Siri, Write the Next Method

Abstract—Code completion is one of the killer features of modern Integrated Development Environments (IDEs), and researchers have proposed different methods to improve its accuracy. While these techniques are valuable to speed up code writing, they are limited to recommendations related to the next few tokens a developer is likely to type given the current context. In the best case, they can recommend a few APIs that a developer is likely to use next.

We present FeaRS, a novel retrieval-based approach that, given the current code a developer is writing in the IDE, can recommend the next complete method (*i.e.,* signature and method body) that the developer is likely to implement. To do this, FeaRS exploits "implementation patterns" (*i.e.,* groups of methods usually implemented within the same task) learned by mining thousands of open source projects. We instantiated our approach to the specific context of Android apps.

A large-scale empirical evaluation we performed across more than 20k apps shows encouraging preliminary results, but also highlights future challenges to overcome.

Index Terms—Code Recommender, Empirical Software Engineering, Mining Software Repositories

I. INTRODUCTION

Developing high-quality software while reducing time-to-market are two classical contrasting objectives in the software industry. This translates into the need for increasing the productivity of software developers, by lowering their learning curves when dealing with unfamiliar code, and by maximizing the quality of the code they write. In response to these needs, researchers have proposed recommender systems for software engineering, defined by Robillard *et al.* as "applications that provide information items valuable for a software engineering task in a given context" [1].

Some recommender systems pursue a long-lasting dream of software engineering research: The (semi-)automatic generation of source code. The goal of these tools is speeding up the implementation of new code. Code completion techniques are nowadays one of the killer features of IDEs [2]. Researchers have proposed different methods to improve code completion accuracy and, more in general, its capabilities [3]–[9]. While these approaches are certainly valuable to speed up code writing, they are limited to recommendations related to the next few tokens a developer is likely to type given the current context. In the best case, they can recommend a sequence of APIs that a developer is likely to use next [5], [8].

We aim at reaching the next level in supporting developers during the writing of new code. We present FeaRS, an approach and an IDE plugin which monitors the code written by Android developers in the IDE and able to recommend the complete code of the next method (*i.e.,* signature and method body) they are likely to implement based on method(s) they already have implemented.

FeaRS relies on a set of implementation patterns that we built by mining 20,713 open-source Android apps available on GitHub. To give a concrete example, the code snippet in Fig. 1 implements an options menu in an Android app. To perform such a task, tutorials recommends as first step to inflate the menu in the `onCreateOptionsMenu(...)` method and, then, to handle the item selection in the `onOptionsItem-Selected(...)` method. Assuming the existence of this implementation pattern in several apps, FeaRS can learn it and recommend the implementation of `onOptionsItem-Selected(...)` once `onCreateOptionsMenu(...)` has been implemented by the developer.

```java
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.my_options_menu, menu);
    return true;
}

public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.about:
            startActivity(new Intent(this, About.class));
            return true;
        case R.id.help:
            startActivity(new Intent(this, Help.class));
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Fig. 1. An implementation pattern in Android

We analyzed 2,721,800 commits performed during the history of the subject apps to identify new methods that are implemented within the same commit. This results, for each analyzed commit c_k, in a set $M_k = \{m_1, m_2, \ldots, m_n\}$ of n new methods created in c_k. By extracting this information for thousands of commits, we can identify implementation patterns repeatedly followed by Android developers, *e.g.,* the implementation of m_1 could imply the implementation of $m_2, \ldots, m_n$. We refer to m_1 as the Left-Hand Side (LHS) of the pattern and to $m_2, \ldots m_n$ as the Right-Hand Side (RHS).

The identification of these implementation patterns is far from trivial. Indeed, two commits c_k and c_j performed in two different repositories may implement different sets of new methods (*e.g.,* $M_k = \{m_1, m_2\}$ and $M_j = \{m_3, m_4\}$) that, however, represent the same implementation pattern (*i.e.,* $m_1 = m_3$ and $m_2 = m_4$).

Recognizing this situation is necessary to identify groups of methods that are repeatedly implemented together in different commits/apps, and not just by chance in a single/few commit(s).

FeaRS identifies clusters of methods likely to implement the same feature in the overall set of mined added methods. Going back to the previous example, this means that m_1 and m_3 are assigned to the same cluster C_1, and m_2 and m_4 to C_2. This results in the flattening of c_k and c_j to the same implementation pattern (*i.e.,* $M_k = M_j = \{C_1, C_2\}$). Once this processing is done for all mined commits, FeaRS applies association rule discovery [10] on all commits, thus creating the set of implementation patterns it relies on.

When monitoring the code written by a developer in the IDE, FeaRS identifies newly written methods and assigns, if possible, each of them to one of the clusters created in the previous step. Then, it checks if an implementation pattern having one or more of the newly implemented methods as LHS is available and, in case a pattern is found, the corresponding RHS is triggered as a recommendation to the developer.

We evaluated FeaRS in a study in which we simulated its usage in the change history of the same 20,713 apps we used to extract the implementation patterns. We used the first 80% of the apps' histories to extract the implementation patterns, the subsequent 10% to tune the FeaRS's parameters, and the last 10% to assess its performance (*i.e.,* test set). For each commit c in the test set, we simulated the scenario in which a developer implemented a subset S of the new methods added in c and used FeaRS to generate recommendations using S as LHS. Then, in case a recommendation is generated, we check if the RHS corresponds to one of the methods actually implemented in c and not part of S.

The achieved results show the feasibility of our approach, but also its limitations. Indeed, while FeaRS is able to generate meaningful recommendations for thousands of methods, several of them concern small methods that are not expected to substantially boost the developer's productivity.

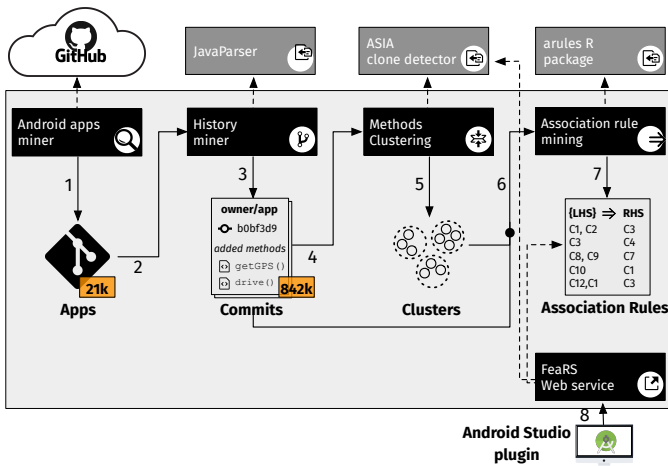## II. FeaRS

Fig. 2 depicts the inner working of FeaRS.

Fig. 2. The FeaRS pipeline

The black boxes represent components that we developed; the grey boxes depict external tools we reused and/or adapted.

All components except the Android Studio IDE plugin reside on a central server providing an access point via the *FeaRS Web service*. Steps 1-7 are executed offline and only once. Step 8 is executed every time the developer completes the implementation of a new method.

A. Mining Android Apps

The *Android apps miner* identifies GitHub repositories related to Android apps. Their history is then analyzed to identify methods implemented within the same commit. We use the GitHub APIs to search for repositories satisfying the following criteria:

They are written in Java. While Android is transitioning to Kotlin as the official language, the majority of Android apps is still written in Java [11]. Note that while we instantiated FeaRS to the specific problem of recommending complete methods for Java Android apps, all the steps in Fig. 2 can be customized to any programming language.

They are Android apps. We ensure that the repository contains a `build.gradle` file with an explicit dependency towards the Android SDKs, indicating the usage of the Gradle build system, the default choice in Android Studio.

They have a limited, but non-trivial change history. We excluded apps with *less than 100 commits* since we are interested in identifying the new methods added by developers within the same commit. Also, we excluded apps having *more than 1,000 commits*, since we do not want FeaRS to learn coding patterns peculiar only to a few apps.

The *Android apps miner* identified and cloned 20,713 GitHub repositories, the set of apps that we use in this work, available in our replication package [12]. The set can be expanded by re-running the Android apps miner.

B. Identifying Methods Added in Commits

The set of cloned repositories is provided as input to the *History miner* (step 2 in Fig. 2). This component extracts the list of commits performed in all branches of each repository by using the `git log --topo-order` command. This command allows analyzing all branches of a project without intermixing their history, avoiding unwanted effects of merge commits.

History miner uses JavaParser [13] to extract, from the Java files added or modified in each commit, the AST nodes which represent the callable declarations (*i.e.,* methods and constructors). In particular, we are interested in the callable declarations added in each commit. Commits not implementing at least two new methods and/or constructors are excluded at this stage, since we want FeaRS to learn implementation patterns in the form of $\{M\} \implies m_i$, where M represents a set of one or more methods and m_i a method that FeaRS can recommend based on the fact that the developer implemented M. Thus, assuming M to be a singleton, at least two new methods must be implemented in a commit (*i.e.,* the one in M and m_i) to make it useful for learning. We excluded commits adding more than 10 new methods (14% of the total number of commits), since these are likely to be tangled commits not representative of any specific implementation pattern [14].

Overall, we processed 2,721,800 commits, of which 841,995 were useful for building FeaRS (*i.e.*, those adding at least two new methods and no more than ten). These commits are provided as input to the module in charge of the methods clustering (step 4 in Fig. 2).

C. Clustering Similar Methods

To identify recurring implementation patterns in the considered commits, FeaRS applies clustering to group methods added in different commits, possibly from different systems, that implement equivalent or very similar functionalities. Two commits c_k and c_j performed in two different repositories may implement different sets of new methods (*e.g.*, $M_k = \{m_1, m_2\}$ and $M_j = \{m_3, m_4\}$) that represent the same implementation pattern (*i.e.*, $m_1 = m_3$ and $m_2 = m_4$). FeaRS can identify, through association rule discovery, that these sets of methods represent a repetitive implementation pattern.

FeaRS builds a weighted undirected graph. Each method added in any of the commits is considered as a node. The weight on the edges connecting each pair of nodes represents the similarity between the two corresponding methods. To assess similarity we use the publicly available ASIA clone detector [15], since it (i) is designed to capture the similarity between two Android methods; and (ii) returns as output an easily interpretable value from 0 (min similarity) to 1 (max). We customized the ASIA similarity algorithm in two ways.

First, in the original implementation all terms in the two methods to compare are lowercased before computing their textual similarity. This is suboptimal in FeaRS, since high precision in the identification of related methods is fundamental.

Experiments revealed that the similarity of methods is artificially boosted by lowercase transformation: Given two methods m_1 and m_2, it happens that a term appearing in the name of m_1 (*e.g.*, `date`) is matched with the type of an object appearing in m_2 (*e.g.*, `Date`). By not transforming `Date` to lowercase, the presence of these two terms does not influence positively the similarity between m_1 and m_2.

Second, while ASIA uses tf-idf (term frequency-inverse document frequency) as a weighting schema for the terms during the textual similarity computation, we only employ term frequency, because we noticed that a single term appearing in both methods and having a very high idf (*i.e.*, being very rare in the corpus) can result in a high similarity between the two methods, even if they implement completely different features. This is especially true in small methods, due to the low number of terms present in them and the strong impact a single shared term can have on their similarity.

The graph we built contains 2,018,479 nodes. We prune all edges with a weight below a threshold λ (λ will be tuned in our evaluation). This creates a set of disconnected subgraphs, each one representing a cluster of methods implementing strongly related functionalities. Within each subgraph (*i.e.*, cluster) we identify the cluster centroid: the method with the highest number of edges, which serves as representative for that cluster. The centroid is used later on by the FeaRS Web service when interacting with the IDE plugin.

D. Association Rule Mining

This module takes as input the list of commits generated by the *History miner* and the clusters output of the previous step (step 6 in Fig. 2) and creates a text file reporting in each line a set of methods added in the same commit and in the same file, using the cluster they belong to. For example, assuming a commit adding three methods m_1, m_2, and m_3 to a file F_i, and those methods being assigned to clusters C_{12}, C_8, and C_{71}, respectively, a line C_{12}, C_8, C_{71} will be added to the file. We decided to split methods added in the same commit but in different files to extract more "cohesive" association rules, and to avoid learning recommendations that span different files (*i.e.*, the developer is working on F_i and FeaRS recommends a method to add in F_j).

FeaRS analyzes the created file using Association Rule Mining [16] to identify implementation patterns, relying on the R `arules` package. We use the first 80% of the apps' commits to extract the association rules, 10% for tuning the parameters of FeaRS and 10% to evaluate it. The output is a set of association rules in the form $\{LHS\} \implies RHS$, where the LHS can be composed by one or more methods, while the RHS always has a single method. This means that FeaRS can only recommend the next method to implement given the one(s) already implemented by the developer.

There are three parameters that we tune in our evaluation: minimum *support* (sup), *confidence* for the mined rules (con), and maximum size of the LHS (max_{LHS}).

The support (sup) indicates how frequently a rule is observed in the dataset and, in our case, represents the percentage of analyzed commits that contains the specific rule.

The confidence (con) assesses how often a given rule is actually true in the dataset. Given a rule $\{LHS\} \implies RHS$, it is computed as the number of commits implementing in the same file all methods in the LHS and RHS divided by the number of commits implementing the LHS in the same file (with or without the RHS). Finally, we also tune the maximum size of the LHS (max_{LHS}).

E. The FeaRS Android Studio Plugin

Fig. 3 shows the FeaRS Android Studio IDE plugin.

The plugin interacts with the server through the Web service (step 8 in Fig. 2). The developer can start and stop FeaRS through simple ◉ and ■ icons in the IDE toolbar. By clicking ◉, FeaRS starts monitoring the code written by the developer and identifies when a new method is added. When this happens, the text of the new methods added by the developer since she pressed the start button is sent to the Web service.

The Web service identifies, for each received method, the cluster it belongs to. Our customized version of the ASIA clone detector computes the similarity between each received method and each centroid representative of the computed clusters. The similarity s for the most similar centroid is compared against a γ threshold (the fifth and last FeaRS parameter to tune): If $s > \gamma$, the method is assigned to the cluster represented by the most similar centroid, otherwise no match is found and the method is discarded.
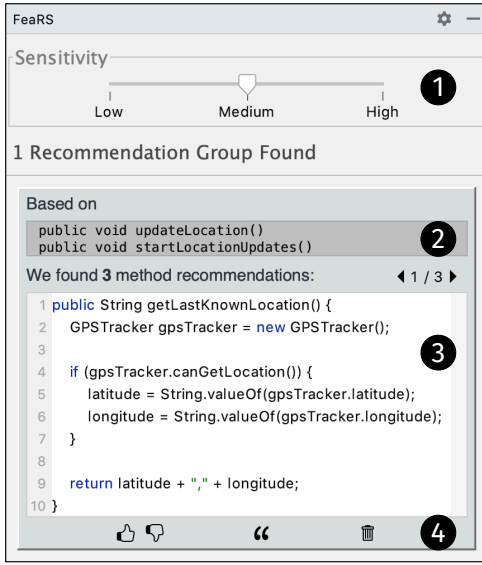
Fig. 3. The FeaRS Android Studio plugin

The slider at the top of the plugin GUI ① allows the developer to customize the "chattiness" of the plugin on three different levels. *Low*, *Medium*, and *High sensitivity* are three different FeaRS configurations that resulted from the calibration of its parameters presented in Section IV-A. By moving the slider towards *Low*, FeaRS becomes more strict and generates fewer, but higher quality, recommendations, while the opposite holds for *High*.

III. STUDY DESIGN

The goal of this study is to assess the performance of FeaRS when used to recommend the next method to implement given one or more (already implemented) methods as input. It thus addresses the following research question:

RQ$_1$: *What is the accuracy of FeaRS in recommending complete methods in the context of Android apps?*

A. Context Selection and Data Collection

We exploit the dataset of 20,713 Android apps as the context of our study. Fig. 4 depicts our experimental design.
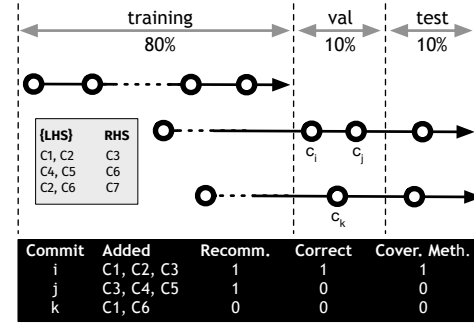

Fig. 4. Study Design

The black arrows represent the change history of the apps considered in our study. Note that the history of the apps is not aligned, meaning that not all the apps exist in the same time period. The vertical dashed lines show how we divide the change history of the apps.

We use the first 80% to extract the association rules used by FeaRS to generate recommendations. We refer to this subset of the history as the "training set". The subsequent 10% is used to tune the parameters of FeaRS to identify the best configurations (*i.e.*, "validation set"), which are used to generate recommendations on the "test set" (*i.e.*, the last 10%), with the goal of assessing the performance of FeaRS.

One important clarification: We do not use the first 80% of each repository as the training set, due to the misalignment of the mined change histories. Instead, given d_s the date of the oldest commit present in all analyzed apps and d_e the date of the most recent commit, we take the first 80% of the time interval going from d_s to d_e as training set. As shown in Fig. 4, this may result in some apps exclusively contributing to the training set (or to the validation/test sets). However, such a design is needed to avoid using "data from the future" when generating recommendations for the validation and test set and, thus, to simulate a real usage scenario for FeaRS.

All combinations of received methods that are matched with a centroid are used to generate different LHSs. For example, if three methods added by the developer are matched to clusters C_1, C_2, and C_3, we generate 7 possible LHSs: $\{C_1\}$, $\{C_2\}$, $\{C_3\}$, $\{C_1, C_2\}$, $\{C_1, C_3\}$, $\{C_2, C_3\}$, and $\{C_1, C_2, C_3\}$.

FeaRS checks if any of these LHSs is equal to the LHS of one of the association rules previously extracted. In case of a match, a recommendation is generated. In the reported example, if $\{C_1, C_2\}$ is matched in a rule $\{C_1, C_2\} \implies C_9$, then the centroid of cluster C_9 is returned by the Web service to the plugin as a recommendation. For the same LHS several different RHSs may be recommended. The matching of the LHS of two rules can lead to redundant recommendations. In the example, let us assume that two rules are matched, one with $\{C_1\}$ and one with $\{C_1, C_2\}$ as LHS, and that both of them have C_9 as RHS. In this case, the Web service returns the centroid of C_9 reporting that it is recommended based on the LHS belonging to the rule having the highest confidence.

The generated recommendations are shown in the IDE as depicted in the bottom part of Fig. 3. ② shows the signatures of the methods implemented by the developer that are part of the LHS of the association rule used to recommend the method shown in ③ (*i.e.*, RHS of the rule). In case several recommendations share the same LHS, the plugin displays them as one recommendation allowing developers to switch between different RHSs using the arrow buttons above ③. The buttons at the bottom of the code snippet ④ allow to: (i) provide a feedback reporting if the recommendation was useful; (ii) copy the snippet; and (iii) delete the recommendation. The feedback, in our current implementation, is stored but not used. We plan to use it in future to adjust the confidence of the recommendations. If the developer decides to copy the snippet, a comment documenting the GitHub repository from when the snippet has been taken is added to the code, so that the developer can check its reusability from a legal perspective.

Indeed, by selecting the first 80% of the history of each app to learn the association rules, it could happen that a given App_x has the last commit of training set made on date d_x, while for App_y the latest commit of its entire history comes on date d_y, with $d_y < d_x$ (*i.e.,* d_y is older than d_x). This would mean that association rules learned on d_x will be applied to generate recommendations for commits performed on date d_y (that will be part of the test set), thus using data from the future to learn how to trigger recommendations, something that cannot happen in a real usage scenario.

TABLE I
FeaRS PARAMETERS TUNING OPTIONS

Parameter	Experimented values
con	0.05, 0.20, 0.35, 0.50, 0.65, 0.80
sup	8.00E-06, 4.80E-05, 8.80E-05, 1.28E-04, 1.68E-04
λ	0.80, 0.85, 0.90, 0.95
max_{LHS}	1, 2, 3, 4, 5, 6, 7, 8, 9

Once the association rules are learned, we assess the performance of FeaRS on the validation set with different parameter configurations (Table I), for a total of 1,080 configurations. Given the number of mined commits, the minimum value of sup we experiment (*i.e.,* 8.00E-06) ensures that an association rule is learned from at least 5 commits to be considered valid.

In all combinations of parameters, we used $\gamma = \lambda$, meaning that the minimum similarity needed to cluster two methods together (*i.e.,* λ) is also the minimum similarity used when generating recommendations to assign a newly implemented method m to a cluster C (*i.e.,* γ, see Section II-E).

As shown in Fig. 4, to identify the best configuration(s) we use 10% of the apps change history (validation set). For each commit in the validation set (c_i, c_j, and c_k in Fig. 4) we match all newly added methods to the clusters that have been defined during the association rules extraction from the training set (using the same similarity threshold as for the clusters definition). This means that we simulated the scenario in which each of the added methods is written by the developer in the IDE, and the FeaRS plugin checks if the added method can be matched with any of the existing clusters (*i.e.,* if its similarity with one of the centroids is higher than γ). If a method is not matched, no further action is taken, while all matched methods are assigned to the corresponding cluster.

Fig. 4 represents our running example, in which the grey box on the left shows the association rules learned on the training set, and the black box at the bottom shows how performance is computed for each commit in the evaluation set. In the case of commit i, three added methods have been matched to clusters C_1, C_2, and C_3. Then, we compute all possible combinations of the matched clusters involving all but one of them. In the case of commit i, this means all possible combinations having length lower than three: $\{C_1\}$, $\{C_2\}$, $\{C_3\}$, $\{C_1, C_2\}$, $\{C_1, C_3\}$, $\{C_2, C_3\}$. Then, we check if any of those combinations match the LHS of one of the rules learned from the training set. In Fig. 4 the pair $\{C_1, C_2\}$ matches the rule $\{C_1, C_2\} \implies C_3$. This means that, assuming C_1 and C_2 to be written before C_3 (more discussion

on this assumption in our threats to validity), FeaRS would be able in a real usage scenario to successfully recommend the next method to implement (*i.e.,* the C_3 centroid). Thus, in Fig. 4, we count the number of recommendations generated by FeaRS (1), column "Recomm.", the number of correct recommendations (1), and the number of methods added in commit i that FeaRS would have potentially been able to recommend (1 out of 3), column "Cover. Meth.". Concerning commit j, it would match the rule $\{C_4, C_5\} \implies C_6$ generating one wrong recommendation (see Fig. 4). No recommendation would be triggered for commit k, since no matched rules are found.

There are two special cases that must be handled:

First, when multiple association rules have the same RHS (*e.g.,* assume $\{C_1\} \implies C_3$ and $\{C_2\} \implies C_3$ are both available in the set of learned association rules). In this case, both rules could be applied, for example, in the context of commit i in Fig. 4. However, considering both rules as successful would inflate the performance of FeaRS since, in a real usage scenario, if $\{C_1\} \implies C_3$ is applied, $\{C_2\} \implies C_3$ cannot be applied, since C_3 already exists.

Second, in case of a "circular dependency" between the LHS and the RHS of two rules, *e.g.,* $r_1 = \{C_1\} \implies C_3$ and $r_2 = \{C_2, C_3\} \implies C_1$. The LHS of r_1 matches the RHS of r_2, and the RHS of r_1 is contained in the LHS of r_2.

In theory both rules could be applied to commit i in Fig. 4, but the application of one rule would exclude the other in a real usage scenario. If we apply r_1, it means that C_1 has been implemented by the developer and it does not make sense to recommend it with r_2. Similarly, if r_2 is applied, this means that C_3 already exists, making r_1 useless.

In both cases we select the rule with the highest confidence.

B. Data Analysis

We assess the performance of each experimented configuration by computing the following metrics:

Recall: $recall = \frac{Comm_{cor}}{Comm_v}$, where $Comm_{cor}$ is the number of commits for which FeaRS generated at least one *correct* recommendation and $Comm_v$ is the set of commits mined in the validation set. A correct recommendation is not necessarily an exact match to the actual implemented code, but the similarity has to be above a certain threshold which is consistent with the predefined clusters. Recall indicates in how many commits FeaRS could be potentially useful for developers.

Precision: $precision = \frac{Comm_{cor}}{Comm_{rec}}$, where $Comm_{rec}$ is the number of commits for which FeaRS generated at least one recommendation (correct or wrong).

Cov$_{commits}$: $cov_{commits} = \frac{Comm_{rec}}{Comm_v}$. This metric indicates the percentage of commits from the validation set that could have triggered FeaRS to generate at least one recommendation (correct or wrong) for developers.

Cov$_{meth}$: $cov_{meth} = \frac{Meth_{cor}}{Meth_{Comm_v}}$, where $Meth_{cor}$ is the number of methods successfully recommended by FeaRS and $Meth_{Comm_v}$ is the total number of methods added in $Comm_v$. This coverage metric indicates the percentage of methods added in all commits from the validation set that could have been automatically generated by FeaRS.
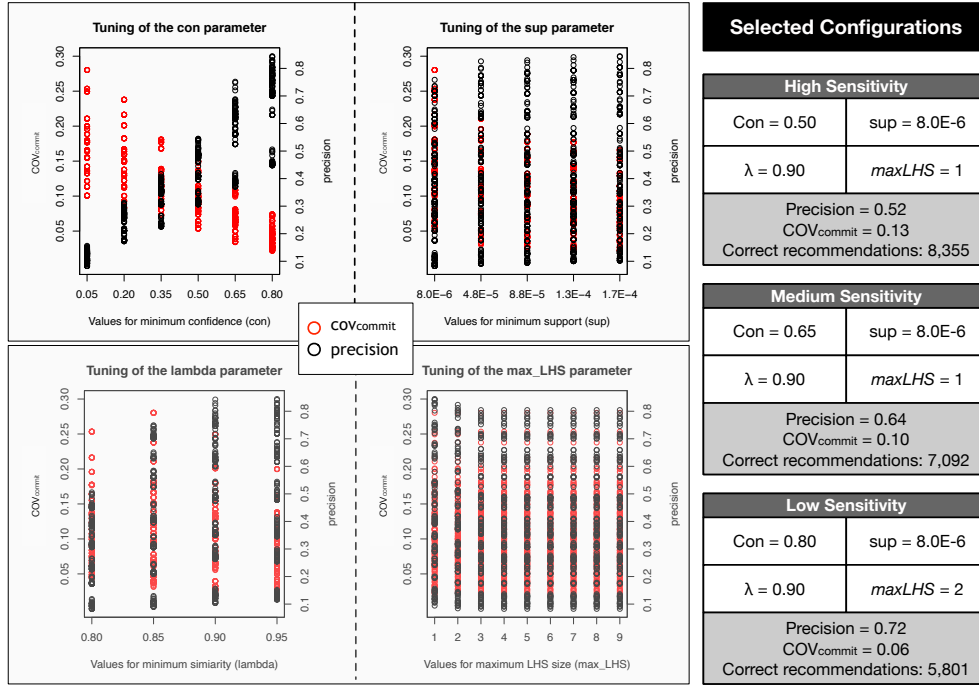
Fig. 5. Tuning of FeaRS's parameters

#Recom: $\#recom$ is the number of recommendations generated by FeaRS in a commit for which it was triggered. We report both the mean and the median values. This metric indicates how many recommendations developers might need to go through while using FeaRS.

Dist$_{tokens}$: $dist_{tokens}$ is the distance (in number of tokens that must be modified, added or deleted by a developer) when a developer receives a correct recommendation from FeaRS, which does not imply an exact match with the code actually implemented by the developer.

IV. RESULTS DISCUSSION

A. FeaRS Parameters Tuning

Fig. 5 shows the results of the parameters tuning performed on the validation set. Each of the four graphs reports on the x-axis the values experimented for a specific parameter; from left to right: minimum confidence (con), minimum support (sup), minimum similarity to cluster two methods (λ), and maximum size of the LHS (max_{LHS}). The y-axis reports the $cov_{commits}$ (left) and the precision (right) achieved, with red dots indicating $cov_{commits}$ values, and black dots precision values. We decided to use these two metrics, over the others, for the parameters tuning since we wanted to contrast the talkativeness of our tool (*i.e.,* in how many commits it generates a recommendation) against the precision of the generated recommendations. To better understand what the black and red dots represent, consider the con graph when its value is set to 0.05. The dots plotted in correspondence of this value represent the performance achieved when fixing $con = 0.05$ and varying all other parameters.

One first observation is related to the range of performance achieved by different configurations: The $cov_{commits}$ varies from 0.02 to 0.28, while the precision from 0.08 to 0.84. While the values of $cov_{commits}$ may look low, it is important to note that the validation set includes 70,562 commits. Thus, even a $cov_{commits}$ of 5% means that FeaRS would have been potentially useful in thousands of commits (>3k).

The trends observed for the four parameters indicate that con has the strongest influence on performance. When the minimum confidence needed to trigger a recommendation grows, as expected the precision linearly increases with a corresponding linear decrease of recall (left part of Fig. 5). Setting con lower than 0.50 does not ensure acceptable precision.

Concerning sup, increasing its minimum value does not substantially increase precision while having a strong negative effect on $cov_{commits}$. Low values of this parameter are preferable. Instead, increasing the λ parameter results in a notable increase in precision, especially when moving from 0.80 to 0.90/0.95. In this case, 0.90 seems to be a good compromise, also considering the minor loss of $cov_{commits}$ as compared to lower values. Finally, the max_{LHS} does not play a big role in the performance of FeaRS.

As the output of this tuning process, we identified three configurations that we linked to the sensitivity bar in our IDE plugin and that are shown in the gray boxes at the right of Fig. 5. These configurations have been picked according to the following process. We started from the assumption that a precision level below 0.50 (*i.e.,* one out of two generated recommendations is correct) is not acceptable. Then, we picked as a *high sensitivity* configuration the one ensuring a precision of at least 0.50 and having the highest $cov_{commits}$.

This configuration is able to generate 8,355 correct recommendations in the validation set, with a precision of 52%. Then, we increase the minimum acceptable precision by 10%, identifying the configuration ensuring at least a 60% precision with the maximum $cov_{commits}$. This resulted in the *medium sensitivity* configuration, that can successfully recommend useful methods in 7,092 cases, with a precision of 64%. Finally, a further increase of the precision level to at least 70%, led to the identification of the *low sensitivity* configuration, that can recommend 5,801 correct methods, with a precision of 72%. These three configurations are the ones we will experiment on the test set with.

B. Quantitative Results

Table II reports the results achieved by the three FeaRS's configurations on the test set. The top part of the table reports the raw data used to compute the performance metrics in the bottom part of the table. In the top part, while "#commits w. corr. recomm." indicates the number of commits with at least one correct recommendation, "#corr. recomm." represents the number of correctly recommended methods, possibly more than one per commit.

TABLE II
PERFORMANCE WHEN CONSIDERING ALL METHODS

	high sensit.	medium sensit.	low sensit.
#commits	69,480	69,480	69,480
#added methods	219,331	219,331	219,331
#commits w. recomm.	8,757	6,447	4,116
#commits w. corr. recomm.	4,878	4,167	3,110
#recommendations	14,642	9,996	7,170
#corr. recomm.	7,383	6,183	5,149
recall	0.07	0.05	0.04
precision	0.50	0.62	0.72
coverage$_{commits}$	0.13	0.09	0.06
coverage$_{meth}$	0.03	0.03	0.02
#recom(median)	1	1	1
#recom(mean)	1.67	1.55	1.74
distance$_{tokens}$(Q1,Q2,Q3)	0,1,2	0,1,2	0,1,2
distance$_{tokens}$(mean)	1.94	2.03	1.81
%distance$_{tokens}$(Q1,Q2,Q3)	0,13,22	0,13,22	0,13,20
%distance$_{tokens}$(mean)	14%	14%	13%

The results achieved by the three configurations are in line with what we observed on the validation set: precision goes from 0.50 (*high sensitivity*) to 0.72 (*low sensitivity*), with recall moves in an inverse direction, decreasing from 0.07 (*high sensitivity*) to 0.04 (*low sensitivity*).

The recall values, while low, still correspond to thousands of methods correctly recommended (>5k in the most conservative configuration). As we learned while performing the qualitative analysis reported in Section IV-C, a correct recommendation does not imply a "useful" recommendation. We noticed that many of the correct recommendations are due to small methods (*e.g.,* a getter method triggers the implementation of the corresponding setter), and decided to re-compute the performance of FeaRS only considering recommended methods with at least four lines of code (including signature but excluding annotations and the closing brace).

To correctly compute recall, this also required us to exclude from our analysis the commits in which a successful recommendation would not be possible at all, due to the absence of newly implemented methods having at least four lines.

TABLE III
PERFORMANCE WHEN EXCLUDING SHORT METHODS

	high sensit.	medium sensit.	low sensit.
#commits	31,088	31,088	31,088
#added methods	83,562	83,562	83,562
#commits w. recomm.	900	763	564
#commits w. corr. recomm.	568	536	413
#recommendations	1,329	1,099	738
#corr. recomm.	778	742	522
recall	0.02	0.02	0.01
precision	0.59	0.68	0.71
coverage$_{commits}$	0.03	0.03	0.02
coverage$_{meth}$	0.01	0.01	0.01
#recom(median)	1	1	1
#recom(mean)	1.48	1.44	1.30
distance$_{tokens}$(Q1,Q2,Q3)	0,3,10	0,3,10	0,3,4
distance$_{tokens}$(mean)	5.08	5.07	3.98
%distance$_{tokens}$(Q1,Q2,Q3)	0,14,28	0,14,28	0,10,18
%distance$_{tokens}$(mean)	17%	16%	13%

Table III reports the results achieved in this scenario. The precision values are in line with before (min: 0.59, max: 0.71), showing that the "quality" of the recommendations is not influenced by the length of the recommended methods. Instead, we observed a drop of recall, that does not go over 2%, with a number of correct recommendations ranging between 522 (low sensitivity) and 778 (high sensitivity).

The number of recommendations generated by FeaRS (#recom) is usually very low (median=1 and mean<2 in both scenarios). This shows that FeaRS does not generate many cases to inspect when triggered. Also, the results of distance$_{tokens}$ indicate that developers need to modify only a few tokens to adapt the received recommendations to their code.

While these results show the potential of FeaRS, they highlight (as in cases discussed for Table II), that the recommended methods are short, with a potential small benefit for developers. Our qualitative analysis will help in better assessing the value of these recommendations.

C. Qualitative Examples

1) Correct Recommendations: Fig. 6 shows an example of a recommendation generated for the Memento app for Android Wear [17].

Suppose that the developer implements the `isExternal-StorageReadable()` method to check whether the external storage of the device is mounted in read-only mode. FeaRS can pop up and recommend the `isExternalStorage-Writable()` method to check also if it is writable or not. This rule had four matching instances in our test set from four different repositories.

Fig. 7 shows an example of providing a custom back navigation for an Android `DrawerLayout`.

Following the implementation of an `onNavigation-ItemSelected(...)` method that uses a `Drawer-Layout`, FeaRS recommends a proper implementation for

Fig. 6. Correct recommendation to the usage of external storage in Android.

Fig. 7. Correct recommendation to provide a custom back navigation for an Android DrawerLayout.

the `onBackPressed()` method. Interestingly, in case of a missing implementation, the `DrawerLayout` might not close properly, as it is discussed in a Stack Overflow question [18]. We found 19 matching instances for this rule in 17 different repositories.

Fig. 8 shows an example a recommendation for the creation of a Google Map object from the Google Maps SDK for Android.

Fig. 8. Correct recommendation for the creation of a GoogleMap instance from the Google Maps SDK for Android.

We found 68 matches for this rule in 62 different repositories. FeaRS matches an `onCreate(...)` method in which an `Activity` creates a `SupportMapFragment` from the SDK. Next, it recommends an initial implementation for the `onMapReady(...)` method, that shows how to add a marker to the map. We found various implementations with the simple difference of the initial marker position (*e.g.,* London, Sydney).

2) Unmatched Implementation Patterns: We present FeaRS's recommendations that have been triggered during the evaluation process (*i.e.,* their LHS has been matched in the test commits) but that have never been successful (*i.e.,* the RHS has not been matched).

Fig. 9 shows an example of recommendation generated for the Artissans Android app [19].

Fig. 9. Unmatched recommendation for user credential validation in sign-up activity.

Suppose that the developer implements the `isValid-Email()` method to check whether the email address provided when creating an new account is valid. FeaRS recommends the `isValidPassword()` method to check, in the same scenario, if the provided password/confirm password fields are valid (*i.e.,* they are composed by at least six characters, and they match each other). This rule had been triggered two times without finding a match for the RHS, thus being classified as an incorrect recommendation. However, when we looked into the two commits in which this recommendation was triggered, we found that both of them actually implemented an `isValidPassword()` method that, however, only validated the password based on its length, do not making the recommended method and the implemented one similar enough to be counted as a correct recommendation. This example is representative of many others we found.

Fig. 10. Unmatched recommendation for creating custom filter for filterable adapter in Android.

For example, Fig. 10 relates to the creation of a custom filter applied to a `RecyclerView.Adapter` in Android. The class `Filter` is used in Android to constrain data according to a specified pattern. Following the implementation of a `UserFilter` constructor, FeaRS recommends a proper implementation of the overridden `publishResults` method from the `Filter` class that, as explained in the Android documentation, is *invoked in the UI thread to publish the filtering results in the user interface*. Again, this recommendation was not matched (and considered wrong) during our study, but also in this case looking into the test commit [20] subject of the recommendation, we found that a similar overridden `publishResults` method was implemented as well following a custom filter constructor. Unfortunately, also in this case the similarity between the RHS of the rule and the implemented `publishResults` was not high enough to automatically identify the recommendation as useful.

These cases show that our experimental design, while useful to provide a first indication about the quality of the recommendations triggered by FeaRS, may have resulted in an underestimation of FeaRS's performance. As previously said, only complementing this mining-based study with experiments with developers can help in better assessing FeaRS's usefulness.

V. THREATS TO VALIDITY

Construct validity. In our experimental design we assumed that if a commit added three methods belonging to clusters C_1, C_2, and C_3 and FeaRS has an association rule $\{C_1\} \implies C_3$, FeaRS would have been useful in that commit to recommend C_3 to the developer. However, we cannot know whether C_3 was written before C_1, thus making FeaRS's recommendation useless in practice. Such a threat can only be addressed by (i) performing a user study in which developers code live using FeaRS, or (ii) recording IDE interaction data of programming sessions. While this is part of our future work, we preferred as first evaluation for FeaRS something that can be large-scale and fully automated, before moving to more costly studies requiring human involvement. In the design of our study, we only consider coding activities from one single commit might perform an implementation task, while ignoring those cases in which a given task can be separated into several commits. Actually we considered the idea of using close commits as a single data point, but we found out that it is hard to define a proper criterion for the selection of multiple commits and it might be risky for the cohesiveness of the task.

Another threat is related to the criterion we used to identify a generated recommendation as "correct". Given a commit c in which m_i and m_j are added, we assume that a recommendation $C_k \implies C_s$ is correct if m_i is matched to an existing cluster C_k and m_j is matched to an existing cluster C_s (or *vice versa, i.e.,* m_i to C_s and m_j to C_k). This implies an assumption, meaning that the assignment of methods to cluster is correct or that, in other words, when a method is assigned to a cluster, the method actually implements functionalities related to those of the cluster. To partially address this threat, two of the authors manually analyzed a set of 100 methods

assigned by FeaRS to a specific cluster, with the goal of verifying whether the assigned cluster actually implements the same feature of the method. After solving conflicts arisen in 7% of cases, they reported an accuracy of 91%. Thus, we acknowledge possible imprecisions in 9% of cases.

Internal validity. We tuned the FeaRS's parameters on a set of commits not used for the learning of the association rules nor for assessment of FeaRS's performance. We experimented with 1,080 combinations of parameters. However, it is possible that better performance can be achieved by considering other possible values. Thus, from this point of view, the reported performance is an underestimation. We adopted a careful experimental design to avoid using "data from the future" when tuning and testing our approach.

External validity. Overall, our study involves a set of 841,995 commits from 20,713 open-source Android apps. The main issue is therefore related to the fact that all used apps are open source, and might not be representative of commercial apps. Also, while FeaRS is general enough to be adapted to other contexts (*e.g.,* Java programming in general), we decided to focus on a more narrow scenario at least for this first work.

VI. RELATED WORK

FeaRS is one of the many recommender systems proposed in the software engineering literature. The latter have been proposed to support many different tasks such as, for example, the recommendation of formal and informal documentation (see *e.g.,* [21]–[23]), the automatic generation of code for different purposes (*e.g.,* [24]–[29]), or the recommendation of relevant code examples/discussions for a task at hand (*e.g.,* [30]–[34]). We focus our discussion on the most related works, and in particular on those dealing with code completion techniques and code search engines.

A. Code Completion Techniques

Basic code completion features of IDEs often rely on the static type system of a programming language and do not consider the actual code context. Suggestions are usually sorted, *e.g.,* in alphabetical order. As a result, relevant recommendations are not always easy to identify.

An alternative approach was presented by Bruch *et al.* [3]. Their *intelligent code completion system* filters out candidates from the list of tokens recommended by the IDE that are not relevant to the current working context, and ranks candidates based on how relevant to the context they are.

Another context-sensitive approach was developed by Nguyen *et al.* [5]. Their *GraPacc* method uses graphs to model API usage patterns, where nodes represent actions (*e.g.,* method calls) and control points (*e.g.,* while), and edges represent control and data flow dependencies between nodes. Context information such as the relation between API elements and other code elements is considered for ranking the most fitted API usage patterns.

Statistical language models have also been used for code completion. In their seminal work on the naturalness of software, Hindle *et al.* developed a code completion engine

for Java, based on an n-gram language model [4]. Their work has been extended by Nguyen *et al.* [9] and Tu *et al.* [6].

A language model approach was implemented by Raychev *et al.* too [8]. They extract sequences of method calls from a large codebase to train a model, which they use to support the autocompletion of method calls, achieving an accuracy of 90% when considering the top three results. Method call completion was also explored by Asaduzzaman *et al.* [35]. Their approach, called CSCC, relies on a database of method call usage contexts collected from open source projects and applies a hash function to find relevant recommendations. From another perspective, Robbes and Lanza proposed to improve code completion by focusing on the recent changes implemented by the developer [7].

Popular IDEs have recognized the importance of supporting context-sensitive recommendations. For example, IntelliJ IDEA has a feature called *Smart completion* to filter and show suggestions applicable to the current context. NetBeans has a *Smart Code Completion* feature to display at the top of the suggestions the most relevant ones for the context. Eclipse has plugins to extend its core code completion, among these, *aiX Code Completer* [36] and *Codota* [37] use AI techniques and can even recommend a full line of code.

While these approaches are undoubtedly valuable to speed up code writing, they are limited to recommendations related to the next few tokens the developer is likely to type given the current context. In the best case, they can recommend a few APIs that the developer is likely to use next. With FeaRS we forge another step ahead, to predict the next full method a developer is likely to implement.

B. Code Search Engines

FeaRS is also related to approaches implementing code search engines that allow retrieving code samples and reusable open source code from the Web.

Early online code search engines (*e.g.,* code-search.google.com, koders.com, and krugle.org) offered keyword-based search and file-level retrieval. These approaches could be improved by considering structural and semantic information of code. Bajracharya *et al.* [38] developed Sourcerer, a code search engine that extracts structural information from the code and stores it in a relational model so it can be queried for code search. It supports queries for control structures, Java types, and micro patterns (*e.g.,* implementation of Semaphore). Reiss developed an approach to combine code search with transformations to map the retrieved code, to meet user specifications [39]. For the searching, it allows the user to specify multiple semantic rules, which also form the basis for the transformations.

Thummalapenta *et al.* developed an approach to support code search engines with static analysis to return fewer, but more relevant code samples for search queries [40], [41]. Their primary goal was to support a user in reusing a given API. Later they extend their approach with SpotWeb [42] to assist users by detecting hotspots that can serve as starting points for reusing APIs.

API usage was also proposed by McMillan *et al.* [43], [44] to return highly relevant matches for a source code search engine. Their approach combines three sources of information to locate relevant software: the textual descriptions of applications, the API calls used inside each application, and the dataflow among those API calls.

Compared to code search engines, FeaRS also relies on an extensive database of methods' source code in open source applications. These methods are organized in clusters based on a similarity algorithm implemented in the ASIA clone detector [15]. FeaRS does not require the user to write a "query" to identify relevant pieces of code, but extrapolates this need by monitoring the IDE.

VII. CONCLUSIONS

Code completion, while provenly useful and extensively used by developers [2] is just a step in the direction of an automated pair programmer, adding complete methods that a developer would have to add anyway and thus removing from the developer the burden of rote work. This was the ambitious goal that we set out to achieve with this work, embodied in the creation of FeaRS, an approach and a tool to automatically recommend to developers the complete next method to write during implementation activities.

FeaRS relies on a simple but intuitive idea: programming is an eclectic activity, which some even go as far as calling it "natural" [4]. What a developer is doing has a high chance of having been done by someone else, somewhere else before. Leveraging this idea, FeaRS mines vast amounts of data to recommend complete methods given a set of methods being implemented by a developer. We evaluated FeaRS on the change history of 20,713 Android apps. The results show the potential of FeaRS, with hundreds of correct methods recommended even in its most conservative configuration.

However, our findings are not conclusive for what concerns the actual usefulness of the generated recommendations in a real usage scenario, in which developers use FeaRS during coding activities. This is due to two observations we made. First, some of the methods recommended by FeaRS are quite short and, while they can still be useful, they could also represent "trivial" recommendation for developers. We believe this can in part be made up by introducing a user feedback loop, which is part of our future work. The quantitative results show that around 15% of the tokens from the recommendations need to be modified, added or deleted to fit the user's code base. One of our future plans is to integrate code adaption techniques into FeaRS to avoid potential conflicts or compilation errors with the user's code environment, and convert the coding convention into the user's style. Second, due to our experimental design, the "unmatched recommendations" are always considered false positives, while we observed that many are actually valuable recommendations. Thus, a deeper evaluation of FeaRS including a well-designed user study represents another main target of our future research.

Data Availability. FeaRS is available as an Android Studio plugin [12].

REFERENCES

[1] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[2] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[3] M. Bruch, M. Monperrus, and M. Mezini, "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*, ser. ESEC/FSE 2009, 2009, pp. 213–222.

[4] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.

[5] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 69–79.

[6] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 269–280.

[7] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[8] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2014, 2014, pp. 419–428.

[9] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR 2016)*, 2016, pp. 362–373.

[10] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 1995, pp. 3–14.

[11] R. Coppola, L. Ardito, and M. Torchiano, "Characterizing the transition to kotlin of android apps: a study on f-droid, play store, and github," in *Proceedings of the International Workshop on App Market Analytics*, 2019, pp. 8–14.

[12] "Replication package. https://github.com/anonymousfears/fears."

[13] "JavaParser. https://javaparser.org/."

[14] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 121–130.

[15] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza, "Automated documentation of android apps," *IEEE Transactions on Software Engineering*, 2019.

[16] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993.

[17] "Memento for Android Wear. https://github.com/inertia-besi-c/Memento-AndroidWear."

[18] "StackOverflow question. https://tinyurl.com/y7pge4l9."

[19] "Artissans Android app. https://github.com/Wess58/Artissans."

[20] "Artie Android app. https://github.com/manbradcalf/Artie-Android/commit/34ccfa3."

[21] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from stack overflow," in *Proceedings of ICSE 2016 (38th International Conference on Software Engineering)*, 2016, pp. 392–403.

[22] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proceedings of ASE 2013 28th IEEE/ACM International Conference on Automated Software Engineering)*, 2013, pp. 562–567.

[23] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," in *Proceedings of ICSE 2017 (39th International Conference on Software Engineering)*, 2017, pp. 94–105.

[24] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 832–837.

[25] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 25–36.

[26] C. Lezos, G. Dimitroulakos, I. Latifis, and K. Masselos, "Automatic generation of code analysis tools: The castql approach," in *Proceedings of the 1st International Workshop on Real World Domain Specific Languages*, ser. RWDSL '16. ACM, 2016.

[27] R. L. Glass, "Some thoughts on automatic code generation," *SIGMIS Database*, vol. 27, no. 2, p. 16–18, Apr. 1996.

[28] H. Liao, J. Jiang, and Y. Zhang, "A study of automatic code generation," in *2010 International Conference on Computational and Information Sciences*, 2010, pp. 689–691.

[29] N. K. Singh, *EB2ALL: An Automatic Code Generation Tool*. London: Springer London, 2013, pp. 105–141.

[30] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proceedings of RSSE 2012*. IEEE Press, 2012, pp. 85–89.

[31] P. Rigby and M. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of ICSE 2013*, 2013, pp. 832–841.

[32] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of SUITE 2011*. ACM, 2011, pp. 17–20.

[33] R. Holmes, R. Walker, and G. Murphy, "Strathcona example recommendation tool," *SIGSOFT Software Engineering Notes*, vol. 30, pp. 237–240, 2005.

[34] ——, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE TSE*, vol. 32, no. 12, pp. 952–970, 2006.

[35] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Context-sensitive code completion tool for better api usability," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 621–624.

[36] "aiX Code Completer. https://tinyurl.com/ydb2ux8x."

[37] "Codota. https://www.codota.com."

[38] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. ACM, 2006, p. 681–682.

[39] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, p. 243–253.

[40] S. Thummalapenta, "Exploiting code search engines to improve programmer productivity," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. ACM, 2007, p. 921–922.

[41] S. Thummalapenta and T. Xie, "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*, ser. ASE '07. Association for Computing Machinery, 2007, p. 204–213.

[42] ——, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 327–336.

[43] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, p. 475–484.

[44] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.