2-2017

Detecting similar repositories on GitHub

Yun ZHANG

David LO
Singapore Management University, davidlo@smu.edu.sg

PAVNEET SINGH KOCHHAR
Singapore Management University, kochharps.2012@phdis.smu.edu.sg

Xin XIA

Quanlai LI

See next page for additional authors

SANER 2017: Proceedings of 24th IEEE International Conference on Software Analysis, Evolution and Reengineering: Klagenfurt, Austria, February 20-24, 2017, http://doi.org/10.1109/SANER.2017.7884605

Detecting Similar Repositories on GitHub

Quanlai Li[1], Yan Li[1], Pavneet Singh Kochhar[2], Xin Xia[1] and David Lo[2]
[1]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]School of Information Systems, Singapore Management University, Singapore
{lql, liyansam, xxia}@zju.edu.cn, {kochharps.2012, davidlo}@smu.edu.sg

***Abstract*—GitHub contains millions of repositories with a number of repositories implementing similar functionalities. Finding similar repositories on GitHub can be helpful for software engineers as it can help them reuse source code, identify alternative implementations, explore related projects, find projects to contribute to, and discover code theft and plagiarism. Previous studies have proposed techniques to detect similar applications by analyzing API usage patterns and software tags. Unfortunately, these prior studies either only make use of a limited source of information or use information not available for projects on GitHub.**

In this paper, we propose an approach that can effectively detect similar repositories on GitHub. Our approach is designed based on three heuristics which leverage additional data sources (i.e., GitHub stars and *readme* files) which are not considered in previous works. The three heuristics are: projects that are *starred* by the same users within a short period of time are likely to be similar with one another, projects that are starred by similar users are likely to be similar with one another, and projects whose readme files contain similar contents are likely to be similar with one another. Based on these three heuristics, we compute two relevance scores (i.e., *star-based relevance* and *readme-based relevance*) to assess the similarity between two repositories. By integrating the two relevance scores, we build a recommendation system called *RepoPal* to detect similar repositories. We compare *RepoPal* to a prior state-of-the-art approach *CLAN* using one thousand Java repositories on GitHub. Our empirical evaluation demonstrates that *RepoPal* achieves a higher success rate, precision and confidence over *CLAN*.

***Keywords*—Similar Repositories, GitHub, Information Retrieval, Recommendation System**

I. Introduction

GitHub contains over 29 million repositories[1] developed by more than 11 million developers spread around the world. A repository is a basic unit in GitHub which typically contains the source code and resource files of a software project. It also stores information related to the project's evolution history and high-level features, and persons who create, contribute, fork, start and watch it. GitHub hosts a myriad of projects ranging from database applications, operating systems, gaming software, web applets, mobile applications and many more. Large organizations like Google, Facebook and Microsoft are using GitHub to host their open-source projects. Many influential open-source projects also move to GitHub; these include popular programming language projects such as Python[2] and Go[3], popular server projects such as Nginx[4] and Cherokee[5], popular development frameworks, platforms and libraries such as Bootstrap[6], Node.js[7], and JQuery[8], and many more. The large number of projects give developers a plethora of options to chose the project they want to use or contribute to.

Among the millions of repositories that GitHub hosts, many implement similar functionalities but are developed by different developers or organizations. Detecting these similar repositories can be useful for code reuse, rapid prototyping, identifying alternative implementations, exploring related projects, finding projects to contribute to, discovering code theft and plagiarism (when they are reused inappropriately), and many more – c.f., [1], [2], [3]. A study showed that often more than 50% of the source code files are reused in more than one open-source projects [4]. Thus, by detecting similar applications, developers can reuse code and focus on implementing the functionalities not provided by any of the existing applications.

Unfortunately, to the best of our knowledge, currently there is no system implemented on GitHub that can detect the similarity of repositories. To help developers find relevant repositories among the millions of repositories it hosts, GitHub provides a search engine. However, it is only a simple text-based search engine that accepts as input a list of query words and returns repository names, files, issue reports and user names that contains the query words. This search engine is certainly not an ideal tool to find similar repositories.

In the literature, past studies have proposed several techniques to detect similar applications. McMillan et al. develop an approach named *CLAN* (Closely reLated ApplicatioNs) that computes similarity between Java applications by comparing the API calls made by the two applications [5]. They show that developers are able to find similar applications and their technique performs better than another a previously proposed technique MUDABlue [6]. Thung et al. propose a technique that leverages software tags instead of the API usage patterns used by *CLAN*, to recommend similar applications [7]. Their approach automatically identifies important tags used by an application by assigning different weights to different tags. The tags along with their weights are then used to compare applications.

[1]https://github.com/about/press
[2]https://github.com/python
[3]https://github.com/golang
[4]https://github.com/nginx/nginx
[5]https://github.com/cherokee/webserver
[6]https://github.com/twbs/bootstrap
[7]https://nodejs.org/en/
[8]https://github.com/jquery/jquery

Although the prior work has made significant progress in the identification of similar applications, there are a number of challenges in applying them to find similar repositories on GitHub. First, GitHub contains millions of repositories that get updated frequently over time and statically analyzing them periodically to retrieve API calls is an expensive task. Second, GitHub does not allow users to tag repositories. In addition to these challenges, prior works do not leverage additional data sources specific to GitHub that can provide new insights. For example, GitHub allows users to *star* repositories to keep track of the repositories that they find interesting. Starring a repository is a public activity that can be viewed and tracked by others. Moreover, repositories often contain *readme* files that describe the high-level features of the applications developed in the repositories.

To deal with the limitations and leverage the additional data sources, in this work, we propose a novel approach to identify similar repositories on GitHub. Our work is based on three heuristics: First, repositories that are starred by the same user within a short period of time are likely to be similar. Second, repositories that are starred by similar users are likely to be similar. Third, repositories whose readme files share similar contents are likely to be similar. Based on these three heuristics, we compute two relevance scores which measure how similar two repositories are. The first relevance score, named *star-based relevance*, is based on the first two heuristics. It is computed by: (1) counting the number of people who star both repositories, weighted based on the period of time that lapsed between the time the two repositories were starred by each person, and (2) counting the number of similar user pairs who starred the repositories. The second relevance score, named *readme-based relevance*, is based on the third heuristic. It is computed by calculating the cosine similarity of the vector space representations of the readme files of the two repositories. Our proposed approach, named *RepoPal*, combines these two relevance scores to identify similar repositories for a given target repository.

We have evaluated *RepoPal* to recommend similar repositories from a collection of 1,000 Java repositories from GitHub. We use a total of 50 queries, each corresponding to a Java repository in GitHub, to evaluate the effectiveness of *RepoPal* and a state-of-the-art approach *CLAN* [5]. We focus on only Java repositories since *CLAN* can only handle Java programs. We evaluate the effectiveness of *RepoPal* and *CLAN* via a user study. The participants in the study rate each recommended similar project with a score ranging from 1 (highly irrelevant) to 5 (highly relevant). Based on the ratings, we use three yardsticks to measure effectiveness: success rate [7], confidence [5], [7], and precision [5], [7] as yardsticks[9]. Our user study results show that *RepoPal* outperforms *CLAN* in terms of success-rate, confidence, and precision by up to 34.48%, 20.14%, and 41.03% respectively.

The contributions of our work are as follows:

- We propose three new heuristics to identify similar repositories on GitHub which leverage two data sources not considered in prior works – i.e., GitHub star and readme files. We integrate these three heuristics in a tool *RepoPal*.
- We have evaluated *RepoPal* and *CLAN* on a dataset of 1,000 Java repositories and showed that our technique outperforms *CLAN* by substantial margins in terms of success-rate, confidence, and precision.

The structure of the remainder of this paper is as follows. In Section II, we describe the three heuristics that our approach uses and some motivating examples. In Section III, we elaborate the details of our approach *RepoPal*. In Section IV, we describe the setup of our experiment which applies *RepoPal* on a dataset of 1,000 GitHub projects and compares it against the state-of-the-art approach *CLAN*. We present the results of our experiment and some threats to validity in Section V and VI respectively. Related work is briefly reviewed in Section VII. Section VIII concludes and mentions future work.

II. Heuristics

In this section, we describe the three heuristics that are used in our system and illustrate them by some examples in Sections II-A, II-B, and II-C.

A. *Heuristic 1: Projects that are* starred *by the same users within a short period of time are likely to be similar with one another*

GitHub users can star a repository to show their approval and interest. The user who stars a GitHub repository is called a *stargazer* of that repository. A stargazer can continuously get updated information of the starred repository. The fact that a GitHub user stars multiple repositories in a short period of time may indicate that the repositories are similar.

GitHub users often have sufficient motivation to star a repository when they find it interesting or useful. If the code developed in a repository is used by a stargazer, she may want to keep track of it in order to update her own code when the original repository is updated. Even if a developer does not use the code in a repository, she can still star an interesting repository to allow her to have an easier access to it, and potentially use the code developed in the repository in the future. Since starring is a public activity in GitHub, starring a repository indicates a stargazer's appreciation and approval to the repository. Starring can be used as a means to promote a repository of interest; it appears on a stargazer's public activity timeline, which allows the repository to be known by the followers of the stargazer. GitHub itself also encourages users to star repositories. Repository star count is used in many GitHub functionalities. For example, GitHub uses the number of stars that various repositories have to help rank repositories returned by its search engine.[10] GitHub also promotes repositories that accumulate stars rapidly by putting them in the *GitHub explore* page.[11]

When a software developer meets with a problem, the developer may seek help from GitHub, hoping to find some repositories for code reuse and inspiration. During surfing on GitHub, the developer may encounter some useful repositories, and subsequently star those that are related to the problem. In this way, multiple related repositories can be starred by the same GitHub user in a short time period. The problem

[9]The definitions of these evaluation metrics are given in Section IV.

[10]https://help.github.com/articles/about-stars/

[11]https://github.com/explore

that a developer meets often changes over time. Intuitively, problems that a single developer meets tend to be similar. This is especially so, if the time gap between when the problems are encountered is short. A developer may solve one problem in the morning, another related problem in the afternoon, and a less related one 5 days later. Our first heuristic is based on the hypothesis that repositories stared by one user in a shorter time period are likely to have higher similarities than those starred by the user in a longer time period.

For example, consider a GitHub user LiqiangZhang[12] and his public activities on Nov 18, 2015. On that date, he starred three repositories in one hour, i.e., bunnyblue/DroidFix, jasonross/Nuwa, and dodola/HotFix. These repositories are all related to Android hot fix.[13] If we look at his activities two days earlier, we would notice that he also starred three repositories in one hour, spongebobrf/MaterialIntroTutorial, alafighting/CharacterPickerView, and fengjundev/DoubanMovie-React-Native. These three repositories are all Android design and user interaction projects. The repositories starred on Nov 18 are highly similar to one another, and those starred on Nov 16 are also highly similar to one another. The two groups of repositories are less but yet still similar to each other, since they are all Android repositories. This example illustrates that repositories that are starred together by a single user are likely to be similar. This is especially true if the period in which they are starred together is short.

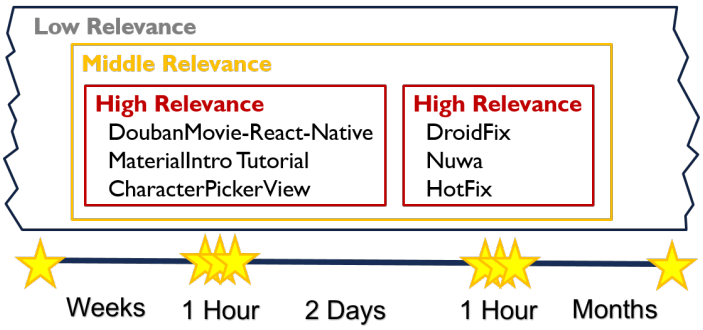


Figure 1: Snapshots of LiqianZhang's Public Activities

This phenomenon is not limited to LiqiangZhang, and we find many similar examples: On Nov 23 and 24, 2015, GitHub user fenixlin[14] starred two repositories: heshibidahe/Active_learning_ml_100k and scikit-learn/scikit-learn. They are both Python machine learning tools or modules. on Jan 26, 2016, GitHub user shichaohao[15] starred two repositories, getlantern/lantern and ziggear/shadowsocks. Once deployed, both of them can be used for assessing public websites in regions where these websites are blocked. Another GitHub user DreaminginCodeZH[16] starred three repositories consequently in Jan 17, 2016, including timusus/RecyclerView-FastScroll, AndroidDeveloperLB/ThreePhasesBottomSheet and daimajia/AndroidImageSlider. These repositories are similar in that they all deal with the scrolling or sliding action on Android.

Moreover, intuitively the more GitHub users star the same set of repositories together, the more likely the set of repositories are similar to one another. For example, GitHub user, yangweidong[17], starred both bunnyblue/DroidFix and dodola/HotFix in one hour, on Nov 18, 2015. These two repositories are both starred by both yangweidong and LiqianZhang. GitHub user vinci7[18] starred both timusus/RecyclerView-FastScroll and AndroidDeveloperLB/ThreePhasesBottomSheet on Jan 27, 2016. As we mentioned before, those two repositories are also starred by user DreamingCodeZH. The fact that many people stars two repositories together strengthens our confidence that the two repositories are similar.

B. Heuristic 2: Projects that are starred by similar users *are likely to be similar with one another*

In the first heuristic, we consider repositories that are starred by the same user to be similar. In this heuristic, we consider repositories that are starred by similar users to be similar too. We consider GitHub users who have many common starred repositories to be similar.

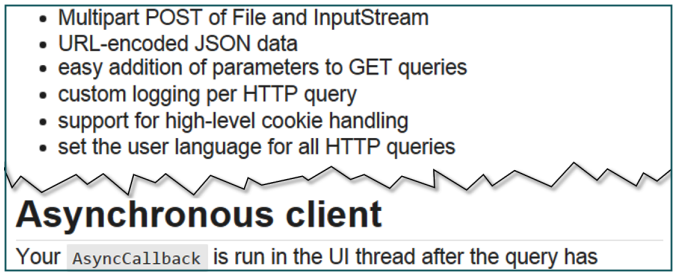
For example, we track two GitHub users who starred at least five common repositories. The two GitHub users, Will Sahatdjian[19] and Aldiantoro Nugroho[20], have starred contra/react-responsive[21], Ramotion/folding-cell[22], so-fancy/diff-so-fancy[23], corymsmith/react-native-fabric[24] and danielgindi/ios-charts[25]. Therefore, we consider them as similar users. Many repositories Will Sahatdjian starred and Aldiantoro Nugroho starred are also similar. For example, Will Sahatdjian starred jessesquires/JSQMessagesViewController[26], and Aldiantoro Nugroho starred facebook/pop[27]. Both repositories are not starred by the other user, however, they are similar. The former one describes itself as *an elegant message UI library for iOS*, while the latter one describes itself as *an extensible iOS and OS X animation library, useful for physics-based interactions*.

C. Heuristic 3: Projects whose readme files contain similar contents are likely to be similar with one another

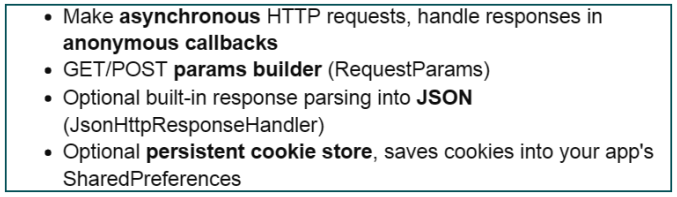
Analyzing the textual similarity of readme files could also help us find similar repositories. It is intuitive that repositories that have similar functionalities have higher likelihood of using similar words in their readme file, even if they are not developed by the same group of developers or organization.

[12] https://github.com/StormGens
[13] Android hot fix allows developers to update Android applications without publishing a new version.
[14] https://github.com/fenixlin
[15] https://github.com/shichaohao
[16] https://github.com/DreaminginCodeZH
[17] https://github.com/yangweidong
[18] https://github.com/vinci7
[19] https://github.com/kwcto?tab=activity
[20] https://github.com/kriwil
[21] https://github.com/contra/react-responsive
[22] https://github.com/Ramotion/folding-cell
[23] https://github.com/so-fancy/diff-so-fancy
[24] https://github.com/corymsmith/react-native-fabric
[25] https://github.com/danielgindi/ios-charts
[26] https://github.com/jessesquires/JSQMessagesViewController
[27] https://github.com/facebook/pop

For example, consider two repositories Android-HttpClient[28] and android-async-http[29]. The two repositories are similar to each other since they implement a number of common functionalities, e.g., asynchronous HTTP client functionality for Android applications. Excerpts of their readme files are shown in Figure 2, and we find that they share a number of words, e.g., asynchronous, HTTP, cookie, JSON, GET, POST, etc.



(a) An example from Android-HttpClient's Readme File



(b) An example from android-async-http's Readme File

Figure 2: Readme Files of Two Similar Projects

III. RepoPal

In this section, we first explain the overall architecture of *RepoPal*. Next, we describe each of its main components.

A. Architecture

Figure 3 shows the overall architecture of our technique, *RepoPal*, with its constituent parts, inputs and output. It takes as input a set of GitHub repositories (GitHub Repository Set), and a query repository (Query Repository). It outputs a ranked list of repositories that are similar to the query repository (Similar Repository List). It consists of three main components: Star Relevance Calculator, Readme Relevance Calculator, and Composer. The first and second component compute the star-based relevance scores and the readme-based relevance scores, respectively. The last component composes the two relevance scores to rank repositories in the Repository Set based on their similarity to the Query Repository. We describe the three components of *RepoPal* in the next sub-sections.

B. Star Relevance Calculator

The Star Relevance Calculator component leverages the GitHub stars to rank repositories. Intuitively, two repositories that are starred at a similar period of time by many people

[28]https://github.com/levelup/Android-HttpClient
[29]https://github.com/loopj/android-async-http

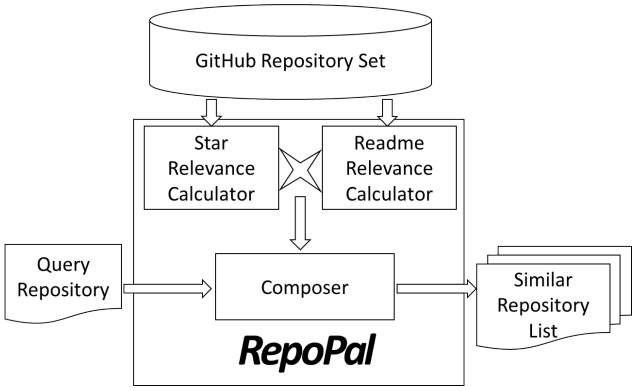


Figure 3: *RepoPal* Architecture

are likely to be similar together. We have shown a motivating example in Section II. We use this intuition to calculate a *star-based relevance score* between two repositories.

Before we define a formula to compute the star-based relevance score, we need to introduce several notations. Let U denotes a single GitHub user, R denotes a repository, and $S(R)$ denote all the users who starred R. Given the fact that user U starred repository R, we use $T(U, R)$ to represent the time-stamp at which user U starred the repository R. Given two repositories R_1 and R_2 which are starred by user U, we can compute the *adjusted time difference* between them being starred by U (denoted as $D(U, R_1, R_2)$) as follows:

$$D(U, R_1, R_2) = |T(U, R_1) - T(U, R_2)| + 30min \quad (1)$$

We set the adjusted time difference as the real time difference (in minutes) plus 30 minutes. This is because there is not much difference of relevance between two repositories whether they are starred within one or a few minutes.

Based on the adjusted time difference, we can calculate star-based relevance score between two repositories R_1 and R_2, denoted as $Relevance_s(R_1, R_2)$, as follows:

$$Relevance_s(R_1, R_2) = \sum_{U \in S(R_1) \bigcup S(R_2)} \frac{1}{D(U, R_1, R_2)} + \sum_{(U_i, U_j) \in SIM} \epsilon \quad (2)$$

In the above equation, SIM is a set of similar GitHub user pairs. We consider a pair of GitHub users who star 5 common repositories as similar. Also, ϵ is a small number equals to the reciprocal of the maximum adjusted time difference of any pair of repositories starred by U_i and any pair of repositories starred by U_j. The motivation of setting ϵ to be such small number is that the similarities of two repositories starred by similar users should not be larger than the similarities of two repositories starred by the same user.

The star-based relevance score is calculated as the sum of the reciprocals of the adjusted time differences plus some contributions due to similar users. In this way, for two repositories, when the time difference between their two stars is smaller, the reciprocal becomes larger, and therefore the relevance is

higher. When there are more users starring the two repositories together, the relevance score will also be higher. When more similar users star the two repositories, the relevance score will also be higher.

C. Readme Relevance Calculator

The Readme Relevance Calculator component computes the readme-based relevance score of two repositories by comparing their readme files. We compute this relevance score using the Vector Space Model (VSM), which is commonly used to find similarity between documents in information retrieval. We pre-process all the readme files by removing stopwords and perform stemming to reduce the words to their root form. We then convert the files into vectors of weights. Each preprocessed word corresponds to an element in the vector and its weight is computed using the standard tf-idf weighting scheme [8]. The weight of word t given document (i.e., readme file) R in a collection of documents C, denoted as $w_{t,R}$ is computed as follows:

$$w_{t,R} = (1 + log\mathrm{tf}_{\mathrm{t,R}}) \times log(\frac{|C|}{df_t}) \quad (3)$$

In Equation 3 above, $tf_{t,R}$ denotes the term frequency of word t in document R, i.e., the number of times t occurs in readme file R. df_t denotes the document frequency of t, i.e., the number of documents (i.e., readme files in C) that contain word t. After the weights are computed, each readme file R can represented as a vector of weights. Given two repositories and their two representative vectors of weights, we can compute their readme relevance score, denoted as $Relevance_r(R_1, R_2)$, by taking the cosine similarity of their representative vectors as follows:

$$Relevance_r(R_1, R_2) = \frac{\sum_{t \epsilon R_1 \cap R_2} w_{t,R_1} \times w_{t,R_2}}{\sqrt{\sum_{t \epsilon R_1} w_{t,R_1}^2} \times \sqrt{\sum_{t \epsilon R_2} w_{t,R_2}^2}} \quad (4)$$

D. Composer

The Composer component composes the two relevance scores and calculates the *overall relevance score* for two repositories R_1 and R_2 as follows:

$$Relevance(R_1, R_2) = Relevance_s(R_1, R_2) \times Relevance_r(R_1, R_2) \quad (5)$$

The pseudocode of the Composer Component is shown in Algorithm 1. Given a query repository, it computes the star-based, readme-based, and overall relevance scores between the query repository and each repository in the Repository Set (RepoSet) – lines 1 – 5. The repositories in Repository Set are then be sorted based on their overall relevance scores (lines 6). Finally, the top-k repositories are output (line 7).

Algorithm 1: Find Top-K Most Similar Repositories

Input : QRepo: Query repository, RepoSet: Set of repositories
Output: Top-k repositories
1 **for** *each repository r in RepoSet* **do**
2 compute $Relevance_s(QRepo, r)$
3 compute $Relevance_r(QRepo, r)$
4 compute $Relevance = Relevance_r \times Relevance_s$
5 **end**
6 Sort the RepoSet repositories in descending order based on $Relevance$ score
7 Output the top-k repositories

IV. Experiment Setup

We compare *RepoPal* with a prior work *CLAN* (Closely reLated ApplicatioNs) [5]. To the best of our knowledge, *CLAN* is the most similar related work.[30] *CLAN* identifies similar applications by measuring the similarity of their Java API (i.e., JDK) method invocations. *CLAN* parses the source code of programs and represents each program by the Java API methods that it calls. *CLAN* then assign weights to the Java API methods following the popular *term frequency - inverse document frequency* (TF-IDF) weighting scheme [8]; method invocations that are called more often are given higher weights, and method invocations that are called in less applications are given higher weights. Next *CLAN* compares two applications based on the weighted JDK method invocations using Latent Semantic Indexing (LSI) [9]. We also combine *RepoPal* and *CLAN* and denote the combined system as *Combined*. Given a pair of repositories R_1 and R_2, *Combined* multiplies the relevance score produced by *RepoPal* with the score that is produced by *CLAN*. The product of the two relevance scores is the score that is returned by *Combined* for R_1 and R_2.

To evaluate the three systems (i.e., *RepoPal*, *CLAN*, and *Combined*), we use a dataset of 1,000 unique[31] popular Java repositories on GitHub which receive more than 20 stars from GHTorrent [10][32]. These repositories are picked randomly and the number of stars ranges from 21 to 6106. We set the requirement for number of stars since many repositories in GitHub are of low quality [11], [12], which is especially true when they do not have many stars. Ideally, only high-quality repositories should be recommended. We only consider Java repositories since we would like to compare our approach with *CLAN* which only works for Java. For each of these 1,000 repositories, we collect its readme file, star events, and source code. We pick 50 repositories among the 1,000 as queries (see Table I), and generate the top five similar repositories using *RepoPal*, *CLAN* and *Combined*.

We invited 50 participants to evaluate the effectiveness of the three systems (i.e., *RepoPal*, *CLAN* and *Combined*) in recommending similar repositories. The participants come from Alibaba Ltd., NetEase Ltd. and Zhejiang University. Out of the 50 participants, 12 of them are Alibaba employees, 9 of them are NetEase employees, 25 of them are senior undergraduates in Zhejiang University and 4 are graduate students from the same university. All of them are skillful programmers with at least 3 years of coding experience. Forty eight participants are GitHub users and 13 of them frequently search interesting repositories in GitHub.

Each participant was given one of the query repositories

[30] Another closely related work by Thung et al. [7] cannot be applied to our setting since it requires the availability of manually assigned tags, while tagging is not supported by GitHub.

[31] None of the repositories are clones of one another.

[32] http://ghtorrent.org/

Table I: Queries Used to Evaluate *RepoPal*, *CLAN* and the combined

Num.	Query	Num.	Query	Num.	Query	Num.	Query	Num.	Query
1	Activiti	11	FlexiImageView	21	jpropel	31	apifest-oauth20	41	FragmentTabHostExample
2	Androzic	12	gatk	22	json	32	circular-progress-button	42	mvp-to-mvvm-transition
3	cli-parser	13	GraphTea	23	jsonde	33	O2OMobile_Android	43	android-number-morphing
4	crunch	14	GwtMobile-UI	24	kalium	34	android-archetypes	44	lucy-xss-servlet-filter
5	me.fantouch.libs	15	Hangouts-UI	25	LyricHere	35	play-rest-security	45	metrics-reporter-config
6	dl4j-examples	16	high-scale-lib	26	mvel	36	MultiActionTextView	46	SalesforceMobileSDK-Android
7	Dumbledroid	17	invokebinder	27	Pydev	37	MultipleChoiceAlbun	47	spring-integration-extensions
8	ehour	18	iText-4.2.0	28	Qiitanium	38	deep_functional_test	48	aws-apigateway-swagger-importer
9	EmojiChat	19	itl-java	29	query	39	FileDownloaderManager	49	shoppinglist-clean-architecture-example
10	Fglass	20	Jobs	30	tempus-fugit	40	appbundle-maven-plugin	50	PhilipsHueSDK-Java-MultiPlatform-Android

listed above and 15 retrieved repositories generated by the three systems (5 from each system). In some cases, there may be several common repositories retrieved by different systems using one query and we omit the duplicates. To reduce experiment bias, we do not inform the participants which of the three systems recommends a repository. Participants were given the URLs of the GitHub repositories and therefore can access the code, authors and contributors, readme file, related links (if any) and other information to assess similarity. To further reduce bias, we do not instruct participants to focus on a specific piece of information. We hope participants could judge the similarity of repositories fairly using various sources of information. Participants are instructed to comprehend all repositories carefully and use at least a total of 30 minutes to assess similarities. After that, they are asked the following questions about the relevance of each retrieved repository to the query:

> How relevant is the retrieved repository to the query repository?
> *Options*:
> (1) Highly Irrelevant,
> the participant finds that there is absolutely nothing in common between the retrieved and query repositories.
> (2) Irrelevant,
> the participant finds that the two repositories have little in common.
> (3) Neutral,
> the participant finds that the two repositories are marginally relevant.
> (4) Relevant,
> the participant finds that the two repositories are similar on a number of aspects.
> (5) Highly Relevant,
> the participant finds that the retrieved and query repositories are similar in most aspects, and even some parts may be identical.

We map each participant response to a score from 1 to 5, with 1 corresponding to "Highly Irrelevant" and 5 corresponding to "Highly Relevant". We use the scores to evaluate the effectiveness of the three recommendation systems (i.e., *RepoPal*, *CLAN*, and *Combined*).

Following prior studies [5], [7], we use three evaluation metrics, i.e., success rate, confidence and precision, to summarize the ratings that we receive from the participants:

1) *SuccessRate@T*. SuccessRate@T is defined as the proportion of successful top-5 recommendations that a system generates. A top-5 recommendation is deemed to be successful if there is at least one retrieved repository with rating T or higher. We consider SuccessRate@4 and SuccessRate@5.
2) *Confidence*. Median and mean confidence is defined as the median and mean ratings participants give to all retrieved repositories recommended by a system.
3) *Precision*. Precision is defined as the proportion of relevant and highly relevant repositories among those that a system recommends for a query. Given a set of queries, we can define the mean and median of the precision scores.[33]

Based on the above evaluation metrics, we investigate the following research questions:

RQ1: *What are the proportions of queries for which RepoPal, CLAN, and Combined return at least a relevant (or highly relevant) search result?*

RQ2: *How high are the median and mean confidence of participants using RepoPal as compared to CLAN?*

RQ3: *What are the precision scores of RepoPal and CLAN?*

V. Experiment Results

In this section we report the evaluation of the three systems on the three metrics, and check the significance of the differences, using data collected from participants.

A. RQ1: Success Rate

The success rates of *RepoPal*, *CLAN* and *Combined* are shown in Table II. We note that *RepoPal* and *Combined* achieve higher success rates than *CLAN*. *RepoPal* performs better than *Combined* showing that adding *CLAN* to *RepoPal* does not help boost performance. *RepoPal*, the best performing system, can generate successful recommendations that contain at least one relevant (highly relevant) repository 88% (78%) of the times, which is a reasonably high percentage. In terms of SuccessRate@4, *RepoPal* and *Combined* outperform *CLAN* by 12.82% and 5.12% respectively. In terms of SuccessRate@5, which is a stricter criteria, *RepoPal* and *Combined* outperform *CLAN* by higher margins, i.e., 34.48% and 24.14% respectively.

[33] When precision is computed, at times recall is computed too. Note that we do not compute recall since we do not know the total number of relevant and highly relevant repositories in our collection of 1,000 repositories. Identifying *all* relevant and highly relevant repositories given a query repository would require too much manual labeling cost.

Table II: Success Rate: *RepoPal* VS. *CLAN* VS. *Combined*

Approach	Success Rate (Score $\geq$ 4)	Success Rate (Score $\geq$ 5)
RepoPal	88%	78%
CLAN	78%	58%
Combined	82%	72%

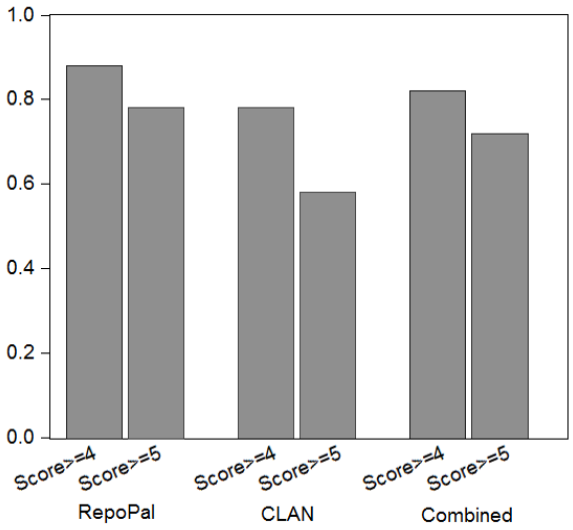


Figure 4: Success Rate

Table III: Confidence: *RepoPal* VS. *CLAN* VS. *Combined*

Approach	Sample Size	Median	Mean
RepoPal	250	4.0	3.52
CLAN	250	3.0	2.93
Combined	250	4.0	3.51

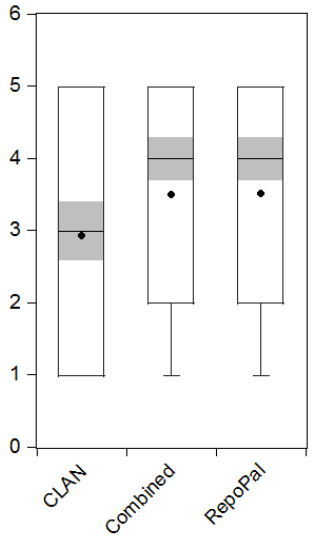


Figure 5: Confidence Box Plot

We note that 22% and 42% query results generated by *CLAN* do not include a single repository rated as relevant (4) or highly relevant (5) respectively. This shows the limitation of using only JDK API method invocations to characterize repositories. Many JDK API method invocations are generic and do not fully characterize the semantics of the applications implemented in repositories. *RepoPal* heuristics more effectively capture the semantics of applications and thus it can better identify similar repositories. Also, the fact that combining *CLAN* with *RepoPal* does not improve performance highlights that including JDK API method invocations may cause noise since unrelated repositories may use similar method calls (e.g., methods from java.util.ArrayList are used by a wide range of applications).

RepoPal outperforms CLAN in terms of SuccessRate@4 and SuccessRate@5 by 12.82% and 34.48% respectively.

B. RQ2: Confidence

Table III and Figure 5 shows the experiment results for confidence. Figure 5 is a box plot diagram showing the distribution of the 250 ratings that *RepoPal*, *CLAN*, and *Combined* each receives. According to the table and the box plot, *RepoPal* and *Combined* have very similar mean and median confidence scores out of the 250 participant ratings, which are higher than the results of *CLAN*. *RepoPal* and *Combined* outperform *CLAN* in terms of mean confidence by 20.14% and 18.80% respectively. The median confidence of *RepoPal* and *CLAN* is 4 (relevant), while that of *CLAN* is 3 (neutral).

Out of the 50 queries, *RepoPal* has higher mean confidence over *CLAN* for 35 queries, and the same mean confidence for 2 queries. *Combined* has higher mean confidence over *CLAN* for 32 queries, and the same mean confidence for 6 queries.

We perform Wilcoxon signed rank test [13] to evaluate whether the improvement of *RepoPal* over *CLAN* is statistically significant in terms of confidence, and we find that the p-value is 0.032. Therefore, the improvement of *RepoPal* over *CLAN* is significant at the confidence level of 95%. We also perform the same test to evaluate whether the improvement of of *Combined* over *CLAN* is statistically significant, and we find the p-value is 0.0020, which indicates the improvement of *Combined* over *CLAN* is significant at the confidence level of 95%.

RepoPal and Combined outperform CLAN in terms of mean confidence by 20.14% and 18.80% respectively. The improvements are statistically significant.

C. RQ3: Precision

Table IV shows the median and mean precision of *RepoPal*, *CLAN*, and *Combined* for the 50 queries. We notice that *RepoPal* and *Combined* has higher median and mean precision than *CLAN*. The median precision of *RepoPal* and *Combined* is 0.6, and their mean precision is 0.55 and 0.56 respectively. Their mean precision scores outperform that of *CLAN* by 41.03% and 43.59% respectively. Figure 6 is the box plot showing the distribution of mean precision out of the 50 queries. We note that the upper quartile for *CLAN* is substantially lower than those of *RepoPal* and *Combined*.

Table IV: Precision: *RepoPal* VS. *CLAN* VS. *Combined*

Approach	Sample Size	Median	Mean
RepoPal	50	0.6	0.55
CLAN	50	0.4	0.39
Combined	50	0.6	0.56

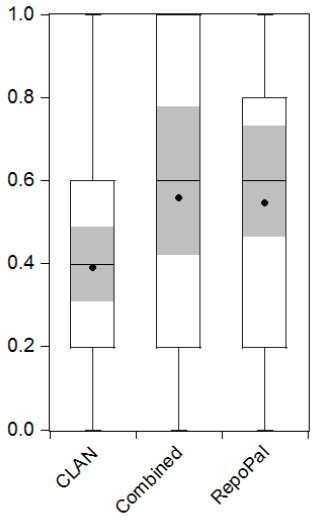


Figure 6: Precision Box Plot

Out of the 50 queries, *RepoPal* has higher precision over *CLAN* for 33 queries, and the same precision for 6 queries. *Combined* has higher precision over *CLAN* for 23 queries and the same precision for 21 queries.

Wilcoxon signed rank test is performed again to test whether the improvements of *RepoPal* and *Combined* over *CLAN* are statistically significant in terms of precision. The p-values of *RepoPal* compared with *CLAN* and *Combined* compared with *CLAN* are $4.007e^{-5}$ and $3.132e^{-7}$, respectively, which indicate that the improvements of *RepoPal* and *Combined* over *CLAN* are statistically significantly.

RepoPal and *Combined* outperform *CLAN* in terms of mean precision by 41.03% and 43.59% respectively. The improvements are statistically significant.

VI. Threats to Validity

In this section, the threats to validity of our system and experiment is discussed. The threats to validity is mainly divided into threat to internal validity, threats to external validity, and threats to construct validity. We also present what steps we have taken to minimize the threats.

A. Threats to Internal Validity

Threats to internal validity relates to experiment bias. We highlight two threats in terms of participants and repositories used to evaluate the three systems below.

Participants: The empirical evaluation is based on the scores given by the 50 participants. Some factors may cause some threats to the validity of the findings; these include: the familiarity of participants with Java and GitHub, participant motivation to give careful evaluation, and consistency in participants' standard of relevance.

Although it is guaranteed that all participants reported themselves to be familiar with Java and GitHub, their proficiency is not independently evaluated by us. The lack of knowledge in Java language and GitHub may influence the participants' judgments. This threat is limited by the fact that all student participants are from computer science department in Zhejiang University and taken sufficient technical courses, and all professional participants are developers or testers in reputed technical companies (Alibaba Ltd.[34] and NetEase Ltd.[35][36]).

Meanwhile, if participants do not have interest or motivation in evaluating the similarity of repositories, they may also make irresponsible choices. We minimize this threat by choosing participants who said they are interested in our research, and asking them to spend enough time to comprehend the repositories.

The inconsistency of evaluation standard among participants may also have negative effect. We try to minimize this by assigning the three system outputs for one query to be rated by the same participant. Thus, the strictness or leniency of this participant in his/her rating, would be fairly distributed to all evaluated systems. Since each query was only assigned to one participant, there is no information as to how reliable these results are across participants. If we have more participants, we can assign the same query to multiple participants and calculate the inter-rater reliability. Note that many past studies that require user study also do not compute inter-rater reliability due to limited number of participants, e.g., [5], [7].

Repositories: In this study, we only use 50 queries to retrieve similar repositories to evaluate *RepoPal*, *CLAN* and *Combined*. We also only retrieve similar repositories from a pool of 999 repositories (i.e., 1,000 minus the one as query). In the future, we plan to use more queries, repositories, and participants to reduce the threats to validity.

The quality of repositories also poses a threat. If a repository is of low quality, possibly with no description or specification and a bad coding style, it is hard for participants to give proper evaluation. We selected repositories with more than 20 stars to build the dataset. Intuitively these popular repositories should have better quality. A similar strategy of filtering repositories using stars, which indicate the popularity of the repositories, was also done in many prior studies, e.g., [12], [14]. Although *CLAN* does not require repositories to have stars, if we do not limit the number of stars, it is likely that *CLAN* will retrieve many repositories that has similar API invocations as the query repository but with low quality, which will also harm *CLAN*'s performance.

B. Threats to External Validity

Threats to external validity mainly deals with the generalizability of our research and experiment. We highlight the

[34]http://www.alibaba.com/
[35]https://en.wikipedia.org/wiki/NetEase
[36]http://www.163.com/

threats in terms of the programming languages and the number of stars of repositories considered in this work.

Programming Languages: GitHub contains numerous repositories written in languages other than Java (e.g. Python, PHP, C++), or combinations of multiple programming languages. RepoPal is not designed for a single language and can be applied to all GitHub repositories. However, since we want to compare *RepoPal* with *CLAN* and *CLAN* only supports Java, we focus on Java repositories in this study. We plan to evaluate *RepoPal* with other repositories written in various programming languages in the future.

Number of Stars: When the repositories' star number reduces, the quality of RepoPal's retrieval may decrease. However, as discussed before, most GitHub repositories with low star number are also of low quality, making them unfavorable to be reused.

C. Threats to Construct Validity

Threats to construct validity relates to the suitability of our evaluation metrics. In this work, we use the same metrics as used by the most closely related work by McMillan et al. [5] and Thung et al. [7]. These metrics are: SuccessRate@T, confidence and precision. These metrics are well known metrics that have also been used in many previous studies [5], [7], [15], [16], [17], [18].

VII. Related Work

Finding Similar Repositories. The closest works to our approach are the studies conduted by McMillan et al. [5] and Thung et al. [7]. McMillan et al. propose an approach *CLAN*, which compares similarity between projects using the API usage patterns [5]. They evaluate their technique on over 8,000 Java applications and find that their approach has a higher precision than previously proposed technique. Thung et al. propose a technique to recommend similar repositories based on software tags mentioned along with the project on SourceForge [7]. They perform a user study which shows that their technique outperforms JavaClan, that only uses Java API method calls.

Unfortunately, GitHub does not support repository tagging and the approach by McMillan et al. only relies on API usage patterns. In this work, we propose a new approach that addresses the limitations of prior approaches to identify similar repositories on GitHub. It relies on two sources of information, GitHub stars and readme files, which were not used in the prior works. We have also compared our approach against the work by McMillan et al. (i.e., *CLAN*) on Java repositories and demonstrated that our work outperforms theirs. We do not compare our approach with Thung et al.'s work since their approach relies on tags which are not available for repositories on GitHub.

Software Recommendation Systems. There have been a number of studies on software recommendation systems. Bajracharya et al. present a technique Structural Semantic Indexing (SSI) which associates words to source code entities based on similarities of API usage, to recommend API usage examples [19]. Thung et al. present a technique that recommends libraries to developers using association rule mining, which is based on the current library usage, and collaborative filtering, which finds libraries used by other similar projects [20]. The evaluation of their technique on 500 Java projects shows high recall rates. Bauer et al. present a technique to detect re-implementations of source code by leveraging identifier based concept location and static analysis [21]. Teyton et al. present an approach that analyzes source code changes in software projects, which have migrated from one third-party library to another, and extract mappings between functions of old library and new library [22].

Our work is orthogonal to the above studies: we recommend similar repositories to a given query repository which is a different problem compared with the the problems addressed by the above mentioned works.

Software Categorization. Several approaches categorize projects into different categories. Kawaguchi et al. propose a technique MUDABlue, that uses source code and applies Latent Semantic Analysis (LSA) to automatically determine different categories from a collection of software systems and classifies these systems into the above categories [6]. They also implement a web-based interface to visualize different categories and compare their technique to some previously proposed techniques based on information retrieval. Wang et al. propose a SVM-based approach to hierarchically categorize software projects by aggregating different online profiles from multiple repositories [23]. They conduct an experiment on over 18,000 projects and find that their technique shows significant improvement in precision, recall and F-measure. These studies classify projects into different categories, however, there can be *many* projects within a category (on average, hundreds [23]). Given a query project, it is not possible to use the above mentioned approaches to differentiate projects within the same category.

Code Search Engine. Several studies have proposed source code search engines, for example, Exemplar [24], Sourcerer [25], SNIFF [26], Portfolio [27], SpotWeb [28], Parseweb [29], and S6 [30]. These search engines recover source code fragments that match a certain natural language query. In this work, we consider a different yet related problem, namely the identification of similar repositories, given a query repository.

Studies on GitHub. A number of studies have analyzed repositories in GitHub. For example, Bissyande et al. study 100,000 GitHub projects to examine the popularity, interoperability and impact of various programming languages [31]. Ray et al. analyse more than 700 projects to understand the effect of programming languages on software quality [32]. Vasilescu et al. analyse thousands of projects and survey GitHub users to investigate the relationship between gender and tenure diversity on team productivity and turnover [33]. Different from the above studies, we focus on an orthogonal problem namely the identification of similar repositories on GitHub.

VIII. Conclusion and Future Work

Detecting similar repositories on GitHub can help software engineers to reuse source code, identify alternative implementations, explore related projects, find projects to contribute to,

discover code theft and plagiarism, among others. A number of prior approaches have been proposed to identify similar applications, unfortunately they are not optimal for GitHub. One approach relies only on similarity in API method invocations [5], while another relies on tags which are not present in GitHub [7]. They do not leverage two sources of information that can intuitively help to identify similar repositories, that is, GitHub stars and readme files. In this work, we propose a new technique named *RepoPal* that leverages the two sources of information. It works based on three heuristics: First, repositories that are starred by the same people within a short period of time are likely to be similar. Second, repositories starred by similar users are likely to be similar. Third, repositories whose readme files share similar contents are likely to be similar. In this study, we have evaluated *RepoPal* on 50 queries run against a pool of 1,000 repositories, and compared its effectiveness against *CLAN*. Our experiment results show that *RepoPal* can outperform *CLAN* in terms of success rate, confidence, and precision.

In a future work, we plan to reduce the threats to validity by including additional queries, repositories, and participants in the evaluation of *RepoPal*. Moreover, we plan to include additional sources of information to boost the effectiveness of *RepoPal* further.

ACKNOWLEDGEMENT

The authors thank to all the developers who participated in this study. This research was supported by National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

REFERENCES

[1] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *KDD*, pp. 872–881, 2006.

[2] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting similar java classes using tree algorithms," in *MSR*, pp. 65–71, 2006.

[3] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *ICSE*, pp. 848–858, 2012.

[4] A. Mockus, "Large-scale code reuse in open source software," in *FLOSS*, 2007.

[5] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *ICSE*, pp. 364–374, 2012.

[6] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue, "Mudablue: an automatic categorization system for open source repositories," in *APSEC*, pp. 184–193, 2004.

[7] F. Thung, D. Lo, and L. Jiang, "Detecting similar applications with collaborative tagging," in *ICSM*, pp. 600–603, 2012.

[8] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[9] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JASIS*, vol. 41, no. 6, p. 391–407, 1990.

[10] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 384–387, ACM, 2014.

[11] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, pp. 92–101, ACM, 2014.

[12] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 155–165, ACM, 2014.

[13] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[14] C. Casalnuovo, P. T. Devanbu, A. Oliveira, V. Filkov, and B. Ray, "Assert use in github projects," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pp. 755–766, 2015.

[15] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.

[16] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 111–120, IEEE, 2011.

[17] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, pp. 475–484, IEEE, 2010.

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

[19] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *FSE*, pp. 157–166, 2010.

[20] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *WCRE*, pp. 182–191, 2013.

[21] V. Bauer, T. Volke, and E. Jurgens, "A novel approach to detect unintentional re-implementations," in *ICSME*, pp. 491–495, 2014.

[22] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *WCRE*, pp. 192–201, 2013.

[23] T. Wang, H. Wang, G. Yin, C. Ling, X. Li, and P. Zou, "Mining software profile across multiple repositories for hierarchical categorization," in *ICSM*, pp. 240–249, 2013.

[24] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ICSE*, pp. 475–484, 2010.

[25] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Min. and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[26] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for java using free-form queries," in *FASE*, pp. 385–400, 2009.

[27] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *ICSE*, pp. 111–120, 2011.

[28] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *ASE*, pp. 327–336, 2008.

[29] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *ASE*, pp. 204–213, 2007.

[30] S. P. Reiss, "Semantics-based code search," in *ICSE*, pp. 243–253, 2009.

[31] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC)*, pp. 303–312, 2013.

[32] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 155–165, 2014.

[33] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov, "Gender and tenure diversity in github teams," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI)*, pp. 3789–3798, 2015.