Using collaborative filtering to recommend Github topics

ABSTRACT

Collaborative filtering is a well-founded technique spreadly used in the recommendation system domain. During recent years, a plethora of approaches has been developed to provide the users with relevant items. Considering the open-source software (OSS) domain, GitHub has become a precious service for storing and managing software source code. To represent the stored projects in an effective manner, in 2017 GitHub introduced the possibility to classify them employing topics. However, assigning wrong topics to a given repository can compromise the possibility of helping other developers reach it and eventually contribute to its development. In this paper, we present <RECOMMENDER ACRONYM>, a recommender system to assist open source software developers in selecting suitable topics to the repositories. <RECOMMENDER ACRONYM> exploits a collaborative filtering technique to recommend libraries to developers by relying on the set of dependencies, which are currently included in the project being This paper show...

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, collaborative filtering, topic recommender

ACM Reference Format:

. 2018. Using collaborative filtering to recommend Github topics. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In recent years, the open source software (OSS) community makes a daily usage of open source repositories to contribute their work as well as to access to projects coming from other developers. GitHub is one of the most well-known platforms that aggregate these projects and render possible the exchange of knowledge among the users. In order to aid information discovery and help developers identify projects that can be of their interest, GitHub introduced *topics*. They are words used to characterize projects, which thus can be annotated by means of lists of words that summarize projects' features. Thanks to the availability of *topics*, several applications are enabled, including the automated cataloguing of GitHub repositories [**?**], further than allowing developers to explore projects by type, technology, and more.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Assigning the right topics to GitHub repositories is a crucial step that, if not properly done, can affect in a negative way their discoverability. In 2017, GitHub presented *repo-topix*, a topic suggestion tool essentially based on information retrieval techniques [**?**]. Although the mechanism works well so far and it is fully integrated in GitHub, in our opinion there is still some room for improvement, e.g., in terms of the variety of the suggested topics, novel data analysis techniques, and the investigation of new recommendation strategies.

In this work, we propose to face these issues by exploiting collaborative filtering, a widely spread technique in the recommendation system domain [34]. By considering GitHub projects as items to recommend, we use user-topic matrixes to perform the similarity and suggest relevant topics given an initial list coming from an input project.

By considering GitHub projects as items to recommend, we use user-topic matrixes to perform the similarity and suggest relevant topics given an initial list coming from an input project. The work gives the following contributions:

- Considering the GitHub projects as products, we suggest relevant topics to the project given an initial list of them;
- We assess the quality of the work employing a well-defined set of metrics commonly used in the recommendation system domain i.e., sales diversity, novelty, and accuracy

The rest of the work is structured as follows. Section 2 shows the issues and the potential challenges in the domain. In Section 3, we present our approach and evaluate it in Section 4. We present the results of the assessment in Section 5 and we discuss the findings. Section **??** summarizes relevant works in the field and we conclude the paper in Section 7 with possible future works.

2 BACKGROUND

Manually assigning topics can be an error-prone activity that can lead to wrongly specified tags. Over the last years, several attempts have been made to *classify* GitHub projects by automatically inferring appropriate topics. In the context of data mining, *classification* is one of the critical operations that are used to dig deep into available data for gaining knowledge and for identifying repetitive patterns [**?**].

In [**?**] the authors present an approach based on *topic modeling* techniques to create categories of GitHub projects. Manual interventions are needed to refine initial sets of categories, which are identified by an LDA-GA technique, that combines two algorithms: Latent Dirichlet Allocation (LDA) and Genetic Algorithm (GA) [**?**]. The approach proposed in [**?**] is unsupervised, meaning that the categories of the catalogue being identified are not known ex-ante.

In a GitHub blog post [**?**] the author presents *repo-topix*, a tool to recommend topics for GitHub repositories. Such a tool combines NLP standard techniques to find an initial set of topics, by parsing the README files and the textual content of a repository e.g., the repository's description. Then, they weight the results with the TF-IDF scheme and remove "bad" topics using a regression model.

Using this refined list, repo-topix computes a custom version of Jaccard Distance to identify additional similar topics. To assess the quality of the framework, they made a rough evaluation based on ROUGE-1 metrics, an n-gram overlap metric that counts the number of overlapping units between the suggested topics and the repository description. Unfortunately, in [**?**] the author discusses an approximation of the repo-topix accuracy, without providing the reader with the complete dataset that was used and the source code of the developed tool.

With the aim of contributing the resolution of the problem of recommending GitHub topics, in the next section we propose to use item-based collaborative filtering to recommend relevant topics. The challenges that we had to cope with for evaluating its performance are mainly the following ones:

▹ *Dataset definition:* the creation of the datasets to be used for evaluating the approach being proposed and comparing it with some baseline is a daunting task: repositories might be moved, heavily changed or even deleted during the initial creation. Thus, the crawling activity can be negatively affected by these continuous changes and lead to lack of data, and poor topic coverage. GHTorrent[1] tries to mitigate this issue by offering daily dumps of the repositories' metadata. However, this kind of data might not be enough or even appropriate (e.g., source code is not available in GHTorrent dumps) to properly classify an entire repository. Even considering directly GitHub data can be difficult: GitHub limits the total number of requests per hour to 5,000 for authenticated users and 60 for unauthorized requests. Considering all these constraints, building a suitable dataset represents a real challenge to be managed carefully.

▹ *Topics distribution:* although tags can be assigned only by the owners of GitHub repositories, users can potentially wrongly specify topics or introduce information overload by inserting too many elements. Thus, creating a reliable ground truth to assess the classification performance of the proposed approach represents another relevant difficulty.

3 PROPOSED APPROACH

In this section we describe <RECOMMENDER ACRONYM>, a system for providing developers with GitHub topics related recommendations. More specifically, <RECOMMENDER ACRONYM> is a *recommender system* [3] that encodes the relationships among various OSS artifacts by means of a semantic graph and utilizes a collaborative filtering technique [34] to recommend GitHub topics. Such a technique has been originally developed for e-commerce systems to exploit the relationships among users and products to predict the missing ratings of prospective items [16]. The technique is based on the premise that *"if users agree about the quality or relevance of some items, then they will likely agree about other items"* [34]. Our approach exploits this premise to solve the problem of library recommendation [30]. Instead of recommending goods or services to customers, we recommend third-party libraries to projects using an analogous mechanism: *"if projects share some libraries in common, then they will probably share other common libraries."*

[1]http://ghtorrent.org/

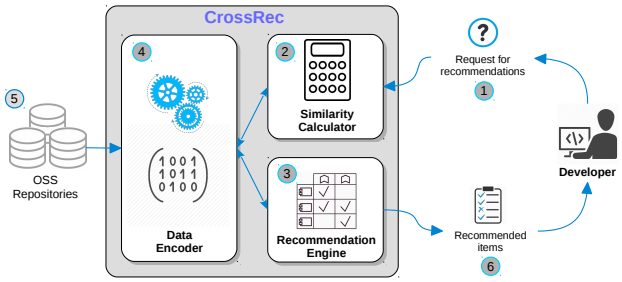


Figure 1: Overview of the <RECOMMENDER ACRONYM> Architecture.

To this end, the architecture of <RECOMMENDER ACRONYM> is shown in Fig. 1, and consists of the software components supporting the following activities:

- *Representing the relationships* among projects and topics retrieved from existing repositories;
- *Computing similarities* to find projects, which are similar to that under development; and
- *Recommending topics* to projects using a collaborative-filtering technique.

In a typical usage scenario of <RECOMMENDER ACRONYM>, we assume that a developer is creating a new system, in which she has already included some topics, or else is evolving an existing system. As shown in Fig. 1, the developer interacts with the system by sending a request for recommendations ①. Such a request contains a list of topics that are already included in the project the developer is working on. The `Data Encoder` ④ collects *background data* from OSS repositories ⑤, represents them in a graph, which is then used as a base for other components of <RECOMMENDER ACRONYM>. The `Similarity Calculator` module ② computes similarities among projects to find the most similar ones to the given project. The `Recommendation Engine` ③ implements a *collaborative-filtering* technique [3],[37], it selects top-k similar projects, and performs computation to generate a ranked list of *top-N* topics. Finally, the recommendations ⑥ are sent back to the developer.

Background data has been collected GitHub [1]. The current version of <RECOMMENDER ACRONYM> [21] supports data extraction from GitHub, even though the support for additional platforms is under development.

In the following, the components `Data Encoder`, `Similarity Calculator`, and `Recommendation Engine` are singularly described.

3.1 Data Encoder

A recommender system for online services is based on three key components, namely *users*, *items*, and *ratings* [33],[25]. A *user-item ratings matrix* is built to represent the mutual relationships among the components. Specifically, in the matrix a user is represented by a row, an item is represented by a column and each cell in the matrix corresponds to a rating given by a user for an item [25]. For topic recommendation, instead of users and items, there are projects and topics, and a project may include various topics to better describe the project. We derive an analogous user-item ratings matrix to

represent the *project-library inclusion* relationships, denoted as ∋. In this matrix, each row represents a project and each column represents a topic. A cell in the matrix is set to 1 if the topic in the column is included in the project specified by the row, it is set to 0 otherwise. For the sake of clarity and conformance, we still denote this as a user-item ratings matrix throughout this paper.

For explanatory purposes, we consider a set of four projects $P = \{p_1, p_2, p_3, p_4\}$ together with a set of topics $L = \{topic_1$=*machine-learning*; $topic_2$=*javascript*; topiclib$_3$=database; $topic_4$=*web*; $topic_5$=*algorithm*$\}$. By extracting the list of defined topics of the projects in P, we discovered the following inclusions: $p_1 \ni topic_1, topic_2$; $p_2 \ni topic_1, topic_3$; $p_3 \ni topic_1, topic_3, topic_4, topic_5$; $p_4 \ni topic_1, topic_2, topic_4, topic_5$. Accordingly, the user-item ratings matrix built to model the occurrence of the topic is depicted in Fig. 2.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{ccccc} topic_1 & topic_2 & topic_3 & topic_4 & topic_5 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figure 2: An example of a user-item ratings matrix to model the inclusion of topics in GitHub repositories.

3.2 Similarity Calculator

The `Recommendation Engine` of <RECOMMENDER ACRONYM> works by relying on an analogous user-item ratings matrix. To provide inputs for this module, the first task of <RECOMMENDER ACRONYM> is to perform similarity computation on its input data to find the most similar projects to a given project. In this respect, the ability to compute the similarities among projects has an effect on the recommendation outcomes. Nonetheless, computing similarities among software systems is considered to be a difficult task [17]. In addition, the diversity of artifacts in OSS repositories as well as their cross relationship makes the similarity computation become even more complicated. In OSS repositories, both humans (i.e., developers and users) and non-human actors (such as repositories, and libraries) have mutual dependency and implication on the others. The interactions among these components, such as developers commit to or star repositories, or projects include libraries, create a tie between them and should be included in similarity computation.

We assume that a representation model that addresses the semantic relationships among miscellaneous factors in the OSS community is beneficial to project similarity computation. To this end, we consider the community of developers together with OSS projects, topics, and their mutual interactions as an *ecosystem*. We derive a *graph-based* model to represent different kinds of relationships in the OSS ecosystem, and eventually to calculate similarities. In the context of mining OSS repositories, the graph model is a convenient approach since it allows for flexible data integration and numerous computation techniques. By applying this representation, we are able to transform the set of projects and topics shown in Fig. 2 into a directed graph as in Fig. 3. We adopted our proposed CrossSim

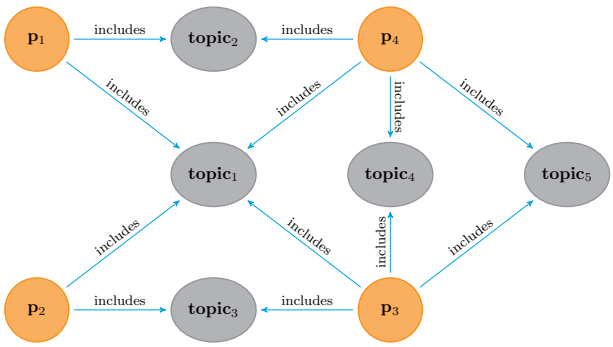


Figure 3: Graph representation for projects and libraries.

approach [22],[23] to compute the similarities among OSS graph nodes. It relies on techniques successfully exploited by many studies to do the same task [10],[6]. Among other relationships, two nodes are deemed to be similar if they point to the same node with the same edge. By looking at the graph in Fig. 3, we can notice that p_3 and p_4 are highly similar since they both point to three nodes $topic_1$, $topic_4$, $topic_5$. This reflects what also suggested in a previous work by McMillan et al. [17], i.e., similar projects implement common pieces of functionality by using a shared set of libraries.

Using this metric, the similarity between two project nodes p and q in an OSS graph is computed by considering their feature sets [10]. Given that p has a set of neighbor nodes $(topic_1, topic_2, .., topic_l)$, the features of p are represented by a vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_i being the weight of node $topic_i$. It is computed as the *term-frequency inverse document frequency* value as follows:

$$\phi_i = f_{topic_i} \times log(\frac{|P|}{a_{topic_i}}) \quad (1)$$

where f_{topic_i} is the number of occurrence of $topic_i$ with respect to p, it can be either 0 and 1 since there is a maximum of one $topic_i$ connected to p by the edge *includes*; $|P|$ is the total number of considered projects; a_{topic_i} is the number of projects connecting to $topic_i$ via the edge *includes*. Eventually, the similarity between p and q with their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_i\}_{i=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$ is computed as given below:

$$sim(p, q) = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n} (\phi_t)^2} \times \sqrt{\sum_{t=1}^{n} (\omega_t)^2}} \quad (2)$$

where n is the cardinality of the set of libraries that p and q share in common [10]. Intuitively, p and q are characterized by using vectors in an n-dimensional space, and Eq. 2 measures the cosine of the angle between the two vectors.

3.3 Recommendation Engine

Phuong ▶*Please reprhare this section*◀ The representation using a user-item ratings matrix allows for the computation of missing ratings [3],[25]. Depending on the availability of data, there are two main ways to compute the unknown ratings, namely *content-based* [27] and *collaborative-filtering* [19] recommendation techniques.

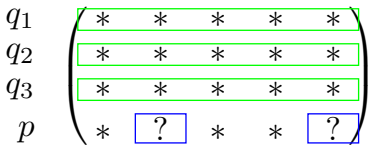


Figure 4: Computation of missing ratings using the user-based collaborative-filtering technique [37].

The former exploits the relationships among items to predict the most similar items. The latter computes the ratings by taking into account the set of items rated by similar customers. There are two main types of collaborative-filtering recommendation: *user-based* [37] and *item-based* [33] techniques. As their names suggest, the user-based technique computes missing ratings by considering the ratings collected from similar users. Instead, the item-based technique performs the same task by using the similarities among items [8].

In the context of <RECOMMENDER ACRONYM>, the term *rating* is understood as the occurrence of a library in a project and computing missing ratings means to predict the inclusion of additional libraries. The project that needs prediction for library inclusion is called the *active project*. By the matrix in Fig. 4, p is the active project and an asterisk ($*$) represents a known rating, either 0 or 1, whereas a question mark (?) represents an unknown rating and needs to be predicted.

Given the availability of the cross-relationships as well as the possibility to compute similarities among projects using the graph representation, we exploit the user-based collaborative-filtering technique as the engine for recommendation [16, 37]. Given an active project p, the inclusion of libraries in p can be deduced from projects that are similar to p. The process is summarized as follows:

- Compute the similarities between the active project and all projects in the collection;
- Select *top-k* most similar projects; and
- Predict ratings by means of those collected from the most similar projects.

The rectangles in Fig. 4 imply that the row-wise relationships between the active project p and the similar projects q_1, q_2, q_3 are exploited to compute the missing ratings for p. The following formula is used to predict if p should include l, i.e., $p \ni l$ [25]:

$$r_{p,l} = \overline{r_p} + \frac{\sum_{q \in topsim(p)} (r_{q,l} - \overline{r_q}) \cdot sim(p,q)}{\sum_{q \in topsim(p)} sim(p,q)} \quad (3)$$

where $\overline{r_p}$ and $\overline{r_q}$ are the mean of the ratings of p and q, respectively; q belongs to the set of *top-k* most similar projects to p, denoted as $topsim(p)$; $sim(p,q)$ is the similarity between the active project and a similar project q, and it is computed using Equation 2.

4 EVALUATION

This section describes the planning of our evaluation, having the *goal* of evaluating the performance of <RECOMMENDER ACRONYM>. In Section 4.1, we introduce the dataset exploited in our evaluation. The evaluation methodology and metrics are presented in Section 4.2. Finally, Section 4.3 describes the research questions.

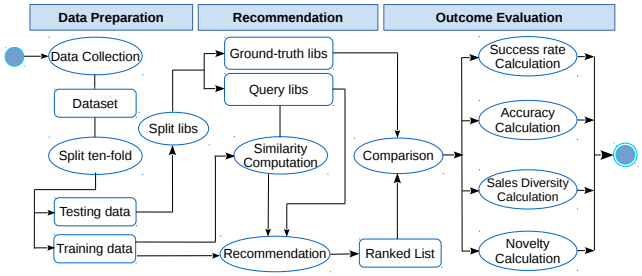


Figure 5: Evaluation Process.

The evaluation process is depicted in Fig. 5 and it consists of three consecutive phases, i.e., *Data Preparation*, *Recommendation*, and *Outcome Evaluation*. We start with the *Data Preparation* phase by creating a dataset from GitHub projects. The dataset is then split into training and testing sets. In the *Recommendation* phase, given a project in the testing set, a portion of its topics is extracted as ground-truth data, and the remaining is used as query to compute similarity and recommendations. Finally, by the *Outcome Evaluation* phase, we compare the recommendation results with those stored as ground-truth data to compute the quality metrics. All the aforementioned phases are detailed in the rest of this section.

4.1 Dataset Extraction

To evaluate the systems, we exploited a dataset with 6258 GitHub repositories that use 15757 topics.

By means of the GitHub API [2] we *randomly* collected a dataset consisting of 6,258 repositories. Claudio ▶*Add how the dataset has been mined (i.e., constraints in term of stars, forks, num of topic, etc)*◀ Juri ▶*Add Statistics*◀

Using a project's name as query, we can retrieve different types of information such as commits, issues, and name of the repository's owner,and the list of topics. By using the GitHub query language [**?**], we applied the following query filter:

$$Qf = "is:featured\ topic:t\ stars:100..80000\ topics:>=2" \quad (4)$$

to consider only GitHub repositories having a number of stars between 100 and 80,000, and tagged with at least two topics. We set such a filter after several query refinements, as the chosen featured topics have different degree of popularity. For instance, the most starred project of the *3d* topic has around 53,000 stars; reversely, some other topics have top rank projects that do not reach 1,000 stars. Thus, we tried to select the best query filter to include a sufficient number of repository for each topic. By imposing such a filter, we tried to avoid skewed repositories that may not have an informative README or projects that are already abandoned for a long time. Forking is another index related to the quality of the project. This feature is typically employed as a starting point for a new project [15]. Furthermore, there is a strong correlation between forks and stars [5]. In this sense, we suppose that a project with a high number of stars means that it gets attention from the OSS community and thus being suitable to identify popular repositories [**? ?**].

4.2 Evaluation Methodology and Metrics' Definition

To assess the performance of <RECOMMENDER ACRONYM> proposed approach, we applied ten-fold cross-validation, considering every time 9 folds (each one contains 625 projects) for training and the remaining one for testing. For every testing project p, a half of its topics are *randomly* taken out and saved as ground truth data, let us call them *GT(p)*, which will be used to validate the recommendation outcomes. The other half are used as testing libraries or query, which are called ***te***, and serve as input for `Similarity Computation` and `Recommendation`. The splitting mimics a real development process where a developer has already included some topics in the current project, i.e., ***te*** and waits for recommendations, that means additional topics to be incorporated. A recommender system is expected to provide her with the other half, i.e., *GT(p)*.

There are several metrics available to evaluate a ranked list of recommended items [25]. In the scope of this paper, *success rate*, *accuracy*, *sales diversity*, and *novelty* have been used to study the systems' performance as already proposed by Robillard et al. [30] and other studies [35],[24]. For a clear presentation of the metrics considered during the outcome evaluation, let us introduce the following notations:

- N is the cut-off value for the ranked list;
- k is the number of neighbor projects exploited for the recommendation process;
- For a testing project p, a half of its libraries are extracted and used as the ground-truth data named as *GT(p)*;
- $REC(p)$ is the *top-N* libraries recommended to p. It is a ranked list in descending order of real scores;
- If a recommended library $l \in REC(p)$ for a testing project p is found in the ground truth of p (i.e., *GT(p)*), hereafter we call this as a library *match* or *hit*.

If $REC_N(p)$ is the set of top-N items and $match_N(p) = GT(p) \bigcap REC_N(p)$ is the set of items in the *top-N* list that match with those in the ground-truth data, then the metrics are defined as follows.

Success rate@N. Given a set of testing projects P, this metric measures the rate at which a recommender system returns at least a topic match among *top-N* items for every project $p \in P$ [35]:

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} \tag{5}$$

where the function *count()* counts the number of times that the boolean expression specified in its parameter is *true*.

Accuracy. Accuracy is considered as one of the most preferred *quality indicators* for Information Retrieval applications [32]. However, *success rate@N* does not reflect how accurate the outcome of a recommender system is. For instance, given only one testing project, there is no difference between a system that returns 1 topic match out of 5 and another system that returns all 5 topic matches, since *success rate@5* is 100% for both cases (see Eq. (5)). Thus, given a list of *top-N* libraries, *precision@N* and *recall@N* are utilized to measure the *accuracy* of the recommendation results. *precision@N* is the ratio of the *top-N* recommended topics belonging to the ground-truth dataset, whereas *recall@N* is the ratio of the ground-truth topics appearing in the N recommended items [22],[10],[9]:

$$precision@N = \frac{|match_N(p)|}{N} \tag{6}$$

$$recall@N = \frac{|match_N(p)|}{|GT(p)|} \tag{7}$$

Sales Diversity. This term originates from the business domain where it is important to improve the coverage as also the distribution of products across customers, thereby increasing the chance that products will get sold by being introduced [36]. Similarly, in the context of topic recommendation, *sales diversity* indicates the ability of the system to suggest to projects as much topics as possible, as well as to disperse the concentration among all items, instead of focusing only on a specific set of topics [30]. In the scope of this paper, *catalog coverage* and *entropy* are utilized to gauge *sales diversity*. Let L be the set of all topic available for recommendation, $\#rec(l)$ denote the number of projects getting library l as a recommendation, i.e., $\#rec(l) = count_{p \in P}(|REC_N(p) \ni l|), l \in L$, and $total$ denote the number of recommended items across all projects.

Catalog coverage measures the percentage of topic being recommended to projects:

$$coverage@N = \frac{\left|\cup_{p \in P} REC_N(p)\right|}{|L|} \tag{8}$$

Entropy evaluates if the recommendations are concentrated on only a small set or spread across a wide range of topic:

$$entropy = -\sum_{l \in L}\left(\frac{\#rec(l)}{total}\right) ln\left(\frac{\#rec(l)}{total}\right) \tag{9}$$

Novelty. In business, the *long tail effect* is the fact that a few of the most popular products are extremely popular, while the rest, so-called the long tail, is obscure to customers [4]. Recommending products in the long tail is beneficial not only to customers but also to business owners [36]. Given that the logarithmic scale is used instead of the decimal one on the x-Axis of Fig. **??**, Fig. **??**, and Fig. **??**, the long tail effect can be spotted there as a few topic are very popular by being included in several projects, whereas most of the topics appear in fewer projects. Specifically, in `D1` just 15 topics are used by more than 200 projects, whereas 14, 876 topics are used by no more than 10 repositories. Juri ►*CHANGE VALUES*◄ In `D2` there are 876 libraries, accounting for 63.85% of the total number, being used by no more than 10 projects. Similarly, in `D3`, 88.24% of the dependencies corresponding to 49, 799 libraries are used by no more than 10 projects. When recommending a library, *novelty* measures if a system is able to pluck libraries from the "long tail" to expose them to projects. This might help developers come across *serendipitous* libraries [11], e.g., those that are seen by chance but turn to be useful for their current project. To quantify *novelty*, we utilize *expected popularity complement* (EPC) which is defined in the following equation [36]:

$$EPC@N = \frac{\sum_{p \in P}\sum_{r=1}^{N} \frac{rel(p,r)*[1-pop(REC_r(p))]}{log_2(r+1)}}{\sum_{p \in P}\sum_{r=1}^{N} \frac{rel(p,r)}{log_2(r+1)}} \tag{10}$$

where $rel(p, r) = |GT(p) \bigcap REC_r(p)|$ represents the relevance of the library at the r position of the *top-N* list to project p, $rel(p, r)$ can either be 0 or 1; $pop(REC_r(p))$ is the popularity of the library

at the position r in the *top-N* recommended list. It is computed as the ratio between the number of projects that receive $REC_r(p)$ as a recommendation and the number of projects that receive the most ever recommended library as a recommendation. Equation 10 implies that the more unpopular libraries a system recommends, the higher the EPC value it obtains and vice versa.

We use the Wilcoxon ranked sum test (considering one data point for each fold of the cross-validation) with a significance level of $\alpha = 5\%$, and the Cliff's delta (d) effect size measure [12]. Due to multiple tests being performed, we adjust p-values using the Holm's correction [14].

4.3 Research Questions

By performing the evaluation, we aim at addressing the following research questions:

- RQ1: How does the variation of training data impact on the prediction performance? To answer this question, we varied the dimension of the dataset to include more training data. In particular, we assess the quality of three different datasets to find out the best one in terms of success rate.
- RQ2: Is the approach able to provide consistent recommendations? We compute the metrics given in Section 4.2 to measure the relevance of our suggested topics considering the distribution of the considered repositories.

We study the experimental results in the next section by referring to these research questions.

5 RESULTS

This section reports and discusses the results of our study by addressing the research questions formulated in Section 4. The section is structured into the following subsections. Section 5.1 introduces an example of the recommendation results by LibRec and <RECOMMENDER ACRONYM>. In Section 5.2, we analyze the obtained results to study the systems' performance. Finally, Section 5.3 discusses the probable threats to validity of our findings.

5.1 Explanatory Example

Before addressing our research questions, we illustrate the recommendations of <RECOMMENDER ACRONYM> through a running example, i.e., the project *peakgames/libgdx-stagebuilder* hosted on GitHub. As shown in Table **??**, such a project uses 16 libraries, which are listed on the left-hand side of the table. Among 16 included libraries, 8 items are extracted and used as the ground-truth libraries that are shown in gray. The remaining 8 items are used as inputs for similarity computation and recommendation, or query. The output obtained by each system is a ranked list in descending order of recommendations with real scores. We took the first 10 libraries, removed the scores and kept only the order of the list to present the results as in Table **??**. The top-10 items are then matched against the ground-truth data. The column **Freq.** reports the frequency of occurrence of the recommended library, over the set of 1, 200 projects. In this case, LibRec only matches **junit:junit** with the ground-truth (which is obviously used by many projects for testing purposes) but, as we can notice, <RECOMMENDER ACRONYM> matches 4 more projects with the ground-truth.

a	a	a	a

Both LibRec and <RECOMMENDER ACRONYM> obtain a *success rate@10=1.0*. However, <RECOMMENDER ACRONYM> has a better *recall@10* compared to LibRec as it returns more relevant items (see Eq. (7)). Furthermore, among the matches by <RECOMMENDER ACRONYM>, 4 items appear in the top rows of the ranked list, indicating that <RECOMMENDER ACRONYM> recommends with a high *precision@N* (see Eq. (6)). LibRec returns only 1 relevant item, which means that both *precision@N* and *recall@N* are considerably lower compared to those of <RECOMMENDER ACRONYM>. Furthermore, LibRec tends to suggest very popular libraries: 6 out of 10 items recommended by LibRec are used by more than 200 projects. For instance, besides **junit:junit**, the second highest frequency item is **org.slf4j:slf4j-api** (473/1, 200). By performing an investigation on the outcome of all queries, we realized that LibRec usually recommends very popular items. The reasons for such differences are explained in Section 5.2.

Four out of five items recommended by <RECOMMENDER ACRONYM> have a low frequency of occurrence. For instance, the first item in the ranked list is **com.badlogicgames. gdx:gdx-platform** and this library is included in only 3/1, 200 projects. Referring to Fig. **??**, it is evident that the top 3 items belong to the long tail, i.e., they are extremely unpopular since each is used by only 3 projects. However, they turn out to be useful as all of them match those stored as ground-truth. In contrast to some existing studies which choose to recommend only popular items to developers [28],[20] we see that popularity is not a good indicator for selecting a library. This implies that the novelty of a ranked list is important: a system should be able to recommend libraries that are *novel* [7], i.e., those that have been rarely seen. In this sense, we expect that <RECOMMENDER ACRONYM> can produce good outcomes, not only in terms of success rate and accuracy, but also sales diversity and novelty.

In summary, for the explanatory example, <RECOMMENDER ACRONYM> obtains a comparable success rate, but better accuracy and novelty than LibRec. This also confirms that success rate is not sufficient for evaluating the recommendation outcomes. A good recommender system is the one that can maintain a trade-off by improving diversity, novelty but still retaining a good accuracy [29]. Consequently, it is necessary to investigate if this trade-off is guaranteed by LibRec and <RECOMMENDER ACRONYM>, and this is done in the next sub-sections by considering the whole dataset discussed in the previous section.

5.2 Empirical Study Results

In this section, we report the results by addressing the research questions **RQ**$_1$, **RQ**$_2$, **RQ**$_3$, **RQ**$_4$, and **RQ**$_5$.

RQ$_1$: *How does <RECOMMENDER ACRONYM> compare with the state-of-the-art approaches in terms of success rate?*

Comparison between LibRec and <RECOMMENDER ACRONYM>. We performed a series of experiments on `D1` using different combinations of number of recommended libraries (i.e., N), and number of neighbor projects exploited in the recommendation phase (i.e., k). Varying N means changing the length of the recommendation list,

whereas increasing k means considering more neighbor projects for recommendation.

Fig. 6(a) shows the *success rate@5* for k={5, 10, 15, 20, 25}. As it can be seen there, the success rate values obtained by <RECOMMENDER ACRONYM> are always superior to those of LibRec. The maximum *success rate@5* of LibRec is 0.876, whereas <RECOMMENDER ACRONYM> obtains success rates being greater than 0.903 for all configurations, with 0.931 being the maximum value. Fig. 6(b) shows the *success rate@10*, for this setting, LibRec gains a comparable performance for $N = 5$. Meanwhile, <RECOMMENDER ACRONYM> achieves a slight improvement in its performance compared to the case with $N = 5$. It is evident that <RECOMMENDER ACRONYM> outperforms LibRec in all test configurations. Fig. 6(a) and Fig. 6(b) imply that changing the number of neighbor projects k does not make a substantial difference in their match rate, as *success rate@N* is stable towards k for both systems.

Next, we investigate the success rate with regards to N. We consider a small number of recommended items, i.e., $N = \{1, 3, 5, 7, 10\}$. In practice, this means that the developer wants to see a short list of recommended libraries. In the first experiment, k is fixed to 10 and the outcomes are depicted in Fig. 6(c). For $N = 1$, LibRec achieves a success rate of 0.647 which is lower than the corresponding value 0.697 produced by <RECOMMENDER ACRONYM>. A success rate of 0.697 implies that <RECOMMENDER ACRONYM> can supply relevant recommendations to the developer at an encouraging match rate, even when she expects only an extremely brief list. Once k is changed from 10 to 20, both systems have a slight increase in *success rate@1* as depicted in Fig. 6(d). However, for other values of N, there are almost no changes in success rate. To further observe this behavior, we conducted more experiments with an increasing k, e.g., $k = \{50, 60, 100\}$. As far as we can see, there are no subtle differences between the conclusions obtained from the new experiments with those previously presented in the paper. Thus, for the sake of clarity, the outcomes of these experiments are omitted from the paper.

Table 1 reports Wilcoxon rank sum test adjusted p-values and Cliff's d, respectively for the comparison of LibRec and <RECOMMENDER ACRONYM> in terms of *success rate*, using $k = 10$ and $N = \{3, 5, 10, 15\}$; the labels in parentheses indicate the magnitude (n:negligible, s:small, l:large). As the table shows, the differences are always statistically significant and in favor of <RECOMMENDER ACRONYM> (effect size is positive), with a large effect size.

In summary, <RECOMMENDER ACRONYM> significantly outperforms LibRec in all considered test configurations concerning *success rate*, with a large effect size. The recommendation time for a fold (120 projects) is relatively faster for <RECOMMENDER ACRONYM> (3s) than for LibRec (20s).

Comparison between LibFinder and <RECOMMENDER ACRONYM

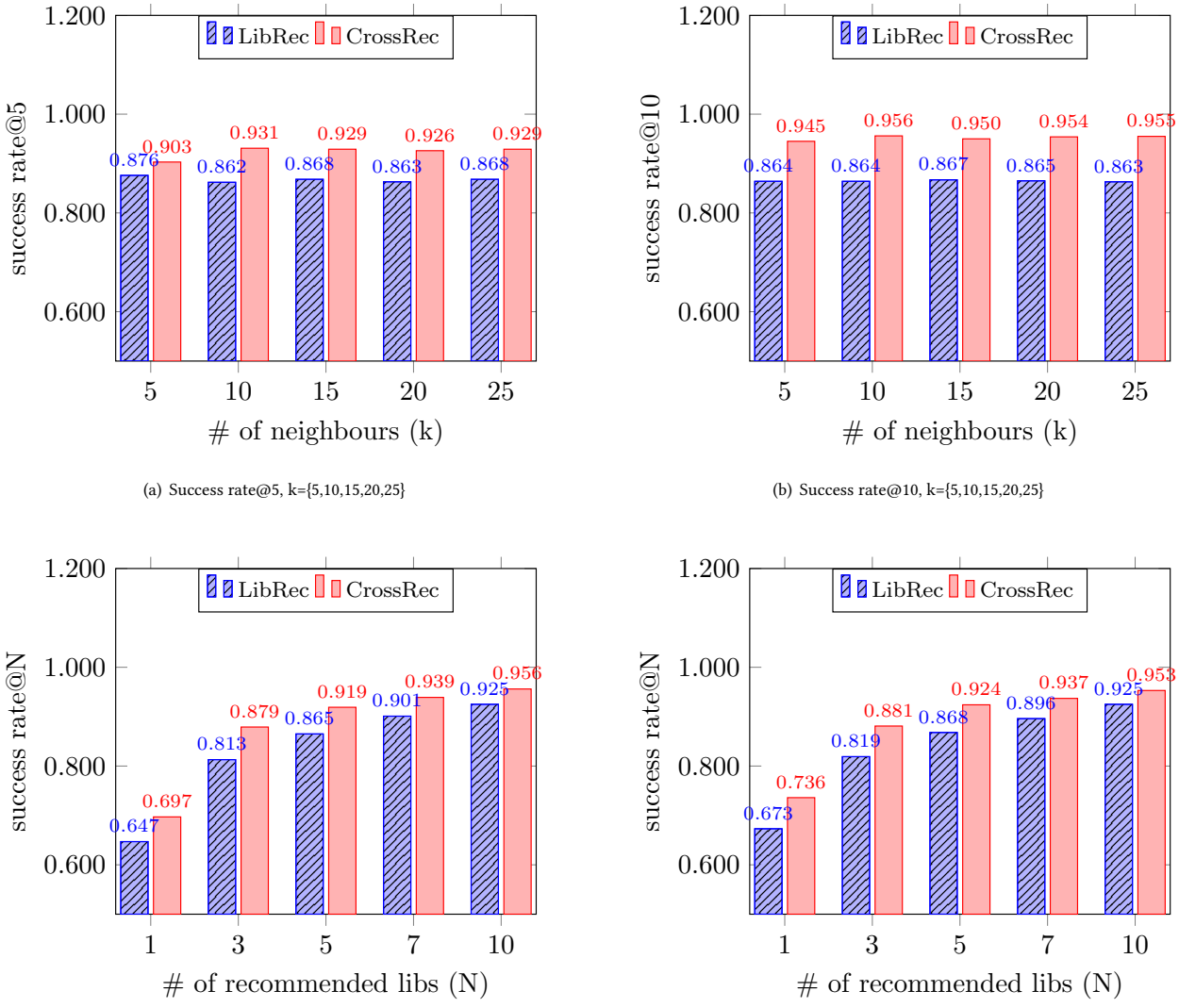
Even though LibFinder is a bit different from <RECOMMENDER ACRONYM> as it aims at providing *on-the-fly* recommendations while the developer is coding by exploiting the existing semantic, we attempted to compare <RECOMMENDER ACRONYM> with LibFinder by applying the same experimental settings. A comparison between LibFinder and <RECOMMENDER ACRONYM> is depicted in Table 2.[2] For small cut-off values, i.e., $N = \{1, 2, 4\}$, <RECOMMENDER ACRONYM> obtains a better success rate than LibFinder does. However, by the higher values $N = \{6, 8, 10\}$, LibFinder gains an improvement in its performance. This suggests that LibFinder tends to provide matches quite late in the ranked list. On one hand, a system with such a recommendation list helps developers approach relevant libraries. On the other hand, this is not useful for those who prefer to get only a short list of recommendations.

By carefully examining the `D2` dataset, we realized that there are several projects containing a small number of dependencies, e.g., 1 or 2 third-party libraries. Such projects are not beneficial to the training of <RECOMMENDER ACRONYM> since the corresponding metadata is not sufficient to compute similarity and, consequently, to provide a recommendation (see Eq. (1) and Eq. (2)). Thus, to further investigate the performance of <RECOMMENDER ACRONYM> on `D2`, we performed a filtering step as follows. Starting from the dataset, we selected projects containing a certain number of libraries, and such a number is called as L. The number was then varied to create different configurations. In practice, this means that we consider only mature projects, i.e., those that include a decent number of libraries for training. Table 3 shows the success rate obtained by varying L. As it can be seen, CrossRec's performance is proportional to L: the more dense (with respect to dependencies) the projects are, the better performance <RECOMMENDER ACRONYM> achieves. For instance, when we consider only projects with at least 4 libraries, i.e., $L = 4$ the success rate obtained for $N = 10$ is 0.914. However, if L is increased to 10, the corresponding success rate improves to reach 0.987. The same trend can be witnessed by other combinations of L and N.

While the `D2` dataset also specifies library versions, LibFinder is unable to deal with this information, resulting in a limitation, as already stated by the authors [26]. In practice, it is crucial to provide developers with not only a suitable library, but also a specific version of that library, otherwise there might be some issues related to version compatibility [18] or, in general, recommending obsolete libraries. Thus, we injected also library version into the training and testing data and fed it to <RECOMMENDER ACRONYM>. The final results for different configurations are reported in Table 4. As it can be seen in the table, given that decent training data is available, <RECOMMENDER ACRONYM> also recommends libraries with version, achieving a high success rate. For instance, even when projects containing at least 2 libraries are considered, or $L = 2$, <RECOMMENDER ACRONYM> obtains a success rate of 0.620 for $N = 1$. This quality metric improves linearly alongside L and N. As an example, when $L = 10$ and $N = 10$, the corresponding success rate is 0.955.

On the `D2` dataset, <RECOMMENDER ACRONYM> obtains a comparable performance compared to that of LibFinder. When it comes to a dataset containing projects with more libraries, <RECOMMENDER ACRONYM> is able to give more precise recommendations. Differently from LibFinder, <RECOMMENDER ACRONYM> is capable recommending also a specific version of a library.

[2] The success rates of LibFinder were extracted directly from the paper [26].



(a) Success rate@5, k={5,10,15,20,25}

(b) Success rate@10, k={5,10,15,20,25}

(c) Success rate@{1,3,5,7,10}, k=10

(d) Success rate@{1,3,5,7,10}, k=20

Figure 6: Success rate of <RECOMMENDER ACRONYM> and LibRec on D1.

Table 1: Wilcoxon rank sum test adjusted p-values and Cliff's d results for N={3,5,10,15}, k=10.

	Cut-off value (N)			
Test	3	5	7	10
Wilcoxon r.s.t. adjusted p-values	0.02	0.02	0.17	0.002
Cliff's d results	0.70 (l)	0.74 (l)	0.93 (l)	0.58 (l)

Comparison between LibCUP and <RECOMMENDER ACRONYM>. For this evaluation, we used the same experimental settings utilized to evaluate LibFinder. In particular, the following cut-off values have been considered: $N = \{1, 3, 5, 7, 10\}$. According to Table 5, the performance obtained by <RECOMMENDER ACRONYM> is always better than that of LibCUP.[3] For example, when the cut-off value $N = 1$, LibCUP gets 0.12 as success rate while the corresponding value by <RECOMMENDER ACRONYM> is 0.21. Similarly, for other values of N, our proposed tool gains a superior performance

[3]The success rate values for LibCUP were extracted from the paper [31].

Table 2: Success rate of LibFinder and <RECOMMENDER ACRONYM> on D2.

	Cut-off value (N)					
System	1	2	4	6	8	10
LibFinder	0.633	0.698	0.813	**0.876**	**0.904**	**0.918**
<RECOMMENDER ACRONYM>	**0.771**	**0.816**	**0.851**	0.860	0.863	0.864

Table 3: Success rate of <RECOMMENDER ACRONYM> on D2 with different number of libraries (L).

		Cut-off value (N)					
# of libraries		1	2	4	6	8	10
L	4	0.811	0.853	0.894	0.906	0.911	0.914
	6	0.890	0.926	0.948	0.958	0.961	0.964
	8	0.930	0.958	0.972	0.977	0.982	0.986
	10	0.938	0.964	0.976	0.981	0.984	0.987

Table 4: Success rate of <RECOMMENDER ACRONYM> on D2 when library version is considered.

		Cut-off value (N)					
# of libraries		1	2	4	6	8	10
L	2	0.620	0.676	0.715	0.729	0.735	0.738
	4	0.785	0.834	0.872	0.893	0.903	0.907
	6	0.848	0.885	0.907	0.921	0.931	0.936
	8	0.888	0.919	0.931	0.937	0.945	0.950
	10	0.906	0.930	0.944	0.948	0.951	0.955

compared to that of LibCUP, in particular when $N = 10$, <RECOMMENDER ACRONYM> achieves a success rate which is nearly twice what LibCUP achieves, i.e., 0.42 compared to 0.22.

Similarly to D2, D3 also features library versions, however LibCUP cannot provide version-specific recommendations. For each library in the dataset, we integrated its version and fed as input for <RECOMMENDER ACRONYM>. For this experiment, we selected only projects with a considerably high number of third-party libraries, i.e., $L = \{7, 8, 9, 10\}$ and the final results are shown in Table 6.

Table 6 shows that <RECOMMENDER ACRONYM> also recommends specific versions of libraries. Among the configurations, the maximum success rate is 0.494, and it is reached when $L = 10$ and $N = 10$. However, in comparison to the outcome by D2, <RECOMMENDER ACRONYM> achieves a considerably lower success rate. Moreover, there is a marginal difference in performance between different values of L. Especially, the system's performance suffers a setback when $L = 9$ in comparison to $L = 8$. By carefully investigating the dataset, we found out that the projects of D2 are very diverse, and they contain a large number of libraries. Moreover, the similarity among the projects is considerably low. This means that the dataset exhibits a high level of heterogeneity and therefore, given a project, generally we cannot find a good set of similar projects. Under the circumstances, it is more difficult to recommend highly relevant libraries, resulting in a low success rate.

On the D3 dataset, <RECOMMENDER ACRONYM> clearly outperforms LibCUP with respect to various configurations. Furthermore, <RECOMMENDER ACRONYM> can recommend also libraries with a specific version, which LibCUP cannot.

RQ$_2$: *How well can LibRec and <RECOMMENDER ACRONYM> recommend third-party libraries with respect to accuracy, sales diversity, and novelty?*

Accuracy. To represent accuracy, we vary N from 1 to 30 to get *precision@N* and *recall@N*. The rationale behind the selection of 30 as the maximum cut-off value is that LibRec normally produces a short list of recommendations, ranging from 32 to 50 items. From the accuracy scores computed using Eq. (6) and Eq. (7), the Precision-Recall curves (PRCs) for all 10 rounds of validation and different values of k were sketched. However, we noticed that the figures representing the folds share a very similar pattern. Thus, for the sake of clarity, we only show in Fig. 7 results of the most representative fold. Since a PRC close to the upper right corner represents a better accuracy [10], we see that by LibRec, changing the number of neighbor k almost makes no difference in its accuracy. Meanwhile for <RECOMMENDER ACRONYM> we can notice that an increase of k brings a slightly better accuracy for some testing folds, however the gain is negligible. For all pieces of testing data, <RECOMMENDER ACRONYM> always produces a superior accuracy compared to that of LibRec.

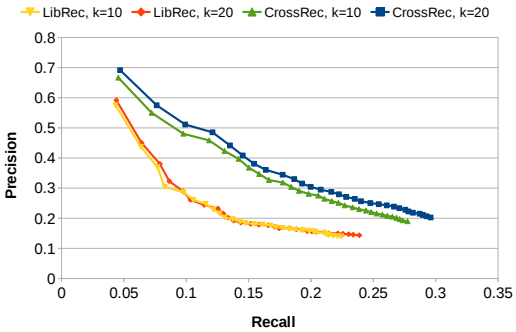


Figure 7: Precision and Recall.

Sales diversity. The catalog coverage scores for LibRec and <RECOMMENDER ACRONYM> are depicted in Table 7. The maximum coverage values are 4.594 and 5.897 for LibRec and <RECOMMENDER ACRONYM>, respectively. According to Eq. (8), a higher

Table 5: Success rate of LibCUP and <RECOMMENDER ACRONYM> on D3.

	Cut-off value (N)				
System	1	3	5	7	10
LibCUP	0.12	0.14	0.15	0.17	0.22
<RECOMMENDER ACRONYM>	**0.21**	**0.31**	**0.36**	**0.39**	**0.42**

Table 6: Success rate of <RECOMMENDER ACRONYM> on D3 when library version is incorporated.

		Cut-off value (N)				
# of libraries		1	3	5	7	10
L	7	0.215	0.313	0.361	0.392	0.423
	8	0.219	0.315	0.362	0.392	0.423
	9	0.214	0.311	0.355	0.383	0.415
	10	0.255	0.368	0.422	0.456	0.494

score means a better coverage. In this sense, the recommendations generated by <RECOMMENDER ACRONYM> cover a wider spectrum of libraries than those by LibRec for both configurations, i.e., $k = 10$ and $k = 20$, using different cut-off values N. Table 8 shows the *entropy* for LibRec and <RECOMMENDER ACRONYM>. Equation (9) suggests that a low entropy value represents a better distribution of items, therefore the recommendations by <RECOMMENDER ACRONYM> have a much better distribution than those obtained by LibRec. For example, for the case $N = 25$ and $k = 20$, <RECOMMENDER ACRONYM> has an entropy of 0.635, which is much better than 2.751, the corresponding value by LibRec.

Table 7: Catalog coverage for N={5,10,15,20,25}, k={10,20}.

	k=10		k=20	
N	LibRec	<RECOMMENDER ACRONYM>	LibRec	<RECOMMENDER ACRONYM>
5	0.857	**1.099**	0.691	**0.814**
10	1.760	**2.157**	1.346	**1.534**
15	2.675	**3.278**	1.937	**2.312**
20	3.577	**4.541**	2.512	**3.143**
25	4.594	**5.897**	3.139	**4.005**

Table 8: Entropy for N={5,10,15,20,25}, k={10,20}.

	k=10		k=20	
N	LibRec	<RECOMMENDER ACRONYM>	LibRec	<RECOMMENDER ACRONYM>
5	0.869	**0.239**	0.552	**0.127**
10	1.752	**0.481**	1.098	**0.254**
15	2.653	**0.723**	1.639	**0.381**
20	3.566	**0.968**	2.193	**0.508**
25	4.500	**1.217**	2.751	**0.635**

Novelty. The *EPC@N* scores for LibRec and <RECOMMENDER ACRONYM> are shown in Table 9. With LibRec, changing k from 10 to 20 decreases *novelty* for all cut-off values. For example, the *novelty* with $N = 25$ and $k = 10$ is 0.349, however once k is changed to 20, it drops to 0.261. For <RECOMMENDER ACRONYM>, changing the number of neighbours k from 10 to 20 does not bring a rise in *novelty*. As shown in Table 9, <RECOMMENDER ACRONYM> always obtains scores that are higher than those of LibRec. For example, when $N = 5$ and $k = 20$, <RECOMMENDER ACRONYM> achieves a value of 0.292 for *novelty*, whereas LibRec gets 0.114.

Table 9: EPC for N={5,10,15,20,25}, k={10,20}.

	k=10		k=20	
N	LibRec	<RECOMMENDER ACRONYM>	LibRec	<RECOMMENDER ACRONYM>
5	0.187	**0.291**	0.114	**0.292**
10	0.264	**0.349**	0.166	**0.344**
15	0.296	**0.376**	0.204	**0.377**
20	0.320	**0.391**	0.236	**0.399**
25	0.349	**0.401**	0.261	**0.416**

This, together with the example in Section 5.1, confirms that <RECOMMENDER ACRONYM> recommends libraries that are closer to the long tail than LibRec can do.

Also in this case (see columns 2-6 of Tables 10 and 11), our analyses are supported by statistical procedures. Differences are always statistically significant. The effect size is negligible/small for *accuracy* (precision and recall), whereas it is large for all other indicators *sales diversity* and *novelty*.

> The results in Fig. 7 and Tables 7, 8, 9 demonstrate that <RECOMMENDER ACRONYM> significantly outperforms LibRec concerning *accuracy*, *sales diversity*, and *novelty*, with a small/negligible effect size for *accuracy* and large elsewhere.

RQ$_3$: *What are the reasons for the performance difference between LibRec and <RECOMMENDER ACRONYM>?*

We attempt to ascertain why <RECOMMENDER ACRONYM> outperforms LibRec. This task might necessitate further investigations, both qualitative and quantitative research. However, by carefully studying the internal design of LibRec, we found out that the improvement attributes to the following facts. In the first place, <RECOMMENDER ACRONYM> employs a completely different approach to represent projects and libraries: it encodes the relationships among them into a graph. Second, to compute the similarity between two projects, <RECOMMENDER ACRONYM> assigns a weight to every library node using tf-idf (see Eq. (1)). In this way, the level of importance of a node is disproportional to its popularity. This is similar to the context of document matching where popular terms are given a low weight [13]. For instance, in Fig. 3, lib_1 is a popular node since it is referred by 4 projects and this makes it have a low weight. As a result, <RECOMMENDER ACRONYM> is able to better capture the similarity between two projects compared to LibRec, which equally treats all libraries. Third, LibRec employs a very

Table 10: Wilcoxon rank sum test adjusted p-values for N={3,5,10,15}, k=10.

	Accuracy		Sales Diversity		Novelty
N	Precision	Recall	Coverage	Entropy	EPC
3	1.58e-16	5.41e-08	0.0005	6.50e-05	4.33e-05
5	1.35e-24	9.10e-12	0.001	6.50e-05	4.33e-05
10	1.07e-21	9.12e-12	0.003	4.33e-05	1.52e-04
15	1.10e-14	–	0.003	6.50e-05	2.06e-04

Table 11: Cliff's d results for N={3,5,10,15}, k=10. Labels in parenthesis indicate the magnitude (n:negligible, s:small, l:large).

	Accuracy		Sales Diversity		Novelty
N	Precision	Recall	Coverage	Entropy	EPC
3	0.18 (s)	0.12 (n)	0.92 (l)	0.96 (l)	1.00 (l)
5	0.23 (s)	0.16 (s)	0.86 (l)	0.98 (l)	1.00 (l)
10	0.22 (s)	0.16 (s)	0.80 (l)	1.00 (l)	0.94 (l)
15	0.17 (s)	–	0.80 (l)	0.98 (l)	0.90 (l)

simple collaborative-filtering technique, though it also considers a set of k-nearest neighbor similar projects for finding libraries, it neglects their similarity level by considering all projects in the same way. Also, the technique assigns more weight to popular libraries without considering the degree of similarity between projects, from where the libraries come. This explains why LibRec recommends very popular items as shown in Section 5.1. In contrast, <RECOMMENDER ACRONYM> improves by assigning a larger weight to libraries that come from highly similar projects (see Eq. (3)). In other words, given a project, <RECOMMENDER ACRONYM> is able to "mimic" the behavior of highly similar projects, it attempts to suggest a comparable set of libraries. Lastly, LibRec exploits association rule mining which indeed mines items that co-exist. This is why the coverage of the recommended items is low compared to what achieved by <RECOMMENDER ACRONYM>.

Our qualitative analysis suggests that the improvements achieved by <RECOMMENDER ACRONYM> with respect to LibRec are due to the weighting scheme being applied, which also considers the projects' similarity, i.e., it rewards recommendation of libraries from similar projects.

RQ$_4$: *Does an increase in the amount of input data help improve CrossRec's overall performance?*

Finally, we studied whether an increase in the input data contributes to an improvement in the performance of <RECOMMENDER ACRONYM>. As it was already mentioned in Section 4.3, r is the ratio of libraries used as query over the total number of libraries that a project contains. For this evaluation, rather than using $r = 50\%$, we varied it along the following values: 20%, 40%, 60%, and 80%, and measured the achieved level of performance. This simulates different levels of a project's maturity: at the beginning when the developer has just included a small number of libraries, or later on when more libraries have been accumulatively populated alongside the project's lifecycle. This research question aims at studying if <RECOMMENDER ACRONYM> can assist the developer in choosing the right libraries at different stages of the development process.

Fig. 8 reports the precision and recall curves (PRCs) obtained by the configurations. As a better accuracy is represented by a PRC close to the upper right corner [10],[9], we see that generally there is an improvement in the recommendation performance when r increases. In particular, there is a sharp growth in precision and recall when r is changed from 20% to 40%. However, the gain tends to stagnate when r becomes larger. For instance, the change in performance between $r = 40\%$ and $r = 60\%$ is small and can be considered as negligible. Similarly, when r goes up to 80% from 60%, there is also a marginal change in accuracy. This corresponds to a saturation: at a certain threshold of r, e.g., $r = 50\%$ or $r = 60\%$, a little or almost no performance gain can be obtained. In essence, that means <RECOMMENDER ACRONYM> can achieve a decent level of accuracy once the developer has included around a half of the total number of libraries.

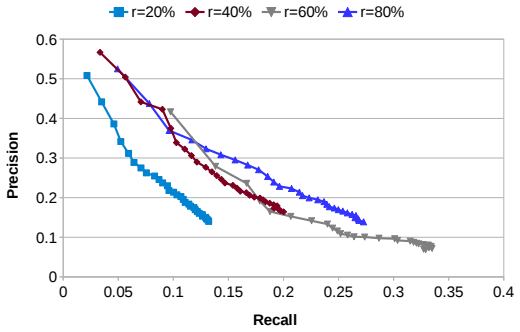


Figure 8: Precision and recall for different amounts of testing and ground-truth data.

For <RECOMMENDER ACRONYM>, an increase in the amount of input data considerably improves the overall performance when the ratio of query to testing data r is small. However, such the gain is disproportionate to r.

5.3 Threats to Validity

We identify the threats that may adversely affect the validity of the experiments as well as the countermeasures taken to mitigate them.

Threats to internal validity are related to any factors internal to our study that can influence our results. The performances achieved by <RECOMMENDER ACRONYM> can depend on the values of

N and k. We showed in the paper results for $k = 10, 20$, and for $N = 5, 10, 15, 20, 25$ (see more in Section 5.4). Results for other values of N and k are consistent with what we already found.

The comparison with LibFinder and LibCUP has been done in an indirect manner through their corresponding datasets, and this might possibly pose a threat to internal validity. We attempted to mitigate such a threat by exploiting the same experimental settings as well as performing various trials on the same datasets. All the additional experiments yielded comparable outcomes to those presented in the paper.

Threats to external validity concern the generalizability of our findings. In the data collection phase, we tried to cover a wide range of possibilities by mitigating also the fact that many repositories in GitHub are of low quality, which is especially true when they do not have many stars. The set of 1, 200 GitHub projects was randomly created by obtaining the following distribution of stars: 14 projects have 0 stars, 135 projects have [1-4] stars, 66 projects have [5-9] stars, 512 projects have [10-99] stars, 300 projects have [100-499] stars, 78 projects have [500-999] stars, and 95 projects have more than 1,000 stars. Moreover, the number of libraries that a project in the considered dataset includes varies considerably from 10 to more than 500.

Threats to construct validity are whether the setup and measurement in the study reflect real-world situations. The threats have been mitigated by applying ten-fold cross validation, attempting to simulate a real scenario of recommending third-party libraries. In the experiments, the dataset is split into two independent parts, namely a *training set* and a *testing set*. In practice, the items in the training data correspond to the OSS projects collected a priori. They are available at developers' disposal, ready to be exploited for any mining purpose. Whereas, an item in the testing data corresponds to the project being developed. In this sense, our evaluation attempts to mimic a real deployment: the recommender systems should produce recommendations for a project based on the data available from a set of existing projects.

5.4 Discussions

<RECOMMENDER ACRONYM> is capable of recommending a library (either just the library, or also a specific version of that library), depending on the availability of the training data. That is, the <RECOMMENDER ACRONYM> approach transcends the limitation of the baselines considered in this paper, i.e., LibRec, LibFinder and LibCUP. Such approaches cannot recommend a specific version of a library. Moreover, <RECOMMENDER ACRONYM> obtains a better performance when the input data is more dense, i.e., more libraries are available for training.

By performing experiments with LibRec and <RECOMMENDER ACRONYM> on the same dataset, and by applying the same experimental settings, we were able to compare their performance in a thorough manner. We have seen that an increase in the number of neighbor projects considered for recommendation from $k = 10$ to $k = 20$ does not make a big distinction in accuracy for both systems. Furthermore, as there are no changes in success rates by increasing k, we can conclude that almost all relevant libraries are in the topmost similar projects. This is further enforced by the fact that the *entropy* improves for both systems when k is increased from 10 to 20. The inclusion of more projects brings various libraries, and this helps to increase the item distribution. With LibRec, the fact that the *novelty* decreases when k increases shows that the additional projects bring only popular libraries. At the same time, the *novelty* does not change with <RECOMMENDER ACRONYM> using the same setting with k. This indicates that the recommended libraries brought by the additional projects do not help improve the overall *novelty*.

In this sense, the ability to compute similarities among projects plays an important role in obtaining a good recommendation performance. In addition, since considering more neighbors means adding more rows to the user-item ratings matrix, which indeed increases the computational complexity, we anticipate that utilizing an appropriate value of k can help speed up the computation, thus increasing the overall efficiency, but still preserving an acceptable effectiveness.

In contrast to LibRec, <RECOMMENDER ACRONYM> is able to maintain a trade-off between *accuracy* and *sales diversity*, it gains better precisions and recalls for all testing folds. Furthermore, <RECOMMENDER ACRONYM> also achieves an adequate catalog coverage and novelty by recommending more unpopular libraries to projects. Although in this paper we performed an evaluation only on Java projects, it is also possible to apply <RECOMMENDER ACRONYM> to search for third-party libraries in other languages, e.g., C++, Python, etc., as long as the relationship between projects and libraries can be represented following the model proposed in Section 3.1 and Section 3.2.

6 RELATED WORK

rw

7 CONCLUSIONS AND FUTURE WORK

conclusion

REFERENCES

[1] [n.d.]. GitHub. https://www.github.com. last accessed 16.06.2019.

[2] [n.d.]. GitHub REST API v3. https://developer.github.com/v3/. last accessed 16.06.2019.

[3] Charu Aggarwal. 2016. *Neighborhood-Based Collaborative Filtering*. Springer International Publishing, Cham, 29–70. https://doi.org/10.1007/978-3-319-29659-3_2

[4] Chris Anderson. 2006. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion.

[5] Hudson Borges, André C. Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 334–344. https://doi.org/10.1109/ICSME.2016.31

[6] Cristian E. Briguez, Maximiliano C.D. Budán, Cristhian A.D. Deagustini, Ana G. Maguitman, Marcela Capobianco, and Guillermo R. Simari. 2014. Argument-based mixed recommenders and their application to movie suggestion. *Expert Systems with Applications* 41, 14 (2014), 6467 – 6482. https://doi.org/10.1016/j.eswa.2014.03.046

[7] P. Castells, S. Vargas, and J. Wang. 2011. Novelty and Diversity Metrics for Recommender Systems: Choice, Discovery and Relevance. In *International Workshop on Diversity in Document Retrieval (DDR 2011) at the 33rd European Conference on Information Retrieval (ECIR 2011)*. Dublin, Ireland. http://ir.ii.uam.es/rim3/publications/ddr11.pdf

[8] Paolo Cremonesi, Roberto Turrin, Eugenio Lentini, and Matteo Matteucci. 2008. An Evaluation Methodology for Collaborative Recommender Systems. In *Proceedings of the 2008 International Conference on Automated Solutions for Cross Media Content and Multi-channel Distribution (AXMEDIS '08)*. IEEE Computer Society, Washington, DC, USA, 224–231. https://doi.org/10.1109/AXMEDIS.2008.13

[9] Jesse Davis and Mark Goadrich. 2006. The Relationship Between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA) *(ICML '06)*. ACM, New York, NY, USA, 233–240. https://doi.org/10.1145/1143844.1143874

[10] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. 2012. Linked Open Data to Support Content-based Recommender Systems. In *Proceedings of the 8th International Conference on Semantic Systems* (Graz, Austria) *(I-SEMANTICS '12)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2362499.2362501

[11] Mouzhi Ge, Carla Delgado-Battenfeld, and Dietmar Jannach. 2010. Beyond Accuracy: Evaluating Recommender Systems by Coverage and Serendipity. In *Proceedings of the Fourth ACM Conference on Recommender Systems* (Barcelona, Spain) *(RecSys '10)*. ACM, New York, NY, USA, 257–260. https://doi.org/10.1145/1864708.1864761

[12] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.

[13] Angelos Hliaoutakis, Giannis Varelas, Epimenidis Voutsakis, Euripides G. M. Petrakis, and Evangelos E. Milios. 2006. Information Retrieval by Semantic Similarity. *Int. J. Semantic Web Inf. Syst.* 2, 3 (2006), 55–73. https://doi.org/10.4018/jswis.2006070104

[14] S. Holm. 1979. A Simple Sequentially Rejective Bonferroni Test Procedure. *Scandinavian Journal on Statistics* 6 (1979), 65–70.

[15] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and How Developers Fork What from Whom in GitHub. *Empirical Software Engineering* 22, 1 (Feb. 2017), 547–578. https://doi.org/10.1007/s10664-016-9436-6

[16] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing* 7, 1 (Jan. 2003), 76–80. https://doi.org/10.1109/MIC.2003.1167344

[17] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting Similar Software Applications. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 364–374. http://dl.acm.org/citation.cfm?id=2337223.2337267

[18] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. Mining Trends of Library Usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops* (Amsterdam, The Netherlands) *(IWPSE-Evol '09)*. ACM, New York, NY, USA, 57–62. https://doi.org/10.1145/1595808.1595821

[19] Catarina Miranda and Alípio M. Jorge. 2008. Incremental Collaborative Filtering for Binary Ratings. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01 (WI-IAT '08)*. IEEE Computer Society, Washington, DC, USA, 389–392. https://doi.org/10.1109/WIIAT.2008.263

[20] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 880–890. http://dl.acm.org/citation.cfm?id=2818754.2818860

[21] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. [n.d.]. CrossRec: Supporting Software Developers by Recommending Third-party Libraries - Online Appendix. https://github.com/crossminer/CrossRec. last accessed 25.09.2019.

[22] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1050–1060. https://doi.org/10.1109/ICSE.2019.00109

[23] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio. 2018. CrossSim: Exploiting Mutual Relationships to Detect Similar OSS Projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 388–395. https://doi.org/10.1109/SEAA.2018.00069

[24] Phuong T. Nguyen, Paolo Tomeo, Tommaso Di Noia, and Eugenio Di Sciascio. 2015. An Evaluation of SimRank and Personalized PageRank to Build a Recommender System for the Web of Data. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15 Companion)*. ACM, New York, NY, USA, 1477–1482. https://doi.org/10.1145/2740908.2742141

[25] Tommaso Di Noia and Vito Claudio Ostuni. 2015. Recommender Systems and Linked Open Data. In *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*. 88–113. https://doi.org/10.1007/978-3-319-21768-0_4

[26] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. 2017. Search-based Software Library Recommendation Using Multi-objective Optimization. *Inf. Softw. Technol.* 83, C (March 2017), 55–75. https://doi.org/10.1016/j.infsof.2016.11.007

[27] Michael J. Pazzani and Daniel Billsus. 2007. *Content-Based Recommendation Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/978-3-540-72079-9_10

[28] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, 102–111. https://doi.org/10.1145/2597073.2597077

[29] Azzurra Ragone, Paolo Tomeo, Corrado Magarelli, Tommaso Di Noia, Matteo Palmonari, Andrea Maurino, and Eugenio Di Sciascio. 2017. Schema-summarization in Linked-data-based Feature Selection for Recommender Systems. In *Proceedings of the Symposium on Applied Computing* (Marrakech, Morocco) *(SAC '17)*. ACM, New York, NY, USA, 330–335. https://doi.org/10.1145/3019612.3019837

[30] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Softw.* 27, 4 (July 2010), 80–86. https://doi.org/10.1109/MS.2009.161

[31] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software* 145 (2018), 164 – 179. https://doi.org/10.1016/j.jss.2018.08.032

[32] Tefko Saracevic. 1995. Evaluation of Evaluation in Information Retrieval. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Seattle, Washington, USA) *(SIGIR '95)*. ACM, New York, NY, USA, 138–146. https://doi.org/10.1145/215206.215351

[33] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the 10th International Conference on World Wide Web* (Hong Kong, Hong Kong) *(WWW '01)*. ACM, New York, NY, USA, 285–295. https://doi.org/10.1145/371920.372071

[34] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. The Adaptive Web. Springer-Verlag, Berlin, Heidelberg, Chapter Collaborative Filtering Recommender Systems, 291–324. http://dl.acm.org/citation.cfm?id=1768197.1768208

[35] Ferdian Thung, David Lo, and Julia Lawall. 2013. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 182–191. https://doi.org/10.1109/WCRE.2013.6671293

[36] Saúl Vargas and Pablo Castells. 2014. Improving sales diversity by recommending users to items. In *Eighth ACM Conference on Recommender Systems, RecSys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014*. 145–152. https://doi.org/10.1145/2645710.2645744

[37] Zhi-Dan Zhao and Ming-sheng Shang. 2010. User-Based Collaborative-Filtering Recommendation Algorithms on Hadoop. In *Proceedings of the 2010 Third International Conference on Knowledge Discovery and Data Mining (WKDD '10)*. IEEE Computer Society, Washington, DC, USA, 478–481. https://doi.org/10.1109/WKDD.2010.54