Empirical Software Engineering (2024) 29:33
https://doi.org/10.1007/s10664-023-10421-9

Using knowledge units of programming languages to recommend reviewers for pull requests: an empirical study

Md Ahasanuzzaman[1] · **Gustavo A. Oliva**[1] · **Ahmed E. Hassan**[1]

Accepted: 6 November 2023 / Published online: 29 December 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
Determining the right code reviewer for a given code change requires understanding the characteristics of the changed code, identifying the skills of each potential reviewer (expertise profile), and finding a good match between the two. To facilitate this task, we design a code reviewer recommender that operates on the knowledge units (KUs) of a programming language. We define a KU as a cohesive set of key capabilities that are offered by one or more building blocks of a given programming language. We operationalize our KUs using certification exams for the Java programming language. We detect KUs from 10 actively maintained Java projects from GitHub, spanning 290K commits and 65K pull requests (PRs). We generate developer expertise profiles based on the detected KUs. We use these KU-based expertise profiles to build a code reviewer recommender (KUREC). We compare KUREC's performance to that of seven baseline recommenders. KUREC ranked first along with the top-performing baseline recommender (RF) in a Scott-Knott ESD analysis of recommendation accuracy (the top-5 accuracy of KUREC is 0.84 (median) and the MAP@5 is 0.51 (median)). From a practical standpoint, we highlight that KUREC's performance is more stable (lower interquartile range) than that of RF, thus making it more consistent and potentially more trustworthy. We also design three new recommenders by combining KUREC with our baseline recommenders. These new combined recommenders outperform both KUREC and the individual baselines. Finally, we evaluate how reasonable the recommendations from KUREC and the combined recommenders are when those deviate from the ground truth. We observe that KUREC is the recommender with the highest percentage of reasonable recommendations (63.4%). Overall we conclude that KUREC and one of the combined recommenders (e.g., AD_HYBRID) are overall superior to the baseline recommenders that we studied. Future work in the area should thus (i) consider KU-based recommenders as baselines and (ii) experiment with combined recommenders.

Keywords Reviewer recommendation · Pull requests · Knowledge representation · Knowledge units · Java

Communicated by Gabriele Bavota.

✉ Md Ahasanuzzaman
ma87@cs.queensu.ca

Extended author information available on the last page of the article

1 Introduction

Code review has a prominent role in modern software development. Indeed, code review systems are baked into the most popular software development platforms (a.k.a, forges), including GitHub, GitLab, and BitBucket.

Code review comprises "a record of the interaction between the owner of a change and reviewers of the change, including comments on the code and signoffs from reviewers" Zanjani et al. (2016). The quality of a code review thus inherently depends on the selection of the reviewer. Yet, finding the right reviewer for a set of code changes is not always a trivial task. This is particularly true in the context of large-scale, distributed software development (Bacchelli and Bird 2013; Rigby and Bird 2013).

Mapping different expertises to individual developers is a key requirement for effective code review. Recent research studies have attempted to develop approaches to detect experts in specific topics (Montandon et al. 2019) and accurately represent expertise Dey et al. (2021). In this paper, we focus on a key facet of expertise, which is *programming language (PL) expertise*. Our rationale is that a piece of code involving concurrency is suitable to be reviewed by someone who has demonstrated experience in dealing with such type of code.

In order to capture the programming language expertise of developers, we introduce the notion of *Knowledge Units (KUs)* of programming languages. We define a KU as *a cohesive set of key capabilities that are offered by one or more building blocks of a given programming language* (see Fig. 1). For instance, some key capabilities offered by the *Exception* building block of the Java programming language include (i) creating custom exception classes and (ii) printing the stack traces of exceptions. To operationalize the definition of KUs for Java, we analyze the topics covered in Java certification exams. Our rationale is that exam topics are precisely aimed at evaluating the degree of one's expertise in applying the key capabilities that are offered by the building blocks of Java. To determine developers' expertise based on KUs, we detect the KUs that are present in the Java files that developers change in their commits.

This paper reports an empirical study on using the knowledge units of the Java programming language to recommend reviewers for GitHub pull requests (PRs). We start with a preliminary study in which we evaluate whether our operationalization of Java KUs can reveal software developers with different programming language expertise. To conduct this study, we detected and analyzed KUs from 10 real-world actively maintained Java projects from GitHub, spanning a total of 290k commits. Next, we applied the k-means algorithm to cluster developers based on the KUs that are present in their changed files. Running the algorithm resulted in the identification 71 different developer clusters of varying sizes. A closer inspection of the clusters indeed revealed developers with different KU expertise profiles.

Given the encouraging results from the preliminary study, we proceeded to build a KU-based reviewer recommendation systems (KUREC). KUREC works as follows. Given a certain PR, we construct a KU-based *development expertise profile* and a KU-based *review expertise profile* for every developer. More specifically, these profiles are created based on the KUs that a developer encountered during his prior code changes and code reviews. Next, a PR-specific *expertise score* is computed for every developer based on how well the KUs that appear in the changed files of the PR in question match the two aforementioned expertise profiles. This score aims to capture whether a candidate reviewer has proven experience with the KUs that are present in the changed files of the PR. Finally, the score gets a bonus depending on how recent the candidate's prior code changes and code reviews are. The intuition is that someone who has worked with concurrent code last week is probably more

suitable to review a PR involving concurrent code than someone who last worked with concurrent code 6 months ago. Finally, we simply rank candidate reviewers based on their expertise score and recommend the top ones. With KUREC in hand, we address the following three research questions:

RQ1: How Accurately can KUREC Recommend Code Reviewers in Pull Requests? We compare the performance of KUREC to that of seven other baseline recommenders, namely: (i) a commit-frequency-based recommender (CF) (Robbes and Röthlisberger 2013), (ii) a review-frequency-based recommender (RF) (Yu et al. 2014), (iii) a modification-expertise-based recommender (ER) Hannebauer et al. (2016), (iv) a comment-network-based recommender (CN) (Yu et al. 2016), (v) a library-expertise-based recommender (CORRECT) Rahman et al. (2016), (vi) a review-history-based recommender (CHREV) (Zanjani et al. 2016) and (vii) a workload-distribution-based recommender (SOFIA) Mirsaeedi and Rigby (2020). Recommender performance is computed using two measures: top-k accuracy and mean average precision (MAP). Our results indicate that:

We observe that KUREC is ranked one along with the top-performing baseline recommender (RF) in Scott-Knott ESD rank, and KUREC outperforms the remaining six baselines. We also note that KUREC has a more stable performance (smaller interquartile range) compared to RF, which is a desired property in practice (Adomavicius and Zhang 2014).

RQ2: Can KUREC be made more Accurate by Combining it with Existing Recommenders? Inspired by our prior work on adaptive heuristics (Malik and Hassan 2008), we combine KUREC with all other baseline recommenders to create three brand-new recommenders: a frequency-based one called AD_FREQ (given a PR, AD_FREQ triggers the baseline recommender that performed the best most of the time), a recency-based one called AD_REC (given a PR, AD_REC triggers the baseline recommender that performed the best in the prior PR), and a frequency/recency hybrid one called AD_HYBRID (given a PR, AD_HYBRID triggers the baseline recommender that performed the best in the prior 10 PRs). We refer to AD_FREQ, AD_REC, and AD_HYBRID as combined recommenders. Our results indicate that:

All three combined recommenders have similar performance. For top-k accuracy, the combined recommenders perform as well as the top baselines (RF and KUREC). For MAP, the combined recommenders outperform the top baselines. More specifically, combined recommenders achieve a median MAP that ranges from 0.43 to 0.60, which is 13.2%–21.1% higher than that of RF and 17.6%–48.3% higher than that of KUREC. Finally, the combined recommenders always outperform the baselines in both top-1 accuracy and MAP@1. We thus conclude that combining KUREC with other recommenders using a simple approach results in a more accurate recommender.

RQ3: How Reasonable are the Recommendations made by KUREC? To evaluate KUREC in RQ1 and RQ2, we compared its recommendations to a ground truth. This ground truth corresponds to the list of actual developers who reviewed the PRs. The results indicated that sometimes KUREC's recommendations differ from such ground truth. We thus wonder whether the recommendations made by KUREC (and our other studied recommenders) would have been reasonable choices in practice when those differ from what happened in reality. We consider a recommendation to be *reasonable* if the (top-1) recommended individual recently committed or submitted PRs that included the majority of the files from the PR in question. Our results indicate that:

KUREC is the recommender with the highest percentage of reasonable recommendations (63.4%). The AD_HYBRID combined recommender, in particular, strikes the best balance between sticking to the ground truth (best recommender from RQ2) and issuing reasonable

recommendations when those deviate from that ground truth (61.3%, third best recommender in this RQ).

Taking together the results from all RQs, we conclude that KUREC and AD_HYBRID are overall superior to the baseline recommenders that we studied.

The Main Contribution of our Paper are as Follows: (i) operationalizing the notion of knowledge units of programming languages using a systematic approach, (ii) defining a method to represent developers' programming language expertise using the knowledge units (KUs) of a programming language, and (iii) introducing a KU-based reviewer recommender (KUREC) and variations (e.g., AD_HYBRID) that are overall superior to the baseline recommenders that we studied. Future work in the area should thus consider KU-based recommenders as baselines and experiment with combined recommenders. A supplementary material package is provided online[1]

Paper Organization Section 2 defines KUs and presents our approach to detect them. Section 3 describes our data collection approach. Section 4 reports our preliminary study in which we analyze the potential behind KUs. Section 5 presents the motivation, approach, and findings of our three research questions. Section 6 discusses the implications of our study. Section 7 discusses related work. Section 8 describes the threats to the validity of our findings. Finally, Section 9 outlines our concluding remarks.

2 Knowledge Units (KUs)

2.1 Definition

Every programming language consists of *building blocks* that developers use to produce code. In the case of Java programming language, we consider the building blocks to be the Java fundamental *language constructs* (e.g., `if/else` statements, `try/catch` statements, and `synchronize` blocks) and *APIs* (e.g., Exception API, Concurrency API, and Generic and Collections API). Each building block offers a *set of capabilities*, which are essentially the things that a developer can do with that building block.

A Knowledge Unit (KU) is a cohesive set of key capabilities that are offered by one or more building blocks of a given programming language. Our definition includes the word "key" and our rationale is to avoid having KUs conceived around capabilities that are too specific or highly context-dependent. Figure 1 illustrates key capabilities that are associated with the Exception API (left-hand side) and the `try/catch` language construct (right-hand side).

2.2 Operational Definition

Certification exams of a programming language (e.g., Oracle Java SE and Java EE certification exams for the Java programming language) aim to determine the skills and knowledge of a developer in using the key capabilities offered by the building blocks of that language. Hence, we can say that *certification exams capture the KUs of a programming language*. For instance, one of the topics covered in the Java SE 8 Programmer II certification exam is "Concurrency". The subtopics of "Concurrency" that are covered in the exam include: (i) create worker threads using `Runnable`, `Callable` and use an `ExecutorService` to concurrently execute

[1] http://www.bit.ly/3bhSFux. The contents will be made available on a public GitHub repository once the paper is accepted.

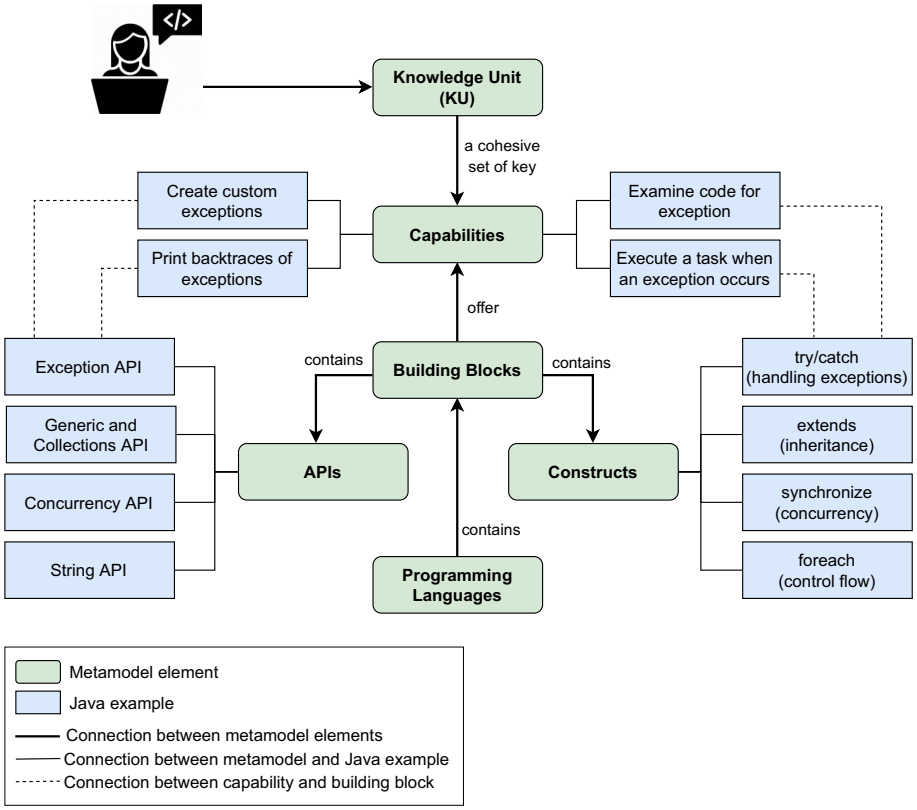


Fig. 1 Our metamodel for knowledge units (KUs)

tasks, (ii) use `synchronize` keyword and `java.util.concurrent.atomic` package to control the order of thread execution, (iii) use `java.util.concurrent` collections and classes including `cyclicBarrier` and `copyonWriteArrayList`, and (iv) use parallel fork/join framework. We interpret such a list of subtopics as the key capabilities that are offered by the "Concurrency" building block of the Java programming language. From this interpretation, we infer that Java has a "Concurrency KU".

In this paper, we operationalize our KUs via Oracle certification exams for the Java programming language. Oracle offers certification exams for different Java editions, such as Java Standard Edition (Java SE) and Java Enterprise Edition (Java EE). For Java SE, Oracle has two exam versions: (i) *Oracle Certified Associate, Java SE 8 Programmer I Certification exam* (Oracle 2022), and (ii) *Oracle Certified Professional, Java SE 8 Programmer II Certification exam* (Oracle 2022). For Java EE, Oracle offers only one version of the exam: *Oracle Certified Professional, Java EE Application Developer Certification exam* (Oracle 2022). In this study, we extract the list of KUs directly from the topics of the aforementioned certification exams. Each topic of each certification exam consists of a list of subtopics, and we interpret such subtopics as the key capabilities of the building blocks of the Java programming language. More specifically, to operationalize KUs we perform the following steps. First, we eliminate subtopics (from any exam) that are inappropriate for our research (i.e., subtopics that explain concepts and cannot be extracted automatically). Subtopics are then rearranged (both inside

Table 1 The identified Java programming language knowledge units (KUs)

Knowledge unit (KU)	Definition
[K1] Data Type KU	This KU describes the declaration and initialization of different types of variables (e.g., primitive type and parameterized type.)
[K2] Operator and Decision KU	This KU describes the usage of different Java operators (e.g., assignment, logical, and bit-wise operators) and conditional statements (e.g, if, if-else, and switch statements).
[K3] Array KU	This KU describes declaration, instantiation, initialization and the usage of one-dimensional and multi-dimensional arrays.
[K4] Loop KU	This KU describes the execution of a set of instructions/methods repeatedly using for, while, and do-while statements and the skipping and stopping of a repetitive execution of instructions and methods using continue and break statements.
[K5] Method and Encapsulation KU	This KU describes the creation of methods with parameters, the use of overloaded methods and constructors, the usage of constructor chaining, the creation of methods with variable length arguments, and the usage of different access modifiers. This KU also describes encapsulation mechanisms, such as creating a set and get method for controlling data access, generating immutable classes, and updating object type parameters of a method.
[K6] Inheritance KU	This KU describes developing code with child class and parent class relationship, using polymorphism (e.g., developing code with overridden methods), creating abstract classes and interfaces, and accessing methods and fields of the parent's class.
[K7] Advanced Class Design KU	This KU describes developing code that uses the final keyword, creating inner classes including static inner classes, local classes, nested classes, and anonymous inner classes, using enumerated types including methods and constructors in an enum type, and developing code using @override annotator.
[K8] Generics and Collection KU	This KU describes the creation and usage of generic classes, usage of different types of interfaces (e.g., List Interface, Deque Interface, Map Interface, and Set Interface), and comparison of objects using interfaces (e.g., java.util.Comparator, and java.lang.Comparable).
[K9] Functional Interface KU	This KU describes developing code that uses different versions of defined functional interfaces (e.g., primitive, binary, and unary) and developing code with user-defined functional interfaces.
[K10] Stream API KU	This KU describes developing code with lambda expressions and Stream APIs. This includes developing code to extract data from an object using peek() and map() methods, searching for data with search methods (e.g., findFirst) of the Stream classes, sorting a collection using Stream API, iterating code with foreach of Stream, and saving results to a collection using the collect method.

Table 1 continued

Knowledge unit (KU)	Definition
[K11] Exception KU	This KU describes the creation of try-catch blocks, the usage of multiple catch blocks, the usage of try-with-resources statements, the invocation of methods throwing an exception, and the use of assertion for testing invariants.
[K12] Date time API KU	This KU describes creating and managing date-based and time-based events using Instant, Period, Duration, and TemporalUnit, and working with dates and times across timezones and managing changes resulting from daylight savings, including Format date and times values.
[K13] IO KU	This KU describes reading and writing data from console and files and using basic Java input-output packages (e.g., java.io.package).
[K14] NIO KU	This KU describes how to interact files and directories with the new non-blocking input/output API (e.g., using the Path interface to operate on file and directory paths. This KU also includes other file-related operations (e.g., read, delete, copy, move, and manage metadata of a file or directory).
[K15] String Processing KU	This KU describes searching, parsing, replacing strings using regular expressions, and using string formatting.
[K16] Concurrency KU	This KU describes many functionalities that are related to thread execution and parallel programming. Some of these functionalities include: creating worker threads using Runnable and Callable classes, using an ExecutorService to concurrently execute tasks, using synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution, using java.util.concurrent classes and using a parallel fork/join framework.
[K17] Database KU	This KU describes the creation of database connection, submitting queries and reading results using the core JDBC API.
[K18] Localization KU	This KU describes reading and setting the locale (Oracle defines a locale as "a specific geographical, political, or cultural region") by using the Locale object, and building a resource bundle for each locale, and loading a resource bundle in an application.
[K19] Java Persistence KU	This KU describes the usage of object/relational mapping facilities for managing relational data in Java applications. With this knowledge, developers can learn how to map, store, update and retrieve data from relational databases to Java objects and vice versa.
[K20] Enterprise Java Bean KU	This KU is the knowledge about managing server-side components that encapsulate the business logic of an application.
[K21] Java Message Service API KU	This KU describes how to create, send, receive and read messages using reliable, asynchronous, and loosely coupled communication.

Table 1 continued

Knowledge unit (KU)	Definition
[K22] SOAP Web Service KU	This KU describes how to create and use the Simple Object Access Protocol for sending and receiving requests and responses across the Internet using JAX-WS and JAXB APIs.
[K23] Servlet KU	This KU describes handling HTTP requests, parameters, and cookies and how to process them on the server sites with appropriate responses.
[K24] Java REST API KU	This KU describes creating web s services and clients according to the Representational State Transfer architectural pattern using JAX-RS APIs.
[K25] Websocket KU	This KU describes how to create and handle bi-directional, full-duplex, and real-time communication between the server and the web browser.
[K26] Java Server Faces KU	This KU describes building UI component-based and event-oriented web interfaces using the standard JavaServer Faces (JSF) APIs.
[K27] Contexts and Dependency Injection (CDI) KU	This KU describes how to manage the lifecycle of stateful components using domain-specific lifecycle contexts and type-safely inject components (services) into client objects.
[K28] Batch Processing KU	This KU describes how to create and manage long-running jobs on schedule or on demand for performing on bulk-data, and without manual intervention.

and across tests) so that they fall under the most specific topic possible. Finally, each topic is converted into a KU (one-to-one mapping), and its subtopics are interpreted as the KU's main capabilities. To mitigate subjectiveness bias, the first two authors collaborated on the entire mapping process and discussed how to handle corner cases. At the end of this process, we identified 28 KUs (Table 1). Appendix A details the mapping process.

2.3 Detection of KUs

We employ a static analysis approach to detect KUs from a given Java project. In this study, we consider all the commits of a project spanning from the first commit to the latest one available as of our data collection period (i.e., January 20th 2022). To detect KUs, we first collect all the commits of a given project. For each commit C of the project, we checkout the commit C and then parse all the java source files that are available at the snapshot of C. We parse Java source files using the Eclipse JDT framework (Eclipse 2020). The JDT framework generates an Abstract Syntax Tree (AST) for the source code and provides visitors for every element of such tree, including *names*, *types* (class, interfaces), *expressions*, *statements*, and *declarations*.[2]

To detect a KU, we refer to the set of key capabilities of that unit (Table 10). For instance, to detect the presence of the *Generics and Collections KU* ([K8]), we use the visitors provided by JDT to detect the capabilities associated with that unit, such as [K8, C1] *create*

[2] The grammar employed by JDT can be seen at https://github.com/eclipse/eclipse.jdt.core/blob/master/org.eclipse.jdt.core/grammar/java.g

and use a generic class and [K8, C2] *create and use* `ArrayList`, `TreeSet`, `TreeMap`, *and* `ArrayDeque` *objects*. For instance, using type visitors, we can determine whether a given class is a generic class ([K8, C1]). Similarly, using statement visitors, we can find instantiations of an ArrayList ([K8, C2]).

During our code analysis, we also collect binding information of methods, classes, and variables (e.g., `org.eclipse.jdt.core.dom.IVariableBinding` resolves variable binding). In our study, we discard type bindings to third-party libraries because our focus is on studying the KUs that are associated with the source code that are written by the developers of the studied subject systems. Finally, we detect the presence of a KU if we can identify the usage of any of the key capabilities associated with that KU in a Java file.

We count the occurrences of KUs per Java source file. For example, one of the capabilities associated with the *Inheritance KU* ([K6]) is to create abstract classes ([K6,C4]). Hence, if we find a declaration of an `abstract` class in a given Java file `X.java` belonging to a subject system *S*, then the counter for *Inheritance KU* that is associated with file `X.java` is incremented by one. In other words, for each file, we produce a vector *V* containing the counts of each KU in that file.

3 Data Collection

Our data collection process contains three main steps. First, we retrieve the source code and pull-request (PR) data from real-world, active Java software projects. We refer to these projects as our *studied projects*. Next, we determine the incidence of KUs in the source code of all commits of our studied projects. Finally, we represent developers' expertise with KUs. Figure 2 presents an overview of our data collection process. We describe each step in more detail below.

- *Step 1) Collect commit data and pull-request data of the studied projects:*

– *Step 1.1) Select actively maintained Java software projects.* In our study, we focus on Java projects hosted on GitHub and that are both popular and actively maintained. However, there exist toy projects, personal projects, academic projects, and websites on GitHub that fulfil those criteria (Kalliamvakou et al. 2014). Hence, in order to select real-world software projects, we use the dataset of *engineered projects* made available by Munaiah et al. (2017). More specifically, we first select the projects from the list that are tagged with the Java language. Next, to study popular and active projects, we filter out projects that have less than 100 stars. We also filter out those that did not receive commits in the past year (relative to our data collection date, which is January 30th, 2022). Then, we sort the list of projects in decreasing order of the *average number of commits per month*. Finally, we select the top-10 projects from this list.
– *Step 1.2) Download the commit history of the selected Java software projects.* To download the commit history of our studied software projects, we `git clone` the repository of each project. The list of projects and their associated GitHub repository links can be found in our supplementary material package. Then, we use PyDriller (Spadini et al. 2018) to collect all the commit information from the projects, such as commit hash id, commit author name, commit author-date, commit message, and list of modified files in a commit. To retrieve the source code (snapshot) of a particular commit of a project, we use the command `git checkout commit_hash`, where `commit_hash` is the commit id.

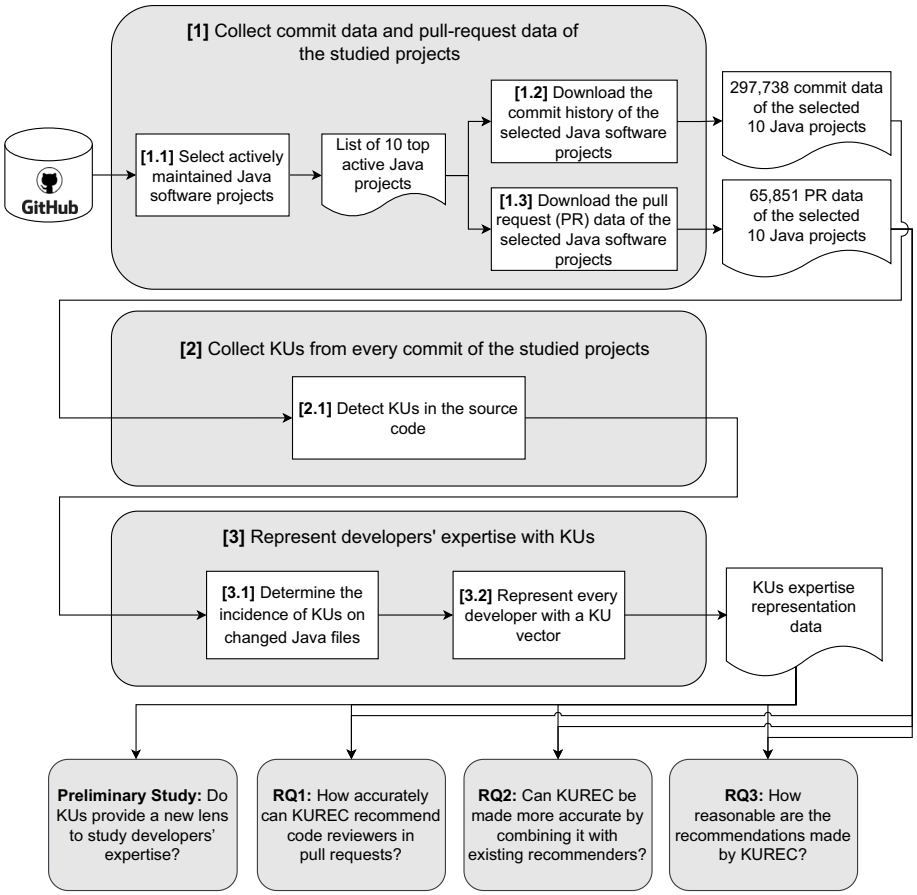


Fig. 2 An overview of our data collection process

– *Step 1.3) Download the pull request (PR) data of the selected Java software projects.* To download the PR data of the studied projects, we use the GitHub Rest API[3]. For example, to collect all the PRs of the Apache HBase project, we issue a `GET` request to `https://api.github.com/repos/apache/hbase/pulls?state=all&per_page=100`. We also use the GitHub API to collect review comments, discussion comments, changed files, and reviewer information for every PR. We use PRs and their associated metadata in RQ1 and RQ2.

- *Step 2) Collect KUs from every commit of the studied projects:*

– *Step 2.1) Detecting KUs in the source code.* We detect the KUs in each Java file of every commit (i.e., every snapshot) of the entire history of our studied projects. To detect KUs, we employ our custom detector built on top of Eclipse JDT (c.f., Section 2.3).

- *Step 3) Represent developers' expertise with KUs:*

[3] https://docs.github.com/en/rest/reference

– *Step 3.1) Determine the incidence of KUs on changed Java files.* First, we identify all the Java files that are changed in every commit by filtering in those files with a `.java` extension. Then, we collect the commit's author name (i.e., the developer who actually performed the code changes), such that we can map changed Java files to developers. Next, we count the occurrences of KUs in each of the changed Java files of every commit. **This count represents the KU expertise gained by the commit's author from having changed those Java files.** Our rationale for counting KUs from the entire file (instead of a diff) is that a developer often needs to understand the entire Java file in order to be able to modify it even a small portion of it. Finally, for every developer, we sum the count of the occurrences for every KU that is identified in each changed file of the developer (see Fig. 3).
– *Step 3.2) Represent every developer with a KU vector.* We represent the expertise of every developer with a 28-position KU vector. Each position corresponds to a different KU (Table 1). In each vector position, we store the ratio of the incidence of a KU. We calculate a developer's ratio of the incidence of a KU by dividing the sum of the occurrences of a KU detected from the changed files of a developer to the total occurrences of the KU detected from all the changed files (see Fig. 3b - for simplicity, we only show three KUs). At the end, we have a KU vector for every developer. This KU vector captures a developer's Java KU expertise. We henceforth refer to this vector as $P_{ku}(D)$, where D is some studied developer.

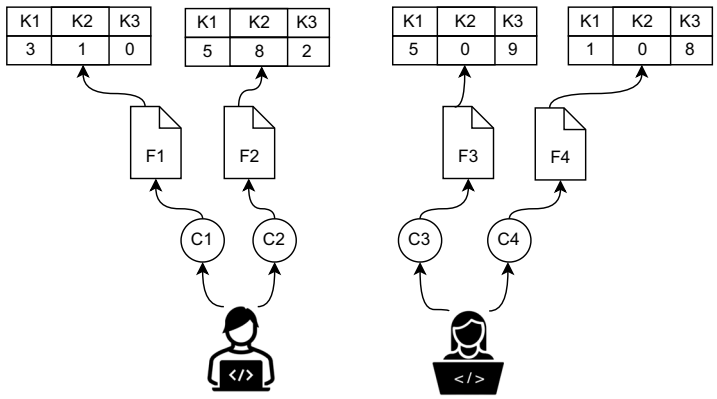
Summary

- Data collection date: January 20th, 2022
- Data source: GitHub
- Number of studied Java projects: 10
- Number of studied commits: 297,738
- Number of pull requests: 65,851
- Pieces of data collected: java source code and pull requests
- Datasets produced: developer expertise representation with KUs

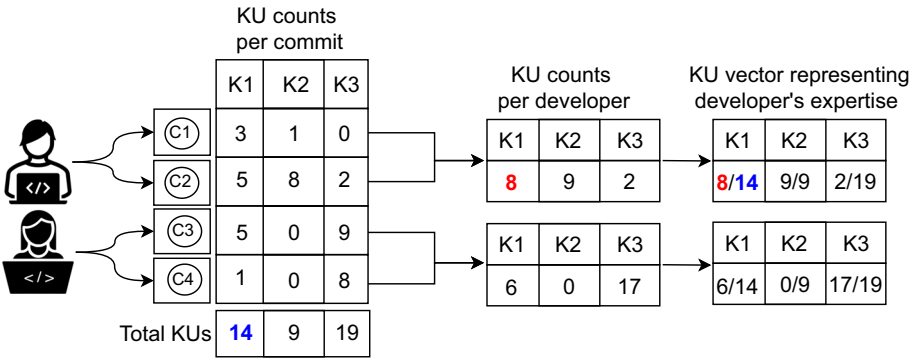
4 Preliminary Study: Do KUs Provide a New Lens to Study Developers' Expertise?

Motivation: Our KU-based expertise profile is derived from the key capabilities of the Java building blocks. In this study, we want to group developers according to their KU-based expertise profiles and study such groups of developers. Finding several groups with unique KU profiles can indicate that KUs might be useful for code reviewer recommendations.
Approach: We first cluster developers based on their KU-based expertise profile. Next, we study the dispersion of clusters' size. In the following, we detail each of the steps that we follow for this study.

- *Step 1) Cluster developers based on their KU-based expertise profiles.* We take the 28 dimensions of the KU-based expertise profile $P_{ku}(D)$ of every developer D across all studied projects and glue them together to build a matrix P_{ku} (the procedure for building $P_{ku}(D)$ is described in the step 3 of Section 3). To cluster developers according to their KU-based expertise profile, we apply the k-means clustering algorithm to a dimentionality-reduced version of P_{ku}. K-means is one of the most popular clustering



(a) KU counts of the changed files of every commit



(b) KU expertise representation

Fig. 3 Representing developers' expertise with KUs. Here, K_i denotes i_{th} knowledge unit, C_j denotes j_{th} commit and F_j denotes a changed file in C_j

algorithms and has been widely used in software engineering research (Bishnu and Bhattacherjee 2012; Yoon et al. 2007; Jureczko and Madeyski 2010; Zhang et al. 2016). We detail the sub-steps of the clustering procedure below.

– *Step 1a) Reduce the dimensionality* P_{ku}. Due to *the curse of dimensionality*, many clustering algorithms, including the k-means algorithm, perform poorly on high dimensional data Parsons et al. (2004). In simple terms, as the number of dimensions in a dataset increases, distance measures become less meaningful, since the distance between any pair of points in a high-dimensional space is almost the same over a wide range of data distributions and distance functions (Li et al. 2003). To overcome the dimensionality issue, a dimensionality reduction technique (e.g., Principal Component Analysis – PCA) can be applied to the data prior to running the clustering algorithm (Google 2020; Goebl et al. 2014). Since PCA has been proven beneficial for the k-means algorithm (Ding and He 2004), we reduce the dimensionality of our data by applying PCA prior to running k-means.

To apply PCA, we use the `prcomp`[4] native function of the R language. PCA transforms the data to a new coordinate plane, and each co-coordinate of that plane is called a principal component. PCA generates as many principal components as the number of dimensions of the data. The first principal component has the largest possible variance in the data, and each successor component has as much of the remaining variability as possible. Similarly to prior studies (Xibilia et al. 2020; Ghotra et al. 2017; Kondo et al. 2019), we reduce data dimensionality by selecting the first-n PCA components that account for at least 95% of the explained variance of the data.

– *Step 1b) Identify optimal number of clusters.* The k-means algorithm takes the number of clusters as an input parameter (K). To identify the optimal number of clusters, we analyze the *silhouette* coefficient (Rousseeuw 1987). The silhouette coefficient is a popular measure that has been widely used in prior studies (Nidheesh et al. 2020; Al-Subaihin et al. 2016; Panichella et al. 2013) for determining the optimal number of clusters. The silhouette coefficient indicates how similar an object (i.e., developer) is to its own cluster (cohesion) in comparison to other clusters (separation). The silhouette coefficient ranges from -1.0 to 1.0. A coefficient close to one indicates that the object well-matches its own cluster and, simultaneously, poorly-matches its neighboring clusters. If most objects have a high coefficient, then the clustering configuration is deemed appropriate.
We run k-means for K values ranging from 2 to 100 and calculate the *median silhouette* for each K value. Our rationale for using the median in lieu of the mean is because the former is more robust to outliers (e.g., a clustering with only few very badly clustered objects should still be considered a good clustering overall) (Rousseeuw 1987). Next, we select the largest K among those that yield a clustering with a median silhouette of 0.90 or higher. We avoid taking the K that maximizes the median silhouette because it might be much lower than some other K with an almost identical median silhouette. Our approach thus prioritizes finding more clusters instead of obtaining just marginally better clusterings (while still ensuring that the obtained clustering are appropriate due to the high threshold for median silhouette).
– *Step 1c) Create clusters of developers.* To cluster developers using the k-means algorithm, we employ the native `kmeans` function of R language[5]. The input dataset corresponds to the selected PCA components in Step 1a. The number of clusters is set based on the results of Step 1b. Finally, we cluster developers using the k-means algorithm and visualize the clusters using the `fviz_cluster` function of the `factoextra` R package (Kassambara and Mundt 2017).

• *Step 2) Characterize developer clusters using KU-based expertise profiles.* To characterize developer clusters using KU-based expertise profiles, we follow two steps. First, we calculate a $diff_{value}$, which we define as the difference between the median value of a given KU within a cluster and the median value of the given KU across all the clusters (i.e., the entire dataset). The $diff_{value}$ thus indicates how the median value of a KU within a cluster compares to the median value of that KU across all clusters. The intuition behind $diff_{value}$ is to determine whether the values of a given KU K in C are either high or low (in relative terms). Formally, we define $diff_{value}$ as follows:

$$diff_{value}(P_{ku}, C, K) = median(Subset(P_{ku}, C, K)) - median(Subset(P_{ku}, K))$$

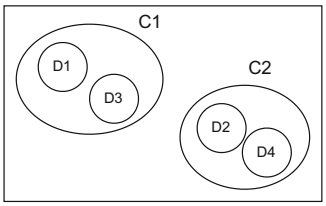
[4] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/prcomp

[5] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/kmeans

Here, $diff_{value}(P_{ku}, C, K)$ returns the $diff_{value}$ for a given cluster C and KU K from the KU-based expertise profile dataset P_{ku} (see Step 1). The $Subset(P_{ku}, K)$ corresponds to the column in P_{ku} that matches the given KU K. The $Subset(P_{ku}, C, K)$ filters $Subset(P_{ku}, K)$ by selecting only those rows that are associated with developers from the cluster C. Figure 4 presents an example scenario of the calculation process of $diff_{value}$.

Next, we plot the $diff_{value}$ of every KU of each cluster in the form of a bar plot. The interpretation of the bar plot is straightforward. A very narrow bar of a KU within a cluster indicates that the median value of the KU in the cluster is very close to the overall median of the KU. Conversely, a very wide bar of a KU within a cluster indicates that the median of the KU in the cluster is very different from the overall median. Hence, wide bars call for special attention. To determine if a bar is too wide, we take $median(Subset(P_{ku}, C, K)$ and determine whether it is outside the interquartile range (Q1 to Q3) of $Subset(P_{ku}, K)$. In the positive case, the bar is emphasized with a vivid (non-transparent) color.

- *Step 3) Study the dispersion of clusters' size.* We calculate the Gini index (Yitzhaki 1979) for the distribution of cluster sizes. The Gini index is a measure of statistical dispersion among values of a frequency distribution. This index ranges from 0.0 to 1.0, with 0.0 representing perfect equality and 1.0 representing perfect inequality. A Gini index close to one indicates greater inequality in the size of the clusters (e.g., a single cluster containing most of the observations).



(a) Developer clusters with KU-based expertise profile. Devleopers D1 and D3 are in Cluster C1. Developers D2 and D4 are in Cluster C2

P_{ku}

	K1	K2
D1	0.1	0.3
D2	0.2	0.1
D3	0.4	0.2
D4	0.1	0.1

$Subset(P_{ku}, C1, K1)$

	K1
D1	0.1
D3	0.4

$Subset(P_{ku}, C1, K2)$

	K2
D1	0.3
D3	0.1

$Subset(P_{ku}, K1)$

	K1
D1	0.1
D2	0.2
D3	0.4
D4	0.1

$Subset(P_{ku}, K2)$

	K2
D1	0.3
D2	0.1
D3	0.2
D4	0.1

$diff_{value}(P_{ku}, C1, K1)$ = median (Subset(P_{ku}, C1, K1)) - median (Subset(P_{ku}, K1))
= median (0.1, 0.4) - median (0.1,0.2, 0.2, 0.2)
= 0.25 - 0.15
= 0.10

(b) The calculation process of $diff_{value}$

Fig. 4 An example scenario of the calculation process of $diff_{value}$

Findings: Observation 1) KU-based expertise profiles form 71 different clusters of developers, indicating that KUs offer a fine-grained lens to study developers' expertise. Figure 5 presents a visualization of the 71 developer clusters that are formed with KU-based expertise profiles. We observe that the generated clusters are separated from each other. We also observe that the number of cluster members are different across the clusters. The Gini index for clusters' size is 0.96, indicating that there exists very strong inequality (i.e., a few clusters contain most of the developers). Indeed, we observe that the largest cluster contains 80% of all developers. That is, the majority of the studied developers have similar KU expertise. To understand the important KUs of this largest cluster, we sort the *diffvalue* of KUs within the cluster in descending order. We observe that the top three KUs are: Array KU, Loop KU and Advanced Class Design KU. Such a result indicates that 80% of all developers contribute mostly to the code changes that are related to Array KU, Loop KU and Advanced Class Design KU.

We also observe that 57% (41 out of 71) of the generated clusters are singleton clusters (i.e., the size of these clusters is one). These singleton clusters are special because the single developer in them has a unique KU-based expertise profile compared to all other studied developers.

Observation 2) Clusters generated from KU-based expertise profiles reveal developers with different code expertise. We observe that each cluster has its own intrinsic characteristics derived from the incidence level of each KU. To illustrate our point, we randomly select six clusters and visualize their characteristics in Fig. 6. The vivid colored KU bar in a cluster indicates that the median value of the KU in that cluster is outside the interquartile

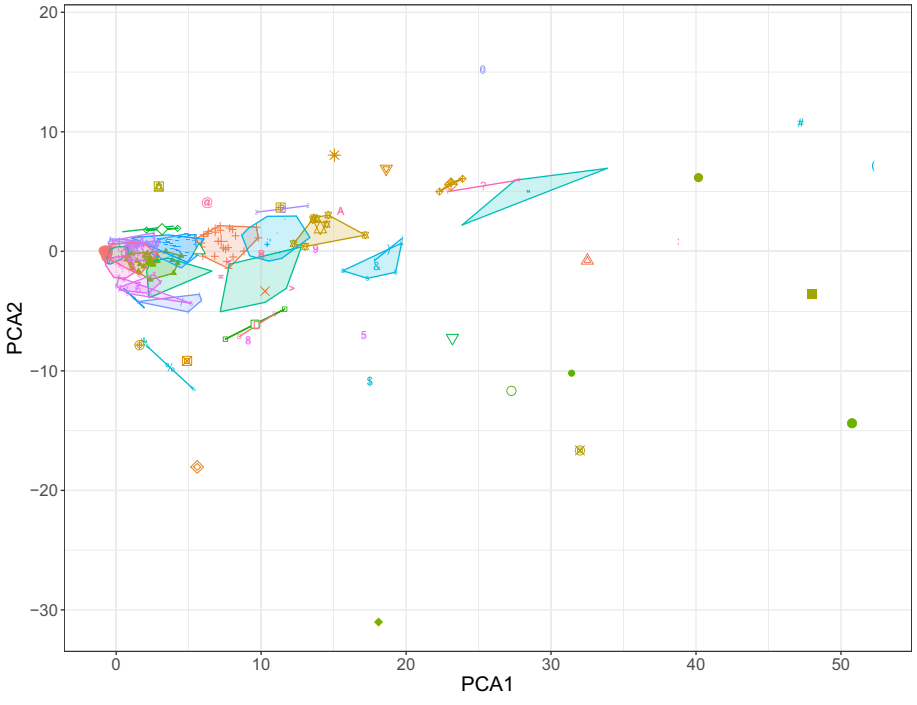


Fig. 5 Visualization of developer clusters with KU-based expertise profiles. Polygons represent clusters and symbols represent developers

range (Q1 to Q3) of the median of the KU in the entire dataset. Hence, these KUs with wide bars indicate important characteristics of the cluster. The wider a vivid coloured KU bar is, the more prominent the KU is among the developers of that particular cluster. As we observe Fig. 6, the "Stream API KU" is wider among the vivid colored KU bars in Cluster-18. As such, the "Stream API KU" is prominent among developers of Cluster-18 and an important characteristic. Similarly, "Java NIO KU" is prominent in Cluster-22, "Rest API KU" is prominent in Cluster-27, "Functional Interface KU" is prominent in Cluster-28, "Database KU" is prominent in Cluster-55 and "Date Time API KU" is prominent in Cluster-65. Such a prominent KU of a cluster indicates that developers of that cluster encounter the KU more frequently than other developers while performing code changes. Hence, developers of the cluster have more expertise in code changes related to that particular wide vivid coloured KU. These findings show that clusters vary from one another and each cluster reveal developers with different code expertise.

We further study for these above-mentioned KUs to understand what percentage of developers have a good experience at these KUs. We define that someone has a good experience with a KU if the developer's KU is above the median KU of the entire dataset. Then, we calculate the percentage of developers who have a good experience at the above-mentioned KUs. The percentage of developers who have a good experience at those KUs is as follows: Stream API KU:31.9%, Java NIO KU:14.8%, Rest API KU:2.9%, Functional Interface KU:11.2%,

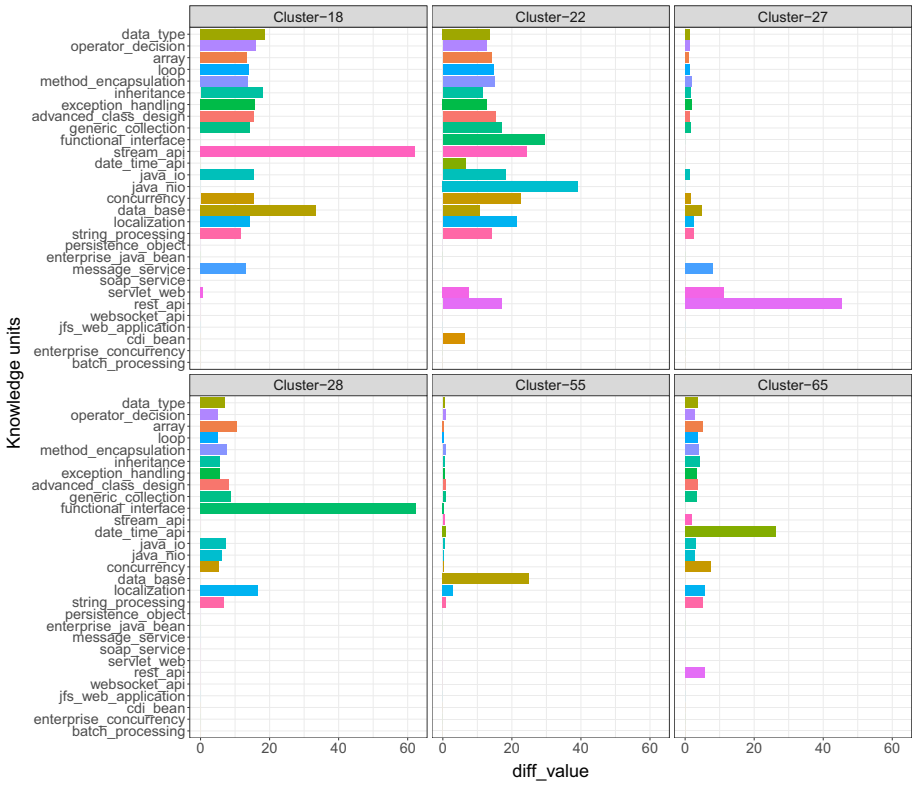


Fig. 6 Visualization of the characteristics of the identified clusters via knowledge units

Database KU:9.9%, and Date Time API KU:13.5%. We observe that a few developers are good at Rest API KU.

Summary

Do KUs provide a new lens to study developers' expertise?

Yes. Through the lens of KUs, we can identify 71 different clusters of developers. Each cluster hosts a set of developers with unique KU-based expertise profile.

5 A Study of Code Reviewer Recommendation Using KUs

In the following, we address our three research questions. For each research question, we discuss our motivation for studying it, the approach used to answer it, and the findings that we observe.

5.1 RQ1:How Accurately can KUREC Recommend Code Reviewers in Pull Requests?

Motivation: As we discuss in Section 1, mapping different expertises to individual developers is a key requirement for effective code review assignment. As we show in the preliminary study, KUs can capture developers' programming language expertise. Hence, in this research question we build a KU-based code reviewer recommender system called KUREC and evaluate its performance. KUREC takes into the code contributions (commits) and prior code reviews performed by an individual when issuing reviewer recommendations.

We clarify that our goal is *not* to design the most accurate code recommender system possible, but rather to determine where its performance stands in comparison to solid baselines. Such a rationale derives from the novelty surrounding the conceptualization and operationalization of KUs.

Approach: We use pull requests to train and test both KUREC and other seven baseline recommenders. **The key assumption of KUREC is that the code reviewers of a given PR are most likely experts in the KUs that appear in the changed files of the given PR.** Our approach for building and evaluating KUREC is summarized in Fig. 7. In the following, we describe each step of our approach in detail.

- *Step 1) Filter the PR dataset.* In Section 3, we discussed how we collected pull request (PR) data. Since our objective is to build a recommender for *code* reviewers, we only select those PRs that (i) include changed Java files, (ii) had at least one reviewer assigned to them, and (iii) are in a `closed` status. To ensure that we have enough PR data for training and testing a recommender, we only select those studied projects that contain at least 100 PRs. As a result of applying this filter, we remove the Cdap and Jruby projects from our study. As a result, we obtain a PR dataset for eight projects. Table 2 presents the statistical summary of our PR dataset.
- *Step 2) Generate train and test PR data.* KUREC is essentially a history-based recommender. Many prior studies (Yu et al. 2016; Strand et al. 2020; Ying et al. 2016; Al-Zubaidi et al. 2020) that also build history-based recommenders also use "training/testing" terminology. Therefore, we opted to use this same type of terminology in our study. Here, "training" terminology in a history-based recommender (e.g., training a recommender)

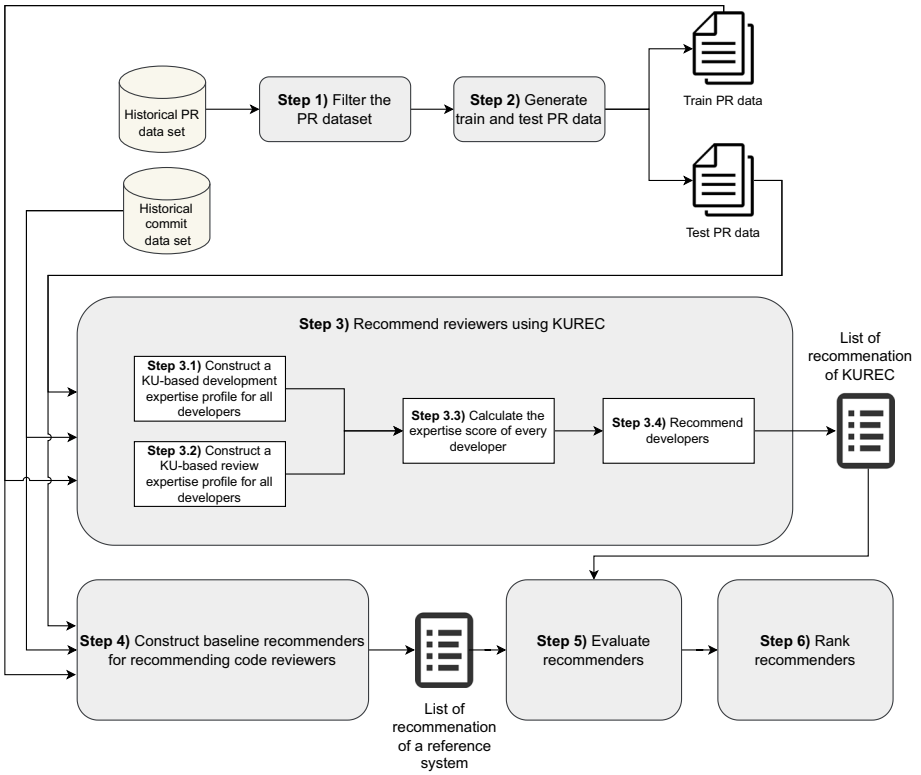


Fig. 7 Our approach for building and evaluating KUREC

indicates that the recommender develops a set of rules or algorithms for making recommendations using historical data.

We need to split the PR data for training and testing the recommenders. To split the PR dataset of a given project, we first sort the dataset chronologically. Our rationale is to avoid training recommenders with future data and testing those projects with past data (data leakage problem (Campos et al. 2011)). Then, in accordance with several prior studies (Xia et al. 2017; Yu et al. 2016; Ouni et al. 2016), we split each dataset into a training dataset containing the initial 80% of the data and a testing dataset containing the remaining 20% of the data.

- *Step 3) Recommend reviewers using KUREC.* We first use the train data to build KU-based development expertise profile and KU-based review expertise profile for every developer. To recommend reviewers of a test data P, we generate expertise score for every developer using the KU-based development expertise profile and KU-based review expertise profile. Then, we recommend reviewers for P. After recommending developers (i.e., reviewers) for P, we update the KU-based development expertise profile and KU-based review expertise profile of developers with P. We perform the following substeps to build KUREC and get its recommendations:

  – *Step 3.1) Construct a KU-based development expertise profile for all developers.* We first select all commits that are performed prior to the opening date of the PR in question.

Table 2 The statistical summary of the PR dataset

Project	Number of PRs	Number of unique reviewers
Apache ActiveMQ	725	17
Apache Groovy	1,660	31
Apache Lucene	520	44
Apache Hbase	3,918	107
Apache Hive	2,817	106
Apache Storm	3,424	51
Apache Wicket	447	16
Elastic Search	35,899	151

Then, we follow the same approach that we present in Step 3 of Section 3 to construct a KU-based development expertise profile for every developer. The collection of all profiles is represented with a matrix $DevExpKU(D_i, KU_j)$, where D_i refers to a developer i that performed at least one of the selected commits and KU_j refers to one of the 28 KUs listed in Table 1.

– *Step 3.2) Construct a KU-based review expertise profile for all developers.* We first select all those PRs that were created prior to the given PR. Then, we collect all changed files that are requested for review in each of the selected PRs. Next, we proceed analogously to Step 3.1, which results in the construction of a KU-based reviewer expertise profile for every developer. The collection of all these profiles is represented with a matrix $RevExpKU(D_i, KU_j)$, where D_i refers to a developer i that performed at least one code review for the selected PRs and KU_j refers to one of the 28 KUs listed in Table 1.
– *Step 3.3) Calculate the expertise score of every developer.* A developer who recently read or worked on a piece of code is likely to have fresher and clearer memories about the properties of such code (including the KUs that appear therein) (Lewis et al. 2013). It is thus not surprising that the recency of code changes and reviews has been taken into account by several reviewer recommenders (Zanjani et al. 2016; Ouni et al. 2016). KUREC also takes recency into account to calculate the final expertise score of every developer.

Following our terminology, let $DevExpKU(D_i)$ be the KU-based development expertise profile of a developer D_i. KUREC adds a *development recency bonus* to each KU_j in $DevExpKU(D_i)$ by (a) determining the last time that D_i committed a file containing KU_j and (b) judging how recent that is. More specifically, let C be the last commit by D_i that includes a file with KU_j. The recency bonus of KU_j for D_i is calculated as the *inverse of the number of days between the commit date of C and the opening date of the PR in question*. Formally, we represent the development recency bonus of KU_j for developer D_i as $RecDevKU(D_i, KU_j)$.

Analogously, KUREC also adds a *review recency bonus* to each KU_j in $RevExpKUs(D_i)$. Let R be the last code review by D_i in which some reviewed file contained KU_j. In this case, the recency bonus of KU_j for D_i is calculated as the *inverse of the number of days between the opening date of the PR that underwent R and the opening date of the PR in question*. We represent the review recency bonus of KU_j for developer D_i as $RecRevKU(D_i, KU_j)$. Lastly, KUREC attributes a final KU-based expertise score $ExpertiseScore(D_i)$ to each

developer as follows:

$$DevScore(D_i) = \sum_{j=1}^{N} [DevExpKU(D_i, KU_j) + RecDevKU(D_i, KU_j)]$$

$$RevScore(D_i) = \sum_{j=1}^{N} [RevExpKU(D_i, KU_j) + RecRevKU(D_i, KU_j)]$$

$$ExpertiseScore(D_i) = DevScore(D_i) + RevScore(D_i)$$

Here, $1 \leq N \leq 28$ is the number of KUs that appear in the changed files of the PR in question (i.e., a PR from the test dataset). Our rationale is that a reviewer of a given PR should be an expert in the KUs that appear in the changed files of such a PR.

– *Step 3.4) Recommend developers.* We recommend the top K ranked developers based on their expertise score, where K ranges from 1 to 5.

- *Step 4) Construct baseline recommenders for recommending code reviewers.*
To evaluate the performance of KUREC, we need to select baseline recommenders. We choose our baselines based on the following two criteria. First, our baseline should have a similar workflow to that of our proposed KUREC recommender. This workflow includes calculating an expertise score for every developer from the historical data of commits and PRs and recommending reviewers of PRs based on their expertise score. Second, the steps to reimplement a baseline should be clearly described and easy to follow so that the replication of the baseline can be made as accurate as possible.
We construct the following seven different reviewer recommenders based on our above-mentioned criteria: (i) a commit-frequency-based recommender (CF) (Robbes and Röthlisberger 2013), (ii) a review-frequency-based recommender (RF) (Yu et al. 2014), (iii) a modification-expertise-based recommender (ER) (Hannebauer et al. 2016), (iv) a comment-network-based recommender (CN) (Yu et al. 2016), (v) a library-expertise-based recommender (CORRECT) (Rahman et al. 2016), (vi) a review-history-based recommender (CHREV) (Zanjani et al. 2016) and (vii) a workload-distribution-based recommender (SOFIA) (Mirsaeedi and Rigby 2020).
We build the CF recommender because *commit frequency* has frequently been used as an indicator for development expertise (Robbes and Röthlisberger 2013). We choose RF as a baseline for its simplicity to build the recommender (Yu et al. 2014). We select ER as other baseline recommender because it has been widely used in other studies (McDonald and Ackerman 2000; Mockus and Herbsleb 2002; Hannebauer et al. 2016; McDonald 2001). We also choose CN for similar reasons (Rong et al. 2022; Liao et al. 2019). We select CORRECT as a baseline because it is closely related to our work. CORRECT leverages the package information of libraries that are used in the source code files to calculate expertise scores and recommends reviewers based on such a score. We choose the CHREV recommender for two reasons. First, CHREV has been shown to outperform three other recommenders, namely REVFINDER (Thongtanunam et al. 2015), xFinder (Kagdi et al. 2008), and REVCOM. Second, the steps to reimplement CHREV are clearly described and easy to follow Zanjani et al. (2016). Finally, we select SOFIA as a state-of-the-art recommender technique that combines the study of Zanjani et al. (2016) with a learning and retention-based scoring mechanism to improve the workload and knowledge distribution in PR recommendations. We briefly describe the construction process of these seven baseline recommenders below.

(i) Commit-frequency-based recommender (CF): For a given PR in the test dataset, CF first selects all the commits that are performed before the opening date of the given

PR. Next, the recommender counts the number of commits that are performed by each developer. Finally, the recommender sorts developers in decreasing order of commit counts and recommends the top-k ones.

(ii) Review-frequency-based recommender (RF): For a given PR in the test dataset, RF first selects all the PRs that are created prior to the opening date of the given PR. Then, RF counts the number of reviews per developer. Next, developers are sorted in decreasing order of review count. Finally, RF recommends the top-k ranked developers.

(iii) Modification-expertise-based recommender (ER): The ER recommender relies on the assumption that a developer who *last* modified a source code file is the most appropriate team member to review new changes to that same file (McDonald 2001). The rationale is that such a developer knows that file and potentially has fresh memories of its characteristics (e.g., structure, dependencies, behavior, and purpose). The ER recommender has been considered in several prior studies (McDonald and Ackerman 2000; Mockus and Herbsleb 2002; Hannebauer et al. 2016; McDonald 2001). We adapt the ER recommender for our context as follows. For a given PR in the test dataset, ER first identifies all the changed files in the PR. We refer to this set of files as S. Next, a list of developers who modified any of those files in the past (i.e., in commits preceding the PR in question) is generated. Then, ER sorts developers in reverse chronological order based on the date of their last changed file in S. Finally, ER recommends the top-k ranked developers.

(iv) Comment-network-based recommender (CN): The CN recommender constructs a comment network by mining the comments of developers performed on the PRs (Yu et al. 2016). The comment network is defined as a weighted directed graph $G_{cn} = (V, E, W)$. Here, the set of developers is indicated as vertices V, the set of relations between nodes is indicated as edges E and the set of weights is represented as W. There is an edge from v_i to v_j if node v_j has reviewed at least one of v_i's PRs. The weight w_{ij} between the nodes v_i and v_j indicates how much common interest these developers have. For a given PR p in the test dataset, CN traverses the graph from the submitter of p using the breadth-first search algorithm. Then, starting from the unvisited edge with the highest weight every time, CN loops to select and mark the nearest neighbor as a candidate. If the number of contributor's neighbors is less than top-k, CN furthers to visit the child nodes until top-k nodes are found. Finally, CN recommend these top-k candidates. More details of the approach can be found in Yu et al. (2016)'s work.

(v) Library-expertise-based recommender (CORRECT): The basic idea of the CORRECT is that the developers who have more experience on the attached external libraries in the changed files of a PR p are more appropriate for code review than the ones having less experience. To identify the expertise in a library, the CORRECT collects the library names from the changed files of PRs (e.g., the import statement of the Java files) and considers them as a bag of tokens. For a given PR p, CORRECT measures cosine similarity between the bag of tokens of p and all PRs preceding p. This similarity value is used as an expertise score for the reviewers of the PRs preceding p. The reviewers are ranked in descending order of their expertise score values and the top-K reviewers are recommended.

(vi) Review-history-based recommender (CHREV): For a given PR p in the test dataset with a changed file f, CHREV tracks the contribution of each past reviewer of that file across all PRs preceding p. CHREV distills review contribution into three measures: (1) total number of review comments contributed to previous code changes in all PRs prior to the given one (C), (2) total number of workdays (a work day is considered as a day on which a reviewer contributed at least one review comment to at least one changed

file) (W) and (3) recency of the review comments (T). To calculate these measures for a changed file f of PR p, all PRs preceding p and that contain the changed file f are used. Finally, a contribution factor termed xFactor is calculated based on these three measures as follows:

$$xFactor(r, f) = \begin{cases} \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + \frac{1}{|T_f - T'_f|} & if\ |T_f - T'_f| > 0 \\ \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + 1 & if\ |T_f - T'_f| = 0 \end{cases}$$

Here, C_f is the number of review comments that are contributed by a reviewer r for the file f and C'_f is the total number of review comments that are written for the file f. W_f is the number of workdays of the reviewer r on which they contributed review comments for the file f and W'_f is the total number of workdays on which review comments were contributed for the file f. T_f is the most recent workday of the reviewer r with the file f and T'_f is the most workday on which a review comment was contributed for the file f.

Finally, the xFactor for all the changed files in the given PR are aggregated for each reviewer, and a score is generated for all the reviewers who contributed to the review of these changed files in past PRs prior to the given one. The final score is calculated as follows:

$$Score(r) = \sum_{i=1}^{N} xFactor(r, f_i)$$

The reviewers are ranked in descending order of their final score values and the top-K reviewers are recommended.

(vii) Workload-distribution-based recommender (SOFIA): The SOFIA combines CHREV with the learning and retention recommender (TurnoverRec) (Mirsaeedi and Rigby 2020) to improve the workload and knowledge distribution in PR recommendations. The scoring function for the TurnoverRec depends on two knowledge aware proxies: (1) LearnRec and (2) RetentionRec. LearnRec guarantees that a developer who has not reviewed or changed all of the files currently under review will be suggested. RetentionRec ensures that non-transient developers who have commitment to the project are recommended. Given the function $Knowledgeable(f)$ that returns the set of developers who have modified or reviewed file f, $SOFIA(D, P)$ selects either a $CHREV(D, P)$ score or a $TurnoverRec(D, P)$ score as defined in the cases below:

$$\begin{cases} \text{CHREV(D, P)}, & \text{if } |Knowledgeable(f)| \leq 2, \text{any } f | f \in P \\ \text{TurnoverRec(D, P)}, & otherwise \end{cases}$$

Here, D is the set of candidate developers and P is the given PR to recommend code reviewers.

The implementation of all baseline recommenders can be found in our replication package.

- *Step 5) Evaluate recommenders.* To evaluate the performance of recommenders, we use top-k accuracy (Xia et al. 2015; Thongtanunam et al. 2015; Zanjani et al. 2016), and MAP (He et al. 2020; Xie et al. 2021), as they are both widely used in the evaluation of recommenders within software engineering. We calculate these measures using the same formulations as Xie et al. (2021)):

$$Top-kaccuracy = \frac{\sum_{r \in R} isCorrect(r, Top-k)}{|R|} \quad (1)$$

Here, R denotes the set of PRs in the test dataset. The $isCorrect(r, Top-k)$ returns 1 if at least one of top-k developers is the correct reviewer of the PR r and returns 0 otherwise.

The top-k accuracy cannot measure how good the recommender's performance is in ranking the recommended reviewers. For example, the correct reviewer of a PR is r_a and two recommenders (A and B) recommend top-3 reviewers. Now, A recommends $\langle r_d, r_e, r_a \rangle$ and B recommends $\langle r_a, r_b, r_d \rangle$. The top-k accuracy is equal for both A and B recommenders because they correctly recommend one reviewer (r_a). However, in terms of ranking the correct reviewer, recommender B's ranking is better than A's ranking. Because recommender B recommends the correct reviewers at the top one ranked position. Whereas recommender A recommends the correct reviewer at the third position. To understand how good the recommender's performance is in ranking the recommended reviewers, we calculate the Mean Average Precision (MAP) metric (He et al. 2020; Xie et al. 2021). MAP assigns higher importance to the results at top ranks. To calculate MAP, we first calculate the Average Precision (AP):

$$AP@k = \frac{\sum_{i=1}^{k} \frac{s(i)}{i} \times rel(i)}{\sum_{i=1}^{k} rel(i)} \quad (2)$$

Here, i is the position of each developer in the recommended list of developers, and $s(i)$ is the sequence number of the correct developer at position i. The $rel(i)$ returns 1 if the i_{th} developer in the list is correct and 0 otherwise. $MAP@k$ is the average of $AP@k$ over all the PRs in the test dataset. For example, we recommend $K = 5$ developers (d1, d2, d3, d4, d5) and the 1-st, 3-rd and 5-th developers are correct. The sequence number of d1, d3, and d5 are 1, 2, and 3, i.e., s(1) = 1, s(3) = 2, and s(5) = 3. Then, the AP@5 can be calculated as $(\frac{1}{1} + \frac{2}{3} + \frac{3}{5})$. MAP@5 averages the AP@5s across all the recommended lists of the PRs in the test dataset.

- *Step 6) Rank recommenders.* To rank the recommenders for a given evaluation metric (e.g., MAP), we first group the results of a recommender across all top-k recommendations (for k in 1 to 5) and studied projects. This grouping leads to a distribution with 40 data points (5 recommendations × 8 studied projects). Next, we apply the Scott-Knott ESD technique (SK-ESD) (Girba et al. 2015; Mittas and Angelis 2013; Tantithamthavorn et al. 2017) to obtain a ranking of the recommenders. The SK-ESD is a method of multiple comparison that leverages a hierarchical clustering to partition the set of treatment values (e.g., medians or means) into statistically distinct groups with non-negligible difference (Mittas and Angelis 2013).

Findings: Observation 3) KUREC is ranked one along with the top-performing baseline recommender (RF) in Scott-Knott ESD rank, and KUREC outperforms the remaining six baselines. Figure 8 presents the distribution of top-k accuracy and Fig. 9 presents the distribution of MAP for KUREC and seven baselines across the studied projects. The recommenders are grouped according to their Scott-Knott ESD rank. We observe that our KUREC achieves the highest rank along with the RF among the baseline recommenders in both top-k accuracy and MAP. We observe that the absolute difference of the median top-k accuracy between the KUREC and the RF is only 0.08 and the absolute difference of median MAP between these two recommenders is only 0.05. Even though the medians differ a little, the performance of our KUREC and the RF, the top-performing baseline recommender, is identical from an effect size standpoint. We also observe that our KUREC achieves a higher rank

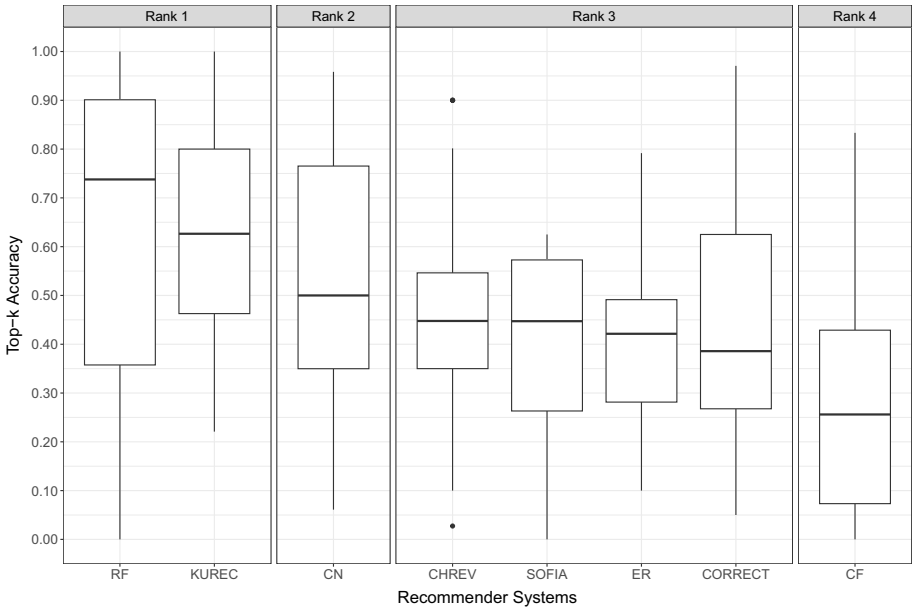


Fig. 8 The distribution of top-k accuracy (k changes from 1 to 5) of baseline recommenders and KUREC across the studied projects. Recommender systems are grouped according to their Scott-Knott ESD rank

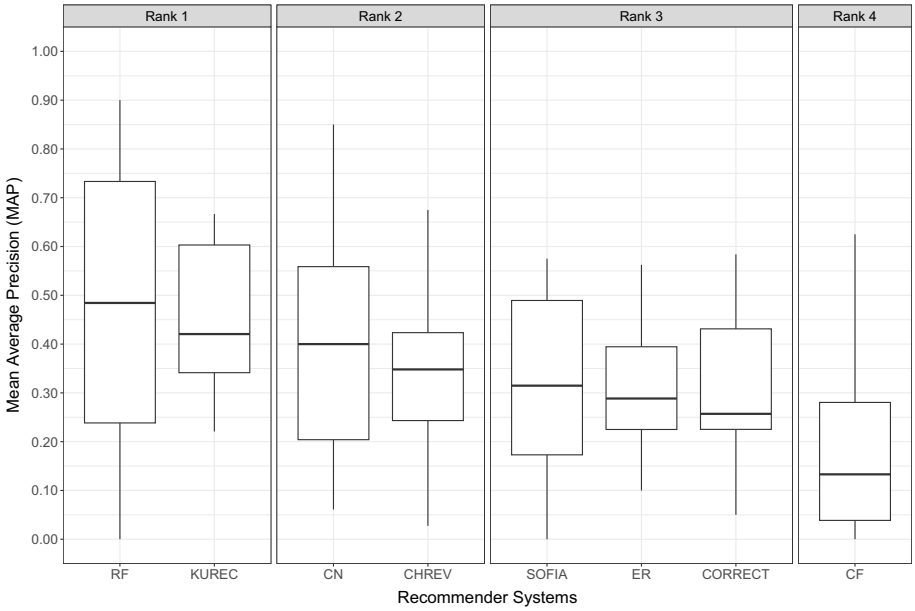


Fig. 9 The distribution of MAP@k (k changes from 1 to 5) of baseline recommenders and KUREC across the studied projects. Recommender systems are grouped according to their Scott-Knott ESD rank

Table 3 The top-k accuracy values of the studied recommenders

Project	Type of Recommender	Recommender Name	Top–k Accuracy					Min Top–k Accuracy
			k=1	k=2	k=3	k=4	k=5	
Apache ActiveMQ	Baseline	CF	0.00	0.00	0.00	0.05	0.05	0.00
		CHREV	0.10	0.20	0.20	0.20	0.30	0.10
		CN	0.35	0.35	0.50	0.50	0.50	0.35
		CORRECT	0.05	0.25	0.35	0.40	0.55	0.05
		ER	0.20	0.25	0.25	0.25	0.25	0.20
		RF	0.75	0.75	0.80	0.85	0.90	0.75
		SOFIA	0.00	0.10	0.10	0.20	0.40	0.00
	KU	KUREC	0.40	0.80	0.80	0.80	0.80	0.40
	Combined	AD_FREQ	0.60	0.65	0.65	0.70	0.75	0.60
		AD_REC	0.70	0.70	0.75	0.85	0.90	0.70
		AD_HYBRID	0.70	0.75	0.75	0.80	0.85	0.70
Apache Groovy	Baseline	CF	0.46	0.75	0.79	0.83	0.83	0.46
		CHREV	0.38	0.42	0.50	0.54	0.54	0.38
		CN	0.63	0.88	0.88	0.92	0.96	0.63
		CORRECT	0.38	0.75	0.79	0.83	0.88	0.38
		ER	0.46	0.63	0.71	0.79	0.79	0.46
		RF	0.54	0.92	0.92	0.92	0.96	0.54
		SOFIA	0.38	0.46	0.63	0.63	0.63	0.38
	KU	KUREC	0.50	0.79	0.92	0.96	0.96	0.50
	Combined	AD_FREQ	0.42	0.67	0.83	0.88	0.88	0.42
		AD_REC	0.38	0.63	0.71	0.79	0.83	0.38
		AD_HYBRID	0.46	0.63	0.75	0.83	0.88	0.46
Apache Lucene	Baseline	CF	0.33	0.56	0.62	0.69	0.73	0.33
		CHREV	0.33	0.36	0.38	0.38	0.40	0.33
		CN	0.42	0.60	0.67	0.69	0.76	0.42
		CORRECT	0.24	0.38	0.44	0.47	0.53	0.24
		ER	0.42	0.47	0.49	0.53	0.58	0.42
		RF	0.53	0.80	0.80	0.84	0.87	0.53
		SOFIA	0.18	0.18	0.40	0.49	0.51	0.18
	KU	KUREC	0.53	0.69	0.80	0.89	0.93	0.53
	Combined	AD_FREQ	0.49	0.67	0.78	0.87	0.93	0.49
		AD_REC	0.47	0.67	0.80	0.89	0.93	0.47
		AD_HYBRID	0.47	0.64	0.78	0.87	0.93	0.47
Apache Hbase	Baseline	CF	0.17	0.22	0.26	0.29	0.32	0.17
		CHREV	0.03	0.31	0.45	0.54	0.57	0.03
		CN	0.06	0.26	0.31	0.44	0.52	0.06
		CORRECT	0.16	0.24	0.34	0.39	0.46	0.16
		ER	0.20	0.32	0.38	0.42	0.45	0.20
		RF	0.23	0.52	0.69	0.69	0.73	**0.23**
		SOFIA	0.35	0.46	0.52	0.55	0.57	0.35

Table 3 continued

Project	Type of Recommender	Recommender Name	Top–k Accuracy k=1	k=2	k=3	k=4	k=5	Min Top–k Accuracy
	KU	KUREC	0.26	0.49	0.59	0.66	0.72	**0.26**
	Combined	AD_FREQ	0.23	0.50	0.65	0.65	0.69	0.23
		AD_REC	0.36	0.49	0.64	0.66	0.69	0.36
		AD_HYBRID	0.35	0.49	0.63	0.65	0.68	0.35
Apache	Baseline	CF	0.05	0.08	0.12	0.17	0.19	0.05
Hive		CHREV	0.24	0.36	0.42	0.45	0.49	0.24
		CN	0.10	0.27	0.35	0.43	0.46	0.10
		CORRECT	0.18	0.28	0.33	0.38	0.42	0.18
		ER	0.20	0.30	0.36	0.41	0.44	0.20
		RF	0.14	0.25	0.38	0.45	0.48	**0.14**
		SOFIA	0.08	0.15	0.23	0.29	0.32	0.08
	KU	KUREC	0.24	0.35	0.44	0.52	0.59	**0.24**
	Combined	AD_FREQ	0.21	0.34	0.41	0.47	0.53	0.21
		AD_REC	0.22	0.33	0.41	0.47	0.55	0.22
		AD_HYBRID	0.22	0.33	0.41	0.48	0.55	0.22
Apache	Baseline	CF	0.01	0.04	0.08	0.27	0.30	0.01
Storm		CHREV	0.51	0.63	0.71	0.76	0.80	0.51
		CN	0.40	0.48	0.50	0.74	0.79	0.40
		CORRECT	0.35	0.60	0.79	0.93	0.97	0.35
		ER	0.28	0.40	0.46	0.47	0.48	0.28
		RF	0.00	0.06	0.71	0.72	0.81	**0.00**
		SOFIA	0.54	0.57	0.58	0.59	0.60	0.54
	KU	KUREC	0.28	0.39	0.51	0.72	0.88	**0.28**
	Combined	AD_FREQ	0.52	0.76	0.84	0.90	0.92	0.52
		AD_REC	0.54	0.80	0.88	0.93	0.96	0.54
		AD_HYBRID	0.56	0.81	0.88	0.92	0.94	0.56
Apache	Baseline	CF	0.00	0.00	0.10	0.35	0.40	0.00
Wicket		CHREV	0.45	0.90	0.90	0.90	0.90	0.45
		CN	0.85	0.85	0.95	0.95	0.95	0.85
		CORRECT	0.20	0.60	0.70	0.80	0.90	0.20
		ER	0.10	0.10	0.30	0.45	0.50	0.10
		RF	0.85	0.95	1.00	1.00	1.00	0.85
		SOFIA	0.55	0.60	0.60	0.60	0.60	0.55
	KU	KUREC	0.30	0.80	0.95	1.00	1.00	0.30
	Combined	AD_FREQ	0.75	0.75	0.85	0.90	0.90	0.75
		AD_REC	0.80	0.85	0.95	0.95	0.95	0.80
		AD_HYBRID	0.80	0.85	0.95	0.95	0.95	0.80
Elastic	Baseline	CF	0.16	0.25	0.34	0.42	0.47	0.16
Search		CHREV	0.23	0.41	0.45	0.51	0.56	0.23
		CN	0.09	0.19	0.25	0.34	0.39	0.09
		CORRECT	0.17	0.25	0.23	0.27	0.32	0.17

Table 3 continued

Project	Type of Recommender	Recommender Name	Top–k Accuracy k=1	k=2	k=3	k=4	k=5	Min Top–k Accuracy
		ER	0.28	0.42	0.50	0.55	0.59	0.28
		RF	0.03	0.12	0.17	0.23	0.29	**0.03**
		SOFIA	0.18	0.27	0.35	0.39	0.44	0.18
	KU	KUREC	0.30	0.43	0.47	0.53	0.56	**0.27**
	Combined	AD_FREQ	0.28	0.41	0.49	0.55	0.59	0.28
		AD_REC	0.28	0.41	0.49	0.55	0.59	0.28
		AD_HYBRID	0.28	0.41	0.49	0.55	0.58	0.28
Median	Baseline	CF	0.11	0.15	0.19	0.32	0.36	0.11
		CHREV	0.29	0.39	0.45	0.52	0.55	0.29
		CN	0.37	0.41	0.50	0.59	0.64	0.37
		CORRECT	0.19	0.33	0.40	0.43	0.54	0.19
		ER	0.24	0.36	0.42	0.46	0.49	0.24
		RF	0.38	0.64	0.76	0.85	0.88	0.38
		SOFIA	0.27	0.37	0.46	0.52	0.54	0.27
	KU	KUREC	0.29	0.59	0.69	0.76	0.84	0.29
	Combined	AD_FREQ	0.45	0.66	0.71	0.78	0.81	0.45
		AD_REC	0.42	0.65	0.73	0.82	0.87	0.42
		AD_HYBRID	0.46	0.63	0.75	0.82	0.88	0.46

We highlight the values in bold for instances where the minimum top-k accuracy of KUREC is higher than that of RF

than the remaining three baseline recommenders (e.g., CN, CORRECT, SOFIA, CHREV, ER and CF) in both top-k accuracy and MAP. Such a result indicates that our KUREC outperforms these baseline recommenders and can be used effectively for recommending code reviewers in PRs.

Observation 4) The performance of our KUREC is more stable than that of the top-performing baseline recommender (RF). As we observe from Figs. 8 and 9, the interquartile range of our KUREC is smaller than the RF in both top-k accuracy and MAP. For MAP, the interquartile range of our KUREC is only 0.25, whereas the interquartile range of the RF is 0.56. Such a result indicates that KUREC is more stable than the RF.

We can also notice from Tables 3 and 4 that the performance of our KUREC is stable. Here, Table 3 presents the top-k accuracy results and Table 4 presents the MAP results across the studied projects. We observe that the minimum top-k accuracy and MAP of our KUREC are always above 0.20, whereas these values could reach 0.00 for the baseline recommenders (see the last column of Tables 3 and 4). For example, in Apache ActiveMQ, the minimum top-k accuracy and the MAP of CF and SOFIA are 0.00. We can even notice unstable performance in the top-performing recommender (RF). The minimum top-k accuracy and MAP are 0.00 for the RF in Apache Storm and Elastic Search projects. In Elastic Search, the top-k accuracy of the RF is the worst among the baseline recommenders. On the other hand, our KUREC outperforms all the baseline recommenders including the RF. Such a result indicates that our KUREC is more stable than the RF. In any recommender, stability is a desired property because a stable recommender can gain users' trust and acceptance of recommendations (Adomavicius and Zhang 2014). Based on our aforementioned results, we

Table 4 The mean average precision (MAP) values of the studied recommenders

Project	Type of Recommender	Recommender Name	Mean Average Precision (MAP)					Min MAP
			k=1	k=2	k=3	k=4	k=5	
Apache ActiveMQ	Baseline	CF	0.00	0.00	0.00	0.01	0.01	0.00
		CHREV	0.10	0.15	0.15	0.15	0.16	0.10
		CN	0.35	0.35	0.40	0.40	0.40	0.35
		CORRECT	0.05	0.15	0.19	0.20	0.23	0.05
		ER	0.20	0.23	0.23	0.23	0.23	0.20
		RF	0.75	0.75	0.76	0.75	0.76	0.75
		SOFIA	0.00	0.05	0.05	0.08	0.12	0.00
	KU	KUREC	0.40	0.60	0.60	0.60	0.69	0.40
	Combined	AD_FREQ	0.60	0.61	0.61	0.62	0.63	0.60
		AD_REC	0.70	0.70	0.71	0.71	0.71	0.70
		AD_HYBRID	0.70	0.71	0.71	0.71	0.71	0.70
Apache Groovy	Baseline	CF	0.46	0.60	0.63	0.61	0.61	0.46
		CHREV	0.38	0.40	0.41	0.41	0.41	0.38
		CN	0.63	0.75	0.73	0.73	0.74	0.63
		CORRECT	0.38	0.56	0.57	0.58	0.58	0.38
		ER	0.46	0.54	0.56	0.55	0.55	0.46
		RF	0.54	0.73	0.73	0.72	0.72	**0.54**
		SOFIA	0.38	0.42	0.47	0.47	0.47	0.38
	KU	KUREC	0.56	0.65	0.65	0.67	0.67	**0.56**
	Combined	AD_FREQ	0.42	0.54	0.57	0.59	0.59	0.42
		AD_REC	0.38	0.50	0.50	0.53	0.54	0.38
		AD_HYBRID	0.46	0.54	0.55	0.57	0.58	0.46
Apache Lucene	Baseline	CF	0.33	0.44	0.47	0.48	0.49	0.33
		CHREV	0.33	0.34	0.35	0.35	0.35	0.33
		CN	0.42	0.51	0.53	0.53	0.54	0.42
		CORRECT	0.24	0.31	0.34	0.34	0.36	0.24
		ER	0.42	0.44	0.44	0.45	0.46	0.42
		RF	0.53	0.67	0.66	0.66	0.65	0.53
		SOFIA	0.18	0.18	0.25	0.27	0.28	0.18
	KU	KUREC	0.54	0.60	0.63	0.65	0.64	0.54
	Combined	AD_FREQ	0.49	0.58	0.60	0.62	0.63	0.49
		AD_REC	0.47	0.57	0.61	0.63	0.63	0.47
		AD_HYBRID	0.47	0.56	0.59	0.61	0.61	0.47
Apache Hbase	Baseline	CF	0.17	0.19	0.20	0.21	0.22	0.17
		CHREV	0.03	0.17	0.22	0.24	0.25	0.03
		CN	0.06	0.16	0.18	0.21	0.23	0.06
		CORRECT	0.16	0.20	0.23	0.24	0.26	0.16
		ER	0.20	0.26	0.28	0.29	0.30	0.20
		RF	0.23	0.38	0.43	0.43	0.44	**0.23**
		SOFIA	0.35	0.40	0.42	0.43	0.43	0.35

Table 4 continued

Project	Type of Recommender	Recommender Name	Mean Average Precision (MAP) k=1	k=2	k=3	k=4	k=5	Min MAP
	KU	KUREC	0.26	0.37	0.41	0.42	0.42	**0.26**
	Combined	AD_FREQ	0.23	0.36	0.41	0.41	0.41	0.23
		AD_REC	0.36	0.43	0.46	0.45	0.45	0.36
		AD_HYBRID	0.35	0.42	0.46	0.44	0.44	0.35
Apache	Baseline	CF	0.05	0.07	0.08	0.09	0.09	0.05
Hive		CHREV	0.24	0.30	0.32	0.33	0.33	0.24
		CN	0.10	0.19	0.22	0.23	0.24	0.10
		CORRECT	0.18	0.23	0.25	0.26	0.27	0.18
		ER	0.20	0.25	0.27	0.28	0.29	0.20
		RF	0.14	0.20	0.24	0.26	0.26	**0.14**
		SOFIA	0.08	0.11	0.14	0.15	0.16	0.08
	KU	KUREC	0.22	0.28	0.31	0.33	0.34	**0.22**
	Combined	AD_FREQ	0.21	0.28	0.30	0.31	0.32	0.21
		AD_REC	0.22	0.27	0.30	0.31	0.32	0.22
		AD_HYBRID	0.22	0.28	0.30	0.32	0.32	0.22
Apache	Baseline	CF	0.01	0.03	0.04	0.09	0.09	0.01
Storm		CHREV	0.51	0.57	0.59	0.59	0.59	0.51
		CN	0.40	0.44	0.44	0.47	0.46	0.40
		CORRECT	0.35	0.47	0.53	0.56	0.56	0.35
		ER	0.28	0.34	0.36	0.35	0.35	0.28
		RF	0.00	0.03	0.25	0.32	0.34	**0.00**
		SOFIA	0.54	0.55	0.56	0.56	0.55	0.54
	KU	KUREC	0.28	0.33	0.37	0.42	0.44	**0.28**
	Combined	AD_FREQ	0.52	0.64	0.67	0.67	0.66	0.52
		AD_REC	0.54	0.67	0.69	0.68	0.68	0.54
		AD_HYBRID	0.56	0.68	0.71	0.69	0.69	0.56
Apache	Baseline	CF	0.00	0.00	0.03	0.10	0.11	0.00
Wicket		CHREV	0.45	0.68	0.66	0.66	0.66	0.45
		CN	0.85	0.85	0.80	0.80	0.80	0.80
		CORRECT	0.20	0.40	0.43	0.45	0.48	0.20
		ER	0.10	0.10	0.17	0.20	0.21	0.10
		RF	0.85	0.90	0.89	0.89	0.87	0.85
		SOFIA	0.55	0.58	0.57	0.54	0.54	0.54
	KU	KUREC	0.30	0.55	0.61	0.63	0.63	0.30
	Combined	AD_FREQ	0.75	0.75	0.77	0.76	0.74	0.74
		AD_REC	0.80	0.83	0.83	0.80	0.78	0.78
		AD_HYBRID	0.80	0.83	0.84	0.81	0.79	0.79
Elastic	Baseline	CF	0.16	0.21	0.23	0.25	0.26	0.16
Search		CHREV	0.23	0.32	0.36	0.37	0.38	0.23
		CN	0.09	0.14	0.16	0.18	0.19	0.09

Table 4 continued

Project	Type of Recommender	Recommender Name	Mean Average Precision (MAP) k=1	k=2	k=3	k=4	k=5	Min MAP
		CORRECT	0.17	0.21	0.23	0.24	0.25	0.17
		ER	0.28	0.35	0.37	0.38	0.39	0.28
		RF	0.03	0.07	0.09	0.10	0.11	**0.03**
		SOFIA	0.18	0.23	0.25	0.26	0.26	0.18
	KU	KUREC	0.30	0.34	0.36	0.38	0.43	**0.30**
	Combined	AD_FREQ	0.28	0.35	0.37	0.38	0.38	0.28
		AD_REC	0.28	0.35	0.37	0.38	0.38	0.28
		AD_HYBRID	0.28	0.35	0.37	0.38	0.38	0.28
Median	Baseline	CF	0.11	0.13	0.14	0.15	0.16	0.11
		CHREV	0.29	0.33	0.35	0.36	0.36	0.29
		CN	0.37	0.39	0.42	0.44	0.43	0.37
		CORRECT	0.19	0.27	0.29	0.30	0.31	0.19
		ER	0.24	0.30	0.32	0.32	0.32	0.24
		RF	0.37	0.52	0.53	0.53	0.53	0.37
		SOFIA	0.27	0.32	0.34	0.35	0.36	0.27
	KU	KUREC	0.29	0.46	0.50	0.51	0.51	0.29
	Combined	AD_FREQ	0.45	0.56	0.59	0.60	0.58	0.45
		AD_REC	0.42	0.53	0.55	0.58	0.58	0.42
		AD_HYBRID	0.46	0.55	0.57	0.59	0.60	0.46

We highlight the values in bold for instances where the min MAP of KUREC is higher than that of RF

can conclude that our KUREC is the best recommender among the baseline recommenders since our KUREC performs identically to the top one but is more stable.

Summary

RQ1: How accurately can KUREC recommend code reviewers in pull requests?

Our KUREC recommender is the best option among the baselines. In particular,

- KUREC is ranked one along with the top-performing baseline recommender (RF) in Scott-Knott ESD rank.
- KUREC has a more stable performance than the top-performing recommender (RF).

5.2 RQ2: Can KUREC be Made More Accurate by Combining it with Existing Recommenders?

Motivation: In Section 5.1, we observe that each recommender has its own strength (e.g., the KUREC leverages both development expertise and review expertise) to perform a better recommendation and its own weakness (e.g., the RF performs the worst when many reviewers perform the reviews in a large project) that results in bad recommendations. Therefore, in this section, we want to see if straightforward combinations of the KUREC with the baseline recommenders result in a better performing recommender.

Approach: To construct a combined recommender by leveraging the recommendations of different recommenders, we are motivated by the work of Malik and Hassan (2008). In this work, the authors propose adaptive meta heuristics that combine various previously researched heuristics and improve the performance of change propagation task (i.e., changes required to other entities of the software recommender to avoid bugs). We follow the same approach of Malik and Hassan (2008). In this approach, all the recommenders uses a Best Recommender System Table (BRST) to track the best-performing recommender.

We briefly explain each of the steps that we follow in this approach. First, all the studied recommenders are trained with the training dataset. Then, a recommender is randomly selected to predict the reviewers of the first PR from the test data. Next, for testing each of the remaining PRs of the test dataset, we select a recommender from the BRST. Once a PR is completed for recommendation, we identify the best-performing recommender and update the BRST. To identify the best-performing recommender for a given PR, we generate a score for each of the studied recommenders by combining both accuracy and MAP. We run every recommender for all previous PRs from our test dataset and calculate the top-k accuracy and MAP. Then, we calculate the average of the accuracy and the average of the MAP for all k recommendations. Next, we average the accuracy and the MAP, and generate a single score for each of the studied recommenders to rank them. We identify the best-performing recommender that has the highest score. Finally, we need to update the BRST when a PR is completed for recommendation. We implement the following three techniques to update the BRST:

1) **Adaptive Frequency Technique (AD_ FREQ).** In this technique, the BRST is a frequency table that stores the count of how many times each recommender becomes the best-performing recommender among all the previous PRs of the test dataset. To update the BRST for a given PR, we increment the count by one to the best-performing recommender. In order to recommend reviewers for a new PR, we select the recommender that has the highest count in the BRST.
2) **Adaptive Recency Technique (AD_ REC).** In this technique, the BRST only stores the best-performing recommender that is identified in the last PR. To update the BRST for a given PR, we replace the currently stored recommender to the identified best-performing recommender for the given PR. In order to recommend reviewers for a new PR, we select the recommender that is stored in the BRST.
3) **Adaptive Hybrid Technique (AD_ HYBRID).** This technique leverages both the frequency and recency techniques. In this technique, the BRST is a frequency table that stores the count of how many times each recommender becomes the best-performing recommender among the last 10 previous PRs of the test dataset. We increment the count by one to the best-performing recommender for the given PR and update the BRST. To recommend reviewers for a new PR, we select the recommender that has the highest count in the BRST among the last 10 previous PRs.

Findings: Observation 5) All the combined recommenders outperform individual recommenders. Figure 10 presents the distribution of top-k accuracy and Fig. 11 presents the distribution of MAP of all recommenders across the studied projects. We observe that all three combined recommenders perform the same from an effect size perspective, ranking first in both top-k accuracy (Fig. 10) and MAP (Fig. 11). For top-k accuracy, the RF, the KUREC and the combined recommenders are all tied (i.e., they all rank first). However, similar to the KUREC, the interquartile range of the combined recommenders is smaller than that of the RF (see Fig. 10). Such a result indicates that the combined recommenders are more stable than the RF. For MAP, the performance of the combined recommenders is

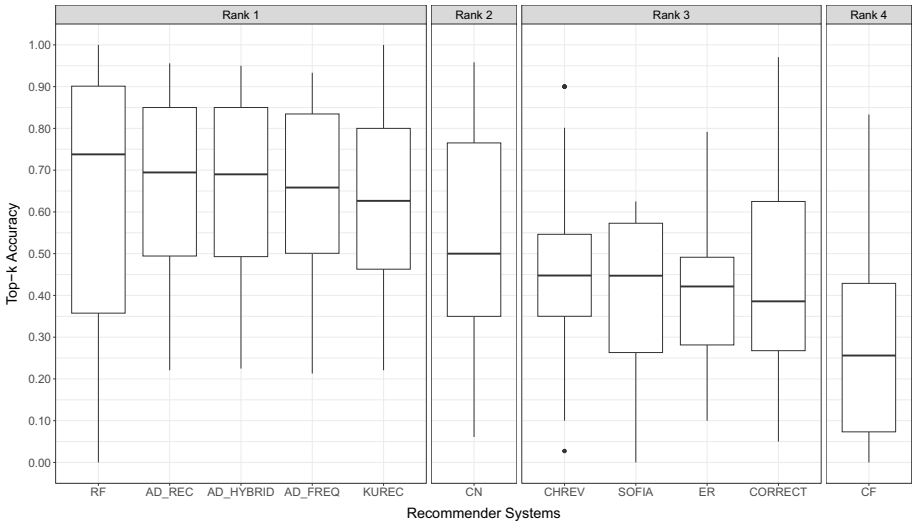


Fig. 10 The distribution of top-k accuracy (k changes from 1 to 5) of recommenders across the studied projects. Recommender systems are grouped according to their Scott-Knott ESD rank

even better. As we observe from Fig. 11, the combined recommenders are the only ranked first recommenders outperforming all individual recommenders. The median MAP of the combined recommenders are higher than all baseline recommenders including the RF and the KUREC (see also the last row of Table 4). For example, the AD_HYBRID has a 9% improvement at top-1 recommendation and 7% improvement top-5 recommendation from the highest performing baseline recommender (i.e., the RF recommender). In addition, the interquartile range of all three combined recommenders is smaller than the RF and is similar

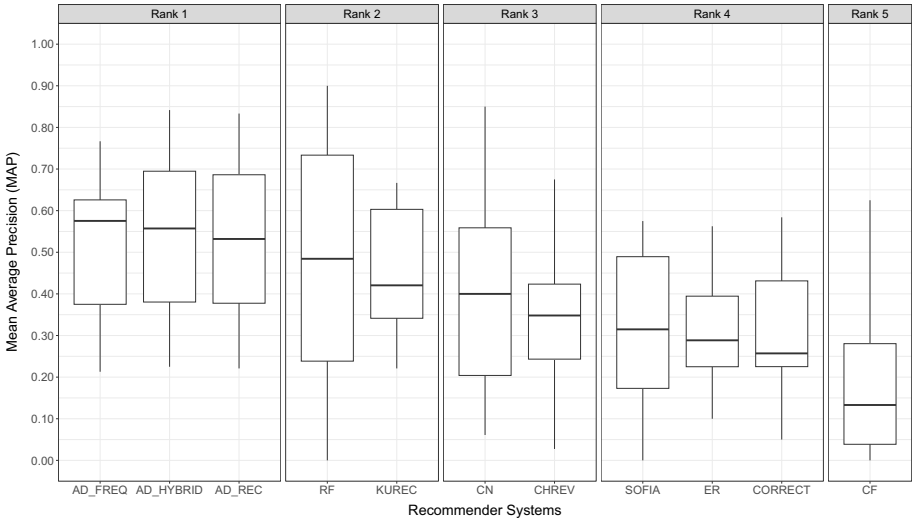


Fig. 11 The distribution of MAP@k (k changes from 1 to 5) of recommenders across the studied projects. Recommender systems are grouped according to their Scott-Knott ESD rank

to the KUREC. As we observe from the last column of Table 4, the minimum MAP of the combined recommenders never goes below 0.22 (whereas the minimum MAP of the RF reaches 0.0).

To determine the effectiveness of KUREC in combined recommenders, we analyze the instances where the combined recommenders opt for KUREC as the choice for recommending reviewers. We observe that KUREC is chosen more often compared to other baselines. For example, AD_HYBRID selects KUREC in 61% (median) of cases. These results indicate that combining the KUREC with baseline recommenders in a straightforward manner outperforms individual recommenders.

Summary

RQ2: Can KUREC be made more accurate by combining it with existing recommenders?

Yes. Combining the KU-based recommender (KUREC) with the baselines in a straightforward manner results in better performing recommenders. In particular,

- All the combined recommenders ranked first among all individual recommenders in terms of both top-k accuracy and MAP.
- The combined recommenders outperform all individual recommenders in terms of MAP.
- The performance of all combined recommenders is more stable than that of the top-performing baseline recommender (RF).

5.3 RQ3: How Reasonable are the Recommendations made by KUREC?

Motivation: To measure the performance of KUREC and the other baseline recommenders, we needed a ground truth. To build our ground truth, we examined the review history of a project. We considered a recommendation to be correct when such a recommendation matched the individual who actually reviewed that PR. This is a standard practice in the code review literature (Zanjani et al. 2016; Thongtanunam et al. 2015; Kagdi et al. 2008; Xie et al. 2021). However, as we observe in RQ1, there are certain projects in which few developers review most of the PRs (c.f., top-5 accuracy/MAP of RF). Prior study also shows that the top-1 recommended reviewers of a recommender who are labeled as incorrect based on the ground truth (i.e., actual reviewers of PRs) would have been comfortable reviewing the given code changes of PRs (Gauthier et al. 2021). We thus wonder whether the recommendations issued by our studied recommenders would have been *reasonable* choices in practice when those differ from what happened in reality. Investigating this hypothesis sheds light into whether KUREC's recommendations would be a valid alternative to recommending the same developers over and over (e.g., to rebalance review effort across the team).

Approach: We consider a recommendation to be *reasonable* if the recommended individual had recent development experience with the majority of the files included in the PR in question. That is, if someone recently contributed changes to the majority of the files in a PR, we assume that that person is likely knowledgeable about those files and would thus be a reasonable reviewer for that PR. More specifically, our approach works as follows.

First, we pick a recommender R. Next, we select the PRs from the test dataset where R's top-1 recommendation D does *not* match the ground truth (the actual reviewer). Let P be one of those PRs. For each P, we collect all files from commits and PRs that were submitted by D in the six months preceding the submission of P. Let F be this collection of files. Finally, if at least 50% of the changed files in P are included in F, then we consider D to be a reasonable recommendation. We choose six months as a time frame to ensure that the reviewer has

very recent expertise with the files included in the PR in question. We report the percentage of *reasonable* recommendations for KUREC and all the other studied recommenders (i.e., KUREC's variations and baseline recommenders).

Findings: Observation 6) KUREC is the recommender with the highest percentage of reasonable recommendations (63.4%). The percentage of reasonable recommendations per recommender (for PRs in which the top-1 recommendation does not match the ground truth) is as follows: KUREC: 63.4%, AD_REC: 61.5%, AD_HYBRID: 61.3%, ER: 60.9%, AD_FREQ: 54.2%, CHREV: 30.4%, CF: 25.4%, CORRECT: 24.1%, SOFIA: 21.6%, RF: 15.2% and CN: 10.6%. Interestingly, both ER and the combined recommenders also have a high percentage of reasonable recommendations. However, the combined recommenders are considerably more aligned to the ground truth compared to ER (c.f., Figs. 10 and 11). The AD_HYBRID combined recommender, in particular, strikes the best balance between sticking to the ground truth (best recommender from RQ2) and issuing reasonable recommendations when those deviate from that ground truth (third best recommender in this RQ). Overall, the aforementioned results provide positive empirical evidence to support the idea that either AD_HYBRID or KUREC's recommendations could be a valid alternative to recommending the same developers over and over. However, more in-depth studies would still be necessary to determine how acceptable those recommendations would be in practice.

To conclude this RQ, we make a final observation. One might notice that the very approach of this RQ could have been converted into a new recommender that would likely outperform all of our studied ones (including KUREC and AD_HYBRID) in terms of the ratio of reasonable recommendations. However, we clarify that such a recommender would have limited applicability in practice because it would *operate exclusively on contribution recency*. In other words, it would only recommend the same individuals over and over. Such a characteristic is undesirable, since (i) it does *not* promote knowledge and expertise sharing (e.g., it aggravates the consequences of developer turnover (Mirsaeedi and Rigby 2020)) and (ii) it prevents adequate review workload distribution. Modern recommenders, such as KUREC and CHREV account for other factors *in addition to recency*. For instance, KUREC considers KU-based development expertise when recommending reviewers. As a consequence, KUREC is able to recommend an individual even if that individual has never performed a code review before.

Summary

RQ3: How reasonable are the recommendations made by KUREC?

KUREC is the recommender with the highest percentage of reasonable recommendations (63.4%). Yet, AD_HYBRID strikes the best balance between sticking to the ground truth (c.f., RQ2) and issuing reasonable recommendations when those deviate from that ground truth (61.3% reasonable recommendations).

6 Discussion

In this section, we briefly discuss the implications of our study.

– **Researchers should consider the stability of evaluation metrics (e.g., top-k accuracy and MAP) for comparing the performance of recommenders.** In RQ1, we observe that KUREC is ranked one along with the top-performing baseline recommender (RF) in Scott-Knott ESD rank. Despite RF's recognition as the best baseline recommender, its top-k accuracy and MAP exhibit a wider interquartile range than those of other rec-

ommenders, highlighting its inconsistent performance. The instability in RF's results is further evident, with the lowest top-k accuracy and MAP recorded as 0.00 in projects such as Apache Storm and Elastic Search. In the case of Elastic Search, RF records the poorest top-k accuracy among the baseline recommenders. On the other hand, our KUREC outperforms all the baseline recommenders including the RF. The importance of stability becomes evident, as a stable recommender garners user trust and acceptance of its recommendations [1]. Therefore, we recommend researchers should consider stability for evaluating the performance of recommenders along with other evaluation metrics.

- **Future research should explore the potential of combining recommenders.** Research papers in the area of reviewer recommendation typically introduce a new recommender and compare it to one or more baselines. However, our paper makes it clear that combining recommenders (even when using very simple approaches to do so) can lead to more stable and accurate recommenders. We observe that the AD_HYBRID demonstrates a 9% improvement in top-1 recommendation and a 7% improvement in top-5 recommendation compared to the best-performing baseline recommender (i.e., the RF recommender). Furthermore, the interquartile range of all three combined recommenders is smaller than the RF and is similar to the KUREC. The selection of the best recommender among the studied recommenders using our heuristic-based techniques (e.g., AD_FREQ, AD_REC, and AD_HYBRID) for a given PR could lead to a higher-performing combined recommender. We observe that KUREC is chosen more often compared to other baselines. For example, AD_HYBRID selects KUREC in 61% (median) of cases. This result indicates that combining the KUREC with baseline recommenders in a straightforward manner outperforms individual recommenders. Since we employ a very straightforward heuristic-based approach to combine multiple baseline recommenders, the cost of building combined recommenders is very limited. However, we want to note that combining multiple recommenders is particularly valuable when the individual recommenders are built from the same raw data (i.e., there is no additional effort in terms of data collection and preprocessing). In fact, combining individual models to obtain more accurate ones is not a new idea (e.g., ensemble models in Machine Learning). Therefore, future research in reviewer recommendation should consider combining models and evaluating their performance.
- **KUREC opens a new door for reviewer recommendation, but also comes with challenges.** Despite our simple operationalization of KUs, KUREC showed surprisingly good results when compared to several established baselines. However, (programming language) KUs do not come for free. KUs, as defined by us, are abstract and require an operationalization. While our operationalization approach is cost-effective and efficient, it does present certain challenges. Specifically, we concentrate on the core functionalities of the programming language, focusing on those outlined in certification exam topics. Therefore, the set of KUs that we conceive in this paper is limited and does not cover all the knowledge embedded in the constructs and APIs of the Java programming language. To further enhance this concept, future research should explore methods to integrate additional KUs and analyze the impact of diverse KUs on the recommendation of code reviewers for pull requests (PRs).

  The second challenge is that Knowledge Units (KUs) are intrinsically tied to specific programming languages. This leads to the requirement of distinct approaches for concept operationalization and detection mechanisms tailored to each language. Thankfully, the presence of certification exams for various languages offers a promising foundation to initiate this process. Perhaps most importantly, we believe that recent breakthroughs in machine-learning-based language models (such as Large Language Models, a.k.a.

LLMs) can possibly help in the operationalization and detection of KUs across different programming languages. An interesting avenue for future investigation thus involves constructing KUREC recommenders for multiple programming languages and delving into strategies for creating a universally applicable recommender by leveraging the concept of KUs.

7 Related Work

(A) Reviewer recommendation In this subsection, we provide a brief overview of the vast literature in reviewer recommendation. We broadly group prior studies into three main categories: (1) studies that focus on an evaluation of recommendation accuracy (e.g., using top-k accuracy and MAP), (2) studies that focus on balancing between the accuracy of recommendations and review workload distribution, and (3) studies that focus on labeling bias and reasonability of recommendation. In the following, we briefly discuss these related studies (a summary is shown in Table 5).

Evaluation of Recommendation Accuracy Yu et al. (2014) study social relations between contributors and reviewers, and propose a recommendation model for reviewers by mining comment networks (CN). The authors compare the performance of the proposed model with the machine learning (ML) based model (e.g., SVM). The authors also design a simple and effective method as a comparison baseline in experiments based on the review frequency of developers. The quantitative evaluations on 10 projects of GitHub show that the CN-based approach achieves significant improvements over the baseline and ML-based method.

In a later study, Yu et al. (2016) investigate whether approaches that are used in bug triaging and code review can be adapted to recommending reviewers for PRs and how to improve the recommendation performance. The authors combined their proposed comment networks (CN) with traditional approaches that are used in bug triaging to evaluate the performance on 84 GitHub projects. The results show that the performance of traditional approaches can be improved by combining them with the information extracted from prior social interactions (e.g., comments) between developers.

Thongtanunam et al. (2015) propose a reviewer recommendation approach, namely REVFINDER, which is based on the past reviews of files with similar names and paths. The authors evaluate their performance on a case study of 42,045 reviews of four popular projects. The results show that REVFINDER achieves an accuracy of 79% at top 10 recommendations and the overall ranking of REVFINDER is three times better than a baseline approach.

Xia et al. (2015) propose another approach for reviewer recommendation, namely TIE. TIE is based on the intuition that the same reviewers are likely to review code changes containing similar terms (i.e., words) as well as review the same files or files in a similar location. The evaluation results show that TIE outperforms REVFINDER on several open source systems.

Zanjani et al. (2016) develop a reviewer recommender approach, namely CHREV, that uses three measures for contributions: (1) the total number of review comments contributed to previous code changes in all PRs prior to the given one, (2) total number of workdays and (3) recency of the review comments. The CHREV first extracts each source code file and generates a reviewer's expertise factor based on their historical contributions. Then the CHREV recommends reviewers based on the score of the cumulative expertise factors. The authors evaluate the performance of CHREV on three open-source systems and a commercial code-

Table 5 The related studies under reviewer recommendation

Category	Study	Year	Considered data types	Evaluated Baselines
Evaluation of recommendation accuracy (e.g., using top-k accuracy and MAP)	Yu et al. (2014)	2014	Comments in PRs	ReviewFrequency(RF) Yu et al. (2014)
	Thongtanunam et al. (2015)	2015	Paths of the changed files in PRs	REVFINDER Thongtanunam et al. (2015) and ReviewBot Balachandran (2013)
	Xia et al. (2015)	2015	Paths of the changed files in PRs	REVFINDER Thongtanunam et al. (2015)
	Zanjani et al. (2016)	2016	Comments in PRs	REVFINDER Thongtanunam et al. (2015), xFinder Kagdi et al. (2008) and REVCOM Zanjani et al. (2016)
	Yu et al. (2016)	2016	Comments in PRs	RF Yu et al. (2016)
	Rahman et al. (2016)	2016	Package name and class name of libraries that are used in the changed files of PRs	REVFINDER Thongtanunam et al. (2015)
	Hannebauer et al. (2016)	2016	The changed files of commits and PRs	–
	Jiang et al. (2017)	2017	Comments in PRs	CN Yu et al. (2016)
	Xia et al. (2017)	2017	Comments in PRs	CN Yu et al. (2016), REVFINDER Thongtanunam et al. (2015), TIE Xia et al. (2015) and Activeness Jiang et al. (2017)
	Fejzer et al. (2018)	2018	Paths of the changed files in PRs	REVFINDER Thongtanunam et al. (2015)
	Liao et al. (2019)	2019	Comments in PRs	CN Yu et al. (2016)
	Jiang et al. (2019)	2019	Paths of the changed files in PRs	CHREV Zanjani et al. (2016), WRC Hannebauer et al. (2016), TIE Xia et al. (2015) and CoreDevRec Jiang et al. (2015)
	Sülün et al. (2019)	2019	Changed files in PRs	REVFINDER Thongtanunam et al. (2015)

Table 5 continued

Category	Study	Year	Considered data types	Evaluated Baselines
	Strand et al. (2020)	2020	Paths of the changed files in PRs	–
	Pandya and Tiwari (2022)	2022	Paths of the changed files in PRs, project, sub-project and author information	REVFINDER Thongtanunam et al. (2015)
Balancing between recommendation accuracy and review workload distribution	Asthana et al. (2019)	2019	Paths of the changed files in PRs	–
	Al-Zubaidi et al. (2020)	2020	Code ownership	–
	Mirsaeedi and Rigby (2020)	2020	Comments in PRs, changed files in commits and PRs	CHREV Zanjani et al. (2016)
	Chouchen et al. (2021)	2021	Paths of the changed files in PRs	CHREV Zanjani et al. (2016), REVFINDER Thongtanunam et al. (2015), ReviewBot Balachandran (2013) and RevRec Ouni et al. (2016)
	Rong et al. (2022)	2022	Paths of the changed files in PRs	CHREV Zanjani et al. (2016), CN Yu et al. (2016), REVFINDER Thongtanunam et al. (2015), AC Jiang et al. (2017), RF de Lima Júnior et al. (2015) and EARec Ying et al. (2016)
Labeling bias and reasonability of recommendation	Tecimer et al. (2021)	2021	Paths of the changed files in PRs	–
	Gauthier et al. (2021)	2021	Comments in PRs and code ownership	–

base at Microsoft. The results show that CHREV outperforms REVFINDER (Thongtanunam et al. 2015), xFinder (Kagdi et al. 2008), and RevCom (Zanjani et al. 2016).

Rahman et al. (2016) propose a code reviewer recommendation tool that recommends reviewers for a pull request by analyzing a developer's past experience with external software libraries and specialized technology (e.g., Google App Engine) that are used by the pull request. The authors evaluate the tool using 13,081 pull requests from 10 subject systems. The results show that the proposed tool outperforms the REVFINDER Thongtanunam et al. (2015).

Hannebauer et al. (2016) compare the performance of review-expertise-based algorithms and modification-based expertise algorithms and study which one of the set of algorithms are suitable for recommending code reviewers. The authors study six algorithms based on modification expertise and two algorithms based on review expertise. They evaluate the performance of these algorithms on four major FLOSS projects. The results indicate that the algorithms based on review expertise yield better recommendations than those based on modification expertise and the Weighted Review Count (WRC) algorithm achieves the best result.

Jiang et al. (2017) analyze different attributes that affect the precision and recall of commenter prediction in code reviewing process. The authors analyze 206,664 comments of 19,543 PRs from 8 popular projects in GitHub. Their results show that the activeness (e.g., recent activity of candidates) is an important attribute in predicting a commenter of a PR. The activeness based approach improves the top-3 precision and recall in recommending commenters in PRs.

Xia et al. (2017) study implicit relationships of data (e.g., identifying common interests between reviewers and contributors) in recommending reviewers. The authors propose a hybrid approach leveraging a latent factor model and a neighborhood method to capture implicit relationships and recommend reviewers based on such relationships. The comparison results with baseline approaches show that the proposed approach improves the precision with top-1 recommendation by 5.8%∼15.3%.

Fejzer et al. (2018) propose a method for recommending code reviewers based on profiles of individual developers. The profile of each developer is the multiset of all file segments from commits that are reviewed by the developer. The profiles can be updated as new reviews are made. The experimental results on four large projects show that the method obtains improved performance.

Liao et al. (2019) propose a reviewer recommendation algorithm that extracts topic information from the comments in PRs and generates an influence network between topics and reviewers. The proposed approach extracts the topic information of PR comments using the LDA topic model Blei et al. (2003). The HITS Anagnostopoulos et al. (2008) and PageRank (Kleinberg 1999) network approaches are used for calculating the influence value of each reviewer. Comparing the five projects, the proposed algorithm can obtain F1-score between 24.3% to 45.9%.

Jiang et al. (2019) propose a reviewer recommendation approach, namely TRFPre which leverages time decaying relationships and time decaying file similarities based on previous pull request decisions. The authors conduct experimental study on 138,373 PRs of 24 GitHub projects. The results show that TRFPre achieves better prediction performance than five state-of-the-art approaches.

Sülün et al. (2019) introduce a novel approach for automatic reviewer recommendation, named RSTrace, that mines the information stored in software artifact (e.g., code and test cases) traceability graphs. To recommend reviewers a score is generated for each developer. This score is proportional to the number of distinct paths available from the candidate reviewer

to a given artifact and inversely proportional to the path lengths. The developer who has the highest score is identified as the most appropriate reviewer for the changed files. An empirical evaluation on 37 open source projects shows that RSTrace yields promising top-k and MRR scores.

Strand et al. (2020) develop a reviewer recommendation tool named Carrot. Carrot uses the collaborative filtering approach to build interaction between the paths of the changed files and potential reviewers to make recommendations. The authors deploy Carrot in one Ericsson project and evaluate the performance of this tool. The results show that Carrot can identify relevant reviewers. However, the results of the static validation with 8 experienced code reviewers show that Carrot can not reduce the lead time and balance the workload of reviewers.

Pandya and Tiwari (2022) propose a hybrid approach named CORMS that computes similarities among file paths, projects/sub-projects, author information and prediction models to recommend reviewers. The authors evaluate the performance of CORMS on 20 projects of both Gerrit and GitHub and compare the results with REVFINDER. The results show that CORMS outperforms REVFINDER.

Balancing Between Recommendation Accuracy and Review Workload Distribution Asthana et al. (2019) develop a reviewer recommender (WhoDo) that uses a hybrid approach with a load balancing mechanism and a scoring function to prioritize developers who have previous experience on files and directories that are involved in code review. The results show that the use of load balancing leads to a skewed review assignment and a decrease in review latency.

The review workload is also considered by Al-Zubaidi et al. (2020). The authors develop a multi-objective search-based approach for recommending reviewers that searches for solutions to maximize the chance of participating in a review and minimize the skewness of review workload. The empirical results demonstrate that including review workload with other important information (e.g., code ownership) improves the performance of recommending reviewers.

Mirsaeedi and Rigby (2020) investigate knowledge loss with the suggestion to distribute knowledge through the code reviewing process to mitigate the files at risk of turnover. The authors evaluate review recommenders in three contexts: (1) ensuring the expertise of reviewers, (2) reducing review workload and (3) reducing the files at risk to turnover. The authors also develop the SOFIA recommender that combines the CHREV recommender, with the learning and retention recommender. The results show that the SOFIA can balance the studied three contexts.

Chouchen et al. (2021) propose a multi-objective search based approach, named WhoReview, that recommends the best set of code reviewers considering their expertise and current workload. WhoReview calculates the expertise of developers based on two pieces of information, such as (1) the frequency and recency of updating files that are similar to the changed files of a PR and (2) the reviewer's collaboration with the developer who submitted the code change. The evaluation results show that WhoReview outperforms state-of- the-art approach.

Rong et al. (2022) propose a hypergraph based recommender named HGRec that automatically recommends code reviewers by modeling high-order relationships (e.g., one PR with multiple reviewers). The authors evaluate the performance of HGRec on 12 OSS projects. The results indicate that HGRec outperforms the state-of-the-art recommenders on recommendation accuracy. In addition, HGRec recommends new reviewers in top-3 recommendations that may help to alleviate the workload congestion issue to some extent.

Labeling Bias and Reasonability of Recommendation Tecimer et al. (2021) study systematic labeling bias in code reviewer recommendation datasets. Systematic labeling bias occurs

due to wrong assumptions made by labelers during the labeling process or intrinsic properties of the observed data. Such bias results in ground truth datasets that contain incorrect labels. The authors introduce an approach to automatically detect and eliminate systematic labeling bias in code reviewer recommendation systems. The authors apply this approach on HIVE and QT Creator datasets. The results show that removing labeling bias with the proposed approach improves the Mean Reciprocal Rank of the evaluated techniques up to 26% in the datasets.

Gauthier et al. (2021) empirically study whether historical data is an appropriate benchmark for reviewer recommendation. The authors replicate two state-of-the-art code reviewer recommenders (e.g., CHREV and WLRREC) and apply them on 9,679 code reviews from GERRIT. The results show that the recommended reviewers of 74% of incorrect recommendations felt that they would have been comfortable reviewing the changes. This result indicates that recommendations labeled as incorrect because they do not match the past assignee may have been correct as well. Such a result inspired us to investigate the reasonability of the recommendations issued by our KU-based recommender and baselines (see RQ3).

Our work is different from the above studies because we introduce KUs and develop a recommender system using KUs that shows effectiveness in recommending code reviewers for PRs and rebalancing review assignments.

(B) Representation of Developers' Expertise Our paper is related to the work of Dey et al. (2021). The authors propose and construct Skill Space, a methodology to represent the expertise of developers by comparing vector representation of four important entities (e.g., developers, projects, programming languages, and APIs). The authors operationalize Skill Space using the World of Code (WoC) data that contains APIs extracted from source code changes of 17 programming languages. To represent Skill Space, the authors employ Doc2Vec text embedding where each document represents the APIs encountered in a single modified file, and the document tags represent the language, the project, and the developer. The evaluation strategy involves fitting the Doc2Vec model on past data and measuring the alignment (e.g., cosine similarity) between vectors representing any pair of developers, projects, and APIs. The results show that Skill Space representations can help identify important aspects between developers, projects, and APIs. For instance, the authors observe that developers are closely aligned to themselves in the Skill Space when developers use new APIs. The results also show that Skill Space affects the probability of a developer's pull requests being accepted.

Our work is different from the above work. Our approach for constructing developers' expertise profile with KUs is considerably easy to deploy (see Fig. 3 of the Section 3) whereas deep learning models (e.g., Doc2Vec) are often difficult to deploy and interpret in practice. In addition, we build reviewer recommender with KUs (e.g., KUREC), compare the performance with baselines and demonstrate the usefulness of our KU-based expertise representation in recommending code reviewers for PRs.

More generally, several studies have attempted to define metrics for measuring expertise of developers using different sources of information, such as source code changes (e.g., commits and pull requests) (McDonald and Ackerman 2000; McDonald 2001; Robbes and Röthlisberger 2013; Mockus and Herbsleb 2002; Girba et al. 2005; Moradi Dakhel et al. 2021), API usage (Ma et al. 2009; Vivacqua and Lieberman 2000; Liang et al. 2022), source code defects (Anvik and Murphy 2007), user interaction data (Fritz et al. 2007, 2010, 2014),

commit messages (Greene and Fischer 2016) and, README files (Wan et al. 2018; Hauff and Gousios 2015). We briefly highlight a few of them in the following paragraphs.

Moradi Dakhel et al. (2021) use the changes in syntax patterns of source code as a proxy for measuring programming expertise. The authors propose a model that relies on the Zipf distribution of syntactic patterns in artifacts that are produced by a developer to evaluate developer's mastery in programming syntax patterns. To evaluate their model's performance, the authors compared the performance of the model with two groups of Clustering and Classifier algorithms. The results show that the proposed approach outperforms the state-of-the-art approaches for classifying novice and expert programmers.

Robbes and Röthlisberger (2013) propose a novel approach to automatically evaluate expertise metrics by applying MSR techniques to the developer interaction repositories (e.g., version control data of projects). These interaction data repositories contain the sequence of activities that developers took to finish a given task, including the overall time elapsed to perform the task, The authors define two expertise metrics based on the past commit activity on the entities that are involved in the tasks. The authors observe that the expertise that exhibits a stronger negative correlation to time and a lower correlation to the task activity if found to be a better expertise metric.

Liang et al. (2022) present a novel method to detect OSS skills and prototype, namely DISKO (**D**etect**I**ng **SK**ills in **O**SS). The approach identifies a set of relevant signals, which are measurable activities or cues associated with four skills: (1) teaches others to be involved in the OSS project, (2) shows commitment towards the OSS project, (3) has knowledge in programming languages and (4) has familiarity with OSS practices. The authors evaluate the approach by surveying 455 OSS contributors. The authors observe that their approach can detect the presence of the skills with precision scores between 77% to 97%.

Our work is different from the above works because we define KUs that we operationalize using certification exams. These KUs act as proxies for knowledge or skill categories of programming languages. In this paper, we conduct an exploratory study of the expertise representation with KUs. We also demonstrate that such expertise representation with KUs can be effective for recommending code reviewers in collaborative platforms (e.g., GitHub).

(C) Knowledge Units (KUs) There have been several works on representing knowledge using KUs (Patel et al. 2018; Peták et al. 2020; Li et al. 2021; Peták and Houška 2018; Cogo et al. 2022). The importance of the hierarchical organization of ontology for an intelligent web is presented by Patel et al. (2018). The authors use KUs to represent the semantic web's ontological concepts. They show that KUs can help achieve context-sensitive inference power and acquire new knowledge easily.

Peták et al. (2020) construct a fuzzy expert system based on fuzzy KUs which are defined as specific and core production rules for the system. These KUs are used to help decide whether to upgrade an enterprise resource planning system. The results demonstrate that KUs can be useful for these expert systems' learning process. In another study, Peták and Houška (2018) demonstrate how complicated issues can be broken down into their most basic forms using a network of fuzzy KUs.

Von Solms and Futcher (2018) represent the important knowledge for studying system engineers' cybersecurity models into different KUs (e.g., Software, Component and Connection, System). To ascertain which topics are deemed to be crucial for inclusion in the module, the authors conduct interviews with engineering professionals from academia and industry. According to the interview results, the three most important areas for cybersecurity are software security, systems security, and organizational security.

Cogo et al. (2022) encode the Rust domain knowledge into 47 different knowledge units (KUs) to study the topical alignment of the Rust documentation. The authors extract these

KUs from both the Question & Answer (Q&A) websites and the official Rust documentation. The authors demonstrate the usefulness of KUs to study the difference between the concrete developers' information needs and the current state of the documentation.

In this paper, we use KUs as a means to create "thematic groupings that encompass multiple, related topics (Bishop et al. 2017)" of the Java programming language. More specifically, we define a programming language KU as a cohesive set of key capabilities that are offered by one or more building blocks of a given programming language (see Section 2). We identify Java programming language KUs from the topics and subtopics of the Java certification exams (e.g., the Oracle Java SE and EE certification exams).

8 Threats to Validity

Construct Validity One of the threats to construct validity is that we did not extract KUs from the third-party libraries that are used by the studied projects. Our objective is to understand the effectiveness of KUs in representing developer expertise. Therefore, we only focus on those KUs that are implemented by the developers of the studied projects. To ensure that we are less likely to miss the KUs that are implemented within the studied projects, we carefully implemented type binding resolutions using the Java JDT library (Eclipse 2020).

In this study, we use the certification exams for the Java programming language as guidelines to identify KUs. Our rationale is that certification exams cover important topics which are used to measure the expertise of developers in using the programming language. The set of KUs that we conceive in this paper are based on the core functionalities of the programming language that are documented in the topics of these exams. Therefore, our identified KUs could be limited and might not cover all the knowledge surrounded with the constructs and APIs of the Java programming language. In addition, certification exams of different programming language versions (e.g., Java SE 8 vs Java SE 11) might cover different topics. Our paper should be seen as a first attempt to introduce the notion of KUs and design how to operationalize these KUs. Future research should focus on refining our conceptualization of KUs and exploring richer operationalizations of KUs (e.g., supporting the detection of different design patterns).

We applied PCA to reduce the dimension of our KU expertise representation data before running the k-means clustering algorithm. To reduce the dimensionality of the data, one needs to select a few PCA components. To select these PCA components, we decided to use a threshold-based approach on the explained variance of the PCA components. We selected the first few PCA components that aggregate up to the 95% of the data which is a reasonable threshold for the explained variance of the data (Xibilia et al. 2020; Ghotra et al. 2017; Kondo et al. 2019).

Internal Validity There could be a threat in identifying the optimal number of clusters using KUs. Identifying an optimum number of clusters is a non-trivial task. However, we follow the silhouette method that has been widely used in prior works (Nidheesh et al. 2020) for determining the optimal number of clusters. In addition, we carefully calculate the silhouette value using the `sklearn.metric` package of Python Sklearn (2022).

External Validity In this study, we only focus on Java projects of GitHub. We were aware of the fact that GitHub contains projects that are not real-world projects (e.g., academic projects and toy projects). Therefore, we carefully filtered our studied projects using the list of real-world software projects curated by Munaiah et al. (2017). We encourage future studies to broaden the scope of our study and investigate how our findings apply to software projects

written in other popular programming languages (e.g., Python) that have certification exams in different development areas (e.g., web development). For example, popular programming languages for web development (e.g., Ruby on Rails, JavaScript, Python, ASP.Net, Node-js, and Objective-C) have their own certification exams (Vekariya 2018).

The results discussed in this paper do not necessarily generalize. We thus encourage future research to (i) use KUs and combine them with other metrics and (ii) evaluate whether our findings generalize to other subject systems that differ substantially from the ones we chose.

9 Conclusion

In this paper, we introduce *Knowledge Units* of programming languages to present developers' expertise and present how KU-based expertise can be useful for recommending code reviewers in PRs. We define a KU as a cohesive set of key capabilities that are offered by one or more building blocks of a given programming language. We operationalize our KUs via certification exams for the Java programming language. As a first paper on the topic, our main objective is to study if KU-based expertise representation (i.e., KUREC recommender) can help recommend code reviewers.

Our results of the study demonstrate that KU-based expertise profiles are useful for recommending code reviewers (RQ1). Combining KUREC with other recommenders resulted in even better recommenders that outperform the baselines (RQ2). And when KUREC deviates from the ground truth, we observe that the recommendations are still frequently reasonable (RQ3). We thus conclude that KUREC and AD_HYBRID are overall superior to the baseline recommenders that we studied. These encouraging results can be helpful to researchers who wish to further study developers' skills or build reviewer recommendation models. We thus invite researchers to (i) explore richer conceptualizations and operationalizations of KUs (e.g., supporting the detection of different design patterns), (ii) evaluate whether our findings generalize to other projects that differ from the ones we chose, (iii) investigate the effectiveness of KUs in solving other practical software engineering tasks that depend on the effective identification and mapping of developers' expertise, such as the development of strategies for increasing the truck factor of a project (Avelino et al. 2016; Ferreira et al. 2019).

Appendix A Java Certification Exams and Knowledge Units

A.1 Java SE 8 Programmer I Exam

Table 6 lists the topics and subtopics covered in the *Java SE 8 Programmer I Exam* as per Oracle's official webpage.

A.2 Java SE 8 Programmer II Exam

Table 7 lists the topics and subtopics covered in the *Java SE 8 Programmer II Exam* as per Oracle's official webpage.

Table 6 Topics and subtopics from the Java SE 8 Programmer I Exam

Exam topic	Exam subtopics
[T1] Java Basics	**[S1]** Define the scope of variables.
	[S2] Define the structure of a Java class.
	[S3] Create executable Java applications with a main method; run a Java program from the command line; produce console output.
	[S4] Import other Java packages to make them accessible in your code.
	[S5] Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.
[T2] Working with Data Types	**[S1]** Declare and initialize variableS(including casting of primitive data types).
	[S2] Differentiate between object reference variables and primitive variables.
	[S3] Know how to read or write to object fields.
	[S4] Explain an object's lifecycle (creation, "dereference by reassignment" and garbage collection).
[T3] Using Operators and Decision Constructs	**[S1]** Use Java operators; use parentheses to override operator precedence.
	[S2] Test equality between strings and other objects using `==` and `equals()`
	[S3] Create and use `if`, `if-else`, and ternary constructs
	[S4] Use a `switch` statement
[T4] Creating and Using Arrays	**[S1]** Declare, instantiate, initialize and use a one-dimensional array
	[S2] Declare, instantiate, initialize and use a multi-dimensional array
[T5] Using Loop Constructs	**[S1]** Create and use `while` loops
	[S2] Create and use `for` loops including the enhanced `for` loop
	[S3] Create and use `do-while` loops
	[S4] Use `break` statement
	[S5] Use `continue` statement
[T6] Working Methods and Encapsulation	**[S1]** Create methods with arguments and return values, including overloaded methods.
	[S2] Apply the `static` keyword to methods and fields.
	[S3] Create an overloaded method; differentiate between default and user defined constructors.
	[S4] Apply access modifiers.
	[S5] Apply encapsulation principles to a class.
	[S6] Determine the effect upon object references and primitive values when they are passed into methods that change the values.

Table 6 continued

Exam topic	Exam subtopics
[T7] Working with Inheritance	**[S1]** Describe inheritance and its benefits.
	[S2] Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object.
	[S3] Determine when casting is necessary.
	[S4] Use `super` and `this` to access objects and constructors.
	[S5] Use abstract classes and interfaces..
[T8] Handling Exceptions	**[S1]** Differentiate among checked exceptions, `RuntimeException`, and `Error`.
	[S2] Create a `try-catch` block and determine how exceptions alter normal program flow.
	[S3] Describe the advantages of exception handling.
	[S4] Create and invoke a method that throws an exception.
	[S5] Recognize common exception classes and categorieS(such as `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`).
[T9] Working with Selected classes from the Java API	**[S1]** Manipulate data using the `StringBuilder` class and its methods
	[S2] Create and manipulate strings.
	[S3] Create and manipulate calendar data using classes from `java.time.LocalDateTime`, `java.time.LocalDate`, `java.time.LocalTime`, `java.time.format.DateTimeFormatter`, `java.time.Period`.
	[S4] Declare and use an ArrayList of a given type.
	[S5] Write a simple Lambda expression that consumes a Lambda Predicate expression.

A.3 Java EE Developer Exam

Table 8 lists the topics and subtopics covered in the *Java EE Developer Exam* as per Oracle's official webpage.

A.4 Mapping Process From Certification Exams to Knowledge Units (KUs)

In our mapping process, the first two authors work collaboratively to minimize subjectivity bias. The authors conducted the entire mapping process together and engaged in extensive discussions regarding the handling of corner cases. The collaboration entails a shared effort between the first two authors to conduct the mapping process together. This involves jointly reviewing the topics and subtopics of exam certifications, mapping the topics to KUs, and discussing how to handle corner cases. It also includes ongoing communication to ensure a

Table 7 Topics and subtopics from the Java SE 8 Programmer II Exam

Exam topic	Exam subtopic
[T10] Java Class Design	**[S1]** Implement encapsulation
	[S2] Implement inheritance including visibility modifiers and composition
	[S3] Implement polymorphism
	[S4] Override `hasCode`, `equals`, and `toString` methods from `Object` class
	[S5] Create and use singleton classes and immutable classes.
	[S6] Develop code that uses the `static` keyword on initialize blocks, variables, methods, and classes.
[T11] Advanced Class Design Knowledge Unit	**[S1]** Develop code that uses abstract classes and methods.
	[S2] Develop code that uses the `final`.
	[S3] Create inner classes including static inner classes, local classes, nested classes, and anonymous innter classes.
	[S4] Use enumerated types including methods, and constructors in an `enum` type
	[S5] Develop code that declares, implements and/or extends interfaces and use the `@Override` annotation.
	[S6] Create and use lambda expressions.
[T12] Generics and Collection Knowledge Unit	**[S1]** Create and use a generic class.
	[S2] Create and use `ArrayList`, `TreeSet`, `TreeMap`, and `ArrayDeque` objects.
	[S3] Use `java.util.Comparator` and `java.lang.Comparable` interfaces.
	[S4] Collections, streams, and filters
	[S5] Iterate using `forEach` methods of `Streams` and `List`.
	[S6] Describe the `Stream` interface and the `Stream` pipeline.
	[S7] Filter a collection by using lambda expressions.
	[S8] Use method references with streams.
[T13] Lambda Built-in Functional Interfaces	**[S1]** Use the built-in interfaces included in the `java.util.function` package such as `Predicate`, `Consumer`, `Function`, and `Supplier`.
	[S2] Develop code that uses primitive versions of functional interfaces.
	[S3] Develop code that uses binary versions of functional interfaces.
	[S4] Develop code that uses the `UnaryOperator` interface.

Table 7 continued

Exam topic	Exam subtopic
[T14] Java Stream API	**[S1]** Develop code to extract data from an object using `peek()` and `map()` methods including primitive versions of the `map()` method
	[S2] Search for data by using search methods of the Stream classes including `findFirst`, `findAny`, `anyMatch`, `allMatch`, `noneMatch`
	[S3] Develop code that uses the Optional class
	[S4] Develop code that uses Stream data methods and calculation methods
	[S5] Sort a collection using Stream API
	[S6] Save results to a collection using the collect method and group/partition data using the Collectors class
	[S7] Use `flatMap()` methods in the Stream API.
[T15] Exceptions and Assertions	**[S1]** Use `try-catch` and `throws` statements
	[S2] Use `catch`, `multi-catch`, and `finally` clauses
	[S3] Use autoclose resources with a `try-with-resources` statement
	[S4] Create custom exceptions and autocloseable resources
	[S5] Test invariants by using assertions
[T16] Use Java SE 8 Data/Time API	**[S1]** Create and manage date-based and time-based events including a combination of date and time into a single object using `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, and `Duration`
	[S2] Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values.
	[S3] Define and create and manage date-based and time-based events using `Instant`, `Period`, `Duration`, and `TemporalUnit`
[T17] Java I/O Fundamentals	**[S1]** Read and write data using the console.
	[S2] Use `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `FileInputStream`, `FileOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter` in the `java.io package`
[T18] Java File I/O (NIO.2)	**[S1]** Use the `Path` interface to operate on file and directory paths
	[S2] Use the `Files` class to check, read, delete, copy, move, and manage metadata a file or directory
	[S3] Use Stream API with NIO.2

Table 7 continued

Exam topic	Exam subtopic
[T19] Object-Oriented Design Principles	**[S1]** Write code that declares, implements and/or extends interfaces.
	[S2] Choose between interface inheritance and class inheritance
	[S3] Develop code that implements "is-a" and/or "has-a" relationships.
	[S4] Apply object composition principles.
	[S5] Design a class using the Singleton design pattern.
	[S6] Write code to implement the DAO pattern.
	[S7] Design and create objects using a factory, and use factories from the API.
[T20] String Processing	**[S1]** Search, parse and build strings
	[S2] Search, parse, and replace strings by using regular expressions.
	[S3] Use string formatting
[T21] Concurrency	**[S1]** Create worker threads using `Runnable`, `Callable` and use an `ExecutorService` to concurrently execute tasks
	[S2] Identify potential threading problems among deadlock, starvation, livelock, and race conditions.
	[S3] Use `synchronized` keyword and `java.util.concurrent.atomic` package to control the order of thread execution.
	[S4] Use `java.util.concurrent` collections and classes including `CyclicBarrier` and `CopyOnWriteArrayList`.
	[S5] Use parallel Fork/Join Framework
[T22] Building Database Applications with JDBC	**[S1]** Describe the interfaces that make up the core of the JDBC API including the `Driver`, `Connection`, `Statement`, and `ResultSet` interfaces and their relationship to provider implementations.
	[S2] Identify the components required to connect to a database using the `DriverManager` class including the JDBC URL.
	[S3] Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections.
[T23] Localization	**[S1]** Read and set the locale by using the `Locale` object.
	[S2] Create and read a `Properties` file.
	[S3] Build a resource bundle for each locale and load a resource bundle in an application.

Table 8 Topics and subtopics from the Java EE Developer Exam

Exam topic	Exam subtopic
[T24] Understanding Java EE Architecture	**[S1]** Describe Java EE 7 standards, containers, APIs, and services
	[S2] Differentiate between application component functionalities
	[S3] Demonstrate understanding of Enterprise JavaBeans and CDI beans, their lifecycle and memory scopes
	[S4] Demonstrate understanding of the relationship between bean components, annotations, injections, and JNDI
	[S5] Create and use singleton classes and immutable classes
	[S6] Develop code that uses the static keyword on initialize blocks, variables, and methods
[T25] Manage Persistence using JPA Entities and BeanValidation	**[S1]** Create JPA Entity and Object-Relational Mappings (ORM)
	[S2] Use Entity Manager to perform database operations, transactions, and locking with JPA entities
	[S3] Create and execute JPQL statements
[T26] Implement Business Logic by Using EJBs	**[S1]** Create session EJB components containing synchronous and asynchronous business methods, manage the life cycle container callbacks, and use interceptors
	[S2] Create EJB timers
	[S3] Demonstrate understanding of how to control EJB transactions, distinguish Container Managed (CMT) and Bean Managed (BMT) transactions
[T27] Use Java Message Service API	**[S1]** Implement Java EE message producers and consumers, including Message-Driven beans
	[S2] Use transactions with JMS API
[T28] Implement SOAP Services by Using JAX-WS and JAXB API	**[S1]** Create SOAP Web Services and Clients using JAX-WS API
	[S2] Create marshall and unmarshall Java Objects by using JAXB API
[T29] Create Java Web Applications using Servlets	**[S1]** Create Java Servlet and use HTTP methods
	[S2] Handle HTTP headers, parameters, cookies
	[S3] Manage servlet life cycle with container callback methods and WebFilters
[T30] Create Java Web Applications using JSPs	**[S1]** Describe JSP life cycle
	[S2] Use JSP syntax, use tag libraries and Expression Language (EL)
	[S3] Handle errors using Java Server Pages
[T31] Implement REST Services using JAX-RS API	**[S1]** Apply REST service conventions
	[S2] Create REST Services and clients using JAX-RS API

Table 8 continued

Exam topic	Exam subtopic
[T32] Create Java Applications using WebSockets	**[S1]** Understand and utilise WebSockets communication style and lifecycle
	[S2] Create WebSocket Server and Client Endpoint Handlers
	[S3] Produce and consume, encode and decode WebSocket messages
[T33] Develop Web Applications using JSFs	**[S1]** Describe JSF architecture, lifecycle and navigation
	[S2] Use JSF syntax and use JSF Tag Libraries
	[S3] Handle localisation and produce messages
	[S4] Use Expression Language (EL) and interact with CDI beans
[T34] Secure Java EE 7 Applications	**[S1]** Describe Java EE declarative and programmatic security and configure authentication using application roles and security constraints and Login Modules
	[S2] Describe WebServices security standards
[T35] Use CDI Beans	**[S1]** Create CDI Bean Qualifiers, Producers, Disposers, Interceptors, Events, and Stereotypes
[T36] Use Concurrency API in Java EE 7 Applications	**[S1]** Demonstrate understanding of Java Concurrency Utilities and use Managed Executors
[T37] Use Batch API in Java EE 7 Applications	**[S1]** Implement batch jobs using JSR 352 API

consistent and coherent approach to data analysis. It is important to note that the authors did not calculate interrater agreement for this mapping process. Our decision to forgo interrater agreement analysis was based on our collaboration approach, which aimed to achieve a shared and consistent understanding of the data (e.g., topics and subtopics of Java certification exams).

We detail our process for mapping from certification exams to KUs. To map from certification exams to KUs, we follow three steps. First, we exclude subtopics (from either exam) that are unsuitable for our study. Next, we rearrange subtopics (within exams and across exams) such that subtopics lie in the most specific topic possible. Finally, we convert each topic into a KU (one-to-one mapping) and interpret its subtopics as the key capabilities of the KU. Tables 6 and 7 show the topics and subtopics of the Java SE 8 certification exam I (E1) and the Java SE 8 certification exam II (E2), respectively. In the following, we explain the mapping process in detail (we use the notation [Ei, Tj, Sk] to refer to a subtopic Sk from topic Tj of exam Ei).

Exclusion of Subtopics We only considered the subtopics that can be automatically detected from the source code using static analysis. For instance, the subtopic "Use autoclose resources with a try-with-resources statement" can be detected by searching for "try-with-resources" code blocks using an appropriate Java parser. Our rationale is to ensure that KUs are computable for any Java system, irrespective of how those are built or used by end-users. In this vein, subtopics that are inherently conceptual or that pose great challenges to be detected in practice (e.g., involve problems that are still under investigation by the SE research community) are excluded from our operationalization. We discuss all individual cases below.

The subtopic [E1, T1, S5] under "[E1, T1] Java Basics" is about comparing and contrasting the features and components of Java, such as: platform independence, object orientation, encapsulation, etc. This topic is thus more of a concept, which can not be operationalized and extracted automatically. After discussion, the first two authors concluded that none of the subtopics of "[E1, T1] Java Basics" can be extracted automatically (e.g., [E1, T1, S5] comparing and contrasting the features and components of Java such as: platform independence, object orientation, encapsulation, etc). As a consequence, we did not include "[E1, T1] Java Basics" in our study.

Some of the subtopics of topics are about the understanding of a concept which can not be operationalized and extracted automatically. For example, the subtopic "[E1, T7, S1] Describe inheritance and its benefits" is about understanding the concept of inheritance. The subtopic "[E1, T8, S3] Describe the advantages of exception handling" is about understanding the concept of exception handling. The authors did not include such cases of subtopics as capabilities of KUs. We highlight all these cases in Table 9.

The authors decided not to include the topic "[E2, T19] Object-Oriented Design Principles" in our study. The topic "[E2, T19] Object-Oriented Design Principles" is about the implementation and detection of different design patterns (e.g., singleton design pattern and factory design pattern). Detection of such design patterns is a non-trivial task and is itself an active research area in the software engineering community (Tsantalis et al. 2006; Xiong and Li 2019). Hence, the authors did not include the topic "[E2, T19] Object-Oriented Design Principles" in this study.

The authors did not include the topic "[E3, T30] Create Java Web Application using JSPs". To create Java web application using JSPs, developers need to create a JSP server pages (e.g., files with ".jsp" extension) containing the JSP tags which are different from Java files. In this study, we only focus on studying the Java files (e.g., files with ".java" extension). Therefore, the authors decided not to include the topic "[E3, T30]" in this study.

Rearrangement of Subtopics Certain subtopics from exam E1 are conceptually related to topics of exam E2 and vice-versa. In those cases, we prioritize the topic that is more specific and move the subtopic in question to that topic. For example, "[E2, T10] Java Class Design" contains the subtopic "[E2, T10, S2] Implement inheritance including visibility modifiers and composition". However, such a subtopic would also fit under "[E1, T7] Working with Inheritance." Since the latter topic is more specific than the former, we move [E2, T10, S2] into [E1, T7]. We also observed rare cases in which it made sense to perform subtopic moving within an exam (as opposed to cross exams). In the following, we describe all subtopic rearrangements in detail.

The subtopics of "[E1, T9] Working with selected classes from the Java API" were moved into several topics of E2:

(1) The subtopics "[E1, T9, S1] Manipulate data using the StringBuilder class and its methods" and "[E1, T9, S2] Create and manipulate strings" were moved into "[E2, T10] String Processing".
(2) The subtopic "[E1, T9, S3] Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate,java.time.LocalTime, java.time.format. DateTimeFormatter, java.time.Period." was moved into "[E2, T16] Use Java SE 8 Date/Time API".

(3) The subtopic "[E1, T9, S4] Declare and use an ArrayList of a given type" was moved into "[E2, T12] Generics and Collection".
(4) The subtopic "[E1, T9, S5] Write a simple Lambda expression that consumes a Lambda Predicate expression" was moved into "[E2, T13] Lambda Built-in Functional Interfaces".

The subtopics "[E1, T8, S2] Create a try-catch block" and "[E1, T8, S4] Create and invoke a method that throws an exception" were moved from "[E1.T8] Handling Exception" into "[E2.T15] Exceptions and Assertions".

All subtopics from the topic "[E2, T10] Java Class Design" were moved into other topics of E1 and E2 as follows:

(1) The subtopics "[E2, T10, S1] Implement encapsulation", "[E2, T10, S4] Override hasCode, equals, and toString methods from Object class", and "[E2, T10, S6] Develop code that uses the static keyword on initialize blocks, variables, and methods" were moved into "[E1, T6] Working with Methods and Encapsulation".
(2) The subtopics "[E2, T10, S2] Implement inheritance including visibility modifiers and composition" and "[E2, T10, S3] Implement polymorphism" were moved into "[E1, T7] Working with Inheritance".
(3) The subtopic "[E2, T10, S5] Create and use singleton classes and immutable classes" was moved into "[E2, T11] Advanced Class Design".
Three subtopics of the topic "[E2, T12] Generics and Collection KU" were moved into "[E2, T14] Java Stream API". The subtopics (1) "[E2, T12, S5] Iterate Collection using forEach methods of Streams", (2) "[E2, T12, S7] Filter a collection by using Stream filter API with lambda expressions", and (3) "[E2, T12, S8] Use method references with streams" were moved into "[E2, T14] Java Stream API".

One-to-one Mappings After performing the exclusions and subtopic rearrangements, we created a one-to-one mapping between topics and KUs. For example, the topic "[E1, T5] Using Loop Constructs" is mapped to the "[K4] Loop KU". All of the subtopics in [E1, T5] are interpreted as key capabilities associated with [K4]. For instance, the subtopic "[E1, T5, S1] Create and use while loops" is considered a key capability associated with the Loop KU. As a result of our mapping process, we identified 28 KUs, which are shown in Table 9.

B Sensitivity Analysis of Choosing a Threshold for Identifying Reasonable Recommendations

The developer who has more prior knowledge about the changed files of the PR is more capable of reviewing that PR.

Prior study also uses a threshold to identify a developer as a reasonable reviewer (Al-Zubaidi et al. 2020). Al-Zubaidi et al. (2020) identify a developer as a reasonable reviewer if the developer previously reviewed at least one of the changed files of the PR in question. We believe that such a threshold could be very low resulting that the developer may not have recent knowledge of the majority of the remaining changed files of the PR. We need to set a threshold that is not very low or very high. A low threshold value means that developers have minimum prior knowledge about the changed files. On the other hand, a high threshold value may not satisfy our objective of distributing workload to other capable reviewers. To identify an appropriate threshold we perform a sensitivity analysis. We calculate the percentage of reasonable recommendations across recommenders using three different threshold values such

Table 9 The mapping from certification exams to Knowledge Unit (KUs)

Exam	Topic	Subtopic	Mapping Action	Knowledge Unit (KU)	Reason
[E1] Java SE 8 Programmer I certification exam	[T1] Java Basics	[S1] Define the scope of variables	Excluded	–	[E1, T1] is basically about the understanding of the concept of the Java programming language and knowledge about how to run a Java program. None of the subtopics of [E1, T1] can be operationalized and extracted automatically. So, [E1, T1] is excluded from this study.
		[S2] Define the structure of a Java class			
		[S3] Understand executable Java applications with the main method			
		[S4] Understand the process of importing packages to make them accessible in your code			
		[S5] Compare and contrast the features and components of Java such as platform independence, object orientation, encapsulation, etc.			
	[T2] Working with Data Types	[S1] Declare and initialize variableS(including casting of primitive data types)	Mapped to KU	[K1] Data Type KU	One to one mapping.
		[S2] Differentiate between object reference variables and primitive variables	Excluded	–	These subtopics of topics are about the understanding of a concept which can not be operationalized and extracted automatically.

Table 9 continued

Exam	Topic	Subtopic	Mapping Action	Knowledge Unit (KU)	Reason
		[S3] Know how to read or write to object fields			
		[S4] Explain an object's life-cycle (creation, "dereference by reassignment" and garbage collection)			
	[T3] Using Operators and Decision Constructs	[S1] Use Java operators; use parentheses to override operator precedence	Mapped to KU	[K2] Operator and Decision KU	One to one mapping.
		[S2] Test equality between strings and other objects using "=" and "equals( )".			
		[S3] Create and use if, if-else, and ternary constructs			
		[S4] Use a switch statement			
	[T4] Creating and Using Arrays	[S1] Declare, instantiate, initialize and use a one-dimensional array	Mapped to KU	[K3] Array KU	One to one mapping.
		[S2] Declare, instantiate, initialize and use a multi-dimensional array			
	[T5] Using Loop Constructs	[S1] Create and use "while" loops.	Mapped to KU	[K4] Loop KU	One to one mapping.
		[S2] Create and use for loops, including the "enhanced for" loop			

Table 9 continued

Exam	Topic	Subtopic	Mapping Action	Knowledge Unit (KU)	Reason
		[S3] Create and use do-while loops			
		[S4] Use break statement			
		[S5] Use continue statement			
	[T6] Working with Methods and Encapsulation	[S1] Create methods with arguments and return values, including overloaded methods	Mapped to KU	[K5] Method and Encapsulation KU	One to one mapping.
		[S2] Apply the static keyword to methods and fields			
		[S3] Create an overloaded method; create default and user-defined constructors			
		[S3] Create an overloaded method; create default and user-defined constructors			
		[S5] Apply encapsulation principles to a class			
		[S6] Determine the effect upon object references and primitive values when they are passed into methods that change the values	Excluded	–	This subtopic [E1, T6, S6] is about the understanding of a concept which can not be operationalized and extracted automatically.
		[S7] Create a constructor chaining (use"this()" method to call one constructor from another constructor	Mapped to KU	[K5] Method and Encapsulation KU	One to one mapping.
		[S8] Use variable length arguments in the methods			

Table 9 continued

Exam	Topic	Subtopic	Mapping Action	Knowledge Unit (KU)	Reason
	[T7] Working with Inheritance	[S1] Describe inheritance and its benefits	Excluded	–	This subtopic [E1, T7, S1] is about the understanding of a concept which can not be operationalized and extracted automatically.
		[S2] Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object	Mapped to KU	[K6] Inheritance KU	One to one mapping.
		[S3] Use casting to subclass			
		[S4] Use super and this to access objects and constructors			
		[S5] Use abstract classes and interfaces			
	[T8] Handling Exceptions	[S1] Differentiate among checked exceptions, RuntimeException, and Error	Excluded	–	The subtopic [E1, T8, S1] is about the understanding of a concept which can not be operationalized and extracted automatically.
		[S2] Create a try-catch block and determine how exceptions alter normal program flow	Moved into [E2, T15] Exceptions and Assertions	[K11] Exception and Assertion KU	Avoid conceptual overlap between the two certification exams.

Table 9 continued

Exam	Topic	Subtopic	Mapping Action	Knowledge Unit (KU)	Reason
		[S3] Describe the advantages of exception handling	Excluded	–	The subtopic [E1, T8, S3] is about the understanding of a concept which can not be operationalized and extracted automatically.
		[S4] Create and invoke a method that throws an exception	Moved into [E2, T15] Exceptions and Assertions	[K11] Exception KU	Avoid conceptual overlap between the two certification exams.
		[S5] Use common exception classes and categories(such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException)			

Table 10 Knowledge Units derived from the Java SE 8 Programmer I Exam, Java SE 8 Programmer II Exam, and Java EE Developer Exam

Knowledge Unit (KU)	Key Capabilities
[K1] Data Type KU	**[C1]** Declare and initialize different types of variableS(e.g., primitive type, parameterized type, and array type), including the casting of primitive data types
[K2] Operator and Decision KU	**[C1]** Use Java operators(e.g., assignment and postfix operators); use parentheses to override operator precedence
	[C2] Test equality between strings and other objects using == and `equals()`
	[C3] Create and use `if`, `if-else`, and ternary constructs
	[C4] Use a `switch` statement
[K3] Array KU	**[C1]** Declare, instantiate, initialize and use a one-dimensional array
	[C2] Declare, instantiate, initialize and use a multi-dimensional array
[K4] Loop KU	**[C1]** Create and use `while` loops
	[C2] Create and use `for` loops, including the `enhanced for` loop
	[C3] Create and use `do-while` loops
	[C4] Use `break` statement
	[C5] Use `continue` statement
[K5] Method and Encapsulation KU	**[C1]** Create methods with arguments and return values
	[C2] Apply the "static" keyword to methods, fields, and blocks
	[C3] Create an overloaded method and overloaded constructor
	[C4] Create a constructor chaining (use "this()" method to call one constructor from another constructor
	[C5] Use variable length arguments in the methods
	[C6] Use different access modifiers (e.g., private and protected) other than "default"
	[C7] Apply encapsulation: identify set and get method to initialize any private class variables
	[C8] Apply encapsulation: Immutable class generation-final class and initialize private variables through the constructor
[K6] Inheritance KU	**[C1]** Use basic polymorphis (e.g., a superclass refers to a subclass)
	[C2] Use polymorphic parameter (e.g., pass instances of a subclass or interface to a method)
	[C3] Create overridden methods

Table 10 continued

Knowledge Unit (KU)	Key Capabilities
	[C4] Create "abstract" classes and "abstract" methods
	[C5] Create "interface" and implement the interface
	[C6] Use "super()" and the "super" keyword to access the members(e.g., fields and methods) of a parent class
	[C7] Use casting in referring a subclass object to a superclass object
[K7] Advanced Class Design KU	**[C1]** Create inner classes, including static inner classes, local classes, nested classes, and anonymous inner classes
	[C2] Develop code that uses the final
	[C3] Use enumerated types including methods and constructors in an "enum" type
	[C4] Create singleton classes and immutable classes
[K8] Generics and Collection KU	**[C1]** Create and use a generic class
	[C2] Create and use `ArrayList`, `TreeSet`, `TreeMap`, and `ArrayDeque`
	[C3] Use `java.util.Comparator` and `java.lang.Comparable` interfaces
	[C4] Iterate using forEach methods of List
[K9] Functional Interface KU	**[C1]** Use the built-in interfaces included in the `java.util.function` packages such as `Predicate`, `Consumer`, `Function`, and `Supplier`
	[C2] Develop code that uses primitive versions of functional interfaces
	[C3] Develop code that uses binary versions of functional interfaces
	[C4] Develop code that uses the `UnaryOperator` interface
[K10] Stream API KU	**[C1]** Develop code to extract data from an object using `peek()` and `map()` methods, including primitive versions of the `map()` method
	[C2] Search for data by using search methods of the Stream classes, including `findFirst`, `findAny`, `anyMatch`, `allMatch`, `noneMatch`
	[C3] Develop code that uses the Optional class
	[C4] Develop code that uses Stream data methods and calculation methods

Table 10 continued

Knowledge Unit (KU)	Key Capabilities
	[C5] Sort a collection using Stream API
	[C6] Save results to a collection using the collect method
	[C7] `UseflatMap()` methods in the Stream API
[K11] Exception KU	**[C1]** Create a try-catch block
	[C2] Use catch, multi-catch, and finally clauses
	[C3] Use autoclose resources with a try-with-resources statement
	[C4] Create custom exceptions and autocloseable resources
	[C5] Create and invoke a method that throws an exception
	[C6] Use common exception classes and categories(such as `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`)
	[C5] Use assertions
[K12] Date Time API KU	**[C1]** Create and manage date-based and time-based events including a combination of date and time into a single object using `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, and `Duration`
	[C2] Formatting date and times values for using different timezones
	[C3] Create and manage date-based and time-based events using Instant, Period, Duration, and Temporal Unit
	[C4] Create and manipulate calendar data using classes from `java.time.LocalDateTime`, `java.time.LocalDate`, `java.time.LocalTime`, `java.time.format.DateTimeFormatter`, and `java.time.Period`
[K13] IO KU	**[C1]** Read and write data using the console
	[C2] Use `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `FileInputStream`, `FileOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter` in the `java.io` package
[K14] NIO KU	**[C1]** Use the Path interface to operate on file and directory paths
	[C2] Use the Files class to check, read, delete, copy, move, and manage metadata a file or directory
[K15] String Processing KU	**[C1]** Search, parse and build strings
	[C2] Manipulate data using the `StringBuilder` class and its methods

Table 10 continued

Knowledge Unit (KU)	Key Capabilities
	[C3] Use regular expression using the `Pattern` and `Matcher` class
	[C4] Use string formatting
[K16] Concurrency KU	**[C1]** Create worker threads using `Runnable`, `Callable` and use an `ExecutorService` to concurrently execute tasks
	[C2] Use `synchronized` keyword and `java.util.concurrent.atomic` package to control the order of thread execution
	[C3] Use `java.util.concurrent` collections and classes including `CyclicBarrier` and `CopyOnWriteArrayList`
	[C4] Use parallel Fork/Join Framework
[K17] Database KU	**[C1]** Describe the interfaces that make up the core of the JDBC API, including the `Driver`, `Connection`, `Statement`, and `ResultSet` interfaces
	[C2] Submit queries and read results from the database, including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections
[K18] Localization KU	**[C1]** Read and set the locale by using the Locale object
	[C2] Build a resource bundle for each locale and load a resource bundle in an application
[K19] Java Persistence KU	**[C1]** Create JPA Entity and Object-Relational Mappings (ORM)
	[C2] Use Entity Manager to perform database operations, transactions, and locking with JPA entities
	[C3] Create and execute JPQL statements
[K20] Enterprise Java Bean KU	**[S1]** Create session EJB components containing synchronous and asynchronous business methods, manage the life cycle container callbacks, and use interceptors.
	[S2] Create EJB timers

Table 10 continued

Knowledge Unit (KU)	Key Capabilities
[K21] Java Message Service API KU	**[S1]** Implement Java EE message producers and consumers, including Message-Driven beans
	[S2] Use transactions with JMS API
[K22] SOAP Web Service KU	**[S1]** Create SOAP Web Services and Clients using JAX-WS API
	[S2] Create marshall and unmarshall Java Objects by using JAXB API
[K23] Servlet KU	**[S1]** Create Java Servlet and use HTTP methods
	[S2] Handle HTTP headers, parameters, cookies
	[S3] Manage servlet life cycle with container callback methods and WebFilters
[K24] Java REST API KU	**[S1]** Apply REST service conventions
	[S2] Create REST Services and clients using JAX-RS API
[K25] Websocket KU	**[S1]** Create WebSocket Server and Client Endpoint Handlers
	[S3] Produce and consume, encode and decode WebSocket messages
[K26] Java Server Faces KU	**[S1]** Use JSF syntax and use JSF Tag Libraries
	[S2] Handle localisation and produce messages
	[S3] Use Expression Language (EL) and interact with CDI beans
[K27] Contexts and Dependency Injection (CDI) KU	**[S1]** Create CDI Bean Qualifiers, Producers, Disposers, Interceptors, Events, and Stereotypes
[K28] Batch Processing KU	**[S1]** Implement batch jobs using JSR 352 API

Table 11 The percentage of reasonable reviewers across recommenders using different thresholds

Recommender Name	Percentage of reasonable recommendation		
	Threshold=30	Threshold=50	Threshold=80
KUREC	70.1	63.4	45.1
AD_REC	68.2	61.5	43.2
AD_HYBRID	68.1	61.3	42.9
ER	68.0	60.9	42.3
AD_FREQ	66.9	54.2	42.6
CHREV	31.8	30.4	24.9
CF	28.4	25.4	17.9
CORRECT	28.3	24.1	16.1
SOFIA	24.3	21.6	15.1
RF	19.2	15.2	9.6
CN	11.7	10.6	8.1

as 30, 50 and 80 (see Table 11). We observe that our initial assumptions hold for all of these threshold values: KUREC and the combined recommender (e.g., AD_HYBRID) achieve the highest percentage of reasonable recommendation. Finally, we select the threshold=50 as an appropriate threshold value for our study since the value is not very low or not very high.

Data Availability Statement (DAS) A supplementary material package is provided online in the following link: http://www.bit.ly/3bhSFux. The contents will be made available on a public GitHub repository once the paper is accepted.

Declarations

Conflicts of interests The authors declared that they have no conflict of interest.

References

Adomavicius G, Zhang J (2014) Improving stability of recommender systems: a meta-algorithmic approach. IEEE Trans Knowl Data Eng 27(6):1573–1587

Al-Subaihin AA, Sarro F, Black S, Capra L, Harman M, Jia Y, Zhang Y (2016) Clustering Mobile Apps Based on Mined Textual Features. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, ESEM'16

Al-Zubaidi WHA, Thongtanunam P, Dam HK, Tantithamthavorn C, Ghose A (2020) Workload-aware reviewer recommendation using a multi-objective search-based approach. In: Proceedings of the 16th ACM international conference on predictive models and data analytics in software engineering, pp 21–30

Anagnostopoulos A, Kumar R, Mahdian M (2008) Influence and correlation in social networks. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, pp 7–15

Anvik J, Murphy GC (2007) Determining Implementation Expertise from Bug Reports. In: Proceedings of the 4th international workshop on mining software repositories, pp 1–8

Asthana S, Kumar R, Bhagwan R, Bird C, Bansal C, Maddila C, Mehta S, Ashok B (2019) Whodo: automating reviewer suggestions at scale. In: Proceedings of the 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 937–945

Avelino G, Passos L, Hora A, Valente MT (2016) A novel approach for estimating truck factors. In: Proceedings of the 24th international conference on program comprehension (ICPC), IEEE, pp 1–1

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 international conference on software engineering, IEEE Press, ICSE'13, p 712–721

Balachandran V (2013) Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 35th international conference on software engineering (ICSE), pp 931–940
Bishnu PS, Bhattacherjee V (2012) Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm. IEEE Trans Knowl Data Eng 24(6):1146–1150
Bishop M, Burley D, Buck S, Ekstrom JJ, Futcher L, Gibson D, Hawthorne EK, Kaza S, Levy Y, Mattord H et al. (2017) Cybersecurity curricular guidelines. In: IFIP world conference on information security education, pp 3–13
Blei DM, Ng AY, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res 3(Jan):993–1022
Campos PG, Díez F, Sánchez-Montañés M (2011) Towards a more realistic evaluation: testing the ability to predict future tastes of matrix factorization-based recommenders. In: Proceedings of the 5th ACM conference on recommender systems, pp 309–312
Chouchen M, Ouni A, Mkaouer MW, Kula RG, Inoue K (2021) Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. Appl Soft Comput 100:106908
Cogo FR, Xia X, Hassan AE (2022) Assessing the alignment between the information needs of developers and the documentation of programming languages: A case study on rust. arXiv:2202.04431
de Lima Júnior ML, Soares DM, Plastino A, Murta L (2015) Developers assignment for analyzing pull requests. In: Proceedings of the 30th annual ACM symposium on applied computing, pp 1567–1572
Dey T, Karnauch A, Mockus A (2021) Representation of Developer Expertise in Open Source Software. In: Proceedings of the 43rd international conference on software engineering, p 995–1007
Ding C, He X (2004) K-Means Clustering via Principal Component Analysis. In: Proceedings of the 21st international conference on machine learning, pp 29–3
Eclipse (2020) Eclipse Java development tools (JDT) . http://www.eclipse.org/jdt/, (Last accessed: April 2023)
Fejzer M, Przymus P, Stencel K (2018) Profile based recommendation of code reviewers. J Intell Inf Syst 50:597–619
Ferreira M, Mombach T, Valente MT, Ferreira K (2019) Algorithms for estimating truck factors: a comparative study. Softw Qual J 27(4):1583–1617
Fritz T, Murphy GC, Hill E (2007) Does a Programmer's Activity Indicate Knowledge of Code? In: Proceedings of the 6th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, p 341–350
Fritz T, Murphy GC, Murphy-Hill E, Ou J, Hill E (2014) Degree-of-Knowledge: Modeling a Developer's Knowledge of Code. ACM Trans Softw Eng Methodol 23(2)
Fritz T, Ou J, Murphy GC, Murphy-Hill E (2010) A Degree-of-Knowledge Model to Capture Source Code Familiarity. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, p 385–394
Gauthier IX, Lamothe M, Mussbacher G, McIntosh S (2021) Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community. In: Proceedings of the 36th IEEE/ACM international conference on automated software engineering (ASE), IEEE, pp 30–41
Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In: Proceedings of the 37th IEEE international conference on software engineering, pp 789–800
Ghotra B, Mcintosh S, Hassan AE (2017) A large-scale study of the impact of feature selection techniques on defect classification models. In: Porceedings of the 14th international conference on mining software repositories, p 146–157
Girba T, Kuhn A, Seeberger M, Ducasse S (2005) How developers drive software evolution. In: Proceedings of the 8th international workshop on principles of software evolution, pp 113–122
Goebl S, He X, Plant C, Böhm C (2014) Finding the optimal subspace for clustering. In: Proceedings of the 2014 IEEE international conference on data mining, pp 130–139
Google (2020) k-Means Advantages and Disadvantages. https://developers.google.com/machine-learning/clustering/algorithm/advantages-disadvantages, (Accessed April 2023)
Greene GJ, Fischer B (2016) Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 804–809
Hannebauer C, Patalas M, Stünkel S, Gruhn V (2016) Automatically recommending code reviewers based on their expertise: an empirical comparison. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, p 99–110
Hauff C, Gousios G (2015) Matching GitHub developer profiles to job advertisements. In: 2015 IEEE/ACM 12th working conference on mining software repositories, pp 362–366
He Q, Li B, Chen F, Grundy J, Xia X, Yang Y (2020) Diversified third-party library prediction for mobile app development. IEEE Trans Software Eng 48(1):150–165

Jiang J, He JH, Chen XY (2015) Coredevrec: automatic core member recommendation for contribution evaluation. J Comput Sci Technol 30:998–1016
Jiang J, Yang Y, He J, Blanc X, Zhang L (2017) Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull based development. Inf Softw Technol 84:48–62
Jiang J, Lo D, Zheng J, Xia X, Yang Y, Zhang L (2019) Who should make decision on this pull request? analyzing time-decaying relationships and file similarities for integrator prediction. J Syst Softw 154:196–210
Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th international conference on predictive models in software engineering, PROMISE'10
Kagdi H, Hammad M, Maletic JI (2008) Who can help me with this source code change? In: 2008 IEEE international conference on software maintenance, pp 157–166. https://doi.org/10.1109/ICSM.2008.4658064
Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and Perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories, pp 92–101
Kassambara A, Mundt F (2017) Package 'factoextra'. Extract and visualize the results of multivariate data analyses 76
Kleinberg JM (1999) Authoritative sources in a hyperlinked environment. J ACM (JACM) 46(5):604–632
Kondo M, Bezemer CP, Kamei Y, Hassan AE, Mizuno O (2019) The impact of feature reduction techniques on defect prediction models. Empirical Softw Eng 24(4):1925–1963
Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead Jr EJ (2013) Does bug prediction support human developers? Findings from a Google Case Study. In: Proceedings of the 35th international conference on software engineering, p 372–381
Li T, Zhu S, Ogihara M (2003) Algorithms for clustering high dimensional and distributed data. Intell Data Anal 7(4):305–326
Liang JT, Zimmermann T, Ford D (2022) Towards mining oss skills from github activity. In: Proceedings of the 44th international conference on software engineering - new ideas and emerging results (NIER track)
Liao Z, Wu Z, Wu J, Zhang Y, Liu J, Long J (2019) Tirr: a code reviewer recommendation algorithm with topic model and reviewer influence. In: Proceedings of the 2019 IEEE global communications conference, pp 1–6
Li X, Peng S, Du J (2021) Towards medical knowmetrics: representing and computing medical knowledge using semantic predications as the knowledge unit and the uncertainty as the knowledge context. Scientometrics pp 1–27
Malik H, Hassan AE (2008) Supporting software evolution using adaptive change propagation heuristics. In: Proceedings of the 24th IEEE international conference on software maintenance, pp 177–18
Ma D, Schuler D, Zimmermann T, Sillito J (2009) Expert recommendation with usage expertise. In: Proceedings of the 25th IEEE international conference on software maintenance, pp 535–538
McDonald DW, Ackerman MS (2000) Expertise recommender: a flexible recommendation system and Architecture. In: Proceedings of the 2000 ACM conference on computer supported cooperative work, pp 231–240
McDonald DW (2001) Evaluating expertise recommendations. In: Proceedings of the 2001 International ACM SIGGROUP conference on supporting group work, p 214–223
Mirsaeedi E, Rigby PC (2020) Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution. In: Proceedings of the 42nd international conference on software engineering, pp 1183–1195
Mittas N, Angelis L (2013) Ranking and clustering software cost estimation models through a multiple comparisons algorithm. IEEE Trans Software Eng 39(4):537–551
Mockus A, Herbsleb JD (2002) Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of the 24th international conference on software engineering, pp 503–512
Montandon JE, Silva LL, Valente MT (2019) Identifying experts in software libraries and frameworks among github users. In: Porceedings of the 2019 16th international conference on mining software repositories (MSR), pp 276–287
Moradi Dakhel A, C Desmarais M, Khomh F (2021) Assessing developer expertise from the statistical distribution of programming syntax patterns. In: Proceedings of the 25th international conference on evaluation and assessment in software engineering, pp 90–99
Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating GitHub for engineered software projects. Empir Softw Eng 22(6):3219–3253
Nidheesh N, Nazeer K, Ameer P (2020) A Hierarchical Clustering algorithm based on Silhouette Index for cancer subtype discovery from genomic data. Neural Comput Appl 32(15):11459–11476
Oracle (2022) Oracle Certified Associate, Java SE 8 Programmer. https://education.oracle.com/oracle-certified-associate-java-se-8-programmer/trackp_333, (Accessed April 2023)

Oracle (2022) Oracle Certified Professional, Java EE 7 Application Developer. https://education.oracle.com/oracle-certified-professional-java-ee-7-application-developer/pexam_1Z0-900, (Accessed April 2023)
Oracle (2022) Oracle Certified Professional, Java SE 8 Programmer. https://education.oracle.com/oracle-certified-professional-java-se-8-programmer/trackp_357, (Accessed April 2023)
Ouni A, Kula RG, Inoue K (2016) Search-based peer reviewers recommendation in modern code review. In: Proceedings of the 32nd IEEE international conference on software maintenance and evolution, pp 367–377
Pandya P, Tiwari S (2022) Corms: a github and gerrit based hybrid code reviewer recommendation approach for modern code review. In: Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering, pp 546–557
Panichella A, Dit B, Oliveto R, Di Penta M, Poshynanyk D, De Lucia A (2013) How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of the 35th international conference on software engineering, pp 522–531
Parsons L, Haque E, Liu H (2004) Subspace Clustering for High Dimensional Data: A Review. SIGKDD Explorations Newsletter 6(1):90–105
Patel A, Jain S, Shandilya SK (2018) Data of semantic web as unit of knowledge. J Web Eng
Peták M, Brožová H, Houška M (2020) Modelling of knowledge via fuzzy knowledge unit in a case of the ERP systems upgrade. Autom Control Comput Sci 54(6):529–540
Peták M, Houška M (2018) Fuzzy knowledge unit. In: Proceedings of 12th international scientific conference on distance learning in applied informatics, pp 491–502
Rahman MM, Roy CK, Redl J, Collins JA (2016) Correct: code reviewer recommendation at github for vendasta technologies. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pp 792–797
Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, pp 202–212. https://doi.org/10.1145/2491411.2491444
Robbes R, Röthlisberger D (2013) Using developer interaction data to compare expertise metrics. In: Proceedings of the 10th working conference on mining software repositories, p 297–300
Rong G, Zhang Y, Yang L, Zhang F, Kuang H, Zhang H (2022) Modeling review history for reviewer recommendation: a hypergraph approach. In: Proceedings of the 44th international conference on software engineering, pp 1381–1392
Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math 20:53–65
Sklearn (2022) dfgdfg. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_samples.html, (Accessed April 2023)
Spadini D, Aniche M, Bacchelli A (2018) PyDriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, p 908–911
Strand A, Gunnarson M, Britto R, Usman M (2020) Using a context-aware approach to recommend code reviewers: findings from an industrial case study. In: Proceedings of the 42nd international conference on software engineering: software engineering in practice, pp 1–10
Sülün E, Tüzün E, Doğrusöz U (2019) Reviewer recommendation using software artifact traceability graphs. In: Proceedings of the 15th international conference on predictive models and data analytics in software engineering, pp 66–75
Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Software Eng 43(1):1–18
Tecimer KA, Tüzün E, Dibeklioglu H, Erdogmus H (2021) Detection and elimination of systematic labeling bias in code reviewer recommendation systems. In: Evaluation and assessment in software engineering, pp 181–190
Thongtanunam P, Tantithamthavorn C, Kula RG, Yoshida N, Iida H, Matsumoto Ki (2015) Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: Proceedings of the 22nd international conference on software analysis, evolution, and reengineering (SANER), pp 141–150
Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST (2006) Design pattern detection using similarity scoring. IEEE Trans Software Eng 32(11):896–909
Vekariya P (2018) Top 7 Programming Language Certifications for Web Developers. https://medium.com/@pdvekariya1/top-7-programming-language-certifications-for-web-developers-a29fce9508e4, (Accessed April 2023)
Vivacqua A, Lieberman H (2000) Agents to assist in finding help. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp 65–72

Von Solms S, Futcher L (2018) Identifying the cybersecurity body of knowledge for a postgraduate module in systems engineering. In: IFIP world conference on information security education, pp 121–132
Wan Y, Chen L, Xu G, Zhao Z, Tang J, Wu J (2018) SCSMiner: mining social coding sites for software developer recommendation with relevance propagation. World Wide Web 21(6):1523–1543
Xia X, Lo D, Wang X, Yang X (2015) Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: Proceedings of the 31st IEEE international conference on software maintenance and evolution, pp 261–270
Xia Z, Sun H, Jiang J, Wang X, Liu X (2017) A hybrid approach to code reviewer recommendation with collaborative filtering. In: Proceedings of the 6th international workshop on software mining, pp 24-31
Xia Z, Sun H, Jiang J, Wang X, Liu X (2017) A hybrid approach to code reviewer recommendation with collaborative filtering. In: Proceedings of the 6th on international workshop software mining, pp 24–31
Xibilia MG, Latino M, Marinković Z, Atanasković A, Donato N (2020) Soft sensors based on deep neural networks for applications in security and safety. IEEE Trans Instrum Meas 69(10):7869–7876
Xie X, Yang X, Wang B, He Q (2021) DevRec: multi-relationship embedded software developer recommendation. IEEE Trans Softw Eng 1–1
Xiong R, Li B (2019) Accurate design pattern detection based on idiomatic implementation matching in java language context. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER), pp 163–174
Ying H, Chen L, Liang T, Wu J (2016) Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In: Proceedings of the 3rd international workshop on crowdsourcing in software engineering, pp 29–35
Yitzhaki S (1979) Relative Deprivation and the Gini Coefficient. Q J Econ 93(2):321–324
Yoon KA, Kwon OS, Bae DH (2007) An approach to outlier detection of software measurement data using the k-means clustering method. In: First international symposium on empirical software engineering and measurement, pp 443–445
Yu Y, Wang H, Yin G, Wang T (2016) Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? Inf Softw Technol 74:204–218
Yu Y, Wang H, Yin G, Wang T (2016) Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? Inf Softw Technol 74:204–2
Yu Y, Wang H, Yin G, Ling CX (2014) Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In: Proceedings of the 21st Asia-Pacific software engineering conference, pp 335–342
Zanjani MB, Kagdi H, Bird C (2016) Automatically recommending peer reviewers in modern code review. IEEE Trans Software Eng 42(6):530–543
Zhang F, Zheng Q, Zou Y, Hassan AE (2016) Cross-project defect prediction using a connectivity-based unsupervised classifier. In: Proceedings of the 38th international conference on software engineering, p 309–320

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

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Md Ahasanuzzaman[1] · Gustavo A. Oliva[1] · Ahmed E. Hassan[1]

Gustavo A. Oliva
gustavo@cs.queensu.ca

Ahmed E. Hassan
ahmed@cs.queensu.ca

[1] Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, Ontario, Canada