Scalable Relevant Project Recommendation on GitHub

Wenyuan Xu
School of Information Engineering
Yangzhou University
Yangzhou, China
x565178035@gmail.com

Xiaobing Sun
School of Information Engineering
Yangzhou University
Yangzhou, China
xbsun@yzu.edu.cn

Xin Xia
Department of Computer Science
University of British Columbia
Canada
xxia02@cs.ubc.ca

Xiang Chen
School of Computer Science and Technology
Nantong University
Nantong, China
xchencs@ntu.edu.cn

ABSTRACT

GitHub, one of the largest social coding platforms, fosters a flexible and collaborative development process. In practice, developers in the open source software platform need to find projects relevant to their development work to reuse their function, explore ideas of possible features, or analyze the requirements for their projects. Recommending relevant projects to a developer is a difficult problem considering that there are millions of projects hosted on GitHub, and different developers may have different requirements on relevant projects. In this paper, we propose a scalable and personalized approach to recommend projects by leveraging both developers' behaviors and project features. Based on the features of projects created by developers and their behaviors to other projects, our approach automatically recommends top *N* most relevant software projects to developers. Moreover, to improve the scalability of our approach, we implement our approach in a parallel processing frame (i.e., Apache Spark) to analyze large-scale data on GitHub for efficient recommendation. We perform an empirical study on the data crawled from GitHub, and the results show that our approach can efficiently recommend relevant software projects with a relatively high precision fit for developers' interests.

KEYWORDS

Software recommendation, parallel processing frame, GitHub

ACM Reference format:
Wenyuan Xu, Xiaobing Sun, Xin Xia, and Xiang Chen. 2017. Scalable Relevant Project Recommendation on GitHub. In *Proceedings of Internetware'17, Shanghai, China, September 23, 2017,* 10 pages.
DOI: 10.1145/3131704.3131706

1 INTRODUCTION

In Github, developers are initiators of all software repositories. They not only create a repository for their own projects, but also explore other projects which they are interested in [21]. For their own projects, they do not need to build everything from scratch. More often, they reuse parts of other projects and then tailor the features to implement their own functionalities. During software development, if there is a tool which could recommend similar or relevant[1] software projects for developers, they can reuse these projects to speed up the development process [8, 12]. In addition, some software projects in open source community develop slowly because only a few developers know them. If we recommend these projects to developers who have similar requirements, their development process can be improved.

In practice, developers may use a search engine to detect similar projects by inputting the query keywords. However, search engines usually focus on text matching [7]. Because software projects are holistic, these keywords may not fully describe the characteristics of a project. In addition, some works focus on the recommendation of software projects for developers. These studies tend to recommend relevant software projects based on project description or source code without considering user behaviors [7, 16, 22], i.e., different developers may have different requirements on relevant projects. These recommendation results are usually inaccurate, which decreases developers' trust in the recommender system.

To address the above challenges, we propose a recommendation approach considering both the developers' behaviors and software projects' features on GitHub[2]. For developers' behaviors, we consider the developers' ratings for their projects as their behaviors like *Create*, *Fork* and *Star*. For project's features, we analyze both the description documents and source code. The source code is the main part of a project, while the documents describe the functionalities and usage of the project [19]. We extract the terms from the source code and documents, and use cosine similarity to measure the textual similarity between two projects [14]. Finally, our approach recommends top *N* most relevant projects generated by the above two components (i.e., developers' behaviors and projects' features) to each developer. Considering there are millions of projects on GitHub,

[1] In this paper, we use the terms "similar" and "relevant" interchangeably.
[2] https://github.com/

to improve the scalability, we parallelize our algorithm on Apache Spark platform[3].

We evaluate our approach with four groups of datasets which represent three different development areas and a mixed one from GitHub. We compare our approach with two state-of-the-art recommendation algorithms, i.e., user-based collaborative filtering (*UserCF*) and an item-based collaborative filtering (*ItemCF*) [11]. The results show that the *Accuracy* of top *5* project recommendation in four groups is 71.59%, 87.64%, 67.58%, and 42.14% respectively, which improves the baseline approaches by a substantial margin. The time for the recommendation can be shortened by increasing the computing nodes in the cluster on Apache Spark platform. This paper makes the following contributions:

- We propose a novel approach to recommend relevant software projects by considering developers' behaviors as well as software projects' features (extracted from description documents and source code) on GitHub platform.
- Our approach can be performed in a parallel way to process the increasing volume of data on Apache Spark platform.
- An empirical study is conducted with four groups of datasets crawled from GitHub to show the effectiveness of our approach.

2 PRELIMINARIES

In this section, we present the preliminary materials, i.e., content-based recommendation, TF-IDF measure, and Apache Spark platform.

2.1 Content-based Recommendation

The content-based recommendation can be described as recommending the items similar to the one the user liked before. It relies on extracting information about items and users' preferences [11].

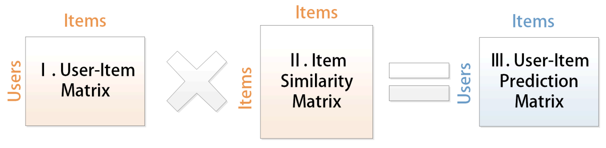


Figure 1: Technique of Content-based Recommendation

A content-based recommendation system is generally composed of two parts, as shown in Figure 1. One is user-item behavior matrix I, which is a $|U| \times |I|$ matrix recording relationships between a set of users U and a set of items I; the other is the similarity matrix of items II, where $Matrix(i_1, i_2) = s$ represents the similarity value between item i_1 and item i_2. This similarity matrix is usually calculated based on the item's features. So content-based recommendation system is mostly used for document recommendation for which the characteristics of the item can be automatically extracted from the document content or their unstructured text description [10, 13]. Finally, the user-item evaluation matrix III is generated from the matrix I and II.

[3]http://spark.apache.org/

In this paper, we use the content-based recommendation to calculate the similarity between different software repositories to recommend top N software projects.

2.2 TF-IDF Measure

TF-IDF is intended to reflect how important a word is to a document in a collection or corpus [6].

Suppose we have a collection of N documents. We define TF_{ij} to be the frequency (number of occurrences) of a term (word) i in document j. Then, the term frequency TF_{ij} is defined as:

$$TF_{ij} = \frac{n_{ij}}{\sum_{k \in j} n_{kj}} \tag{1}$$

where n_{ij} represents the times of the word i appearing in document j. The denominator is the sum of all words appeared in the document j.

Inverse Document Frequency (IDF) measures how much information the word provides, which means whether the word is common or rare across all documents. The *IDF* of a word i in the corpus N is defined as:

$$IDF_{iN} = \log\left(\frac{|N|}{|\{d|d \in N, i \in d\}|}\right) \tag{2}$$

where $|N|$ represents the total number of documents in the corpus N, and $|\{d|d \in N, i \in d\}|$ means the number of documents where the word i appears. The main usage of *IDF* is to find the specificity of a word. The smaller is the denominator, the greater is the *IDF* value, which shows that word i has a good classification capability.

Finally, *TF-IDF* is calculated as $TF_{ij} \times IDF_{iN}$, which we can learn that the results based on TF-IDF tend to filter out common words and preserves vital words.

In this paper, we use TF-IDF to calculate the values of each word in the source code files or documents of a project. And the features of a project are identified based on the TF-IDF values of the words.

2.3 Apache Spark

Apache Spark is a fast and general engine for large-scale data processing[4]. The engine realized distributed computing based on MapReduce algorithm [9]. Apache Spark is suitable for data mining and machine learning which needs to perform iterations with the MapReduce algorithm.

Resilient distributed dataset (RDD) is the core concept of Apache Spark. It is composed of distributed data collection performing two main operations: transformation and action [3]. *Transformation* is an operation such as filter (), map () to generate another RDD, and *action* is an operation such as count () and collect () to trigger a computation. Here we list the involved Apache Spark APIs used to implement our recommendation algorithm:

- HashingTF () and IDF (): these two functions are used to calculate TF-IDF values of words in a project.
- Cartesian (): this function is used to combine the projects into pairs.
- Map (), FlatMap (), and GroupByKey (): we mainly use these functions to transform our RDDs to complete our similarity calculation and the recommendation process.

[4]https://spark.apache.org/

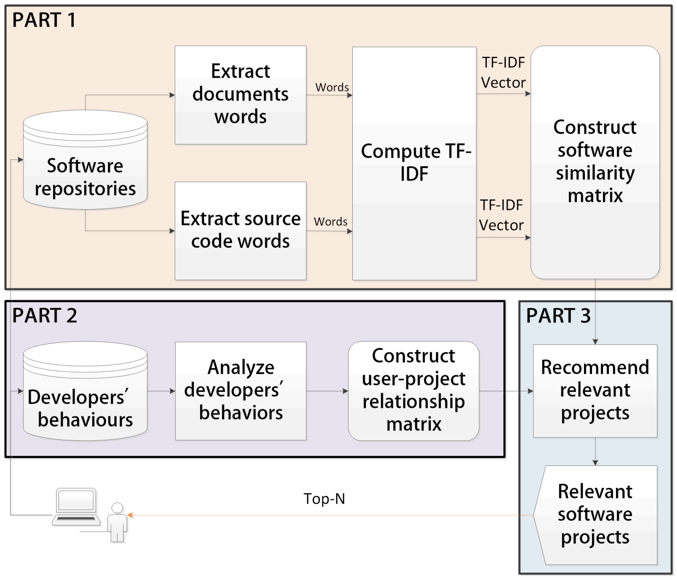

PART 1
Software repositories
Extract documents words
Extract source code words
Words
Words
Compute TF-IDF
TF-IDF Vector
TF-IDF Vector
Construct software similarity matrix
PART 2
Developers' behaviours
Analyze developers' behaviors
Construct user-project relationship matrix
PART 3
Recommend relevant projects
Relevant software projects
Top-N

Figure 2: An overview of the architecture of our approach.

- SubtrackByKey (): this function is used to delete projects known to the user.

3 OUR APPROACH

Our approach is built on GitHub, where developers can create software repositories, fork or give stars to other repositories. An architectural overview of our approach is shown in Figure 2. First, we analyze software repositories and calculate the similarity matrix between the projects on GitHub (Part 1). Second, we analyze developers' behaviors to get user-project matrix (Part 2). Finally, top N software projects are recommended for each developer using the recommendation algorithm based on the content similarity and users' behaviors (Part 3).

3.1 Extracting Features of Projects

A software project on GitHub contains different types of files, such as source code files, binary files, and description files (e.g., README files). Different types of files reflect different characteristics of a project, which can be used to extract features from a project. For example, source code files are one of the main parts of a project while the description files describe the usage of the project. We mainly analyze these two kinds of files to extract the features of a project.

Before extracting features from a project, we usually need to preprocess those files, e.g., removing noisy information. For project description documents (e.g., README files), they are mainly ended with "md" and "txt". These files are recorded to describe the project for users to understand, which are mainly composed of natural language descriptions. We first remove the code part in them, which may be noisy information to describe a project. In addition, there are some meaningless stop words (e.g., "we", "be", etc.), so we need to filter them out. Finally, we obtain a list of words from the project description documents.

For source code files, the suffix name of these files is also distinctive which is relevant to their programming language. Thus we extract these source code files based on their suffix names. Next, we remove the numbers and escape characters since they are meaningless in measuring the similarity of projects. Then, we transform the compound words (in our case, the identifiers) into several single words. The names of the identifiers form a large part in the source code which reflects the characteristics of a project. There are usually two ways to combine several words to define an identifier. One is Camel-Case, and the other is Underline-Case. We use a regular expression to match and separate them into single words. Finally, we obtain a list of words from the project source code files.

After preprocessing the description files and source code files, we extract the features of a project based on the TF-IDF measure. The TF-IDF measure calculates a word vector which is used to indicate a project's features. In this paper, the features of a project are represented by two vectors, i.e., TF-IDF vector of description documents and TF-IDF vector of source code [15]. It is difficult to compute the similarity values based on different lengths of TF-IDF vectors. Here, we unify the length of TF-IDF vectors by using a hash table and mapping each word to a hash integer value. A long hash vector may increase the size of calculation, while a short hash vector may decrease the accuracy. According to experimental results, when the vector length is 3000 ~ 4000, the effect is relatively good [4].

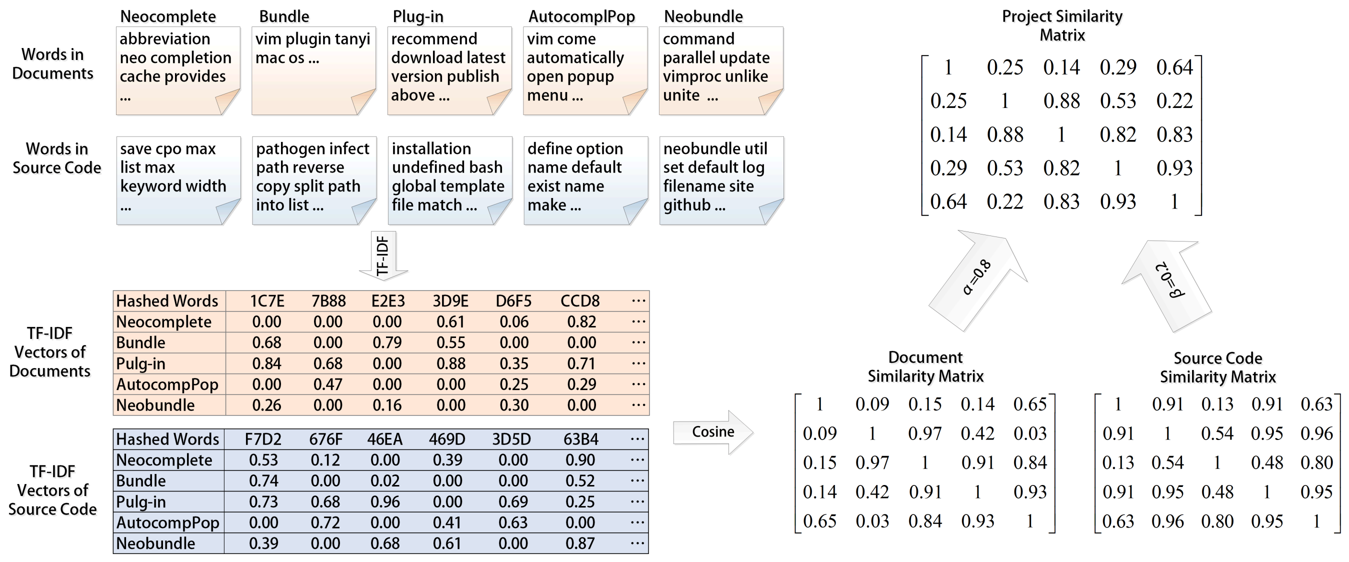
To recommend relevant and similar projects, similarity between different projects needs to be calculated. Here we calculate the similarity between projects based on the two vectors of description documents and source code, respectively. Specifically, we use the cosine measure to calculate two similarity matrices of projects on GitHub. One matrix represents the similarity values of the projects based on the description documents, and the other represents the similarity values of the projects based on the source code. In each matrix, the similarity between two projects p and q is calculated as $sim(p,q) = \frac{V_p \cdot V_q}{\sqrt{\|V_p\| \cdot \|V_q\|}}$, where V_p and V_q donate the TF-IDF vector of the projects p and q on description files or source code files.

Next, we use linear combination to combine the similarity of description files and source code files, and we compute the similarity of two projects p and q as:

$$\begin{aligned} &SIM(p,q) = \alpha \cdot sim_{doc}(p,q) + \beta \cdot sim_{src}(p,q) \\ &s.t., \alpha + \beta = 1 \end{aligned} \tag{3}$$

In the above equation, α and β represent the weights of similarity between the description files and source code files of two projects, respectively. Different weights reflect their different capability to reflect the characteristics of a project. The values of α and β can be empirically determined. By default, we set α to 0.8 and β to 0.2.

An Example. Figure 3 shows an example of the procedure of extracting features of the projects and calculating the similarity values between them. We have five projects, *Neocomplete*, *Bundle*, *Plug-in*, *AutocomplPop*, and *Neobundle*. We first extract the words of these five projects from their documents and source code. Then we calculate the TF-IDF vectors of documents and source code for each project, and calculate the similarity values between different


Words in Documents
Neocomplete
abbreviation neo completion cache provides ...
Bundle
vim plugin tanyi mac os ...
Plug-in
recommend download latest version publish above ...
AutocomplPop
vim come automatically open popup menu ...
Neobundle
command parallel update vimproc unlike unite ...
Words in Source Code
save cpo max list max keyword width ...
pathogen infect path reverse copy split path into list ...
installation undefined bash global template file match ...
define option name default exist name make ...
neobundle util set default log filename site github ...
TF-IDF
TF-IDF Vectors of Documents
Hashed Words 1C7E 7B88 E2E3 3D9E D6F5 CCD8 ···
Neocomplete 0.00 0.00 0.00 0.61 0.06 0.82 ···
Bundle 0.68 0.00 0.79 0.55 0.00 0.00 ···
Pulg-in 0.84 0.68 0.00 0.88 0.35 0.71 ···
AutocompPop 0.00 0.47 0.00 0.00 0.25 0.29 ···
Neobundle 0.26 0.00 0.16 0.00 0.30 0.00 ···
TF-IDF Vectors of Source Code
Hashed Words F7D2 676F 46EA 469D 3D5D 63B4 ···
Neocomplete 0.53 0.12 0.00 0.39 0.00 0.90 ···
Bundle 0.74 0.00 0.02 0.00 0.00 0.52 ···
Pulg-in 0.73 0.68 0.96 0.00 0.69 0.25 ···
AutocompPop 0.00 0.72 0.00 0.41 0.63 0.00 ···
Neobundle 0.39 0.00 0.68 0.61 0.00 0.87 ···
Cosine
Document Similarity Matrix
Source Code Similarity Matrix
α=0.8
β=0.2
Project Similarity Matrix

$$\text{Project Similarity Matrix} \quad \begin{bmatrix} 1 & 0.25 & 0.14 & 0.29 & 0.64 \\ 0.25 & 1 & 0.88 & 0.53 & 0.22 \\ 0.14 & 0.88 & 1 & 0.82 & 0.83 \\ 0.29 & 0.53 & 0.82 & 1 & 0.93 \\ 0.64 & 0.22 & 0.83 & 0.93 & 1 \end{bmatrix}$$

$$\text{Document Similarity Matrix} \quad \begin{bmatrix} 1 & 0.09 & 0.15 & 0.14 & 0.65 \\ 0.09 & 1 & 0.97 & 0.42 & 0.03 \\ 0.15 & 0.97 & 1 & 0.91 & 0.84 \\ 0.14 & 0.42 & 0.91 & 1 & 0.93 \\ 0.65 & 0.03 & 0.84 & 0.93 & 1 \end{bmatrix}$$

$$\text{Source Code Similarity Matrix} \quad \begin{bmatrix} 1 & 0.91 & 0.13 & 0.91 & 0.63 \\ 0.91 & 1 & 0.54 & 0.95 & 0.96 \\ 0.13 & 0.54 & 1 & 0.48 & 0.80 \\ 0.91 & 0.95 & 0.48 & 1 & 0.95 \\ 0.63 & 0.96 & 0.80 & 0.95 & 1 \end{bmatrix}$$

Figure 3: An example of extracting features and calculating the similarity values between different projects.

projects. Suppose we set $\alpha = 0.8$ and $\beta = 0.2$, and we get the similarity matrix of these projects.

3.2 Extracting User Behaviors

In GitHub, a developer can *Create*, *Fork*, or *Star* a project, which constitutes a developer's behavior. Here we simply introduce the meanings of these behaviors[5].

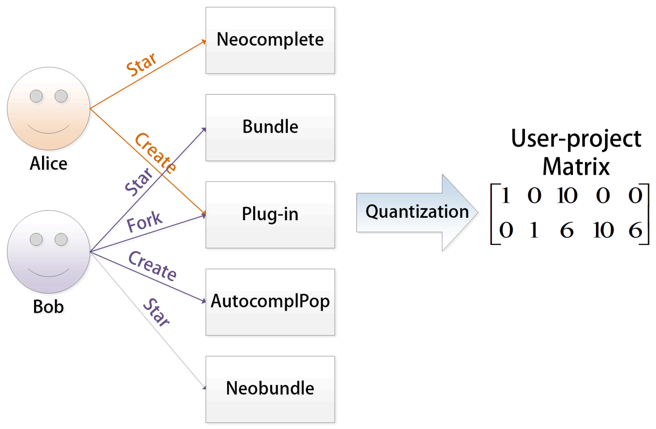
- *Create* behavior: GitHub allows users to create repositories and then creators or co-developers can update repositories.
- *Fork* behavior: A fork behavior is a copy of a repository that you manage. You can fetch updates from or submit changes to the original repository with pull requests.
- *Star* behavior: Starring a repository allows you to keep track of projects that you find interesting, even if you are not associated with the project.

These behaviors reflect the degree of a developer's demand for different projects [21]. For example, a developer created a project, which means that the project is directly related to him/her. In this case, he/she may need other similar projects for reference. In another case, if a developer stared a project, it may show that he/she is just interested in this project. A developer's behaviors on his/her known project is described as a triple $< user\_id, project\_id, value >$, where the *value* describes the degree of the user's ($user\_id$) needs for project ($project\_id$). Each *value* is decided by what kind of user's behavior. In this paper, we assign different values for different behaviors as Table 1 shows. All these behaviors constitute a user-project matrix UP, and $UP(a, b)$ represents the behavior value of the user a for the project b. In GitHub, a user may have different behaviors to the same project. In this case, we select the highest value from these behaviors for the project.

[5]https://help.github.com/

Table 1: Value of each behavior.

Behavior	Create	Fork	Star
Value	10	6	1



$$\begin{bmatrix} 1 & 0 & 10 & 0 & 0 \\ 0 & 1 & 6 & 10 & 6 \end{bmatrix}$$

Figure 4: An example of modeling the user behaviours.

Continuing with the previous example, in Figure 4, Alice stared *Neocomplete*, and created *Plug-in*; Bob stared *Bundle* and *Neobundle*, forked *Plug-in*, and created *AutocomplPop*. According to the above mapping rules, we can get the user-project matrix on the right of the figure.

3.3 Recommending Software Projects for Developers

When we get the project similarity matrix and user-project matrix, we can recommend software projects for these developers. The recommendation is performed based on a *Demand* measure, which is

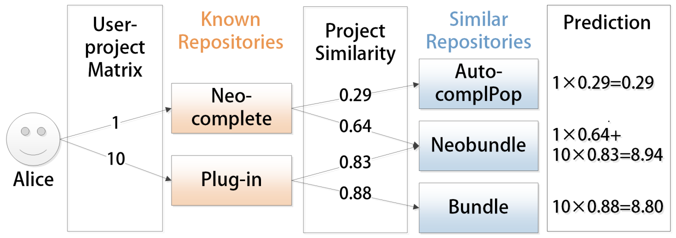


Figure 5: An example of recommending projects for developers.

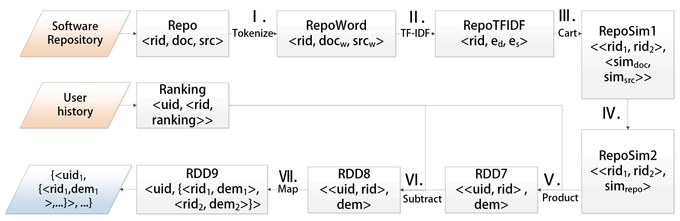

Software Repository
Repo <rid, doc, src>
I. Tokenize
RepoWord <rid, doc_w, src_w>
II. TF-IDF
RepoTFIDF <rid, e_d, e_s>
III. Cart
RepoSim1 <<rid_1, rid_2>, <sim_doc, sim_src>>
IV.
RepoSim2 <<rid_1, rid_2>, sim_repo>
V. Product
RDD7 <<uid, rid> , dem>
VI. Subtract
RDD8 <<uid, rid>, dem>
VII. Map
RDD9 <uid, {<rid_1, dem_1>, <rid_2, dem_2>}>
{<uid_1, {<rid_1,dem_1 >,...}>, ...}
User history
Ranking <uid, <rid, ranking>>

Figure 6: The process of RDD transformation.

used to show a user's needs for different projects. Given a user u and a project p, the $Demand(u, p)$ measure is defined as follows:

$$Demand(u, p) = \sum\nolimits_{i \in proj(u)} UP(u, i) \times SIM(i, p) \qquad (4)$$

In the above equation, $i \in proj(u)$ represents the projects which a user has labelled before (i.e., create, fork, or star), $UP(u, i)$ denotes the user u's demand for project i, and $SIM(i, p)$ is the similarity value between project i and p. Hence, the sum can be used to predict the user's need for unknown projects. We recommend top n projects with higher *demand* values.

An Example. Figure 5 shows an example of this procedure. We get Alice's behaviors from user-project matrix. Suppose we set $n = 2$. From the project similarity matrix, the two most similar projects with *Neocomplete* are *AutocomplPop* and *Neobundle*, and the similarity values are 0.29 and 0.64, respectively. The most two similar projects with *Plug-in* are *Neobundle* and *Bundle*, and the similarity values are 0.83 and 0.88, respectively. Then, we can predict the degree of Alice's needs for these projects as shown in the right rectangle. Finally, a Top-N recommendation is given to each user. In this example, if we recommend two similar projects, they are *Neobundle* and *Bundle*.

3.4 Parallel Computing

Suppose the number of projects is n and length of each vector is m, when calculating the similarity values between each pair of repositories, the time complexity is $O(n^2m)$, which consumes the most amount of time in our approach. In GitHub, there are over 10.6 million projects since 2014[6]. Based on the above analysis, a single machine is difficult to complete the recommendation. So we perform our recommendation in a parallel framework—Apache Spark. Apache Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of tuples with fixed formatting that can be processed in parallel [20]. Designing the transformation of RDD in a Directed Acyclic Graph (DAG) is the key point to realize our parallel algorithm. Figure 6 describes the transformation of our algorithm.

We first need to load all documents and source code in software repositories into RDD. As shown in Step I, we extract the words from documents and source code files and get the triples like $< rid, < doc, src >>$. Here rid is the identity of a repository; doc represents the word set of documents; and src represents the source code. The set of these tuples constitute a RDD (*repos*).

[6]https://github.com/features

In Step II, we use *HashingTF ()* and *IDF ()* functions to calculate the TF-IDF vector for document and source code, respectively. After that, we have $< rid, e_d, e_s >$ of each repository, where e_d represents the TF-IDF vector of documents and e_s represents the source code.

In Step III, we use *Cartesian()* function to get pairs of vectors, like $repoPair(<< rid_1, e_{d1}, e_{s1} >, < rid_2, e_{d2}, e_{s2} >>)$. Catresian product returns a tuple from two sets. For example, the RDD R has element $\{a, b\}$, then $Cartesian(R, R) = \{(a, a), (a, b), (b, a), (b, b)\}$.

In Step IV, we use $repoPair$ to do a mapping transformation to calculate the similarity values between different projects based on the document and the source code, respectively. This map applies the defined *Cosine()* function to every item of $repoPair$, which is a function of calculating the cosine similarity between vectors. So we get $repoSim1(<< rid1, rid2 >, < sim_{doc}, sim_{src} >)$.

In Step V, we use a mapping transformation to combine document and source code to get the similarity between different projects. With *Add()* function, we get $<< rid_1, rid_2 >, sim_{repo} >$, which is sorted in $repoSim2$, where sim_{repo} represents the similarity value between rid_1 and rid_2.

In Step VI, we get the tuple $behavior < uid, < rid, ranking >>$ from the user-project behaviour. Here uid is the identity of a user; rid is the identity of a repository; and $ranking$ is the score of the developer's (uid) behaviour on the software repository (rid). Then use it to calculate the user-project demand matrix with $repoSim2$. After Step VI, we get the tuple like $<< uid, rid >, val >$, which is sorted in $recRes1$. val shows the user uid's interest in software rid.

Next step, we use $resRes1$'s $< uid, rid >$ as key and use *subtractByKey ()* with $behavior$ to remove the projects the developer has known.

Finally, we use $resRes2$'s uid as key to do *groupByKey ()* to collect all the projects related to user uid and select the top-N projects. In this way, we can get the tuple $< uid, S >$, where S represents the top-N repositories recommended for user uid.

4 EMPIRICAL STUDY

4.1 Research Questions

In our study, we discuss the following three research questions:

RQ1: Does our approach perform better than the state-of-the-art recommendation approaches?

RQ2: Is it necessary to extract the features from both the description documents and source code of a project?

RQ3: Is our parallelization algorithm scalable to large-scale data?

Our approach is proposed for repository recommendation, so we study *RQ1* to investigate whether the recommendation results are accurate. When calculating the similarity, we first extract features of a project from both its description documents and source code for software project recommendation. Then, we assign the source code files and description documents to different weights. We are also concerned about the impact of different weights on our recommendation approach (to answer *RQ2*). Finally, as we use the parallelization algorithm to improve the scalability, we investigate *RQ3* to show whether our approach can be scalable to large-scale data.

4.2 Empirical Environment

We implemented our approach with Python 2.7.9 and Apache Spark 2.0.1. For *RQ1~RQ2*, we launched our experiment on 8GB RAM and 8 Cores CPU with 2.40 GHz in each core. In *RQ3*, we set up a cluster with Amazon EC2[7] clusters consisting of $n = 1 \sim 5$ worker nodes. Each node includes a 4-core, 64-bit machine with 8GB of RAM memory.

4.3 Parameters Setting

In our approach, there are a series of parameters which need to be set, such as k most similar programs to perform the similarity calculation. We set k to 100 which is shown to be effective [5]. In addition, the parameters α and β represent the weight of description documents and source code. For *RQ1*, we set α to 0.8 and β to 0.2 since we empirically find these values achieve a reasonable good performance, which will be proved by the results of *RQ2*. In *RQ2*, we set α from 0 to 1 with a step of 0.1, thus β was set from 1 to 0 to find the influence of these parameters on accuracy. In *RQ3*, since these values do not affect the speed of calculations, we still use the default values i.e., $a = 0.8$, $b = 0.2$. Finally, the parameter N represents top N similar programs needed to be recommended. In *RQ1* and *RQ2*, we set N to be 3, 5 and 10 for our study. For other research questions, we set N to 10.

4.4 Methods and Evaluation Metrics

4.4.1 Methods. We use GitHub API[8] to fetch the data. For *RQ1~RQ2*, we used four groups of data. The first three groups are extracted from three organizations on GitHub named *vim-jp*[9], *Formidable*[10], and *harvesthq*[11]. *vim-jp* is a vim community for Japanese developers and users, and they mainly use Java and C/C++ to develop projects. *Formidable* focuses on web application development such as PHP, Javascript, Ruby, and CSS development. *harvesthq* focuses on Android app development. These three groups represent three different development areas so that we can figure out the performance of our recommendation approach. The reason we selected these three groups is that there are significant relevant projects in them. For example, in *vim-jp*, projects *vital.vim*[12], *javacomplete*[13], and *unite.vim*[14] are relevant, since they all provide support tools to vim development. We first crawled all the developers in these 3 organizations, and then we crawled the projects that these developers create, fork, and star[15]. In the last group, we crawled 1,010 active users on GitHub (they have at least 20 software repositories on GitHub) and 11,518 repositories related to them. The preferences of these users and their development areas are different, and their projects are developed in different languages. We experiment with this group to test the effect of our system in the real environment. Specifically, we chose 60% of the developers' behaviors and their software projects as our input, and used the remaining 40% for testing. Each experiment was repeated ten times and we used the average value. The details of each group are shown in Table 2.

Table 2: Statistics of four groups on GitHub

Group Name	Users	Projects	Development Areas
vim-jp	22	562	Vimscript
Formidable	16	185	Web
harvesthq	43	540	Android
Mixed	1010	11518	/

In *RQ1*, our approach recommends top N projects to each developer in each group. Then, we compared our recommendation results with other two typical recommendation algorithms, i.e.,*UserCF* and *ItemCF*. *UserCF* first identifies a collection of users that are similar to the target user's interests, and detects the projects that the user likes in the collection and recommends these projects to the target user [11]. *ItemCF* calculates the similarity of projects based on users' behaviors, which generates a recommendation list according to the similarity of the items and the user's historical behaviors [11].

In *RQ2*, we changed parameter α from 0 to 1 with a step of 0.1, thus β from 1 to 0, and observe the change in the accuracy of our approach.

In *RQ3*, we crawled more data with different sizes to test the scalability of our recommendation approach. We fetched four groups with 400, 600, 800, and 1,010 users, and 5,818, 8,378, 10,473 and 11,518 projects for this study. The four groups of data are 250MB, 360MB, 450MB, and 500MB, respectively, using $1 \sim 5$ worker nodes to calculate the running time of our algorithm.

4.4.2 Evaluation Metrics. For the first two research questions (*RQ1* and *RQ2*), we use the *Accuracy*, *Recall*, *Precision*, and *F*1 metrics to answer them, respectively. These four metrics are defined as follows:

$$Accuracy = \frac{|\{u|u \in U, R(u) \cap T(u) \neq \emptyset\}|}{|U|} \tag{5}$$

$$Recall = \frac{\sum_{u \in U} |R(u) \cap T(u)|}{\sum_{u \in U} |T(u)|} \tag{6}$$

[7] https://aws.amazon.com/ec2/dedicated-hosts/
[8] https://developer.github.com/
[9] https://github.com/vim-jp
[10] https://github.com/FormidableLabs
[11] https://github.com/harvesthq

[12] https://github.com/vim-jp/vital.vim
[13] https://github.com/Shougo/javacomplete
[14] https://github.com/Shougo/unite.vim
[15] The data were crawled in October 6th, 2016. The number of developers in these 3 organizations might be changed.

$$Precision = \frac{\sum_{u \in U} |R(u) \cap T(u)|}{\sum_{u \in U} |R(u)|} \tag{7}$$

$$F1 = \frac{Recall * Precision * 2}{Recall + Precission} \tag{8}$$

U represents all the users in test data. $R(u)$ represents the number of relevant project repositories recommended to user u by the recommendation approach. $T(u)$ represents the number of relevant project repositories about user u in testing data which are extracted from the remaining 40% of the user behaviors.

For *RQ3*, we calculate the time efficiency of the data sets with different data sizes on different numbers of worker nodes [2]. Obviously, when the size increases, the computing time will increase [1]. Besides, we introduce the speedup ratio, which is the performance achieved by reducing the running time in parallel computing, which is calculated as $Speedup = T_s/T_d$, where T_s is the time consumed by the algorithm (i.e., on a single node) and T_d is the time consumed by the parallel algorithm (i.e., on d identical worker nodes). The higher is the speedup value, the less the relative time is consumed by parallel computing, and the higher is for parallel efficiency and performance. We performed each experiment three times and used the average time as the execution time.

5 EMPIRICAL RESULTS

5.1 Answer to RQ1

First, we show that whether the recommendation results are effective, as well as compared with two typical personalized recommendation approaches, i.e., user collaborative filtering (*UserCF*) and item collaborative filtering (*ItemCF*) [17].

The empirical results are shown in Table 3. The results show that the *Accuracy*, *Recall*, *Precision*, and $F1$ values of our approach are much higher than that of *UserCF* and *ItemCF*. Both *UserCF* and *ItemCF* have poor results because there are so many unpopular projects on GitHub, which makes user-project matrix sparse. Thus, *UserCF* cannot accurately detect similar developers based on users' behaviors and *ItemCF* cannot use it to calculate the similarity. So in our algorithm, we calculate the similarity of projects using documents and source code, as well as developers' behaviors. This greatly improves the accuracy of our recommendation, for example, in the top 10 recommendation of Formidable group, the *Accuracy* value of our approach reaches 95.00%. Even in the worst case, the *Accuracy* of our approach is 54.72%. This ensures that over a half of users could find their interested projects—at least one, rather than getting nothing.

Therefore, from the results discussed above, compared with *UserCF* and *ItemCF*, our approach can recommend more accurate results for developers based on their behaviors and characteristics of the software projects.

The reason for the low *Recall* in *Mixed* group is that the user behavior in the test is less, while our algorithm should recommend top N repositories for each user. We can notice that the precision of top 10 recommendation is 17.52%, which means when recommending 10 projects, there are 1.7 relevant projects. In practice, developers might be satisfied with this approach considering there are millions of projects on GitHub.

5.2 Answer to RQ2

In this section, we investigate whether it is necessary to analyze both the description documents and source code to extract features for a project on GitHub and how to set their (α and β) weights.

Through figure 7(a), we can see that α and β have little effect on $Accuracy$. However, we can notice that when $\alpha = 0$ (i.e., only considering the source code) and $\beta = 0$ (i.e., only considering the description document), the $Accuracy$ results are not the best, which indicates that the features of a software project cannot be extracted only from documentation or source code. Figure 7(b) and Figure 7(c) show the $Recall$ and $Precision$ results of each group with different α and β values. Similarly, we can see that when $\alpha = 0$ or $\beta = 0$ the $Precision$ and $Recall$ is bad. However, when $\alpha \in [0.7, 0.9]$ (i.e. $\beta \in [0.1, 0.3]$.), their values are relatively high. In addition, Figure 7(d) demonstrates the $F1$ scores with different α values. The results show that when α increases, $F1$ increases at the beginning; but when α value increases to 0.6, the growing rate slows down. When α achieves 0.8, $F1$ value starts to decline. We should notice that in case of $\alpha = 0$ or $\alpha = 1$, which represents that the features of a project are extracted only from the source code or description documents, the $F1$ scores are not the best. Based on the $Accuracy$, $Precision$, $Recall$, and $F1$ scores with different α and β values, we can conclude that it is necessary to extract features of a software project using both the description documents or source code.

From the results discussed above, both software source code and description documents are important for content recommendation in our approach, and the weights assigned to them do affect the accuracy results. To let more people find their interests—at least one—and to ensure the recommendation accuracy, we can set $\alpha = 0.8, \beta = 0.2$ considering the results as shown in Figure 7.

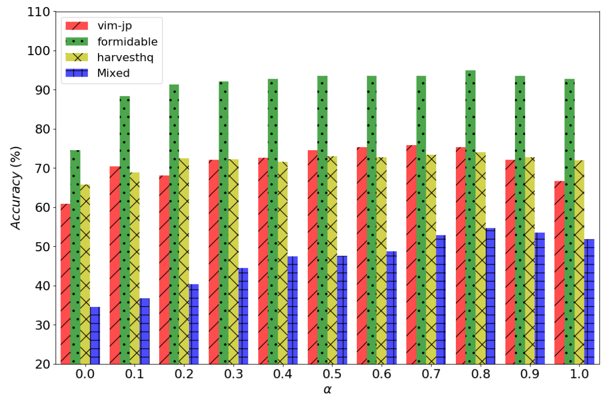
5.3 Answer to RQ3

In GitHub, there are a large number of software developers and software projects. So to analyze such large-size data needs a lot of time for recommendation. Moreover, the size of the data on GitHub is still increasing. In our approach, we implement our approach on Apache Spark to process such large-size data. In this section, we evaluate the scalability of our approach by showing the performance of different datasets on different numbers of worker clusters.

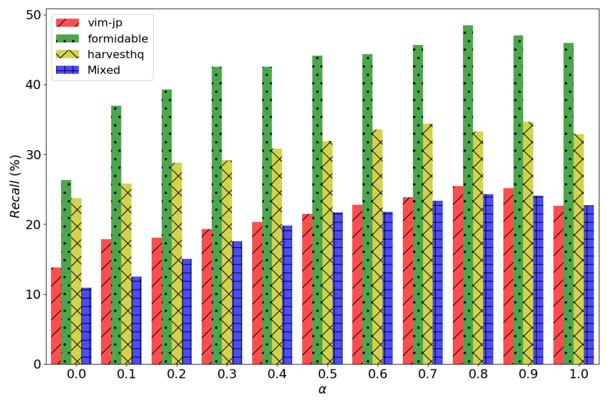
The experimental results are shown in Figure 8. From Figure 8(a), we can notice that our parallel algorithm can reduce the execution time by increasing the worker nodes. For example, when the number of worker nodes increases from 1 to 2, the reduction rate of execution time is obvious. In addition, when the number of worker nodes is 1, the runtime of different datasets is different; but when the number of worker nodes is 5, the difference is significantly smaller. It means that by controlling the number of worker nodes, the execution time of our approach is controllable. In Figure 8(b), we can notice that the acceleration effect is different for different datasets when the number of worker nodes increases. For example, when the dataset is only 250MB, the speedup effect is small if we added another node on the basis of 2 nodes. However, when the datasets are 450M and 500M, there is a big speedup when the number of nodes is increased from 3 to 4. This is because when the dataset is small, one or two nodes can finish the calculation process in a short

Table 3: The empirical results of our system, *UserCF* and *ItemCF*

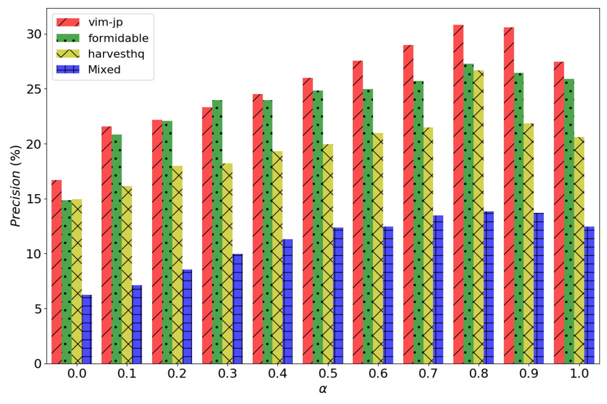
		vim-jp			Formidable			harvesthq			Mixed		
		Top3	Top5	Top10	Top3	Top5	Top10	Top3	Top5	Top10	Top3	Top5	Top10
Accuracy	Ours	**65.13%**	**71.06%**	**75.32%**	**84.72%**	**89.07%**	**95.00%**	**61.80%**	**67.85%**	**73.99%**	**36.58%**	**42.98%**	**54.72%**
	UserCF	5.37%	6.46%	8.60%	6.57%	6.57%	6.57%	5.78%	8.53%	0.71%	1.47%	1.47%	2.41%
	ItemCF	4.26%	6.98%	8.62%	7.31%	7.31%	7.31%	5.78%	7.41%	0.71%	1.47%	1.47%	2.31%
Recall	Ours	**11.27%**	**16.51%**	**25.43%**	**30.14%**	**37.63%**	**48.46%**	**18.69%**	**25.51%**	**33.29%**	**10.32%**	**14.32%**	**24.28%**
	UserCF	0.44%	0.55%	0.82%	1.28%	1.28%	1.28%	0.97%	1.47%	2.10%	0.28%	0.28%	0.47%
	ItemCF	0.39%	0.63%	0.90%	1.42%	1.42%	1.42%	0.96%	1.24%	1.78%	0.28%	0.28%	0.45%
Precision	Ours	**45.00%**	**39.80%**	**30.82%**	**56.07%**	**42.22%**	**27.29%**	**38.98%**	**32.22%**	**26.68%**	**19.55%**	**16.27%**	**13.83%**
	UserCF	2.32%	1.89%	1.48%	1.80%	1.80%	1.80%	2.82%	2.63%	2.03%	0.46%	0.46%	0.40%
	ItemCF	1.88%	2.26%	1.69%	1.90%	1.90%	1.90%	2.87%	2.23%	1.72%	0.46%	0.46%	0.39%
F1	Ours	**18.01%**	**23.31%**	**27.83%**	**39.17%**	**39.76%**	**34.89%**	**25.25%**	**28.46%**	**27.47%**	**13.51%**	**15.23%**	**17.62%**
	UserCF	0.73%	0.84%	1.04%	1.45%	1.45%	1.45%	1.43%	1.88%	2.06%	0.35%	0.35%	0.43%
	ItemCF	0.63%	0.98%	1.16%	1.57%	1.57%	1.57%	1.44%	1.59%	1.74%	0.35%	0.35%	0.41%



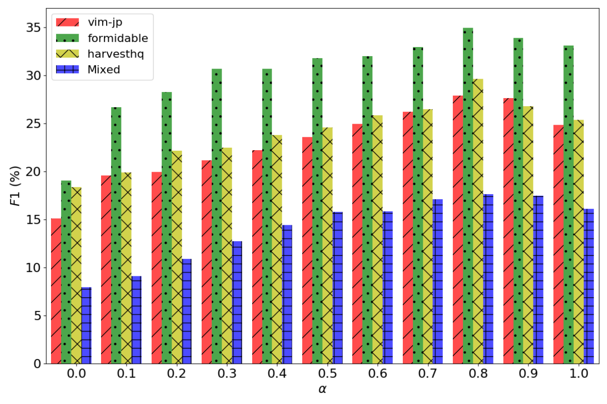
(a) *Accuracy* of each group.



(b) *Recall* of each group.



(c) *Precision* of each group.

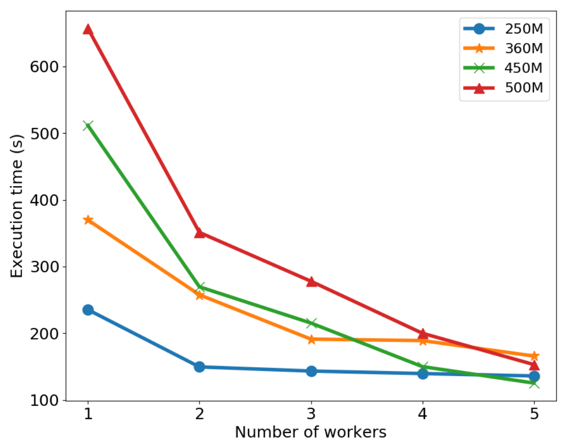


(d) $F1$ of each group.

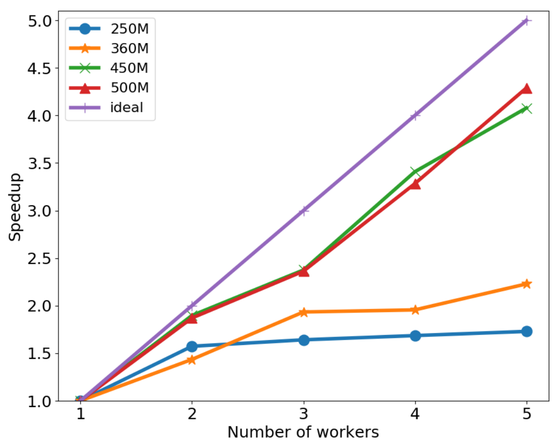
Figure 7: Results of RQ2.

time. In this situation, increasing the number of nodes will increase the time of resource allocation. The speedup effect is not obvious. However, when the dataset becomes large, the situation is different. One or two nodes cannot complete the calculation. And the execution time is greatly reduced when we add nodes. Especially in the dataset with 450M and 500M, we get a linear growth in the runtime which shows the scalability of our approach in large-scale data.

Based on the results discussed above, the parallel framework in our approach can process large-scale data in a short time by adding



(a) Runtime of each group.



(b) Speedup of each group.

Figure 8: Results of RQ3.

more machines to the cluster to ensure the scalability of the increasing data in GitHub.

6 THREATS TO VALIDITY

6.1 Threats to Internal Validity

We highlight the internal threats in terms of availability of description files and weights in the textual similarity.

Availability of Description Files: As we noticed, some projects might not have the description files, in such a case, our approach will only evaluate the textual similarity of two projects based on their source code file similarity. Although it might reduce the accuracy of our approach, as shown in RQ2, even without the description file similarity, our approach still achieve a substantial improvement over the baseline approaches.

Weights: We set the default values of the weights (α and β) as 0.8 and 0.2, respectively. In practice, the optimal value of the parameters might be different for different datasets, and in our study we empirically find $\alpha = 0.8$ and $\beta = 0.2$ could achieve a reasonable performance. In the future, we plan to design an automated algorithm to tune these two weights.

6.2 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 experiment size.

Programming Languages: GitHub contains numerous repositories written in multiple programming languages (e.g. Java, Python, PHP, C++), or combinations of multiple programming languages. Our approach is not designed for a single language and can be applied to all GitHub repositories. In this paper, we set up our experiments on projects on multiple languages such as Java, Python, C/C++. In the future, we plan to perform more experiments to further reduce this threat.

Experiment Size: In RQ1-RQ2, we applied our technique to four groups on GitHub. The first three groups represent three different development domains on GitHub, and the experiment size is similar to a previous related work by Zhang et al. [22] which uses 50 queries and search for similar projects from a pool of 1,000 projects. The last group contains a total of more than 10,000 projects and 1,000 developers. In the future, we plan to include more developers and projects to reduce the threats to validity.

7 RELATED WORK

Our work is inspired by the empirical study of Zhang et al. [21]. They illustrated four types of user behavior data, including fork, watch, pull-request and member which are suitable for finding relevant projects. And different user behavior data sets are suitable for different recommendation purposes. In this paper, we proposed our approach on GitHub to recommend software projects for developers by considering their different behaviours.

There are some works focusing on detecting similar repositories [7, 16, 22]. McMillan et al. design a tool *CLAN*, which uses Latent Semantic Indexing (LSI) to measure the similarity of repositories on API usage [7]. Their approach achieves a higher precision than previous studies. Thung et al. propose a different way which combines tags given by SourceForge to detect similar repositories [16]. They performed a user study with over a hundred thousand of applications from SourceForge and proved to be more efficient than *JavaCLAN* that only consider APIs. Zhang et al. further improve *CLAN*, which can detect similar repositories on GitHub based on two data sources (i.e., GitHub stars and README files) [22]. They evaluated their technology with 4 participates and 1,000 java repositories and got more accurate results than CLAN.

All of their work tend to recommend projects based on a query project, i.e., a developer needs to enter a query project, and their works will recommend similar projects. Different from their study, our study solve a related but different problem: we focus on scalable, personalized and relevant project recommendation, i.e., we consider the developers' historical behaviors and project similarity to recommend relevant projects, and we do not have the user query, while their tools are similar to a search engine inside GitHub. Moreover, we consider different types of users' behaviors on GitHub, and we not only consider the README files (documents). But also source code to improve the recommendation accuracy. As we observed, not all projects in GitHub has the README files. Moreover, CLAN only considers projects written in Java, while our approach can process projects in all programming languages.

REPERSP is a realized tool we created based on this paper's method [18]. It provides a web service to interact with users, including fetching data from GitHub, providing recommendation results, and receiving feedbacks.

8 CONCLUSION AND FUTURE WORK

In this paper, we proposed a scalable software project recommendation on GitHub. The recommendation is performed based on the developers' behaviors and the features extracted from the description and source code. In addition, our approach is implemented in Apache Spark to process an increasing size of data on GitHub. We evaluated our approach with four groups of data from Github, and the results showed that our approach can recommend more accurate results. Moreover, our approach can be used to efficiently process large-scale data by adding more machines to the cluster to ensure the scalability of the increasing data in GitHub.

However,there is still some further work needed to be done from different aspects. First, we need to do more experiments, especially online experiments with actual developers, to figure out whether our recommendation approach is actually useful. Second, we need to improve our approach, for example, to find a better way to detect user behaviors, to further improve the recommendation accuracy. Finally, we can take into account the user feedback on our recommendation results, which will make our recommendation approach more practical and effective.

ACKNOWLEDGMENTS

This work is supported partially by Natural Science Foundation of China under Grant No. 61402396 and No. 61472344, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2016B21, partially by the Jiangsu Qin Lan Project, partially by the China Postdoctoral Science Foundation under Grant No. 2015M571489.

REFERENCES

[1] Steven H.H. Ding, Benjamin C.M. Fung, and Philippe Charland. 2016. Kam1N0: MapReduce-based Assembly Clone Search for Reverse Engineering. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 461–470.

[2] Zekeriya Erkin, Michael Beye, Thijs Veugen, and Reginald L. Lagendijk. 2012. Privacy-preserving Content-based Recommender System. In *Proceedings of the on Multimedia and Security (Sec '12)*. ACM, New York, NY, USA, 77–84.

[3] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227.

[4] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark: Lightning-Fast Big Data Analytics* (1st ed.). O'Reilly Media, Inc.

[5] Daniel T Larose. 2005. k-Nearest Neighbor Algorithm. *Discovering Knowledge in Data: An Introduction to Data Mining* (2005), 90–106.

[6] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of massive datasets*. Cambridge University Press.

[7] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting Similar Software Applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 364–374.

[8] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. 2012. Recommending Source Code for Use in Rapid Software Prototypes. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 848–858.

[9] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241.

[10] Michael J Pazzani and Daniel Billsus. 2007. Content-based recommendation systems. In *The adaptive web*. Springer, 325–341.

[11] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. *Introduction to recommender systems handbook*. Springer.

[12] Wei Shi, Xiaobing Sun, Bin Li, Yucong Duan, and Xiangyue Liu. 2015. Using feature-interface graph for automatic interface recommendation: A case study. In *Advanced Cloud and Big Data, 2015 Third International Conference on*. IEEE, 296–303.

[13] Xiaobing Sun, Bin Li, Yucong Duan, Wei Shi, and Xiangyue Liu. 2016. Mining Software Repositories for Automatic Interface Recommendation. *Scientific Programming* 2016 (2016).

[14] Xiaobing Sun, Bin Li, Yun Li, and Ying Chen. 2015. What Information in Software Historical Repositories Do We Need to Support Software Maintenance Tasks? An Approach Based on Topic Model. In *Computer and Information Science*. 27–37. https://doi.org/10.1007/978-3-319-10509-3_3

[15] Xiaobing Sun, Xiangyue Liu, Jiajun Hu, and Junwu Zhu. 2014. Empirical Studies on the NLP Techniques for Source Code Data Preprocessing. In *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies (EAST 2014)*. ACM, New York, NY, USA, 32–39. https://doi.org/10.1145/2627508.2627514

[16] Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Detecting similar applications with collaborative tagging. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 600–603.

[17] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. 2006. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 501–508.

[18] Wenyuan Xu, Xiaobing Sun, Jiajun Hu, and Bin Li. 2017. REPERSP: Recommending Personalized Software Projects on GitHub. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE.

[19] Cheng Yang, Qiang Fan, Tao Wang, Gang Yin, and Huaimin Wang. 2016. RepoLike: Personal Repositories Recommendation in Social Coding Communities. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware (Internetware '16)*. ACM, New York, NY, USA, 54–62.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[21] Lingxiao Zhang, Yanzhen Zou, Bing Xie, and Zixiao Zhu. 2014. Recommending Relevant Projects via User Behaviour: An Exploratory Study on Github. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies (CrowdSoft 2014)*. ACM, New York, NY, USA, 25–30.

[22] Yun Zhang, David Lo, Kochhar Pavneet Singh, Xin Xia, Quanlai Li, and Jianling Sun. 2017. Detecting Similar Repositories on GitHub. In *2017 IEEE 24rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE.