CIGAR: Contrastive Learning for GitHub Action Recommendation

Jiangnan Huang
Radboud University
Nijmegen, The Netherlands

Bin Lin
Radboud University
Nijmegen, The Netherlands

***Abstract*—GitHub Actions was introduced in 2019 as an integrated solution for CI/CD to automate software development workflow. Since then, it has gained tremendous popularity among developers. In a GitHub Actions workflow, actions refer to custom applications for performing complex but frequently repeated tasks. Actions can be typically found in GitHub Marketplace or public GitHub repositories. Prior studies have already disclosed that developers often reuse actions to reduce double work and improve productivity. However, it is not trivial for developers, especially novices, to figure out which action to reuse due to the large number of actions available and the limited search functionality GitHub Marketplace provides. To address this issue, we propose CIGAR (ContrastIve learning for GitHub Action Recommendation). Given the textual description of a task developers want to execute, CIGAR will recommend the most relevant actions. CIGAR exploits a pre-trained RoBERTa model to convert sequences of words into high-dimensional vector representations, and is fine tuned through a contrastive learning objective. The performance of CIGAR was evaluated on a novel dataset curated based on prior research, and the results demonstrate that CIGAR can reliably recommend actions needed by developers and significantly outperforms the GitHub Marketplace search engine. Our study indicates the promise of employing contrastive learning for GitHub action recommendation. The promising performance achieved can potentially drive a wider adoption of GitHub Actions and facilitate the automation of software development workflows.**

***Index Terms*—GitHub Actions, CI/CD, Development Workflows, Contrastive Learning, Recommender System**

I. Introduction

GitHub [1] is the largest social coding platforms, hosting over 330 million software repositories and accommodating over 100 million developers in June 2023. By incorporating features like issue tracking and pull requests into distributed version control tools, GitHub has revolutionized the way developers collaborate [2], [3]. While the adoption of a pull-based development workflow presents many opportunities and benefits, it also raises high requirements for repository maintainers to coordinate the activities during the development process such as internal communication, code review, testing, and merging pull requests [4]. To facilitate the development and collaboration, continuous integration and delivery (CI/CD) tools have been widely adopted to automate a wide range of activities including building, testing, quality assurance [5].

In 2019, GitHub publicly released GitHub Actions[1] (GHA, hereafter) as its own solution for CI/CD. GHA is fully integrated into GitHub, and since its release, lots of repositories have started adopting it or migrated their old CI/CD services to GHA [6]. Like other CI/CD tools, developers can define automated tasks in a GHA workflow. What makes GHA workflow special is the adoption of "actions", which are custom applications performing complex but frequently repeated tasks. These actions, commonly distributed on the GitHub Marketplace and/or in public repositories, allow developers to easily integrate specific tasks (*e.g.,* set up environment). By reusing actions, developers can avoid writing repeated code in the workflow and improve their productivity.

However, it is not always easy for developers to figure out which actions to reuse. GitHub Marketplace[2], as the official portal where developers can search for actions, currently provides over 19k actions. The number is only increasing as third-party developers can list their own actions on the marketplace. It is virtually impossible for a developer to know all the existing actions. While GitHub Marketplace provides a search functionality, its usability leaves much to be desired. Navigating through the searching bar to discover appropriate actions can be a frustrating experience as the search engine often lacks precision and yield irrelevant or incomplete results, making it challenging to find the desired actions. Moreover, there are also many actions hosted on GitHub but not listed on Marketplace. These usability issues hinder the overall user experience and undermine the full potential of reusable actions.

To address the above issues, we propose a new approach named CIGAR (**C**ontrast**i**ve learning for **G**itHub **A**ction **R**ecommendation). CIGAR takes as input the textual description of the tasks developers want to execute in the workflow, and outputs the most relevant actions which can be reused. CIGAR exploits the pre-trained model RoBERTa [7] to get representations of task description, and uses contrastive learning in a supervised way [8] to "pull" together the representations of task description (*i.e.,* the anchor) and matching actions and "push" apart the anchor from non-matching actions in high-dimensional vector space.

To evaluate the effectiveness of our proposed approach, we crafted our own dataset based on the dataset created by Decan *et al.* [6]. Experimental results indicate that CIGAR significantly outperforms the search functionality provided by GitHub Marketplace.

[1]https://github.com/features/actions

[2]https://github.com/marketplace

Compared to GitHub Marketplace, our approach improves the success rate@1 (*i.e.,* the first recommended action is exactly the one adopted by developers) from 0.02 to 0.724 (a 3520% increase). Additionally, when comparing CIGAR with RoBERTa without contrastive learning, our approach increases the success rate@1 and @5 by 129.1% and 50.6%, respectively. We also visualized the vector representations learned from CIGAR and its variants to give readers a more intuitive idea on the effectiveness of each component of our approach.

To the best of our knowledge, our study is the first to recommend GitHub actions. The good performance indicates the possibility of using contrastive learning to aid CI/CD workflow composition and facilitate development process.

II. Background

A. GitHub Actions

In order to use GHA in a GitHub repository, developers need to specify the tasks they want to execute in the GHA workflow using YAML syntax. All the workflow files should have either `.yml` or `.yaml` extensions, and be placed within the designated "`.github/workflows`" directory. An example of such YAML files can be found in Listing 1, which defines a GHA workflow named `Maven test`[3].

```yaml
name: Maven test
on:
  pull_request:
    types: [ opened, reopened, edited ]
  push:
    branches: [ develop, master ]
jobs:
  mvn_verify:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Maven Central Repository
        uses: actions/setup-java@v2
        with:
          java-version: '8'
          distribution: 'adopt'
      - name: Run the Maven verify phase
        run: mvn --batch-mode -P dev test
```

Listing 1: Example of a GHA workflow file.

Each GHA workflow consists of a set of *jobs* that are triggered by a collection of *events* (*e.g.,* push, pull request in Listing 1). Each job comprises a series of *steps*, which are the smallest units of work in a workflow. Steps can be defined by specifying the commands that should be executed using the `run` key (*e.g.,* the last line of Listing 1) or by delegating tasks to "actions" defined in the same repository, a public repository, or a published Docker container image through the `uses` key. In this study, we mainly focus on actions from public repositories (referred to as "action repositories"). In Listing 1, two actions are used: "`actions/checkout`" checks out the repository so that the workflow can have the access, and "`actions/setup-java`" sets up the required Java environment so that the workflow can work with Java projects. Developers can optionally specify the name of steps. The step names are usually used to describe the task developers want to execute. For example, the second step in Listing 1 is named "Setup Maven Central Repository".

[3]The example is adapted from a randomly picked GHA workflow, available on https://github.com/abel533/Mapper/blob/master/.github/workflows/test.yml

B. Deep learning & Transfer learning

Deep learning (DL) is a rising field in machine learning that utilizes neural networks to mimic human decision-making [9]. Unlike traditional models such as Decision Trees or SVM, DL models automatically learn task-specific features from data, reducing the need for complex feature engineering. However, a key challenge is the reliance on massive training data due to the large number of parameters, making them prone to overfitting and poor generalization. Transfer learning addresses this issue by leveraging knowledge from previously learned tasks. It involves a pre-training phase to capture relevant knowledge and a fine-tuning phase to apply it to new tasks, enabling models to well handle target tasks with limited samples [10].

C. Pre-trained Model

The development of deep neural networks in Nature Language Processing (NLP) has led to the introduction of Transformers [11]. Pre-trained models like GPT [12] and BERT [13], based on Transformer architectures, have shown impressive performance on various NLP tasks.

Transformer [11] is a deep learning model architecture introduced in 2017, initially designed for machine translation. It utilizes self-attention mechanisms to capture relationships between sequence elements without recurrent neural networks (RNNs). The Transformer enables significantly more parallelization, making it much easier to train on a large dataset.

BERT (Bidirectional Encoder Representations from Transformers) [13] is a pretraining technique that applies bidirectional training of Transformers to language modeling. By training on a large number of unlabeled texts using tasks like Masked Language Modeling (MLM) and Next Sentence Prediction (NSP), BERT captures bidirectional contextual information and achieves high performance in NLP tasks.

RoBERTa (Robustly Optimized BERT Approach) [7] is an extension of BERT, which builds upon BERT's architecture and is trained using a similar methodology but with some modifications. RoBERTa removes the NSP task and employs dynamic masking during training. Its superior performance and robustness in various NLP tasks make it an ideal candidate for our approach in action recommendation. We further enhance its capabilities by fine-tuning it with contrastive learning.

D. Contrastive Learning

Contrastive learning was proposed as a learning paradigm by Hadsell et al [14] for classification and has several different formulations and variations. Unlike most loss functions that sum over samples (*e.g.,* cross-entropy loss and mean squared error loss), contrastive loss functions explicitly handle the similarity and dissimilarity between samples. They define positive pairs (similar samples) and negative pairs (dissimilar samples) and aim to maximize agreement within positive pairs while minimizing agreement amongst negative pairs.

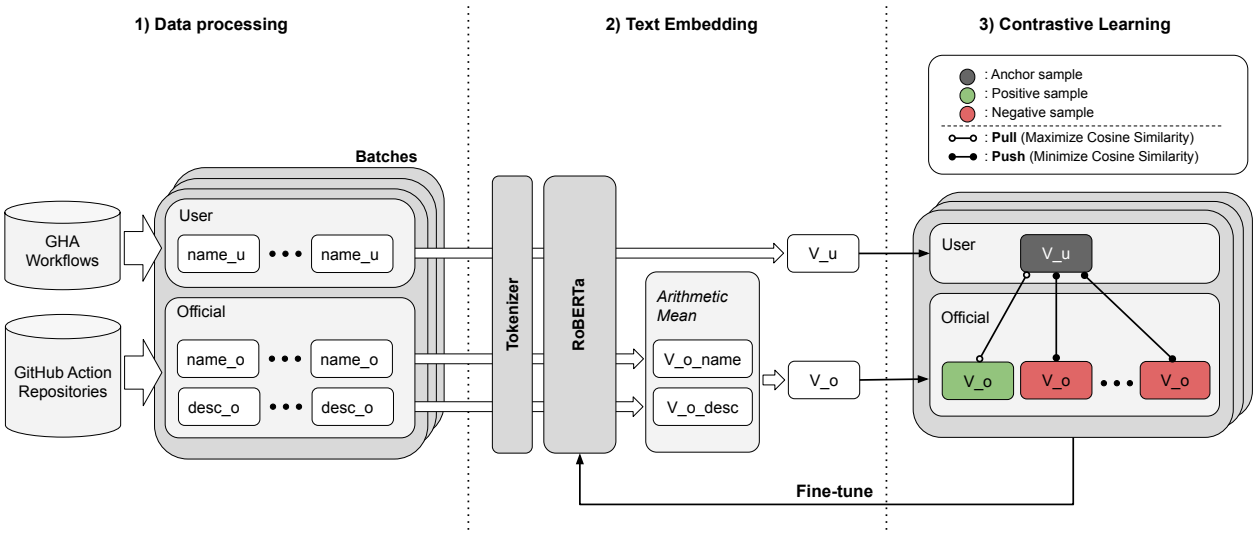

Fig. 1: Overall framework of CIGAR

The overall framework of contrastive learning can be described as follows:

- Define an encoder function $f(x)$ that converts a given sample x into latent vector representation.
- Define a function $sim(v_i, v_j)$ to measure the similarity between two representations v_i and v_j (*e.g.,* Cosine Similarity, Euclidian distance).
- Define a contrastive loss function $L(p_i, p_j)$ to compute the loss between pairs of representations p_i and p_j. The loss function encourages similar samples to have higher similarity scores and dissimilar ones to have lower scores.

Assume a batch of examples $x_1, ..., x_n$ where n represents the batch size. For each sample x_i, create positive pairs and negative pairs. Convert them into vector representation $v_1, ..., v_n$ with $f(x)$. Compute the similarity scores for each pair of representation with $sim(v_i, v_j)$, and calculate the contrastive loss $L(p_i, p_j)$ based on the similarity scores.

III. CIGAR

We propose a novel approach CIGAR (**C**ontrast**i**ve learning for **G**itHub **A**ction **R**ecommendation), leveraging a pre-trained model RoBERTa and contrastive learning. Given the textual description of a task developers want to execute in the GHA workflow, CIGAR recommends the most relevant GitHub actions.

A. Overall Framework

Fig. 1 illustrates the overall framework of our model. Our framework consists of three major components: 1) Data Processing, 2) Text Embedding, and 3) Contrastive Learning. In the subsequent subsections, we will provide detailed explanations of these components.

B. Data Processing

Data Processing is the first step in our approach. We process data from two sources: GHA workflows and action repositories. For all the steps in each GHA workflow, we first extract 1) the user description of the task, *i.e.,* the name of the step (`name_u` in Fig. 1, *e.g.,* "Set up Maven Central Repository") and 2) the used action denoted as `action_id` (*e.g.,* `actions/setup-java`). For this used action, we download its GitHub repositories. Each action repository must contain a metadata file named either `action.yml` or `action.yaml`, which defines the inputs, outputs, and runs configuration. To represent the action, we retrieve two features from the YAML file: the official names (`name_o`) and descriptions (`desc_o`). For example, the action `actions/setup-java`[4] has an official name "Setup Java JDK" and its description is "Set up a specific version of the Java JDK and add the command-line tools to the PATH". In the end, we obtain a list of data entries D in the format of `(action_id, name_u, name_o, desc_o)` after data processing.

C. Text Embedding

In this step, for each entry in D, we aim to construct a vector representation V_u (referred to as User Vector) for `name_u` and one single vector V_o (referred to as Official Vector) for both `name_o` and `desc_o`.

First, the text (`name_u`, `name_o`, or `desc_o`) is split into a list of words and fed into the RoBERTa tokenizer. The output tokens are then concatenated with a special $[CLS]$ token at the beginning and a $[SEP]$ token at the end to form a single sequence. The sequence is further converted into token IDs using the tokenizer's `convert_tokens_to_ids()` function. The token IDs are finally passed to the RoBERTa model for generating vector representations.

[4]https://github.com/actions/setup-java

It is worth noting that to effectively use the RoBERTa model, input sequences of equal length are necessary. Thus, we select a block size of 128 tokens as all the individual texts in our train set are much shorter (with a maximum size of 29 words). The block size of 128 ensures that no relevant information is omitted during training, and allows for potential longer inputs during validation and testing without the need of truncating and sacrificing essential information. If the token IDs are shorter than 128, one-padding is applied to maintain consistent input dimensions for the RoBERTa model.

The outputs of the RoBERTa model are vector representations of `name_u`, `name_o`, and `desc_o`, denoted as V_u, V_o_name, and V_o_desc, respectively. We further aggregate V_o_name and V_o_desc using arithmetical mean value [15] and obtain the Official Vector V_o. To conclude, during this step, we convert D into a list of data in 3-tuple structure: `(action_id, V_u, V_o)`.

D. Contrastive Learning

Our approach exploits Contrastive Learning to pull together text referring to the same actions and to push apart that referring to distinct actions. Specifically, we train our model using a contrastive loss function that combines the N-pair Loss and Soft-Nearest Neighbor Loss, which was originally proposed by Tao *et al.* [16]. With the aim of increasing the similarity between texts referring to the same actions while reducing the similarity between texts referring to different actions, the loss function is defined as follows:

$$L_N = -log\frac{exp(sim(v^\top, v^+)/\tau)}{exp(sim(v^\top, v^+)/\tau) + \sum_{j=1}^{N-1} exp(sim(v^\top, v_j{}^-)/\tau)} \quad (1)$$

where $v^\top$ is the vector representation of the anchor text sample, v^+ and v^- stand for representations of positive/negative text samples, and τ is the temperature, a hyperparameter that can be tuned to scale the penalties on negative samples.

For the similarity evaluation $sim(v_i, v_j)$, we adopt Cosine Similarity. A simple linear transformation is added as we expect a co-domain of [0, 1]:

$$sim(u, v) = (\frac{u \cdot v}{\|u\|\|v\|} + 1) * 0.5 \quad (2)$$

Algorithm 1 illustrates the pseudocode for constructing positive/negative samples and computing the contrastive loss.

We loop over each batch B of data, for each 3-tuple `(action_id, V_u, V_o)` we take the User Vector as an anchor example $V\_u^\top$, which represents the step name assigned by developers (*i.e.,* description of the task they want to execute in a workflow). Then we use the Official Vector, *i.e.,* the arithmetic mean of the two vectors of official name and description, as a positive example V_o^+. To generate negative examples, we again iterate over B, any Official Vectors that do not represent the same action as the anchor example (*i.e.,* having a different `action_id`) are considered to be negative, denoted as V_o^-. After constructing positive and negative examples, we compute their cosine similarity with anchor example. Then we apply Func. 1 to compute the contrastive loss L_i for each anchor example in batch B, and eventually obtain the batch contrastive loss L_B using the average value.

Algorithm 1: Contrastive loss computation

Input: A batch of data, B;
Output: Batch contrastive loss, L_B;

for *each tuple* T_i *in B* **do**
  $V\_u^\top, V\_o^+ \leftarrow T_i$;
  $S_{pos} \leftarrow sim(V\_u^\top, V\_o^+)$;
  for *each tuple* $T_j (i \neq j)$ *in B* **do**
    if $T_i.action\_id \neq T_j.action\_id$ **then**
      $\_, V\_o_j{}^- \leftarrow T_j$;
      $S_{neg_j} \leftarrow sim(V\_u^\top, V\_o_j{}^-)$;
    end
  end
  $L_i = -log(\frac{exp(S_{pos}/\tau)}{exp(S_{pos}/\tau)+\sum exp(S_{neg}/\tau)})$;
end
$L_B = \sum_{i=1}^{|B|} L_i/|B|$;

We initialize our model with the same parameters used in the Hugging Face's implementation of RoBERTa[5], and exploits the contrastive loss computation process demonstrated above to fine-tune the model.

IV. Study Design

The *goal* of this study is to evaluate the ability of CIGAR in terms of recommending correct actions to developers.

A. Research Questions

We aim at answering the following research questions (RQs):

- **RQ**$_1$: *To what extent is* CIGAR *able to provide accurate recommendations for actions?* In this RQ, we aim to analyze whether the recommended actions are actually the ones adopted by developers. The results will give a general picture of how reliable our approach is.
- **RQ**$_2$: *How do different features impact the performance of* CIGAR*?* We aim to analyze performance of CIGAR when training on different combinations of data features to understand how each feature contributes to the performance.
- **RQ**$_3$: *How do fine-tuning and constractive learning impact the performance of* CIGAR*?* Two important components in our approach is fine-tuning RoBERTa and contrastive learning. We aim to understand how much each component contributes to the performance.
- **RQ**$_4$: *How does* CIGAR *perform compared to the GitHub Marketplace search functionality?* GitHub Marketplace is currently the online official portal where developers can search for actions to reuse. We aim to compare the recommendation capacities of CIGAR and the GitHub Marketplace.

[5]https://huggingface.co/docs/transformers/model_doc/roberta

TABLE I: EXAMPLES OF DATASET

action	official name	official description	user-assigned step names	# user-assigned step names
actions/upload-artifact	Upload a Build Artifact	Upload a build artifact that can be used by subsequent workflow steps	upload package archive, upload covered self goc binary, ...	4416
actions/cache	Cache	Cache artifacts like dependencies and build outputs to improve workflow execution time	restore build files from cache, cache/restore node_modules, ...	2179
...	...	...	...	...
mikeal/merge-release	Merge Release	Deploy to npm!	release, publish, publish to npmjs	3
blinktag/nfpm	NFPM packager action v2	NFPM packaging tool for deb, rpm, and apk packaging	create apk package, create deb package, create rpm package	3

B. Data Collection

We built our dataset based on `secos-gha`, a publicly available dataset created by Decan *et al.* [6]. `secos-gha` contains 67,870 repositories, encompassing 70,278 workflows, 108,500 jobs, and 567,352 steps.

For these steps, we first extracted the actions by parsing the `uses` key (*e.g.,* `uses: actions/setup-python@v4`). We skipped those which did not contain the `uses` key. In our study, we also removed the version number (*e.g.,* `@v4`), as we found in most cases the workflows would still work when a different version, especially newer ones, are used. The name of the steps were also recorded. This resulted in 2,468 unique actions, called in 180,593 steps across 24,875 repositories. We then filtered out those actions which are associated with less than three different user-assigned task names, in order to get rid of too specific or toy actions. This resulted in 802 distinct actions.

For these 802 actions, we downloaded their corresponding GitHub repositories and extracted their action YAML files (named either `action.yml` or `action.yaml`). We then further retrieved the official names and descriptions from YAML files. As action YAML files are mandatory for GitHub action repositories, we consider a repository incomplete such the action YAML file is missing. After removing these actions with incomplete repositories, we obtained 756 distinct actions, associated with 24,993 distinct step names.

Table I presents some examples of our dataset. Apparently, some actions are reused much more frequently than others, which indicates the imbalance of the dataset, which corresponds to the reality.

C. Data splitting

To train and evaluate our approach, we split our data into train, valid, test sets with a ratio of 4:1:1 on the level of distinct user-assigned step names. For example, an action with 60 distinct step names will have 40, 10, and 10 data points in the train, validation, and test sets, respectively.

Furthermore, we only focus on the top-100 actions with the highest number of distinct step names for the validation and test sets, while all the actions were included in the train set. The reason is that most of those actions after top-100 have a very limited number of distinct step names, and cannot be easily split based on the ratio of 4:1:1. We also believe that limiting to the top 100 actions during testing enables a targeted evaluation of the model's effectiveness on actions that have a higher likelihood of being utilized by users, ensuring that the evaluation results are more indicative of real-world usage scenarios.

D. Evaluation Metrics

To evaluate our approach, we collect the following data for each input m (user-assigned step name):

- $top_N(m)$: the top-N recommended actions;
- $a(m)$: the actual action used in the step;
- $match_N(m)$: a binary variable, 1 if $a(m) \in top_N(m)$, otherwise 0.

To measure the performance of CIGAR, we utilize two evaluation metrics: success rate@N and catalog coverage. These metrics have been widely adopted in the related studies [17].

Success rate@*N*. Given a set of user-assigned step names M, which refer to a set of actions A, the success rate@N is defined as the ratio of the actions falling within the top-N suggested actions to the total number M:

$$Success\ rate@N = \frac{\sum_{i=1}^{|M|} match_N(m)}{|M|} \tag{3}$$

In our study, the number N was set from 1 to 5. Since our data is highly biased (*i.e.,* some actions were associated with thousands of distinct names while some are only associated with a few), there is a need to reduce the potential bias. Therefore, for those actions with more than 10 distinct step names, we only randomly picked 10 data points.

Catalog coverage. Ideally, we would like CIGAR to be able to provide diverse suggestions. This metric is calculated as the ratio of recommended distinct actions to the number of all the available actions for recommendation:

$$Coverage\ rate@N = \frac{|\cup_{m \in M} top_N(m)|}{|A|} \tag{4}$$

TABLE II: SUCCESS RATE ON TEST SET, N = [1-5], τ = [0.1-10000]

Temperature τ	N				
	1	2	3	4	5
0.1	0.046	0.077	0.105	0.128	0.153
1	0.198	0.319	0.408	0.477	0.528
5	0.523	0.69	0.781	0.834	0.875
10	0.661	0.833	0.889	0.936	0.953
50	0.724	**0.878**	**0.927**	**0.948**	0.955
100	**0.74**	0.871	0.907	0.944	**0.959**
500	0.708	0.851	0.895	0.927	0.955
1000	0.703	0.852	0.908	0.94	0.958
5000	0.673	0.841	0.906	0.934	0.952
10000	0.684	0.841	0.901	0.926	0.944

TABLE III: CATALOG COVERAGE ON TEST SET, N = [1-5], τ = [0.1-10000]

Temperature τ	N				
	1	2	3	4	5
0.1	0.69	0.82	0.92	0.96	0.97
1	0.84	0.92	0.96	0.97	0.99
5	0.97	1	1	1	1
10	0.96	1	1	1	1
50	0.97	1	1	1	1
100	0.97	1	1	1	1
500	0.96	1	1	1	1
1000	0.97	1	1	1	1
5000	0.95	1	1	1	1
10000	0.96	1	1	1	1

E. Experimental Setting

Our model is implemented with PyTorch, based on the RoBERTa model provided by Hugging Face[6]. The hyperparameters chosen for our approach were as follows: CIGAR was trained for 10 epochs with a batch size of 16. Adam optimizer was used with a learning rate of $5e^{-5}$. To keep the random process in our approach reproducible, the seed value was set to 123456.

V. RESULTS

This section presents our experimental results and answers our four research questions regarding the overall effectiveness of CIGAR, the impact of different features, fine-tuning, and contrastive learning in CIGAR, and to what extent it can outperform GitHub Marketplace search engine.

A. RQ1: To what extent is CIGAR able to provide accurate recommendations for actions?

To investigate the effectiveness of our approach, we conducted experiments on the test data with various temperature τ. Table II shows the success rate@N of our approach when τ and N were set to different values. When $N = 1$ (*i.e.,* only recommending one action), CIGAR can already provide reasonable results with a temperature τ higher than 10 (success rate over 0.661). For most Ns, the success rate is the highest when τ is set to 50.

To get better overview on how the performance changes along with the different τ and N, we visualized the results in Fig. 2. The left part shows the success rate@N of CIGAR with different values of N ranging from 1 to 10, when τ is set to 50. We added the values for N from 6 to 10 in this case, just to get an idea on potential saturation of success rate. Apparently, when N increases, the corresponding success rate also improves, which is expected as the more actions recommended, the higher chance to hit the adopted action. The improvement in the success rate is substantial when $N < 5$. After that, the improvement becomes relatively less noticeable. When $N = 5$, our model demonstrates a success rate higher than 95%. The right part of Fig. 2 illustrates the success rate@N of CIGAR with various values of temperature τ from 10^{-1} to 10^4. A significant enhancement in success rate can be observed while increasing τ from 10^{-1} to 10^1. However, once exceeding 50, a higher τ does not help in terms of generating more accurate suggestions in most of the cases. Instead, an excessively high temperature can even damage the model's performance especially when N is relatively small. For example, when the temperature is increased from 10^2 to 10^4, the success rate@1 decreases from 74% to 68.4%.

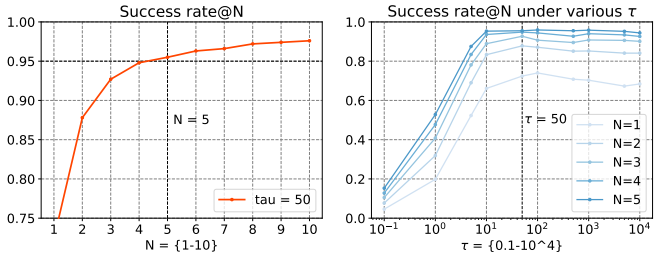

Fig. 2: Success rate of CIGAR with various N and τ

Table III shows the catalog coverage rate of our approach with various N and τ selected. When $\tau \geq 5$ and $N > 1$, CIGAR almost always achieves a full coverage; even when N is set to 1, the coverage is still at least 95%. The high coverage indicates that a large portion of the available actions are considered for recommendation, *i.e.,* CIGAR can effectively make suggestions across various types of available actions.

To be consistent and utilize the parameters which can bring the best performance, when answering the remaining RQs, we use τ of 50.

[6]https://huggingface.co/

TABLE IV: SUCCESS RATE ON DIFFERENT FEATURE COMBINATIONS, N = [1-5], τ = 50

Selected features	N				
	1	2	3	4	5
name only	0.656	0.838	0.902	0.929	0.943
description only	0.672	0.837	0.89	0.918	0.936
both w/ vector mean	**0.724**	**0.878**	**0.927**	**0.948**	**0.955**
both w/ mean similarities	0.721	0.874	0.917	0.943	0.952

TABLE V: SUCCESS RATE ON EACH VARIANT OF OUR APPROACH, N = [1-5], τ = 50

Model variants	N				
	1	2	3	4	5
CIGAR	0.724	0.878	0.927	0.948	0.955
RoBERTa w/ fine-tuning	0.316	0.432	0.521	0.577	0.634
RoBERTa w/o fine-tuning	0.081	0.116	0.153	0.171	0.197

B. RQ2: How do different features impact the performance of CIGAR*?*

To answer this RQ, we investigated the performance of the model when different features were used and when different feature combination strategies were used. More specifically, we consider the following scenarios:

- **Name only.** We only use the official name to represent actions during training, discarding the description information.
- **Description only.** We only use the description to represent actions during training, discarding the official names.
- **Both features with vector mean.** This is our original approach described in Section III, namely we use both name and description to represent actions. The two features are aggregated by calculating the arithmetic means of their high-dimensional vector representations.
- **Both features with mean cosine similarity.** Both features (name and description) are used. However, instead of taking the arithmetic means of their vector representations, in this setting we aggregate the cosine similarities between the vectors of these two features and the vector of the corresponding user-assigned step names (V_u). In other words, the model's suggestions for actions are made based on the average cosine similarity.

The success rate@N can be found in Table IV. For all the different values of N, CIGAR achieves better performance when training with both features of actions (*i.e.,* names and descriptions), instead of training only on one of the two features. Incorporating multiple features allows the model to capture complementary information, resulting in a more comprehensive representation of the data, which potentially improve the performance. Moreover, aggregating the vector representations of the two features tends to yield better performance compared to aggregating the cosine similarity. This implies that aggregating vectors can better capture the complex relationships and nuances between the two features, enabling CIGAR to make more accurate suggestions. However, it is worth noting that the improvement is rather limited.

Another fact is that when N is becoming larger, the performance difference is becoming smaller even fewer features are used. However, if we only want to take the first recommendation, using both features can bring much more visible enhancement for recommending the correct actions.

C. RQ3: How do fine-tuning and constractive learning impact the performance of CIGAR*?*

In our approach, we exploited the RoBERTa model to generate high-dimensional vector representations and used contrastive learning to fine tune the model. In this RQ, we are interested in the impacts of different components on the performance, namely the fine-tuning and contrastive learning. Therefore, we compared our approach with two variants:

- **RoBERTa without fine-tuning.** We directly use the pretrained RoBERTa to suggest actions based on user-assigned step names without fine-tuning and contrastive learning.
- **RoBERTa with fine-tuning.** We fine tune the RoBERTa model without contrastive learning. During the fine-tuning process, we use Mean Square Error (MSE) as the loss function that is used in the original RoBERTa model.

Table V shows the results of these two variants and CIGAR. Comparing the results of RoBERTa with and without fine-tuning, we find that our approach significantly benefits from fine-tuning. The success rate@5 is only 0.197 if we directly use the representations in RoBERTa to recommend GitHub actions. With fine-tuning, the success rate increases by 0.437. If we fine-tune with contrastive learning, the success rate@5 further increased by 0.321. Both fine-tuning RoBERTa and contrastive learning can bring significant improvement in the model's performance.

To have an intuitive understanding on how these variants differ from each other, we also visualized the code vectors. We randomly selected five actions and get embeddings of user-assigned step names that refer to these actions, then we use t-SNE [18] to reduce the dimensionality from 128*768 to 2 for visualization. As shown in Fig. 3, points in the same shape/color represent user-assigned step names referring to the same actions. Points with the same shape/color in Fig. 3 (a) are scattered across the graph, meaning that the embedding of name is irregular without fine-tuning. Some of the points with the same shape/color in Fig. 3 (b) converge together, but their boundaries are not always clear and some of them tangled together, which means it can tell the differences of only some actions, but not all of them. As illustrated in Fig. 3 (c), contrastive learning can keep dissimilar pairs away while pulling similar pairs together, which greatly increases the success rate when recommending GitHub actions.

(a) w/o finetune (b) w/ finetune (c) CIGAR: w/ finetune + contrastive learning

Fig. 3: Visualization of the vector representations of the user-assigned names

TABLE VI: EFFECTIVENESS OF CIGAR VS. MARKETPLACE, N = [1-5], τ = 50

Models	N				
	1	2	3	4	5
CIGAR	0.724	0.878	0.927	0.948	0.955
Marketplace	0.018	0.025	0.027	0.027	0.028

D. RQ4: How does CIGAR perform compared to the GitHub Marketplace search functionality?

As GitHub Marketplace is currently the official portal for developers to search for actions, we compared our approach with the search engine provided by GitHub Marketplace. We adopted the same test set which was used for answering RQ1. More specifically, we input the user-assigned step names in the search engine of Marketplace, and collected the returned list of actions. This process was automated with scripts. We then calculated the success rate for GitHub Marketplace and compared the results with CIGAR.

As show in Table VI, our approach remarkably outperforms the GitHub Marketplace search engine in terms of success rate@N. For most of the cases, GitHub Marketplace was not able to return any results with the step name assigned by developers. This result implies that the algorithm behind this search engine is probably trivial and does not take into account different aspects (*e.g.,* official description) in actions. We recommend that GitHub can improve the search functionality to make it easier for developers to locate the actions they need. Moreover, some actions publicly available might not be listed on GitHub actions, a search engine which contains these actions will also be appreciated.

E. Replication Package

To facilitate replication studies, our scripts and dataset are publicly available online, which can be found on https://github.com/jiangnanpro/CIGAR.

VI. DISCUSSION

Our experimental results have demonstrated the promise of CIGAR. However, it would be beneficial to dig deeper and understand when our approach does not generate good recommendations for future improvement. Therefore, in this section, we investigate the underlying reasons behind the occasional failures of CIGAR.

A. When Does CIGAR Fail When $N = 1$?

We first inspected the cases in which CIGAR fails when $N = 1$, *i.e.,* the first recommended action is not the one adopted by developers. Out of the 100 distinct actions used in the test set, 69 of them contain at least one failure. In total, 236 recommended actions are not what developers adopted. Table VII provides some examples of failed attempts.

With a closer examination on the failed recommendations, we found that these failures often include duplicated actions. For example, both `actions/setup-ruby` and `ruby/setup-ruby` serve the purpose of setting up a Ruby environment. When looking into their official repositories, we discovered that the action `actions/setup-ruby` was marked as deprecated, and users were advised to migrate to `ruby/setup-ruby`, which is actively maintained by the official Ruby organization at this moment. Similar situations were observed for the actions such as `actions/create-release` and `marvinpinto/action-automatic-releases`. In the repository of `actions/create-release`, action owners indicated that the action was no longer maintained and suggested users to consider other alternative actions.

TABLE VII: EXAMPLES OF CIGAR FAILURES @$N = 1$

user-assgn. step name	recommended action	action adopted
"setup ruby 2.6"	actions/setup-ruby	ruby/setup-ruby
"cache .gradle"	actions/cache	burrunan/gradle-cache-action
"create new release"	actions/create-release	marvinpinto/action-automatic-releases

These examples clearly highlight the presence of outdated actions. As GitHub Actions is still an emerging realm, action deprecation and updates can occur frequently. We would recommend that developers pay attention to the potential changes of the actions they use. To improve the usability, action recommender systems can also detect duplicate actions and consider whether the recommended actions are still actively maintained. Preferences should be given to those still active.

B. When Does CIGAR Fail When $N = 5$?

We further investigated the failed cases when $N = 5$, namely the adopted actions do not appear among the top 5 recommendations. Out of the 100 actions in our test set, 23 of them had at least one failed case. In total, CIGAR was not able to recommend actions for 45 different task descriptions (step names). Interestingly, most of the failed examples were associated with user-side issues. As depicted in Table VIII, many of the failed examples were caused by the overly general assigned step names (*e.g.,* "build" and "test"). Another interesting example is the action `coverallsapp/github-action` in Table VIII. The develop assigned the step name "send parallel finished" and used this action. However, this assigned step name is irrelevant to the official name ("Coveralls GitHub Action") or the description ("Send test coverage data to Coveralls.io for analysis, change tracking, and notifications") of the action. Instead, it refers to an optional input named "parallel finished". Consequently, CIGAR faced challenges in deducing the correct suggestion due to the mismatch between the step name and official name/description.

To improve the performance of CIGAR, we could incorporate more information from action repositories. By including details about each input of actions, CIGAR might be able to provide more relevant suggestions. However, it is also important to consider that more data leads to increased computational power requirements. Besides, some noise might be introduced as some inputs are very general and not relevant to the task itself. Therefore, more study should be conducted to understand the benefit of integrating input information in our models.

Another lesson we could learn is that, developers are not always able to describe the task they want to execute in GHA workflow concisely and accurately. A intelligent assistant might be introduced to take a proactive approach and guess what kind of steps developers would like to have at the first place.

TABLE VIII: EXAMPLES OF CIGAR FAILURES @$N = 5$

user-assgn. step name	recommended action	action adopted
"build"	goreleaser/goreleaser-action uraimo/run-on-arch-action borales/actions-yarn eskatos/gradle-command-action gradle/gradle-build-action	pypa/cibuildwheel
"test"	reactivecircus/android-emulator-runner eskatos/gradle-command-action gradle/gradle-build-action borales/actions-yarn goreleaser/goreleaser-action	cypress-io/github-action
"send parallel finished"	octokit/request-action styfle/cancel-workflow-action actions/github-script geekyeggo/delete-artifact rtcamp/action-slack-notify	coverallsapp/github-action

VII. THREATS TO VALIDITY

Threats to internal validity concern the extent in which the evidence supports our claim about cause and effect. In this paper, we exploited the pre-trained RoBERTa model to convert descriptive keywords or names into the high-dimensional code vector space. The scale of our data is not huge, CIGAR may tend to memorize the examples in the dataset instead of learning general patterns. This can lead to overfitting, as the model becomes too specialized to the train set and performs poorly on new data. To address this threat, several regularization techniques have been performed in our approach, including early stopping and exploiting a validation set.

Threats to construct validity concern the relationship between theory and observation. In our study, we used success rate@N and catalog coverage as the metrics for performance evaluation. One issue is that our dataset is highly imbalanced, which means some actions are associated with much more step names than others. To mitigate the bias it could introduce in the success rate, we focused on the top-100 actions with the highest number of distinct step names for validation and testing, and only picked 10 randomly selected data points for those actions with more than 10 distinct step names. However, it is unclear how our approach performs for less popular actions. Another issue is that we only consider a recommendation correct when it is exactly the one used by developers. However, sometimes, it is possible that several actions would work. Therefore, our reported success rate is actually the lower bound for our dataset.

Threats to external validity mainly concern the generalizability of our results. We built our dataset based on the data published in 2022 [6] and only used actions associated with over two distinct names. Currently, the number of available actions are increasing rapidly, and much more workflows have been created since the data were collected. It is unclear where our recommender system can achieve similar performance when the data points significantly increase. However, we believe our study has sufficiently demonstrated that the applied method is promising, and the high catalog coverage also indicates the ability of recommending a wide range of actions.

VIII. RELATED WORK

In this section, we discuss the related work on GitHub Actions and recommendation systems for software engineering.

A. GitHub Actions

Since the emergence, GitHub Actions has been widely adopted by developers. However, it has not been extensively studied by researcher, given that it is a rather young tool.

Golzadeh *et al.* [19] conducted an empirical study based on 91K+ active GitHub repositories to understand the evolving CI landscape on GitHub. By analyzing the development history of these repositories, they observed that GHA has become the dominant CI/CD service only 18 months after its introduction. After analyzing 14K migrations of CI tools in 13K+ repositories, they found that most of them (12K+ cases) targeted GHA.

Kinsman *et al.* [20] conducted the pioneering study exploring developers' utilization of GHA and the subsequent changes in various activity indicators following their adoption. Through an extensive analysis of 3,190 repositories, they found that the implementation of GHA led to an increase in the number of rejected pull requests while reducing the number of commits in merged pull requests. By manually inspecting 209 issues related to GHA, they concluded that most developers had a positive perception of GHA.

Valenzuela-Toledo and Bergel [21] investigated the use and maintenance of GHA workflows. They conducted a manual inspection of 222 commits associated with workflow changes and revealed 11 distinct types of workflow modifications. Additionally, they discovered several shortcomings in the production and maintenance of GHA workflows, highlighting the need for adequate tools that facilitate the creation, editing, refactoring, and debugging of GHA workflow files.

Decan *et al.* [6] conducted an empirical study based on a dataset of 68K repositories, with 43.9% utilizing GHA. They indicated that the reuse of actions in GHA is a common practice, but it is concentrated in a limited number of actions. Overall, they provided an overview of GHA's usage, contributing to a better understanding of both GHA and its implications for collaborative software development on GitHub.

B. Recommendation systems for software engineering

Recommendation systems for software engineering (RSSEs, hereafter) are software applications that provide information items estimated to be valuable for a software engineering task in a given context [22]. They assist developers in navigating large information spaces and getting instant recommendations that are helpful to solve a particular development task.

Nguyen *et al.* [23] presented FOCUS, a tool which mines open-source software repositories to recommend API method invocations and usage patterns by analyzing how APIs are used in projects similar to the current project. FOCUS has been evaluated on a large number of Java projects collected from GitHub and Maven Central. Results show that it outperforms the state-of-the-art approach PAM [24] with regard to the success rate, accuracy, and execution time.

Di Rocco *et al.* [25] presented TopFilter, a tool to assist open source software developers in selecting suitable topics for GitHub repositories. They built a project-topic matrix and applied a syntactic-based similarity function to recommend missing topics by representing repositories and related topics in a graph. The results show that TopFilter exhibits good performance. Furthermore, by leveraging the results obtained from the state-of-the-art topic recommender system MNB network [26] as its initial set of topics, TopFilter can experience a substantial enhancement in its performances of topics recommendation.

Rubei *et al.* [27] introduced PostFinder, which provides suggestions on posts that contains highly relevant answers from Stack Overflow according to a user-given context consisting of the source code under development. The approach has been validated utilizing a user study involving a group of 12 developers.Experimental results indicate the suitability of PostFinder to recommend relevant Stack Overflow posts and concurrently show that the tool outperforms a well-established baseline FaCoY [28].

Wei *et al.* [29] proposed an approach named CLEAR for automatic API recommendation. They point out that existing methods have limitations in capturing sequential semantic information and distinguishing similar queries. CLEAR addresses these issues by using BERT sentence embeddings and contrastive learning. Experimental results indicate that CLEAR achieves significant improvements in method-level and class-level API recommendation compared to other approaches.

A systematic literature review has also been conducted for RSSEs. Gasparic and Janes [22] assessed 46 papers on RSSEs from 2003 to 2013. Their findings indicate that RSSEs mainly output source code artifacts to enhance system quality, streamline development, reduce cognitive load, and aid decision-making. Unexploited opportunities lie in the development of recommendation systems outside the source code domain.

Di Rocco et al. [30] also presented an experience report on various RSSEs that have been developed in the context of EU CROSSMINER project. They indicated that multiple data mining and machine learning techniques are adopted while building RSSEs when a large amount of data are available.

To the best of our knowledge, none of the RSSEs have tried to recommend GitHub actions or help the composition of other CI/CD tools.

IX. Conclusions and Future Work

In this paper, we presented CIGAR, a new approach to recommend actions to developers for facilitating their GHA workflow composition. The evaluation with action adoptions from real GHA workflows indicates that CIGAR can generate accurate recommendations and cover a wide range of available options. CIGAR also remarkably outperforms the default search engine provided by GitHub Marketplace. The satisfactory performance reveals the promise of using the RoBERTa model and contrastive learning to create recommender systems for software engineering tasks.

As for future work, we will focus on two aspects. Firstly, we will focus on improving the performance of our approach. CIGAR will be trained on a larger scale of dataset which is up-to-date and covers more actions on GitHub Marketplace, as the current dataset was crafted based on the data released in 2022. Furthermore, more features, such as the inputs of each action and the `ReadMe.md` files in action repositories will be taken into account when constructing vector representations. Secondly, we will focus on other recommendation tasks related to CI/CD workflows. The core techniques in CIGAR can also be applied in different areas in the field of software engineering. In the future, we plan to adopt deep learning for tasks including automatic workflow completion/generation, GHA workflow debugging and refactoring.

Acknowledgments

We thank Frits W. Vaandrager for the feedback on the earlier draft of this paper.

REFERENCES

[1] "Github." [Online]. Available: https://github.com/

[2] L. Dabbish, H. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," 02 2012, pp. 1277–1286.

[3] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 345–355. [Online]. Available: https://doi.org/10.1145/2568225.2568260

[4] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 285–296.

[5] P. Rostami Mazrae, T. Mens, M. Golzadeh, and A. Decan, "On the usage, co-usage and migration of ci/cd tools: A qualitative analysis," *Empirical Software Engineering*, vol. 28, no. 2, p. 52, 2023.

[6] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, "On the use of github actions in software development repositories," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 235–245.

[7] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[8] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," 2021.

[9] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[10] X. Han, Z. Zhang, N. Ding, Y. Gu, X. Liu, Y. Huo, J. Qiu, Y. Yao, A. Zhang, L. Zhang, W. Han, M. Huang, Q. Jin, Y. Lan, Y. Liu, Z. Liu, Z. Lu, X. Qiu, R. Song, J. Tang, J.-R. Wen, J. Yuan, W. X. Zhao, and J. Zhu, "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666651021000231

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[12] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[14] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2, 2006, pp. 1735–1742.

[15] W. Schwarz and J. Spilker, *Arithmetical functions*. Cambridge University Press, 1994, vol. 184.

[16] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive cross-language code clone detection," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 413–424.

[17] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2009.

[18] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[19] M. Golzadeh, A. Decan, and T. Mens, "On the rise and fall of ci services in github," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 662–672.

[20] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use github actions to automate their workflows?" in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 420–431.

[21] P. Valenzuela-Toledo and A. Bergel, "Evolution of github action workflows," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 123–127.

[22] M. Gasparic and A. Janes, "What recommendation systems for software engineering recommend: A systematic literature review," *Journal of Systems and Software*, vol. 113, pp. 101–113, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002605

[23] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1050–1060.

[24] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 254–265.

[25] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. Nguyen, and R. Rubei, "Top-filter: an approach to recommend relevant github topics," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[26] C. Di Sipio, R. Rubei, D. Di Ruscio, and P. T. Nguyen, "A multinomial naïve bayesian (mnb) network to automatically recommend topics for github repositories," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020, pp. 71–80.

[27] R. Rubei, C. Di Sipio, P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, "Postfinder: Mining stack overflow posts to support software developers," *Information and Software Technology*, vol. 127, p. 106367, 2020.

[28] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 946–957.

[29] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "Clear: Contrastive learning for api recommendation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 376–387.

[30] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, and R. Rubei, "Development of recommendation systems for software engineering: the crossminer experience," *Empirical Software Engineering*, vol. 26, no. 4, p. 69, 2021.