.

8.1.4 Section 4: Usability and Usefulness Ratings

We ask the professional participants to rate the perceived usability of the FACER tool for their development activities and the usefulness of FACER's recommendations on a scale of

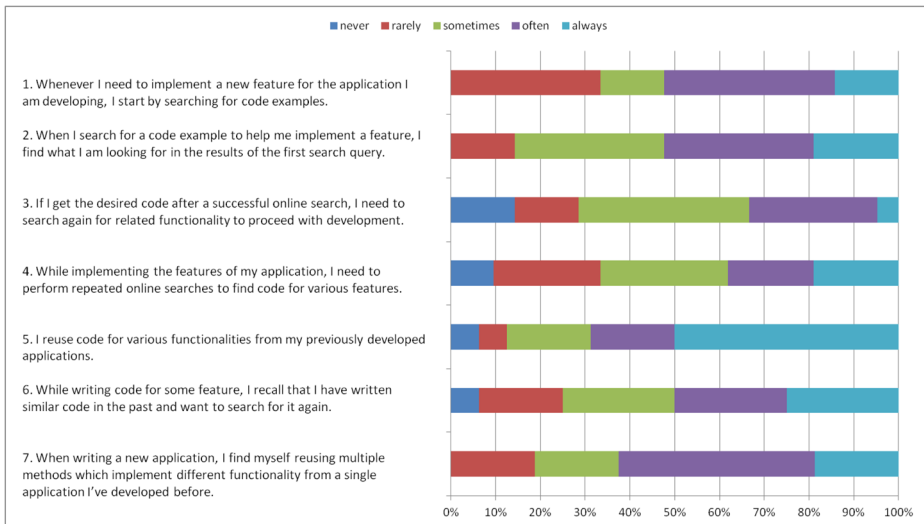


Fig. 16 Analysing developer's code search and reuse practices

1 to 5 where 1 indicates a low rating and 5 indicates a high rating. We ask the developers to provide ratings as follows:

1. Rate the usability of this recommendation tool for your development activities on a scale of 1-5.
2. Rate the usefulness of these recommendations based on their ability to provide relevant functionality for your application on a scale of 1-5.

For student participants, we only ask for a rating on the usefulness of FACER's recommendations and we ask them to provide any comments as feedback.

8.2 User Survey Results

Developer code search and reuse practices We measure the frequency of code search or reuse activities performed by professional developers and obtain their feedback on a scale of 1 to 5 (where 1=never, 2= rarely, 3= sometimes, 4= often, 5= always). The results are shown in Fig. 16. We plot these results in a series of 100% stacked bar charts which show the percentage of subjects responding to a certain value between 1 to 5 indicated by 5 different colors respectively. When reporting results to indicate that a developer does a given activity, we consider ratings 3 (sometimes) to 5 (always).

The first bar in Fig. 16 indicates that 65% of the developers start implementing a new feature by first searching for code examples with 50% doing this at least often. This observation is in line with previous studies on developer's need to search for code examples. The second bar plot indicates that the first search attempt is successful for 85% of the developers. They do not need to reformulate their query again. Next, we investigate whether the developers need to search again for related functionality after getting code for their initial search. We observe that 70% of the developers need to find functionality related to code that is obtained from their initial search to proceed with development. This strengthens our motivation to provide developers with code for related features. From the fourth bar plot,

we observe that 65% of the developers face the problem of performing repeated searches for finding code for various features of an application they are developing. This also indicates the need for a code recommender that assists developers in providing related code for their application being developed. We also investigate whether the developers tend to search and reuse the code they already wrote in the past. Our findings from the fifth and sixth bar plot indicate that 85% of the developers reuse their own code from previously developed applications and 75% need to write code for features they have previously implemented and thus search for their already written code. This indicates that applications share some common features which is coherent with our approach of mining repeatedly co-occurring features across applications. The last bar plot in Fig. 16 indicates that 80% of the developers reuse multiple functionality from a single application that they previously developed. Thus, building the FACER repository on an organization's code base can discover such repeatedly co-occurring functionality and using FACER can allow the developer to receive related code recommendations without explicitly searching for them.

RQ3: The survey results show that 70% of the developers face the need to search for related features which supports the motivation of our work.

Analyzing developer's feedback on FACER's recommendations for motivating example

We analyze the feedback of seven professional Android developers who review the recommendation scenario from the motivating example we discussed in Section 2. In response to whether the recommended methods are related to the given method (select image) and system being developed (photo sharing application), one developer strongly agreed, while four agreed and two were neutral. In response to whether the recommended methods are useful and can be reused in the context of their method and system being developed, six developers agreed and one developer was neutral. The high level of agreement shows that FACER can provide relevant recommendations that can be reused for the development of our motivation example of the photo sharing application.

Developer feedback on FACER tool's interface and its usefulness Figure 17 summarizes developer's feedback on FACER. It shows a series of bar plots which capture the percentage of developers agreeing to some statements describing the FACER tool. The levels of agreement are on a scale from 1 to 5 with 1 indicating a strong disagreement and 5 indicating a strong agreement. We observe that 75% of the developers agree (on level 4 or 5) that the organization of information on the tool screens is clear. 65% of the developers agree (on level 4 or 5) that the tool can speed up their development. 75% agree (on level 4 or 5) that they would be interested in using the tool. It is very encouraging to see that 75% of the developers perceive (on level 4 or 5) that FACER can reduce the need to perform repeated online searches to find various features of an application. 50% of the developers agree (on level 4 or 5) that overall FACER can successfully recommend related functionality.

Developers' comments on FACER's recommendations We received positive comments from the professional developers regarding FACER's IDE-integrated interface which eliminates the need to leave the development IDE to search for code. One professional also claims to never have seen such a recommendation approach for Android development. The following are quotes of some of the feedback we got:

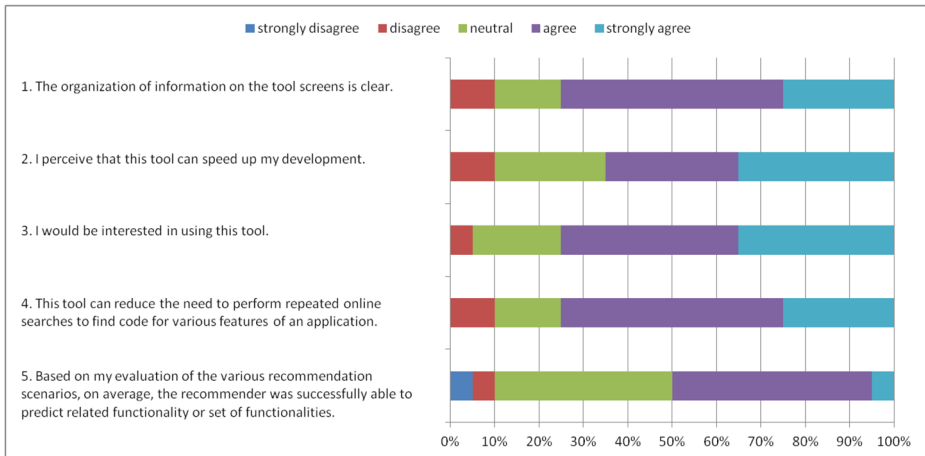


Fig. 17 Analysing developer’s feedback on FACER

- *“The best thing about this approach is that everything is on a single interface, which would make the process of searching methods quite simple and efficient. This would help generate more relevant ideas to the developers, which would improve the overall functionality of application.”*
- *“This work is really good and appreciable. No such thing is available so far in Android Development. By using this user/Developer can easily search for related code/solution and implement it, using this plugin, without moving outside Eclipse/AndroidStudio”*
- *“Developer can search the related code in the IDE window rather than going to the web.”*

The professional developers also commented on the ability of FACER to save time as shown below:

- *“It can speed up the creating of basic structure of any application feature. And after that a person can customize that according to his need.”*
- *“That seems quite reasonable and helpful!... This tool will save a lot of time.”*
- *“Time saving. Less time will be spent on checking each link shown in Google search.”*

There were some general comments of the professional subjects indicating a positive impact of the FACER tool for helping developers. One subject was of the view that FACER can not totally eliminate online searches for required code but can act as a supplementary tool to speed up development.

- *“I am really impressed by the overall idea of your tool. It will definitely help developers in the long run”*
- *“As a beginner in the industry, I used to do some online searches and check some snippets and understand the underlying objective and start implementing code as per my requirement. your FACER is good but what I’m thinking is using your tool, I don’t think every suggested method solves my problem and maybe checking all suggestions and re-write the method save some time compared to my approach. so we need to both and can’t replace one another, and I’d love to use it in future.”*

The professional developers also made some suggestions on improving FACER’s recommendations. One participant expresses the need for more relevant recommendations. This

is definitely something we can address in future. Another participant suggests making predictions based on business use case, which we think can be implemented by filtering and/or prioritizing the recommendations that directly relate to the business logic of the application. Another comment refers to the need for the source of code recommendations to be always up to date, so that new solutions are available. We plan to address this need by having a continuous repository update mechanism in place.

We also received comments suggesting UI enhancements like the recommendations to appear on the right side. FACER's Eclipse plugin allows the developer to move the panel containing code recommendation to their desired position, so this is not difficult to achieve. Furthermore, we also received suggestions to show details of the parent class of the recommended methods. In future, we can integrate the ability to browse the complete class for a recommended method. One participant mentioned the need for providing alternate solutions against recommended functionalities. Sometimes developers are looking for optimal implementations of some functionality in terms of conciseness, exception handling and other quality factors. The fact that our recommended methods come from clone groups with multiple implementations of the same functionality provides a solution for the subject's requirement. In future, we can learn to distinguish between different methods of a clone group using quality parameters to provide alternate solutions for a recommendation.

Students' comments on FACER's recommendations We also received some positive comments about FACER's recommendations from the subjects who are Master's students. They indicated the usefulness of recommendations in writing code faster and thus saving time. One student commented that FACER's recommendations having concise implementations added to his knowledge of writing improved code. Some of their comments are as follows:

- *“Recommendations are quite good and can aid a developer to code [faster] given he knows where to head”*
- *“These recommendations contained concise code. In my past experience, I remember doing things [with a comparably] difficult approach”*
- *“I found 4 useful method[s] out of 5 so I like these recommendations”*
- *“[Avoids the need for] writing the whole code or searching for [a] new module... quite helpful [recommendations].... time saving...”*

The students further expressed their opinion on using FACER for future personal projects and stated its benefit of enhancing a developer's capabilities and reusability of code.

- *“Thanks for this, I hope I can use it for my future projects if [need] be”*
- *“I think it is good idea to build this system which [enhances] the capability of [a] developer”*
- *“..in OOP driven development environments, I can see this system having a lot of value in enhancing re-usability of code... can act as code auto-complete ...”*

A few student participants had some concerns regarding the recommendations. One participant pointed out that some of FACER's results are inaccurate. Another participant pointed to the fact that some method recommendations are very generic. These comments are as follows:

- *“[some] prediction[s] are very close and some ...are very [inaccurate]”*
- *“the method you have [recommended] is very generic not ... serving a specific purpose”*

The reason for inaccurate and generic recommendations may be due to the recommended methods containing API calls that do not necessarily translate to a core feature for an application's product domain, instead they may be implementing Android framework-specific code which glues together the core features of an application. In future, we want to be able to distinguish between domain-relevant features and other more generic framework-specific helper features.

Analyzing developer's ratings on the usefulness and usability of FACER As discussed in Section 8.1.4, we ask the professional subjects to rate the usability and usefulness of our approach on a scale of 1 to 5 where 1 indicates a low rating and 5 indicates a high rating. Figure 18 shows that 90% of the professional developers give moderate to high ratings (ranging from 3 to 5 on 5-point Likert scale) on the usability of the tool for their development activities. We also find that 95% of the professionals give a moderate to high rating (ranging from 3 to 5) on the usefulness of FACER's recommendations to provide relevant functionality for their application; 70% give a high usefulness rating (ranging from 4 to 5). This indicates that professional developers find the tool and its recommendations helpful for their development tasks.

Figure 19 shows ratings from the students on the usefulness of our approach. It indicates that 85% of the students give a moderate to high rating (ranging from 3 to 5) on the usefulness of FACER's recommendations to provide help in their development; 68% give a high usefulness rating (ranging from 4 to 5). This indicates that students, like professionals, find the tools recommendations helpful for their development tasks.

Based on the above feedback, we can conclude that, overall, participants expressed the desire to use the system for their needs and feel that it could help save time by avoiding the need to search or write code.

RQ4: The survey results show that 90% of the professional developers give a moderate to high rating on the usability of FACER tool for their development activities. Furthermore, 95% of the professional developers and 85% of the student developers find the related method recommendations from FACER useful.

9 Threats to Validity

Internal Validity We rely on third-party tools in FACER's implementation such as the JDT parser (Eclipse Java development tools 2020) and clustering algorithms. Any inaccuracies

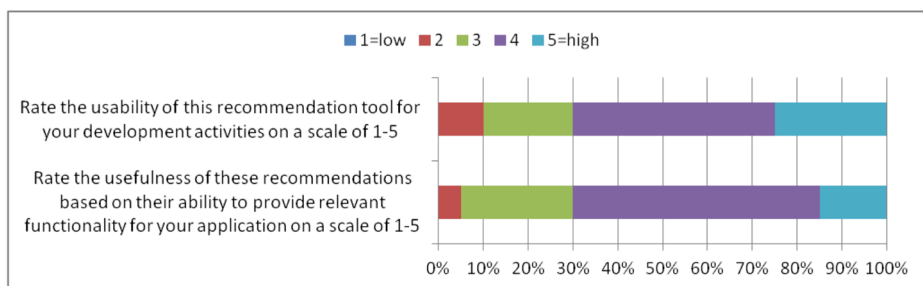


Fig. 18 Professional developer's ratings on the usefulness and usability of FACER

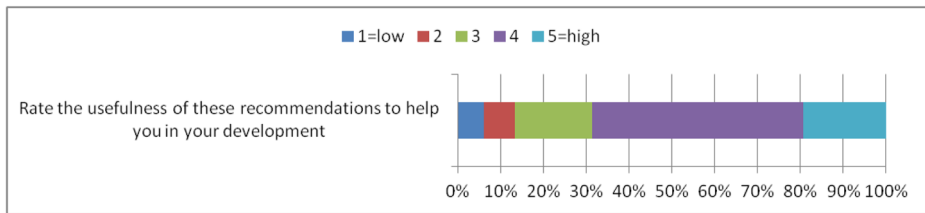


Fig. 19 Student developer's ratings on the usefulness of FACER

in these tools will affect our results. However, most of these tools have been widely used and tested. That said, we notice that there are certain types of Android framework-specific calls that have no object reference; such calls are not detected by the JDT parser (Eclipse Java development tools 2020) as method invocations and are, therefore, not parsed as API calls. This can result in some functionalities being ignored while clustering methods into clone groups. Furthermore, in the calculation of the number of statements of methods containing API calls, we proxied statements using line numbers which has resulted in density percentage values higher than 100% for some methods. While this can have an effect on the clustering of methods into clone groups, our manual analysis of the validity of clone groups formed as a result of the clustering algorithm gives us reassurance in the results.

For the evaluation of inter-clone group dissimilarity, the human evaluators manually validate only the clone group descriptions of pairs with a TF-IDF similarity threshold greater than 0.5. By relying on TF-IDF, we may have incorrectly automatically marked some clone groups with a TF-IDF score less than 0.5 as dissimilar. However, we decided to use this technique, because it is impossible to ask an external validator to manually validate thousands of combinations.

Construct Validity The ground truth we use in our automated evaluation is a proxy for a subjective decision that should be made by the developer. There could be a method from another project that is actually relevant but that we consider as a false positive. However, our automated evaluation only helps us in determining the appropriate thresholds. We engage professional developers and students for a manual evaluation to determine FACER's precision in practice.

We do not currently report the number of times FACER is able to provide related method recommendations against all methods retrieved in Stage 1 of FACER. Currently, the FACER repository is built on 120 projects only. Once we increase the size of the repository, we can calculate the overall success rate of providing related method recommendations.

Our manual validation of the recommendations as well as user survey setup does not represent a real development scenario that a developer would go through to really use FACER. Our setup, while contrived, reduces the cognitive load and expectations of actual development from participants and allows them to focus on a well-defined task. Since we provide the description of the test method as well as the domain of the application, participants can easily determine whether a recommended method is indeed related or not. Such an evaluation gives us a fair indication of FACER's precision by human subjects. In the future, we plan to conduct a long-term user study where developers use FACER for real tasks and we evaluate how often they use the recommended related features and what their perception of the tool is.

External Validity The limited number of professional developers involved in evaluating FACER is a threat to external validity.

Only one professional developer performed the evaluation of 126 clone groups. We reduce the threat to validity of evaluation by ensuring that the evaluator is experienced and also by including all authors of this paper in the evaluation.

We reported the precision of FACER based on our dataset that consists only of Android applications. Applications from our datasets were chosen to have some common domains such that we can indeed find meaningful co-occurring features. While we cannot generalize beyond our analyzed applications, we do not see any conceptual reasons why FACER cannot be applied to more projects and domains with similar results. While the number of queries we use is limited, we wanted to make sure we evaluate with meaningful queries that represent actual functionality. Since FACER's Stage 2 requires that the user has selected a method from FACER's repository in Stage 1, this limits our ability to use external queries from Stack Overflow, for example, since they may represent features not in FACER's repository.

10 Discussion and Future Work

FACER got positive feedback from users participating in our survey. However, there is still room for improvement. One of the professional developers who participated in our survey had a concern regarding the results of FACER being bound to data in the FACER repository. For now, we can build the FACER repository on any collection of Java source code projects whether they are downloaded from online sources (GitHub, SourceForge, etc.) or found within local code repositories in an organization. FACER may miss out on code recommendations if they are not part of the raw source code projects it mines. This has two implications: (1) The repository needs to be constantly updated. This can be an offline process applied once a month for example. Or in future, we may switch to mining Method Clone Structures on-the-fly from the results of online code searches. (2) In an industrial context, an organization often needs to develop similar products for a product line (Apel et al. 2013). If FACER is built on an organization's code repositories, then developers can get effective recommendations for the software being developed. Our findings from our survey indicate that 85% of the developers reuse their own code from previously developed applications and 75% need to write code for features they have previously implemented and thus search for already written code. This indicates that applications share some common features which is coherent with our approach of mining repeatedly co-occurring features across applications. Our results show that 80% of the developers reuse multiple functionality from a single application that they previously developed. Thus, building the FACER repository on an organization's code base can discover such repeatedly co-occurring functionality and using FACER can allow the developer to receive related code recommendations without explicitly searching for them.

In our motivating example from Section 2, the StackOverflow user is having problems finding "a complete and simple code". The user states that the code she tested did not work or was not good. FACER prioritizes making recommendations from the same project from which a user selects a method in the first stage of FACER. This way, FACER is likely to recommend code that can integrate without problems in a developer's existing code. However, ensuring that a recommended method can integrate and work without problems requires an understanding and awareness of the developer's context. In the future, we plan to work on making FACER context-aware to make context-sensitive recommendations that

integrate well. The quality of the recommended code snippet may depend on several qualitative factors such as ease of integration, conciseness, readability, and optimized execution. Currently, FACER does not incorporate these qualitative aspects of code while making recommendations. However, this is something we plan to pursue in future.

We also plan to extend our evaluation to larger Android/Java code bases and evaluate with more queries including those from Stack Overflow. Additionally, we plan to evaluate FACER by conducting a live study in an open forum or in the context of an organized workshop, where we can also investigate whether FACER can effectively speed up development in real settings. In future, we intend to have developers perform some programming tasks using their conventional methods and compare task completion time with that of tasks completed using FACER. From a survey, we can check which code search tools developers use currently, and use them as baselines for comparison.

11 Conclusion

Current code recommendation and code search systems focus on the immediate requirements of the developer. They retrieve code against a specific query. For a new but related task, the developer has to perform a new search. The need to perform repeated searches for associated functionality can impede the performance and productivity of the developer. In this paper, we proposed a solution that allows developers to receive recommendations for their potential future requirements. Our main contribution is a recommendation system FACER that provides developers with method recommendations having functionality relevant to their current feature under development.

FACER works in two stages. The first stage is a simple code search engine which given a query returns a code snippet implementing the feature in the query. The second stage, which is the main contribution of this paper, is a recommender which given a selected method from Stage 1 recommends related methods that implement related functionality. For example, if a developer is currently implementing the “connect to Bluetooth” feature and found a relevant method to reuse, FACER would recommend a method implementing the “disconnect from Bluetooth” functionality as a related feature the developer may need to implement. To accomplish this, FACER relies on clustering methods according to their API usages, where a cluster represents methods implementing the same or similar functionality. It then finds frequently co-occurring method clusters which it uses to recommend related functionality.

To evaluate FACER, we extracted data from 120 open-source GitHub projects from four different domains. We first performed a manual validation of the detected method clusters to ensure that clustering methods based on API usages results in meaningful clusters with functionally similar methods. Our results show that 91% of the analyzed method clusters are valid. We then performed an automatic evaluation with different FACER settings to determine the best configuration for related method recommendations. Once we determined the best configuration, we performed a user evaluation with 10 professional and 39 experienced student participants to determine whether the related methods recommended by FACER are indeed relevant to the original method and feature. Our results show that FACER’s related method recommendations are, on average, 80% precise. We also received positive feedback about FACER’s functionality, which encourages us to further improve FACER and release it to developers soon. The current prototype implementation for FACER, along with all the data from our evaluations is available on our online artifact page (FACER Artifacts 2020).

Acknowledgments The authors are thankful to the Lahore University of Management Sciences (LUMS), Ignite National Technology Fund Pakistan (SRG-257), and Prince Sultan University for funding our research. The last author's research is funded by the Canada Research Chairs Program. The authors would also like to thank the professional developers and students who participated in the user survey and evaluation.

References

- Abdalkareem R, Shihab E, Rilling J (2017) On code reuse from stackoverflow: an exploratory study on android apps. *Inf Softw Technol* 88:148–158
- Abid S (2019) Recommending related functions from api usage-based function clone structures. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 1193–1195
- Abid S, Javed S, Naseem M, Shahid S, Basit HA, Higo Y (2017) Codeease: harnessing method clone structures for reuse. In: 2017 IEEE 11th international workshop on Software clones (IWSC). IEEE, pp 1–7
- Abstract Syntax Trees (2020) <https://www.eclipse.org/jdt/core/r2.0/dom> [Online; accessed 28-Sep-2020]
- Android SDK Classes (2020) <https://developer.android.com/reference/classes>. [Online; accessed 28-Sep-2020]
- Android Studio SDK (2020) <https://developer.android.com/studio>. [Online; accessed 28-Sep-2020]
- Apel S, Batory D, Kästner C, Saake G (2013) Software product lines. In: Feature-oriented software product lines. Springer, pp 3–15
- Asaduzzaman M, Roy CK, Schneider KA, Hou D (2016) A simple, efficient, context-sensitive approach for code completion. *J Softw Evol Process* 28(7):512–541
- Bajracharya SK, Ossher J, Lopes CV (2010) Leveraging usage similarity for effective retrieval of examples in code repositories. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, pp 157–166
- Baker BS (1993) A theory of parameterized pattern matching: algorithms and applications. In: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp 71–80
- Bavota G, De Lucia A, Marcus A, Oliveto R (2014) Recommending refactoring operations in large software systems. In: Recommendation systems in software engineering. Springer, pp 387–419
- Bielik P, Raychev V, Vechev M (2015) Programming with” big code”: Lessons, techniques and applications. In: LIPICs-Leibniz International Proceedings in Informatics, vol 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
- Brandt J, Guo PJ, Lewenstein J, Dontcheva M, Klemmer SR (2009) Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, pp 1589–1598
- binaryDist (2020) <https://github.com/NikNakk/binaryDist/>
- Brandt J, Guo PJ, Lewenstein J, Klemmer SR (2008) Opportunistic programming: How rapid ideation and prototyping occur in practice. In: Proceedings of the 4th international workshop on End-user software engineering, pp 1–5
- Bruch M, Monperrus M, Mezini M (2009) Learning from examples to improve code completion systems. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, pp 213–222
- Cambronero J, Li H, Kim S, Sen K, Chandra S (2019) When deep learning met code search. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 964–974
- Chatterjee S, Juvekar S, Sen K (2009) Sniff: A search engine for java using free-form queries. *Fund Approach Softw Eng*:385–400
- Chen L, Ye W, Zhang S (2019) Capturing source code semantics via tree-based convolution over api-enhanced ast. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, pp 174–182
- Chen X, Zou Q, Fan B, Zheng Z, Luo X (2018) Recommending software features for mobile applications based on user interface comparison. *Requir Eng*:1–15
- Cohen J (1960) A coefficient of agreement for nominal scales. *Educ Psychol Measur* 20:37–46
- Defays D (1977) An efficient algorithm for a complete link method. *Comput J* 20(4):364–366
- Dumitru H, Gibiec M, Hariri N, Cleland-Huang J, Mobasher B, Castro-Herrera C, Mirakhorli M (2011) On-demand feature recommendations derived from mining public product descriptions. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp 181–190

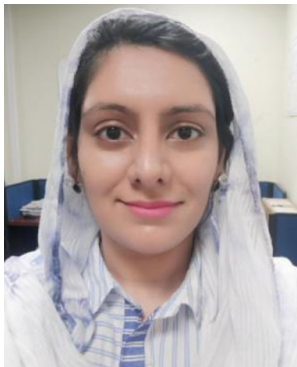
- Eclipse code recommender (2018) <http://www.eclipse.org/recommenders>. [Online; accessed 16-May-2018]
- Eclipse Java development tools (2020) <https://www.eclipse.org/jdt/>. [Online; accessed 28-Sep-2020]
- FACER Artifacts (2020) https://github.com/shamsa-abid/FACER_Artifacts
- Fiverr - freelance services marketplace for businesses (2021) <https://www.fiverr.com/>. [Online; accessed 3-Feb-2021]
- FPClose (2019) <https://www.philippe-fournier-viger.com/spmf/FPClose.php>. [Online; accessed 1-Feb-2019]
- GitHub (2020) <https://github.com/>. [Online; accessed 28-August-2020]
- Grahne Gösta, Zhu Jianfei (2005) Fast algorithms for frequent itemset mining using fp-trees. *IEEE transactions on knowledge and data engineering* 17(10):1347–1362
- Gu X, Zhang H, Kim S (2018) Deep code search. In: 2018 IEEE/ACM 40th international conference on software engineering (ICSE). IEEE, pp 933–944
- Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp 631–642
- Han J, Kamber M, Pei J (2012) 9 - classification: Advanced methods, 3rd edn. Han J, Kamber M, Pei J (eds), Morgan Kaufmann, Boston. ISBN 978-0-12-381479-1. <http://www.sciencedirect.com/science/article/pii/B9780123814791000095>
- Han J, Pei Jx, Kamber M (2011) Data mining: concepts and techniques. Elsevier
- Hartmann B, Doorley S, Klemmer SR (2008) Hacking, Mashing, gluing: Understanding opportunistic design. *IEEE Pervasive Comput* 7(3):46–54
- He J, Zhang J, Li X, Ren Z, Lo D, Wu X, Luo Z (2019) Recommending new features from mobile app descriptions. *ACM Trans Softw Eng Methodol (TOSEM)* 28(4):22
- Heirarchical clustering (2019) <https://rdrr.io/r/stats/hclust.html>, December 2019
- Hill R, Rideout J (2004) Automatic method completion. In: Proceedings of the 19th IEEE international conference on Automated software engineering. IEEE Computer Society, pp 228–235
- Holmes R, Murphy GC (2005a) Using structural context to recommend source code examples. In: 2005. ICSE 2005. Proceedings. 27th international conference on Software engineering. IEEE, pp 117–125
- Holmes R, Walker RJ, Murphy GC (2005b) Strathcona example recommendation tool. In: ACM SIGSOFT Software engineering notes, vol 30. ACM, pp 237–240
- Holmes R, Walker RJ, Murphy GC (2006) Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans Softw Eng* 32(12)
- Hong Y, Lian Y, Yang S, Tian L, Zhao X (2016) Recommending features of mobile applications for developer. In: International conference on advanced data mining and applications. Springer, pp 361–373
- Hsu S-K, Lin S-J (2011) Macs: Mining api code snippets for code reuse. *Expert Syst Appl* 38(6):7291–7301
- Ichii M, Hayase Y, Yokomori R, Yamamoto T, Inoue K (2009) Software component recommendation using collaborative filtering. In: 2009 ICSE Workshop on search-driven development-users, infrastructure, tools and evaluation. IEEE, pp 17–20
- Inoue K, Yokomori R, Yamamoto T, Matsushita M, Kusumoto S (2005) Ranking significance of software components based on use relations. *IEEE Trans Softw Eng* 31(3):213–225
- Ishihara T, Hotta K, Higo Y, Igaki H, Kusumoto S (2012) Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects. In: 2012 19th working conference on Reverse engineering (WCRE). IEEE, pp 387–391
- Ishihara T, Hotta K, Higo Y, Kusumoto S (2013) Reusing reused code. In: 2013 20th working conference on Reverse engineering (WCRE). IEEE, pp 457–461
- Jaccard P (1901) Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat* 37:547–579
- Jansen S, Brinkemper S, Hunink I, Demir C (2008) Pragmatic and opportunistic reuse in innovative start-up companies. *IEEE Softw* 25(6):42–49
- Java Class Libraries (2020) <https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html>. [Online; accessed 28-Sep-2020]
- Java Development Kit (2020) <https://www.oracle.com/java/technologies/javase-downloads.html>. [Online; accessed 28-Sep-2020]
- Jiang H, Nie L, Sun Z, Ren Z, Kong W, Zhang T, Luo X (2016) Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*
- Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans Softw Eng* 28(7):654–670
- Kanwal J, Maqbool O, Basit HA, Sindhu MA (2019) Evolutionary perspective of structural clones in software. *IEEE Access* 7:58720–58739
- Keivanloo I, Rilling J, Zou Y (2014) Spotting working code examples. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp 664–675

- Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2009) Sourcerer: mining and searching internet-scale software repositories. *Data Min Knowl Disc* 18(2):300–336
- Lucene (2017) <https://lucene.apache.org>. [Online; accessed 28-August-2017]
- Lucene Core (2020). <https://lucene.apache.org/core/> (2020). [Online; accessed 29-Sep-2020]
- Lucene Document (2020). <https://lucene.apache.org/core/7.2.0/core/org/apache/lucene/document/Document.html>. [Online; accessed 29-Sep-2020]
- Luan S, Di Y, Barnaby C, Sen K, Chandra S (2019) Aroma: Code recommendation via structural code search. *Proc ACM Programm Lang* 3(OOPSLA):1–28
- Luan S, Di Y, Sen K, Chandra S (2018) Aroma: Code recommendation via structural code search. arXiv:1812.01158
- Lv C, Jiang W, Liu Y, Hu S (2014) Apisynth: a new graph-based api recommender system. In: *ICSE Companion*, pp 596–597
- Lv F, Zhang H, Lou J-g, Wang S, Zhang D, Zhao J (2015) Codehow: Effective code search based on api understanding and extended boolean model (e). In: *2015 30Th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, pp 260–270
- Mandelin D, Xu L, Bodík B, Kimelman D (2005) Jungloid mining: helping to navigate the api jungle. In: *ACM SIGPLAN Notices*, vol 40. ACM, pp 48–61
- Market-basket analysis (2019) <https://www.kdnuggets.com/2016/10/association-rule-learning-concise-technical-overview.html>, December 2019
- McIlroy S, Ali N, Hassan AE (2016) Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empir Softw Eng* 21(3):1346–1370
- McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: finding relevant functions and their usage. In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, pp 111–120
- McMillan C, Hariri N, Poshyvanyk D, Cleland-Huang J, Mobasher B (2012) Recommending source code for use in rapid software prototypes. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, pp 848–858
- Mccarey F, Cinnéide M, Kushmerick N (2005) Rascal: A recommender agent for agile reuse. *Artif Intell Rev* 24(3-4):253–276
- Mens K, Lozano A (2014) Source code-based recommendation systems. In: *Recommendation systems in software engineering*. Springer, pp 93–130
- Mishne A, Shoham S, Yahav E (2012) Typestate-based semantic code search over partial programs. In: *Acmsigplan notices*, vol 47. ACM, pp 997–1016
- Mojica JJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE (2013) A large-scale empirical study on software reuse in mobile apps. *IEEE Softw* 31(2):78–86
- Moreno L, Bavota G, Di Penta M, Oliveto R, Marcus A (2015) How can i use this method?. In: *2015 IEEE/ACM 37th IEEE international conference on Software engineering (ICSE)*, vol 1. IEEE, pp 880–890
- Nguyen P, Di Rocco J, Ruscio D, Ochoa L, Degueule T, Di Penta M (2019) Focus: A recommender system for mining api function calls and usage patterns. In: *41St ACM/IEEE international conference on software engineering (ICSE)*
- Nguyen AT, Nguyen TT, Nguyen HA, Tamrawi A, Nguyen HV, Al-Kofahi J, Nguyen TN (2012) Graph-based pattern-oriented, context-sensitive source code completion. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, pp 69–79
- Niu H, Keivanloo I, Zou Y (2017) Api usage pattern recommendation for software development. *J Syst Softw* 129:127–139
- Ohtani A, Higo Y, Ishihara T, Kusumoto S (2015) On the level of code suggestion for reuse. In: *2015 IEEE 9th international workshop on Software clones (IWSC)*. IEEE, pp 26–32
- parallelDist (2020) <https://cran.r-project.org/web/packages/parallelDist/index.html>. [Online; accessed 14-Sept-2020]
- Pascarella L, Geiger F-X, Palomba F, Di Nucci D, Malavolta I, Bacchelli A (2018) Self-reported activities of android developers. In: *2018 IEEE/ACM 5Th international conference on mobile software engineering and systems (MOBILESoft)*. IEEE, pp 144–155
- Rahman MM, Roy CK, Lo D (2016) Rack: Automatic api recommendation using crowdsourced knowledge. In: *Software analysis, evolution, and reengineering (SANER), 2016 IEEE 23rd international conference on*, vol 1. IEEE, pp 349–359
- Raychev V, Vechev M, Yahav E (2014) Code completion with statistical language models. In: *ACM SIGPLAN Notices*, vol 49. ACM, pp 419–428
- Richard Landis J, Koch GG (1977) The measurement of observer agreement for categorical data. *Biometrics*, pp 159–174

- Sachdev S, Li H, Luan S, Kim S, Sen K, Chandra S (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, pp 31–41. ISBN 978-1-4503-5834-7. <https://doi.org/10.1145/3211346.3211353>
- Sadowski C, Stolee KT, Elbaum S (2015) How developers search for code: a case study. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, pp 191–201
- Sammut C, Webb GI (eds.) (2010) TF-IDF, Springer, Boston. ISBN 978-0-387-30164-8. https://doi.org/10.1007/978-0-387-30164-8_832
- Score boosting (2020) https://lucene.apache.org/core/3_5_0/scoring.html#Score [Online; accessed 27-Sep-2020]
- Shimada R, Hayase Y, Ichii M, Matsushita M, Inoue K (2009) A-score: Automatic software component recommendation using coding context. In: 2009 31st international conference on software engineering-companion volume. IEEE, pp 439–440
- Sparse matrix clustering (2019) https://stackoverflow.com/questions/30944701/clustering-a-large-very-sparse-binary-matrix-in-r/30945176?noredirect=1#comment106303086_30945176
- Stack Overflow developer survey (2020) Most loved, dreaded, and wanted platforms. <https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-platforms>. [Online; accessed 16-September-2020]
- Stack Overflow Question. (2020) <https://stackoverflow.com/questions/25490928/androidselect-image-from-gallery-then-crop-that-and-show-in-an-imageview>. [Online; accessed 16-Sep-2020]
- Stephen E et al (1995) Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gattford Okapi at trec-3. Nist Special Publ Sp 109:109
- Subramanian S, Inozemtseva L, Holmes R (2014) Live api documentation. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp 643–652
- Svajlenko J, Keivanloo I, Roy CK (2013) Scaling classical clone detection tools for ultra-large datasets An exploratory study. In: 2013 7th international workshop on software clones (IWSC). IEEE, pp 16–22
- Takuya W, Masuhara H (2011) A spontaneous code recommendation tool based on associative search. In: Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation. ACM, pp 17–20
- Thummalapenta S, Xie T (2007) Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, pp 204–213
- Thung F, Lo D, Lawall J (2013) Automated library recommendation. In: 2013 20th working conference on reverse engineering (WCRE). IEEE, pp 182–191
- Thung F, Oentaryo RJ, Lo D, Tian Y (2017) Webapirec: Recommending web apis to software projects via personalized ranking. arXiv:1705.00561
- Tsunoda M, Kakimoto T, Ohsugi N, Monden A, Matsumoto K-I (2005) Javawock: A java class recommender system based on collaborative filtering. In: SEKE, pp 491–497
- Umarji M, Sim S, Lopes C (2008) Archetypal internet-scale source code searching. Open source development, communities and quality, pp 257–263
- Vechev M, Yahav E et al (2016) Programming with “big code”. Found Trends@ Programm Lang 3(4):231–284
- Venkatasubramanyam RD, Gupta S, Singh HK (2013) Prioritizing code clone detection results for clone management. In: 2013 7th international workshop on software clones (IWSC). IEEE, pp 30–36
- Wan Y, Shu J, Sui Y, Xu G, Zhao Z, Wu J, Yu PS (2019) Multi-modal attention network learning for semantic source code retrieval. arXiv:1909.13516
- Wang J, Dang Y, Zhang H, Chen K, Xie T, Zhang D (2013) Mining succinct and high-coverage api usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, pp 319–328
- Wang L, Lu F, Wang L, Li G, Xie B, Yang F (2011) Apiexample: An effective web search based usage example recommendation system for java apis. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp 592–595
- Wilcoxon test (2020) <https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/wilcox.test>. [Online; accessed 18-Oct-2020]
- Xia X, Bao L, Lo D, Kochhar PS, Hassan AE, Xing Z (2017) What do developers search for on the web? Empir Softw Eng 22(6):3149–3185
- Xie T, Pei J (2006) Mapo: Mining api usages from open source repositories. In: Proceedings of the 2006 international workshop on Mining software repositories. ACM, pp 54–57

- Yan S, Yu H, Chen Y, Shen B, Jiang L (2020) Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 344–354
- Yang P, Fang H, Lin J (2017) Anserini: Enabling the use of lucene for information retrieval research. In: Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp 1253–1256
- Yao Z, Peddamail JR, Sun H (2019) Coacor: Code annotation for code retrieval with reinforcement learning. In: The world wide web conference, pp 2203–2214
- Ye Y, Fischer G (2002) Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the 24th international conference on Software engineering. ACM, pp 513–523
- Yu Y, Wang H, Yin G, Bo L (2013) Mining and recommending software features across multiple web repositories. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware. ACM, pp 9
- Yun U, Leggett JJ (2005) Wlpminer: weighted frequent pattern mining with length-decreasing support constraints. In: Pacific-asia conference on knowledge discovery and data mining. Springer, pp 555–567
- Zhang J, He J, Ren Z, Chen X (2018) Recommending apis for api related questions in stack overflow. IEEE Access 6:6205–6219
- Zhao J, Liu Y (2017) Detecting and ranking api usage pattern in large source code repository: A lfm based approach. In: International cross-domain conference for machine learning and knowledge extraction. Springer, pp 41–56
- Zhou S, Shen B, Zhong H (2019) Lancer: Your code tell me what you need. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 1202–1205

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Shamsa Abid is a Ph.D. Candidate in the Department of Computer Science at the Lahore University of Management Sciences (LUMS), Lahore, Pakistan. She received her B.S. degree in Computer Science from Lahore College for Women University, Lahore, Pakistan, in 2004 and the M.S. degree in Computer Science from the Lahore University of Management Sciences (LUMS), Lahore, Pakistan, in 2013. Her research interests include software reuse, code recommendation systems and information retrieval. Before joining her Ph.D. program, she worked in both the software industry as well as in the academia. She has experience working as a Senior Software Engineer in Techlogix Pvt. Ltd., where her work experience included the development of Android applications. She has held Visiting Lecturer positions at the National University of Computer and Emerging Sciences (NUCES FAST), Lahore, Kinnaird College for Women University, Lahore, and Beaconhouse National University, Lahore, Pakistan. She has also served as a member of the Shadow Program Committee for the Mining Software Repositories (MSR) conference in 2021.



Shafay Shamail completed his BSc Electrical Engineering from UET Lahore, MSc Electronics from University of Wales, UK and Ph.D. in Electrical Engineering from University of Bath U.K. He has over 30 years of teaching experience during which he has also been involved in curriculum design and implementation. He has worked in the software industry as well where he gained experience in E-commerce technologies. His current research interests encompass study of quality aspects of software engineering including software development tools, software process improvement, software quality prediction, & autonomic systems, and utilization of cloud infrastructure and services for developing e-government and e-commerce solutions. He is currently working as Professor in the Department of Computer Science, LUMS, Lahore. He is a Senior Member of IEEE.




Hamid Abdul Basit is an Associate Professor at the department of Computer Science, College of Computer and Information Sciences, Prince Sultan University. He received his Ph.D. from National University of Singapore. His area of research is software engineering; focusing on software reuse, software maintenance, code clones, code recommendation systems, and secure software development. He has several publications in the top conferences and journals of the field.



Sarah Nadi is an Assistant Professor in the Department of Computing Science at the University of Alberta, and a Tier II Canada Research Chair in Software Reuse. She obtained her Master's (2010) and PhD (2014) degrees from the University of Waterloo in Canada. Before joining the University of Alberta in 2016, she spent approximately two years as a post-doctoral researcher at the Technische Universität Darmstadt in Germany. Sarah's research focuses on providing intelligent support for software maintenance and reuse across three main themes: developing variability analysis strategies to help developers deal with the complexity of highly configurable software systems designed to enable large-scale code reuse, providing software integration support for consolidating changes from multiple versions of the same system as they evolve over time, and creating recommender systems to guide developers through correctly and securely reusing individual functionality from external libraries.

Affiliations

Shamsa Abid¹  · Shafay Shamail¹ · Hamid Abdul Basit² · Sarah Nadi³

Shafay Shamail
sshmail@lums.edu.pk

Hamid Abdul Basit
hbasit@psu.edu.sa

Sarah Nadi
nadi@ualberta.ca

¹ Lahore University of Management Sciences, Lahore, Pakistan

² Prince Sultan University, Riyadh, Saudi Arabia

³ University of Alberta, Edmonton, Canada