Detecting Similar Repositories on GitHub

Yun Zhang[1], David Lo[2], Pavneet Singh Kochhar[2], Xin Xia[1‡], Quanlai Li[3], and Jianling Sun[1]
[1]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]School of Information Systems, Singapore Management University, Singapore
[3]University of California, Berkeley, USA
{yunzhang28, xxkidd, sunjl}@zju.edu.cn, {davidlo, kochharps.2012}@smu.edu.sg liquanlai1995@gmail.com

***Abstract*—GitHub contains millions of repositories among which many are similar with one another (i.e., having similar source codes or implementing similar functionalities). Finding similar repositories on GitHub can be helpful for software engineers as it can help them reuse source code, build prototypes, 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. However, 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 a novel approach that can effectively detect similar repositories on GitHub. Our approach is designed based on three heuristics leveraging two data sources (i.e., GitHub stars and *readme* files) which are not considered in previous works. The three heuristics are: repositories whose *readme* files contain similar contents are likely to be similar with one another, repositories starred by users of similar interests are likely to be similar, and repositories starred together within a short period of time by the same user are likely to be similar. Based on these three heuristics, we compute three relevance scores (i.e., *readme-based relevance*, *stargazer-based relevance*, and *time-based relevance*) to assess the similarity between two repositories. By integrating the three 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***.**

***Index Terms*—Similar Repositories, GitHub, Information Retrieval, Recommendation System**

I. Introduction

GitHub is a large and popular open-source project platform, which hosts various open-source projects including database applications, operating systems, gaming software, web applets, and mobile applications. Large organizations like Google, Facebook and Microsoft are using GitHub to host their open-source projects. Many influential open-source projects are also on GitHub; these include popular programming language projects such as Python[1] and Go[2], popular server projects such as Nginx[3] and Cherokee[4], popular development frameworks, platforms and libraries such as Bootstrap[5], Node.js[6], and JQuery[7]. In total, GitHub contains over 38 million repositories[8] developed by more than 15 million developers spread around the world. A repository, which is a basic unit in GitHub, 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.

While GitHub meets various needs of developers due to its various open-source projects, there exists a problem. Mockus shows that more than 50% of the source code files are reused in more than one open-source projects [1], which indicates that a large proportion of open-source projects (i.e., repositories) on GitHub are similar. Two repositories are regarded as similar if they have some similar source code files or implement some similar functionalities. The large number of similar repositories provides developers a plethora of options to choose the project they want to use or contribute to, which may cost developers much selection time and even interfere them to select a proper project. Although GitHub provides a search engine to help developers find relevant repositories among the millions of repositories it hosts, the search engine is only a simple text-based tool that receives 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. Therefore, there is a heavy need for developers to have a tool that can detect similar repositories on GitHub.

Detecting similar repositories can be useful for code reuse, rapid prototyping, identifying alternative implementations, exploring related projects, finding projects to contribute to, and discovering code theft and plagiarism (when they are reused inappropriately) [2], [3], [4]. Moreover, by detecting similar repositories, developers can reuse code and focus on implementing the functionalities which are not provided by any existing projects. In the literature, past studies have proposed several techniques to detect similar projects. McMillan et al. develop an approach named *CLAN* (Closely reLated ApplicatioNs) that assess similarity between Java projects by comparing the API calls made by the two projects [5]. They

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

show that their technique performs better than 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 projects [7]. Their approach automatically identifies important tags used by a project and assign different weights to different tags. The tags along with their weights are then used to assess the similarity of projects.

Although the prior work has made significant progress in the identification of similar projects, there are a number of challenges in applying them to detect 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 a task with too much cost. Second, GitHub does not allow users to tag repositories. In addition to the above two main 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 projects developed in the repositories.

To deal with the limitations and leverage the additional data sources, in this work, we propose a novel approach to detect similar repositories on GitHub. Our approach is based on three heuristics: First, repositories whose *readme* files contain similar contents are likely to be similar with one another. Second, repositories starred by users of similar interests are likely to be similar. Third, repositories starred together within a short period of time by the same user are likely to be similar. Based on these three heuristics, we compute three relevance scores which measure how similar two repositories are. The first relevance score, named *readme-based relevance*, is based on the first heuristic. It is computed by calculating the cosine similarity of the vector space representations of the readme files of the two repositories. The second relevance score, named *stargazer-based relevance*, is based on the second heuristic. It is computed by calculating the similarities of stargazers[9] of the two repositories. The third relevance score, named *time-based relevance*, is based on the third heuristic. It is computed based on the period of time that lapsed between the time the two repositories were starred by each person. By integrating these three relevance scores, we build a recommendation system named *RepoPal* to detect similar repositories for a given query repository on GitHub.

To evaluate the effectiveness of *RepoPal*, we compare *RepoPal* with a state-of-the-art approach *CLAN* [5]. We conduct an empirical study on 1,000 Java repositories from GitHub. We focus on only Java repositories since *CLAN* can only handle Java programs. We first randomly choose 50 repositories as query repositories among the 1,000 Java repositories and achieve a top-5 recommendation for each query repository. Then, we invite 4 participants to evaluate the relevance of the top-5 recommendations generated by *RepoPal* and *CLAN*. 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, i.e., success rate [7], confidence [5], [7], and precision [5], [7], to measure effectiveness. Our user study results show that *RepoPal* outperforms *CLAN* in terms of success-rate, confidence, and precision by up to 66.67%, 46.30%, and 97.06% respectively.

The contributions of our work are as follows:

- We propose a novel recommendation system *RepoPal* to detect similar repositories on GitHub. The system is based on three new heuristics leveraging two data sources (i.e., GitHub star and readme files) which are not considered in prior works.
- We evaluate *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 rest of our paper is organized as follows. In Section II, we introduce the three heuristics that our approach uses with some motivating examples. In Section III, we present the overall architecture of our proposed system *RepoPal* and elaborate its main components. In Section IV, we describe our experiments and the results. We discuss the threats to validity in Section V. Related work is briefly reviewed in Section VI. Section VII concludes and mentions future work.

II. Three Heuristics

In this section, we describe the three heuristics that are used in our system and illustrate them by some motivating examples.

Heuristic 1: Repositories whose readme files contain similar contents are likely to be similar with one another. It is intuitive that repositories that implement similar functionalities or have similar source code files 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.

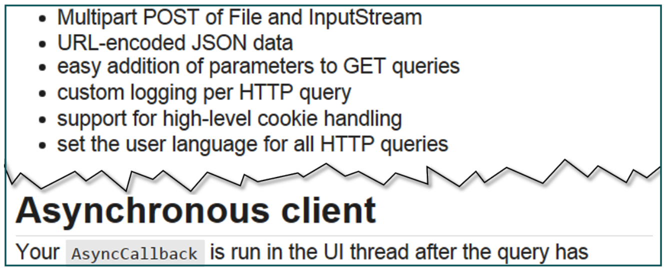
For example, consider two repositories Android-HttpClient[10] and android-async-http[11]. The two repositories are similar to each other since they implement a number of common functionalities, e.g., asynchronous HTTP client functionality for Android applications. Excerpts of their readme files are shown in Figure 1, and we find that they share a number of words, e.g., asynchronous, HTTP, cookie, JSON, GET, POST, etc.

Heuristic 2: Repositories starred by users of similar interests are likely to be similar. 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

[9]A stargazer is a user in Github who stars a repository. A stargazer can continuously get updated information of the starred repository.

[10]https://github.com/levelup/Android-HttpClient

[11]https://github.com/loopj/android-async-http

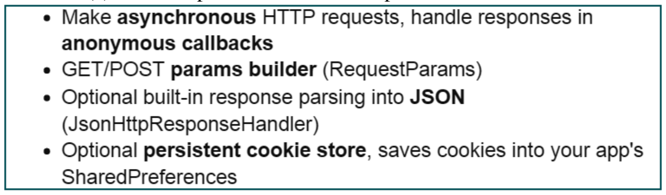


- Multipart POST of File and InputStream
- URL-encoded JSON data
- easy addition of parameters to GET queries
- custom logging per HTTP query
- support for high-level cookie handling
- set the user language for all HTTP queries

Asynchronous client

Your `AsyncCallback` is run in the UI thread after the query has

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



- Make **asynchronous** HTTP requests, handle responses in **anonymous callbacks**
- GET/POST **params builder** (RequestParams)
- Optional built-in response parsing into **JSON** (JsonHttpResponseHandler)
- Optional **persistent cookie store**, saves cookies into your app's SharedPreferences

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

Figure 1: Readme Files of Two Similar Projects

get updated information of the starred repository. 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, he/she may want to keep track of it in order to update his/her own code when the original repository is updated. Even if a developer does not use the code in a repository, he/she can still star an interesting repository to allow his/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.[12] GitHub also promotes repositories that accumulate stars rapidly by putting them in the *GitHub explore* page.[13]

Intuitively, repositories starred by two users who have starred many common repositories are likely to be similar. For example, we track two GitHub users who starred at least five common repositories. The two GitHub users, Will Sahatdjian[14] and Aldiantoro Nugroho[15], have starred contra/react-responsive[16], Ramotion/folding-cell[17], so-fancy/diff-so-fancy[18], corymsmith/react-native-fabric[19] and danielgindi/ios-charts[20]. Many other repositories Will Sahatdjian starred and Aldiantoro Nugroho starred are similar. For example, Will Sahatdjian starred jessesquires/JSQMessagesViewController[21], and Aldiantoro Nugroho starred facebook/pop[22]; both repositories are UI libraries for iOS.

Heuristic 3: Repositories starred together within a short period of time by the same user are likely to be similar.

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.

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 third 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[23] 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 (Android hot fix is a framework that allows developers to update Android applications without publishing a new version). 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.

This phenomenon is not limited to LiqiangZhang, and we find many similar examples: On Nov 23 and 24, 2015, GitHub user fenixlin[24] 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

[12]https://help.github.com/articles/about-stars/
[13]https://github.com/explore
[14]https://github.com/kwcto?tab=activity
[15]https://github.com/kriwil
[16]https://github.com/contra/react-responsive
[17]https://github.com/Ramotion/folding-cell
[18]https://github.com/so-fancy/diff-so-fancy
[19]https://github.com/corymsmith/react-native-fabric
[20]https://github.com/danielgindi/ios-charts
[21]https://github.com/jessesquires/JSQMessagesViewController
[22]https://github.com/facebook/pop
[23]https://github.com/StormGens
[24]https://github.com/fenixlin

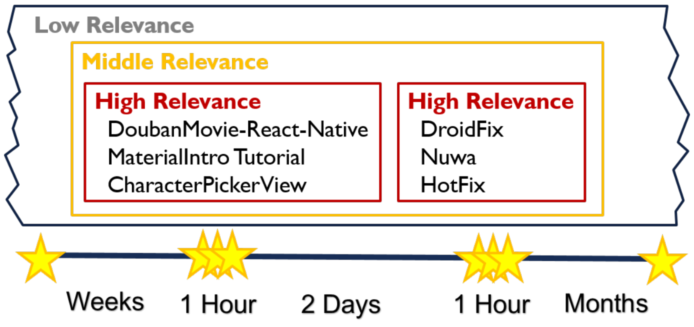


Figure 2: Snapshots of LiqianZhang's Public Activities

shichaohao[25] 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 Dreaming-inCodeZH[26] starred three repositories consequently on 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.

III. RepoPal

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

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 four main components: Readme Relevance Calculator, Stargazer Relevance Calculator, Time Relevance Calculator and Composer. The first, second, and third components compute readme-based, stargazer-based, and time-based relevance scores, respectively. The last component composes the three relevance scores to rank repositories in the Repository Set based on their similarity to the Query Repository. We describe these four components of *RepoPal* in the next sub-sections.

B. 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 performing 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:

$$w_{t,R} = (1 + log\mathrm{tf_{t,R}}) \times log(\frac{|C|}{df_t}) \tag{1}$$

In Equation 1 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-based relevance score, denoted as $Relevance_r(R_1, R_2)$, by taking the cosine similarity of their representative vectors as follows:

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

C. Stargazer Relevance Calculator

The Stargazer Relevance Calculator component leverages the GitHub stars to rank repositories. This component is based on the second heuristic described in Section II. It calculates a *stargazer-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 R denote a GitHub repository, U denote a single GitHub user, $S(R)$ denote all the users who starred R, and $S(U)$ denote all the repositories a user starred. Given a user U_1 who starred repositories $S(U_1)$, and a user U_2 who starred repositories $S(U_2)$, we compute the similarity score between the two users (denoted as $Sim(U_1, U_2)$) as follows:

$$Sim(U_1, U_2) = \frac{|S(U_1) \bigcap S(U_2)|}{|S(U_1) \bigcup S(U_2)|} \tag{3}$$

In the above equation, we denote the size of set S as $|S|$. Two users achieve a higher similarity score if they starred more common repositories.

Then given two repositories R_1 and R_2 which are starred by users $S(R_1)$ and users $S(R_2)$, we can calculate stargazer-based relevance score between these two repositories (denoted as $Relevance_s(R_1, R_2)$) as follows:

$$Relevance_s(R_1, R_2) = \underset{\substack{U_i \in S(R_1) \\ U_j \in S(R_2)}}{Avg} Sim(U_i, U_j) \tag{4}$$

For repositories R_1 and R_2, we compute all user pairs' similarity scores between sets $S(R_1)$ and $S(R_2)$. The stargazer-based relevance score is calculated as the average score of the user pairs' similarity scores. In this way, for two repositories, when the user similarity scores of the two repositories are higher, the relevance score will be higher.

[25]https://github.com/shichaohao
[26]https://github.com/DreaminginCodeZH

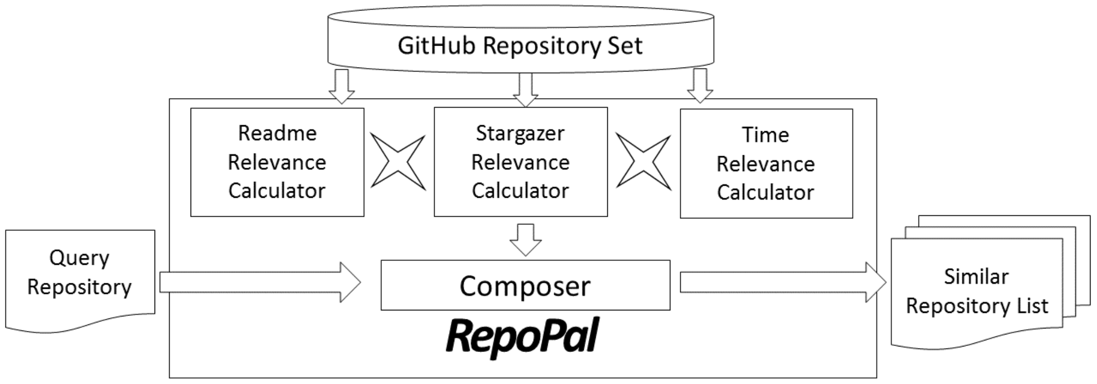

GitHub Repository Set
Readme Relevance Calculator
Stargazer Relevance Calculator
Time Relevance Calculator
Query Repository
Composer
Similar Repository List
RepoPal

Figure 3: *RepoPal* Architecture

D. Time Relevance Calculator

The Time Relevance Calculator is based on the third heuristic described in Section II. It calculates a *time-based relevance score* between two repositories. Before we define a formula to compute the time-based relevance score, we need to introduce a new notation. Given the fact that user U starred repositories R_i and R_j, we use $D(U, R_i, R_j)$ to represent the difference in time at which user U starred the two repositories rounded up to the nearest number of hours. Using this notation, given two repositories R_1 and R_2 which are starred by users $S(R_1)$ and users $S(R_2)$, we can calculate time-based relevance score between these two repositories (denoted as $Relevance_s(R_1, R_2)$) as follows:

$$Relevance_t(R_1, R_2) = \underset{U_i \in S(R_1) \bigcap S(R_2)}{Avg} \frac{1}{|D(U_i, R_1, R_2)|} \tag{5}$$

For repositories R_1 and R_2, we compute all similarity scores for users in the intersection of $S(R_1)$ and $S(R_2)$. The time-based relevance score is calculated as the average of these scores. In this way, for two repositories, when the *time differences* are smaller, the relevance score will be higher. If the intersection is empty, the time-based relevance score is set to 0.

E. Composer

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

$$\begin{aligned} Relevance(R_1, R_2) = Relevance_r(R_1, R_2) \times \\ Relevance_s(R_1, R_2) \times \\ Relevance_t(R_1, R_2) \end{aligned} \tag{6}$$

The pseudocode of the Composer Component is shown in Algorithm 1. Given a query repository, it computes the readme-based, stargazer-based, time-based, and overall relevance scores between the query repository and each repository in the GitHub Repository Set (RepoSet) – lines 1 – 6. The repositories in RepoSet are then be sorted based on their overall relevance scores (lines 7). Finally, the top-k most similar repositories are output (line 8).

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 $Relevance_r(QRepo, r)$
3     compute $Relevance_s(QRepo, r)$
4     compute $Relevance_t(QRepo, r)$
5     $Relevance=$
      $Relevance_r \times Relevance_s \times Relevance_t$
6 end
7 Sort the RepoSet repositories in descending order
  based on $Relevance$ score
8 Output the top-k most similar repositories
```

IV. Experiments and Results

In this section, we evaluate the effectiveness of our recommendation system *RepoPal*. The experimental environment is an Intel(R) Core(TM) i7-4710HQ 2.50 GHz CPU, 16GB RAM desktop running Windows 10 (64-bit). We first present our experiment setup in Sections IV-A, then introduce user study and evaluation metrics in Sections IV-B and IV-C respectively. We then present three research questions and our experiment results that answer the three research questions in Section IV-D.

A. Experiment Setup

We compare *RepoPal* with a prior work *CLAN* (Closely reLated ApplicationNs) [5]. To the best of our knowledge, *CLAN* is the most related work we can compare with. Actually, there is another closely related work by Thung et al. [7]. However, the work cannot be applied to our setting since it requires the availability of manually assigned tags, while tagging is not supported by GitHub. *CLAN* identifies similar applications by measuring the similarity of their Java API (i.e., JDK) method invocations. *CLAN* first parses the source code of programs and represents each program by the Java API methods that it calls. *CLAN* then assigns weights to the Java API methods following the popular *term frequency - inverse document frequency* (TF-IDF) weighting scheme [8], in which API methods that are called more often are given higher weights, and API methods that are called in less applications are given lower weights. In the end *CLAN* compares two applications based on the weighted Java API methods using Latent Semantic Indexing (LSI) [9].

To evaluate the two systems (i.e., *RepoPal* and *CLAN*), we use a dataset containing 1,000 unique (i.e., none of the repositories are forked of one another) popular Java repositories on GitHub which receive more than 20 stars from GHTorrent [10][27]. The repositories in the dataset are randomly picked according to the two requirements. First, 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. Among the selected repositories the number of stars ranges from 21 to 1712. Second, 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 then randomly pick 50 repositories among the 1,000 as queries (see Table I), and generate the top five similar repositories using *RepoPal* and *CLAN*.

B. User Study

We perform a user study to evaluate the effectiveness of the two systems (i.e., *RepoPal* and *CLAN*) in recommending similar repositories. We invited 4 participants for the user study, who are PhD students in Zhejiang University. All participants are skillful programmers with at least 4 years of Java coding experience and have good English reading skills. The 4 participants are GitHub users and frequently search interesting repositories in GitHub.

We randomly partitioned the 4 participants into 2 groups, each of which has 2 participants. Each participant in each group was randomly assigned to evaluate 25 of the query repositories. Each query repository was manually labeled by two participants. For each query repository, the user study contains two parts. First, we ask the two participants to independently label the 10 retrieved repositories generated by the two systems (5 from each system). Second, the two annotators discussed their disagreements (if any) to reach a common decision; for cases where the two annotators cannot reach an agreement, they asked other participants to be involved in the decision process.

In the labeling process, participants were given the URLs of all the query and retrieved repositories so that they 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 to assess similarities and assign relevance degrees of each retrieved repository and the query repository.

The degree of relevance of the retrieved repository to the query repository is shown in Table II. 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 final scores generated after the participants have discussed and resolved their differences to evaluate the effectiveness of the two recommendation systems (i.e., *RepoPal* and *CLAN*).

In the labeling process, we randomly mix the top-5 repositories generated by our approach *RepoPal* and the top-5 repositories generated by *CLAN* – participants do not know which result is produced by which approach. We used Fleiss Kappa [13] to evaluate the agreement between the two annotators. The overall Kappa value[28] between the two participants considering all queries is 0.614, which indicates substantial agreement between them.

C. Evaluation Metrics

Following prior studies [5], [7], we use three evaluation metrics, i.e., success rate, confidence and precision, to summarize the ratings from the participants and evaluate the effectiveness of the two recommendation systems (i.e., *RepoPal* and *CLAN*).

1) SuccessRate@T: SuccessRate@T is defined as the proportion of successful top-5 recommendations among all the recommendations a system generates. A top-5 recommendation is deemed to be successful if there is *at least one* retrieved repository among the top 5 with rating T or higher; T refers to the score of each repository marked by participants in the user study. For example, if a top-5 recommendation receives ratings 4, 1, 2, 3, 1, considering T to be 4, the recommendation is deemed successful. On the other hand, if T is set to 5, the recommendation is deemed unsuccessful since none of the top-5 recommendations receives a rating of 5. If a system can generate successful top-5 recommendations for 800 out of 1,000 repositories, the SuccessRate@T of the system is 80%. In our work, we only consider SuccessRate@4 and SuccessRate@5 since lower ratings (i.e., 1, 2 and 3) can not be regarded as *successful* recommendations.

2) Confidence: Median and mean confidence is defined as the median and mean relevance degrees participants give to all retrieved repositories recommended by a system. There are 50 query repositories, and each query repository has top-5 retrieved repositories. Therefore, the median and mean confidence is computed from 250 relevance degrees for each system.

[27]http://ghtorrent.org/

[28]Kappa values of < 0, [0.01, 0.20], [0.21, 0.40], [0.41, 0.60], [0.61, 0.80], [0.81, 1.00] are considered as poor agreement, slight agreement, fair agreement, moderate agreement, substantial agreement, almost perfect agreement, respectively

Table I: Queries Used to Evaluate *RepoPal* and *CLAN*

Num.	Query	Num.	Query	Num.	Query	Num.	Query	Num.	Query
1	Xinstaller	11	android-pin	21	make-it-easy	31	Garum	41	AndroidSlidingUpPanel-ScrollView
2	RxApacheHttp	12	AndroidImageCrop	22	xPlodMusic	32	RC4A	42	Android-CountdownTimer
3	Swiftnotes	13	SpringSecurityOAuth2	23	MultipleModel	33	ADP	43	spring-security-oauth2-google
4	dragqueen	14	better-java-monads	24	CubicChunks	34	android-linq	44	android-keystore-password-recover
5	CopperMod	15	emberjs-plugin	25	Protocoder	35	viritin	45	Uber-Android-SDK
6	Goreinu	16	MultipleChoiceAlbun	26	WelikeAndroid	36	ez-vcard	46	Android-TrackingRingWidget
7	NativerSDK	17	simple-netty-source	27	android-bind	37	pellet	47	Android-SVProgressHUD
8	KJController	18	SourceWall	28	houdini	38	Subspace	48	material-navigation-drawer
9	WearPomodoro	19	restfulie-java	29	MyAppList	39	Rosetta	49	FastScrollRecyclerView
10	Avaritia	20	Simple-Image-Blur	30	presenta	40	archi	50	elasticsearch-analysis-kuromoji

Table II: Degree of relevance of the retrieved repository to the query repository

Score	Relevance	Explanation
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 only 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.

3) Precision: Precision is defined as the proportion of relevant and highly relevant repositories (i.e., whose final relevance degrees are equal to or greater than 4) among the top-5 recommendations that a system generates for a query. Given a set of queries, we can define the mean and median of the precision scores. Note that recall is often computed along with precision. The precision metrics reflects the accuracy of the similarity search. However, we do not compute recall since we do not know the total number of relevant and highly relevant repositories (i.e., whose average scores are equal to or greater than 4) 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.

D. Research Questions

Our experiments are designed to answer the following research questions:

RQ1: *What are the proportions of queries for which* RepoPal *and* CLAN *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*?*

The three research questions correspond to the three evaluation metrics introduced in the above Section. We present the experiment results for the three research questions in the follow text.

1) Result for RQ1 – Success Rate: The success rates of *RepoPal* and *CLAN* are shown in Table III. We note that *RepoPal* achieves higher success rates than *CLAN*. *RepoPal* can generate successful recommendations that contain at least one relevant (highly relevant) repository 88% (60%) of the times, which is a reasonably high percentage. *CLAN* only achieves SuccessRate@4 score and SuccessRate@5 score of 62% and 36%, respectively. In terms of SuccessRate@4, *RepoPal* outperforms *CLAN* by 41.94%. In terms of SuccessRate@5, which is a stricter criteria, *RepoPal* outperforms *CLAN* by higher margins of 66.67%.

Table III: Success Rate: *RepoPal* VS. *CLAN*

Approach	Success Rate (Score $\geq$ 4)	Success Rate (Score $\geq$ 5)
RepoPal	88%	60%
CLAN	62%	36%

We note that 38% and 64% 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 capture the semantics of applications more effectively, and thus it can better identify similar repositories.

We perform Wilcoxon signed rank test [14] to evaluate whether the improvement of *RepoPal* over *CLAN* is statistically significant in terms of SuccessRate@4 and SuccessRate@5, and we find that the two p-values are < 0.001. Therefore, the improvement of *RepoPal* over *CLAN* is significant at the confidence level of 99.9%.

RepoPal outperforms CLAN in terms of SuccessRate@4 and SuccessRate@5 by 41.94% and 66.67% respectively. The improvement is statistically significant.

2) Result for RQ2 – Confidence: Table IV and Figure 4 show the experiment results for confidence. Figure 4 is a box plot diagram showing the distribution of the 250 ratings that *RepoPal* and *CLAN* each receives. According to the table

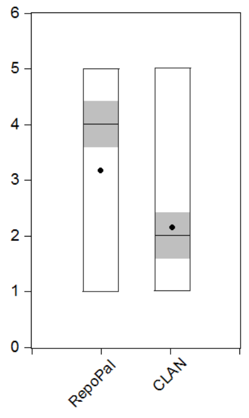


Figure 4: Confidence Box Plot

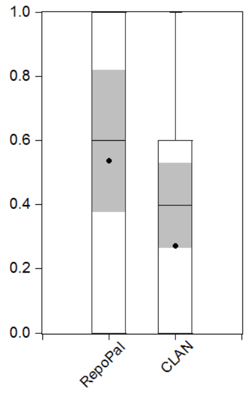


Figure 5: Precision Box Plot

and the box plot, *RepoPal* achieves higher mean and median confidence scores than the results of *CLAN* out of the 250 ratings. The mean confidence scores of *RepoPal* and *CLAN* are 3.16 and 2.16, respectively. *RepoPal* outperforms *CLAN* in terms of mean confidence by 46.30%. The median confidence of *RepoPal* is 4 (relevant), while that of *CLAN* is 2 (Irrelevant).

Table IV: Confidence: *RepoPal* VS. *CLAN*

Approach	Sample Size	Median	Mean
RepoPal	250	4.0	3.16
CLAN	250	2.0	2.16

We perform Wilcoxon signed rank test [14] to evaluate whether the improvement of *RepoPal* over *CLAN* is statistically significant in terms of confidence, and we find that the p-value is < 0.001. Therefore, the improvement of *RepoPal* over *CLAN* is significant at the confidence level of 99.9%.

RepoPal outperforms CLAN in terms of mean confidence by 46.30%. The improvement is statistically significant.

3) Result for RQ3 – Precision: Table V shows the median and mean precision of *RepoPal* and *CLAN* for the 50 queries. We notice that *RepoPal* has higher median and mean precision than *CLAN*. The median precision of *RepoPal* is 0.6, and the mean precision is 0.536. The median precision of *CLAN* is 0.4, and the mean precision is 0.272. The mean precision scores of *RepoPal* outperforms that of *CLAN* by 97.06%. Figure 5 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 that of *RepoPal*.

Wilcoxon signed rank test is performed again to test whether the improvement of *RepoPal* over *CLAN* is statistically significant in terms of precision. The p-values of *RepoPal* compared with *CLAN* is < 0.001, which indicate that the improvement of *RepoPal* over *CLAN* is statistically significantly at the confidence level of 99.9%.

Table V: Precision: *RepoPal* VS. *CLAN*

Approach	Sample Size	Median	Mean
RepoPal	50	0.6	0.536
CLAN	50	0.4	0.272

RepoPal outperforms *CLAN* in terms of mean precision by 97.06%. The improvement is statistically significant.

V. 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 two systems below.

Participants: The empirical evaluation is based on the scores given by the 4 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 college of computer science and technology in Zhejiang University and have taken sufficient technical courses.

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 (in our study, each participant spend around 6 hours to label the data).

The inconsistency of evaluation standard among participants may also have negative effect. We try to minimize this by assigning the two 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. Also, we assign each query to two participants for evaluation, and ask them to discuss with the disagreements, which we believe can much reduce the evaluation bias.

Repositories: In this study, we only use 50 queries to retrieve similar repositories to evaluate *RepoPal* and *CLAN*. 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 select 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], [15]. 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 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.

Poor Readme Files: RepoPal may fail to recommend relevant repositories with poor readme files (e.g., default or blank readme files). However, such repositories tend to be of low quality. Among the 1,000 repositories that we use to evaluate RepoPal (which receive more than 20 stars and thus of substantially good quality), none of them have blank readme files.

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], [16], [17], [18], [19].

VI. Related Work

We classify related work into several parts. We first introduce the most related works to ours in Section VI-A. Then we introduce several other related works about recommendation systems, software categorization and code search. At last, we briefly introduce some studies on GitHub.

A. Detecting 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.

B. Recommendation Systems

There have been a number of studies on software recommendation systems [20], [21], [22], [23], [24], [25], [26], [27]. 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 [20]. 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 [21]. 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 [22]. 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 [23]. They evaluate their approach on a large dataset from repositories such as GitHub and SourceForge and find that their technique is able to detect migrated code segments and achieves better results than context-based approach. Cubranic et al. propose a tool named Hipikat that recommends artifacts from the archives which might be useful for a newcomer to a project [24]. They evaluate their tool by conducting a qualitative study on graduate students in software engineering and a case study for a task on Eclipse.

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.

C. Software Categorization

Several approaches categorize projects into different categories [6], [28], [29], [30], [31]. 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 into multiple repositories [28]. They conduct an experiment on over 18,000 projects and find that their technique shows significant improvement in terms of precision, recall and F-measure.

The above studies can classify projects into different categories. However, there can be many projects in a category without any ranking by similarity. Given a query project, it is not possible to use the above mentioned approaches to rank the projects by similarity and find the most similar projects. In our work, our proposed recommendation system can rank similar repositories with each of them having a similarity score and recommend the most similar repositories according to the rank.

D. Code Search Engine

Several studies have proposed source code search engines, for example, Exemplar [32], Sourcerer [33], SNIFF [34], Portfolio [35], SpotWeb [36], Parseweb [37], and S6 [38]. These search engines can discover source code fragments that match a certain natural language query. However, they are not good at detecting similar projects. In this work, we consider a different yet related problem, namely the detection of similar repositories on GitHub, given a query repository.

E. Studies on GitHub.

A number of studies have analyzed repositories on GitHub [39], [40], [41], [42]. For example, Bissyande et al. study 100,000 GitHub projects to examine the popularity, interoperability and impact of various programming languages [39]. Ray et al. analyse more than 700 projects on GitHub to understand the effect of programming languages on software quality [40]. Vasilescu et al. investigate the interplay between StackOverflow activities and the development process, which is reflected by code changes committed to the largest open source project platform GitHub. [41]. They show that active GitHub committers ask fewer questions and provide more answers than others. In a later work, Vasilescu et al. analyse thousands of projects and survey GitHub users to investigate the influence of gender and tenure diversity on team productivity and turnover [42]. Different from the above studies, we focus on an orthogonal problem namely the detection of similar repositories on GitHub.

VII. 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 or proper 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]. In this work, we propose a novel recommendation system named *RepoPal* to identify similar repositories on GitHub. *RepoPal* leverages two data sources (i.e., GitHub stars and *readme* files) which can intuitively help to identify similar repositories but are not considered in previous works. And it is designed based on three heuristics: First, repositories whose *readme* files contain similar contents are likely to be similar with one another. Second, repositories starred by users of similar interests are likely to be similar. Third, repositories starred together within a short period of time by the same user are likely to be similar. To evaluate the effectiveness of *RepoPal*, we perform experiments on 1,000 Java repositories on GitHub and compare it against a state-of-the-art approach *CLAN* [5]. We invite several participants to evaluate our experiment results and the evaluation shows 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, Code, and Dataset

The authors would like to thank the user study participants for their help in rating the recommendations. This research was supported by NSFC Program (No. 61602403). Our code and dataset can be downloaded from https://github.com/yunzhang28/RepoPal.

References

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

[2] 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.

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

[4] 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.

[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] J. L. Fleiss, "Measuring nominal scale agreement among many raters.," *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] D. Cubranic and G. Murphy, "Hipikat: recommending pertinent software development artifacts," in *ICSE*, pp. 408–418, 2003.

[25] X. Xia and D. Lo, "An effective change recommendation approach for supplementary bug fixes," *Automated Software Engineering*, pp. 1–44, 2016.

[26] X. Xia, D. Lo, Y. Ding, and J. M. Al-Kofahi, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, pp. 1–1, 2016.

[27] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," vol. 8144, pp. 72–81, 2013.

[28] 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.

[29] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *The Ieee/acm International Conference*, pp. 51–62, 2016.

[30] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining software metrics and text features for vulnerable file prediction," in *International Conference on Engineering of Complex Computer Systems*, pp. 40–49, 2015.

[31] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *IEEE Computer Software and Applications Conference*, pp. 107–116, 2014.

[32] 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.

[33] 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.

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

[41] B. Vasilescu, V. Filkov, and A. Serebrenik, "Stackoverflow and github: Associations between software development and crowdsourced knowledge," in *Social Computing (SocialCom), 2013 International Conference on*, pp. 188–195, IEEE, 2013.

[42] 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.