Автономная некоммерческая организация высшего образования «Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS (BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы «Информатика и вычислительная техника»
Area of Specialization / Academic Program Title: «Computer Science»

Тема / Topic	**Оценка характеристик качества проекта на основе метрик из открытых репозиториев / Project quality characteristics scoring based on metrics from open-source repositories**

Работу выполнил / Thesis is executed by	**Бариев Азат Радикович / Bariev Azat Radikovich**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Круглов Артём Васильевич/ Kruglov Artem Vasilevich**	подпись / signature

Иннополис, Innopolis, 2023

Contents

List of Tables

List of Figures

Abstract

Software quality is one of the key parameters of project success. Maintainable code reduces costs of implementing new features. Secure code reduces risk of sensitive data leak. Easily deployable projects and automated workflows reduce bugs and save time of development. In this paper we collect different metrics on a number of open source repositories. Then we use these metrics to cluster repositories in different groups, and interpret the result with SHAP values. Also, we propose new metrics, which are based on data retrieved from GitHub Actions files. We discovered that our new metrics proved to be the most effective for clustering in our dataset.

Chapter 1

Introduction

Poor quality makes applications prone to bugs and failures. It is hard to maintain and update such projects.

In order to improve software quality, programmers use different tools to measure quality and find weak spots. Common tools are linters or formatters, static analyzers, security checkers, and testing frameworks.

Linters are mostly checking the code for codestyle matching, and some can check whether the directory tree is according to the standards, if there are any. Static analyzers parse code to find bugs or syntax errors. Some are capable of finding maintainability problems, such us high complexity of the code or code duplications. Security checkers find unsafe code dependencies and common vulnerabilities in the code. Testing frameworks run different tests and optionally calculate code coverage.

While many researchers propose different models and metrics for quality estimation, not all of them are applicable or tested on real projects. Thus, in this paper we want to research what parts of software quality can be measured on the data and metrics retrieved from open source repositories, and to study what technologies developers use to maintain their project quality, and look at correlations between different metrics. To achieve this, we collect different metrics: statistical information from GitHub API, maintainability metrics of the code, dependency metrics, and workflow metrics.

While most of these metrics are easily accessible, and many tools have been developed for it already, the workflow metrics are rarely used. Most CI\CD pipelines have very different format and behaviour, which complicates the development of tools for measuring workflow quality. Yet these configurations have some common features: each configuration defines conditions and order of execution of different commands. This is what we will try to measure: we will collect numerical data of amount of jobs and steps, and try to understand what is being executed. In this work we are only consider the tools developers use for quality validation.

We expect that this work will show some results on the way different metrics influence the quality of a product and propose some new metrics, which can be used in developing better quality measurement tools.

To achieve this, we want to know which characteristics are possible to measure on open source repositories:

RQ 1: Which quality characteristics can be measured on open source repositories?

In this paper, we chose deployability, security, and maintainability. Apart from other characteristics (usability, reliability, and portability), these can be measured without running applications or applying any subjective analysis. This is a crucial part, as we need to process many repositories with different running procedures; some of them are just libraries, others are big services with complex architectures.

Thus, each repository would require manual installation and running, which is extremely time consuming. With similar reasoning, we can not apply any subjective metrics.

Then, after we chose characteristics to measure, we need to define which metrics we will use to measure characteristics:

RQ 2: Which metrics can we use to score quality characteristics?

We will focus on metrics that are either already available in a public repository or can be calculated without running the application.

First, we retrieve common repository info: number of commits, contributors, forks, issues, pulls and stars. This information is unlikely to contain information about chosen characteristics, but it can be useful in the validation part.

After that, we retrieve metrics that can help us measure the chosen characteristics. For deployability, we will collect workflow statistics in the repository and analyze workflow files. For security, we will collect information about dependency vulnerabilities and usage of security frameworks. Finally, we collect some common maintainability metrics, such as linter or formatter usage, code coverage usage. An exact list will be provided in the methodology section.

After we define metrics, we need to find a way to retrieve all of them: which APIs and tools are available and which can be derived from collected metrics.

RQ 3: How can identified quality characteristics be measured?

For most of them, the Github API is enough, and some require downloading repositories and analyzing their files.

After all metrics are collected, we preprocess it to make a dataset. After preprocessing, we will try to apply some machine learning techniques for clustering and see the results.

RQ 4: Can we apply these metrics to predict or measure the quality of a project?

As a result, we expect to create an application that measures multiple metrics and categorizes any Python repository into different groups by Github link.[1].

[1]All article related files stored in this GitHub repository: https://github.com/Ziucay/thesis-project

Chapter 2

Literature Review

I Overview of software quality characteristics

According to ISO/IEC 25010 [1], such characteristics of product quality defined: a) Functional suitability b) Performance efficiency c) Compatibility d) Usability e) Reliability f) Security g) Maintainability h) Portability.

Let us take a look at each of these characteristics:

From the standard [1], functional suitability is "degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions". It has three sub-characteristics: functional completeness, functional correctness, and functional appropriateness.ă

These sub-characteristics require clearly defined tasks and objectives, which are rarely found in open source repositories. Moreover, they require some subjective measurement of the degree to which software satisfies these requirements, which is unlikely to be done by analyzing the metrics available from public repository.

In the same manner, we will not research performance efficiency, reliability, compatibility, usability, and portability.ă

These characteristics are either subjective or need requirements or can be measured from testing and benchmarking, which is outside the scope of this paper.

While we are unable to measure reliability, we can try to measure another software quality aspect, which relates to it - deployability. According to Google's 2022 State of DevOps report [2]: "High performers who meet reliability targets are 1.4x more likely to use Cl." They also found that "Teams that combine version control and continuous delivery are 2.5x more likely to have high software delivery performance than teams that only focus on one." Thus, the state of CI(Continuous Integration) and CD(Continuous Delivery) pipelines in the project affects the reliability of the project.

As we investigate several characteristics, we should review existing papers on each of them.

II Overview of works grouped by characteristics

A. Deployability

Deployability, and DevOps, the practice of implementing this characteristic in software, despite not being present in the ISO standard, plays a crucial role in maintainability, security, and reliability of the software. CI\CD pipeline automates testing, code style and security checks, and deployment of the application, thus making the whole lifecycle of the software more efficient and reliable. Forsgren et al. [3] shows what "...organizations software delivery capability can in

fact provide a competitive advantage to your business". In the same book [3] authors propose these software metrics: lead time, deployment frequency, mean time to restore, and change failure percentage. However, in this case, only the deployment frequency metric is available for us, as the GitHub repository is unlikely to contain retrievable data for the other three metrics.

B. Security

Security is one of the most crucial concerns in software quality and has various studies and tools. Chowdhury et al. [4] propose several system- and code-level security metrics and proposeăsome ways to apply them. They overview existing approaches to measure security: cumulative metrics (number of bugs detected), numbers of mentions in CERT advisories, and code-level metrics. We can try to quantify these metrics in our work and create a scoring function with them. Mellado et al. [5] investigate existing security models. They conclude that despite the existence of many approaches, there is a need for a more complete model that will enable the measurement and scoring of project security. Also, the authors review various standards and initiatives regarding software security. They notice that many of these standards show a broad but unspecific image of common software vulnerabilities. From these sources, we can create a set of security-related metrics, which we can measure from public repositories.

C. Maintainability

Then, we will review articles regarding software maintainability.

Dubey et al. [6] reviews object-oriented metrics for measuring maintainability and analyses them. They also proposed a model based on Chidamber & Kemerer metrics [7]. To avoid reinventing already implemented tools, we should review existing tools for measuring maintainability. Lincke et al. [8] reviews such free and commercial tools and found that different tools may calculate the same metrics differently. Dagpinar et al.[9] conclude that significant metrics for maintainability are size and import direct coupling metrics.

Oman and Hagemeister [10] explore many maintainability metrics and arrange them in a tre structure. Particularly, they focus on the Target Software System metrics - maturity attributes, source code metrics, and supporting documentation. While maturity attributes(attributes of language and particular framework) are outside the scope of our work, we can try to measure some maintainability and source code metrics presented.

D. Literature review conclusion

To summarize, there are many papers suggesting models and metrics for these three software quality characteristics, but not many trying to apply them to real projects. Based on this, we can try to make our own solution to analyze open-source repositories with metrics and models from these articles.

Chapter 3

Methodology

In this chapter, we describe all the steps that will be done in this work.

I Criteria for choosing repositories

First, we need a list of repositories with which we will work. To ease the calculation of metrics, we decided to choose repositories in one language, and this language should have many repositories. We chose Python due to its popularity.ă

To create a representative dataset, we need to filter the projects in the unfinished stage. We define "unfinished" as projects that have no working (or at least prototype) version.ă

To exclude these projects, we will set some requirements:

- Project is at least one-year-old
- It has at least 500 stars
- Language of the repository Python
- It has at least one push for the last year

Based on empirical observations, most projects require being at least one year old to be usable. In addition, we introduce stars and fork criteria for the same reason.

As a result, we expect to have a list with as many repositories as possible.

II Metrics

Then, we move to collect metrics from repositories.

A. General repository info

These metrics are not for measurement quality characteristics but rather indicators of the popularity of the repository. All of them can be collected automatically via the GitHub API.

TABLE I
General repository metrics

Metric name	Explanation
The number of stars	How many people awarded the repository.
The number of forks	How many times people cloned the repository to modify or expand it
The number of issues	The number of questions and suggestions to the repository
Pulls: amount	Amount of pull requests
The number of commits	How many commits does the repository have
Amount of contributors	How much people contributed to the project

B. Security

To analyze python dependencies, we will use data from requirements.txt files in repositories - this file commonly contains information about dependencies to use the package. From our empirical observation, many repositories do not have repositories.txt files or are renamed to some other name.

To solve this issue, we will check all files whose names suit to this regex: $.*\mathrm{requirements}.*\backslash.\mathrm{txt}$. From them, we will choose the first in alphabetical order. For repositories with no such file or renamed repositories, we assume that they do not have any dependencies.

TABLE II
Security metrics

Metric name	Explanation
Percentage of vulnerable dependencies to the total	Amount of dependencies with at least one detected vulnerability to the amount of safe ones
Vulnerabilities per vulnerable dependency	Average amount of vulnerabilities in vulnerable dependency
Presence of security testing frameworks	Any identified security checker found in workflow

C. Deployability

These metrics are used to measure the deployability of the project.

TABLE III
Deployability metrics

Metric name	Explanation
The number of workflows	Number of workflow files
The total number of jobs	Total number of jobs in all workflows
Total step amount	Total number of steps in all workflows
Total action amount	Total number of actions in all workflows
Any testing in workflow	Does any workflow run tests
Any testing framework in workflow used	Does any testing framework found in workflow
Deployment frequency	How often the project releases

Each workflow is a separate $*.\mathrm{yml}$ file. It consists of various configuration options, conditions and jobs. In this paper, we are interested in job steps. Each step of a job is either a command or GitHub Action. Github Action is an yml file with some commands. They are typically used to reuse workflow steps.

For numerical metrics, we just count them directly from the files.

Binary metrics require further analysis. We save each command and Github Action in one file. Then, after all repositories are processed this way, we have one big plain text file. After that, we need to tokenize it - split a whole file into an array of strings by spaces. Then, we remove duplicates, common commands ($\mathrm{echo}, \mathrm{ls}$, for example) and Actions, and other irrelevant tokens.

After this processing, we expect to have a list of all programs and actions used for checking the quality of the project - testing frameworks, static analyzers, linters, formatters, etc. Then, for each repository, we check what tools it uses in its workflows and save that information in the dataset as boolean values.

D. Maintainability

These metrics are used to measure maintainability of the project.

TABLE IV
Maintainability metrics

Metric name	Explanation
Mean Cyclomatic complexity (CC)	Number of linearly independent paths in a method, average per file
Mean Maintainability Index (MI)	Metric to measure software maintainability, measured mean per file
Mean LOC	Lines of code, mean per file
Mean LLOC	Logical lines of code, mean per file
Mean SLOC	Source lines of code, mean per file
Mean Comments	Number of comments, mean per file
Mean Blank lines	Number of blank lines, mean per file
Code coverage tools presence	Does any code coverage tool found in workflow
Linters or formatter presence	Does any linter or formatter found in workflow
Document linter or formatter presence	Does any document formatter found in workflow

The formula for Maintainability Index:

$$\mathrm{MI} = \max\left[0, 100\frac{171 - 5.2\ln\mathrm{V} - 0.23\mathrm{G} - 16.2\ln\mathrm{L} + 50\sin(\sqrt{2.4\mathrm{C}}))}{171}\right]$$

III Analyzing dataset

A. Preprocessing

Before building a model, we need to make some preprocessing step.

First, we remove all rows which had some error during collecting stage. It can be connectivity error during downloading, repository deleted or any other problem which made metrics collection failed.

B. Clustering

For clustering we use HDBSCAN [11] clustering technique. HDBSCAN is a hierarcical density-based clustering algorithm, inspired by HDBSCAN [12] and OPTICS [13] alorithms. The advantage of this algorithm is better performance for clusters with varying density.

C. Visualization

To visualize the clusters, we will apply tSNE [14] dimensionality reduction technique. As shown in the original article, this algorithm perform better than some other state of the art algorithms for now.

D. Interpretation of the dataset

To explain the clusters, we will calculate SHAP [15] values and plot its graphs, to see why the clusters formed as they are. This approach can calculate the contribution of each feature value to a particular prediction. Thus, we can define an human understandable characteristic for each cluster found, understand whether this cluster consists of better or worse performing projects.

Chapter 4

Implementation

In the first section, we will retrieve a list of repositories and describe obstacles found along the way. In the second chapter, we will describe metrics retrieval.

I Retrieving repositories

Before retrieving metrics, we need to collect a list of repositories that suit our criteria, as defined in the methodology section.

The GitHub API for repositories has two major limitations: a limit for accessing RESTs and a limit of 1000 repositories retrieved at a time.

A. *Avoiding API limit*

For registered users, the API limit is 5,000 requests per hour. As there can only be one account per user, we decided to add delays to the metrics retrieval code. This way, when the limit is reached, the application sleeps until the limit is restored.

```
g = Github(token)
names = []
```

```
  while (len(names) < num_repos):
    if check_rate_limits(token) == 0:
      print(f'Token reached limit at {cur_year} and {cur_month}')
      time.sleep(600)
      continue
    ...
  return names
```

Listing 4.1: Check for rate limits

B. Avoiding limit of retrieving repositories

To retrieve metrics, we need a list of repositories. But if we simply use our criteria in the search without making any changes, only 1000 repositories will return. To avoid this, we used time-slicing: starting from a certain date, we search to get metrics that were created only this month, then repeat the search for the next months until the current date. This way, we retrieved less than 1000 repositories for each month, but we are now able to get as many repositories as there are in compliance with our criteria.

```
cur_year = start_year
cur_month = start_month
  while (len(names) < num_repos):
  ...
      start = f'{cur_year}-{add_zeros_to_num(cur_month)}-01'
      end = f'{cur_year}-{add_zeros_to_num(cur_month)}-{
   get_number_of_days_in_month(cur_year,cur_month)}'

      repos = g.search_repositories(f"created:{start}..{end} stars:>500 forks
   :>5 pushed:>2022-01-24 language:Python")

      for repo in repos:
        names.append(repo.full_name)

      cur_year, cur_month = next_date(cur_year, cur_month)
```

```python
...
return names
```

Listing 4.2: Searching for repositories

II Retrieving metrics

First, we need to find out which metrics can be retrieved from GitHub. According to the official documentation, these metrics are available for non-collaborators and are relevant for this research:

a) Commits: total amount, and amount by weeks b) Amount of contributors c) Amount of forks d) Issues: amount, tags e) Pulls: amount and amount by weeks f) Amount of stars

A. *Security metrics*

For measuring security metrics, we used the Python package safety. It can, provided a requirements file exists, find all known vulnerabilities of listed dependencies:

```python
...
requirements_path = []
    for root, dirs, files in os.walk('./temp'):
        for file in files:
            if regex.match(file):
                print(file)
                requirements_path.append(os.path.join(root, file))
    requirements_path.sort()
    if len(requirements_path) != 0:
        least_path = requirements_path[0]
        os.system(f'safety check -r {least_path} --output json > ./
   requirements_results.json')
```

```
...
```

Listing 4.3: Measuring security metrics

B. Maintainability metrics

For maintainability metrics, we used package radon. It can calculate some basic metrics (LOC, Cyclomatic complexity) and some sophisticated metrics - Maintainability Index. We retrieved them and then calculated the mean values.

```
...
mi = []
        data = json.load(results)
        for file in data:
            if 'mi' in data[file]:
                mi.append(data[file]['mi'])
        if len(mi) == 0:
            df.loc[df['name'] == repo, 'cc_mean'] = 0

            df.loc[df['name'] == repo, 'cc_variance'] = 0
        else:
            mean = sum(mi) / len(mi)

            df.loc[df['name'] == repo, 'mi_mean'] = mean

            df.loc[df['name'] == repo, 'mi_stdev'] = statistics.stdev(mi)
...
```

Listing 4.4: Measuring maintainability metrics, example for Maintainability index

C. Deployability metrics

The number of workflows and the number of workflows that run are directly accessible from the GitHub API.

To measure workflow statuses, we divide all workflows runs into three groups: success runs, failure runs, and other runs. This is important as workflows can have many statuses. Thus, success runs are those that have ’success’ or ’completed’ status. Failed ones have ’failure’ status. Other statuses are put in another section.

```
...
    match status:
        case 'success' | 'completed':
          status = 'completed'
        case 'failure':
          pass
        case _:
          status = 'other'

      runs.append((created_at,status))
...
```

Listing 4.5: Filtering statuses

Then we measure average workflow run frequency (Listing 4.6).

```
...
if workflow_runs_count <= 1:
      pass
    else:
      sorted_dates = sorted(runs, key= lambda x: x[0])

      time_difference = datetime.timedelta(0)

      for i in range(1, len(sorted_dates), 1):
        time_difference += sorted_dates[i][0] - sorted_dates[i-1][0]

      frequency = time_difference / (len(sorted_dates) - 1)

    df.loc[df['name'] == repo,'workflow_frequency'] = frequency
```

```
...
```

Listing 4.6: Measuring deployment frequency

D. Workflow metrics

For workflow metrics we have both numerical and boolean metrics.

1) Numerical metrics: Numerical metrics are simply parsed from yml files found in repository.

```
def get_metrics(df, regex, repo):
    workflows_path = []
    for root, dirs, files in os.walk('./temp'):
        for file in files:
            if regex.match(file):
                workflows_path.append(os.path.join(root, file))

    workflows_path.sort()
    if len(workflows_path) != 0:

        jobs_count = 0
        steps_count = 0
        uses_count = 0
        run_count = 0
        run_commands = []
        uses_commands = []

        for workflow in workflows_path:
            with open(workflow, 'r') as stream:

                data_loaded = yaml.safe_load(stream)
                jobs = data_loaded['jobs']
                jobs_count += len(jobs)

                for job in jobs:
                    current_job_steps = jobs[job]['steps']
```

```python
                    steps_count += len(current_job_steps)

                    for step in current_job_steps:
                        if 'uses' in step:
                            uses_count += 1
                            run_commands.append(step['uses'])
                        elif 'run' in step:
                            run_count += 1
                            uses_commands.append(step['run'])


```

Listing 4.7: Parsing data from workflow files

2) Boolean metrics: These are:

a) Presence of security testing frameworks b) Any testing in workflow c) Any testing framework in workflow used d) Code coverage tools presence e) Linters or formatter presence f) Document linter or formatter presence

First, we need to collect and save each step and action. We do it also by parsing workflow files. In result, we have a column with array of actions, and a column with array of terminal commands for each repository. After that, transform these columns to a dictionary with tokens as keys, and values as occurrences. Listing 4.8 shows this process for actions column. Terminal commands are processed in a similar way.

```python
    # read dataset
df = pd.read_csv('./dataset-with-workflow-7.csv')

# get actions
df_actions = df['actions']

# convert to list of lists
df_actions_list = df_actions.values.tolist()
df_actions_list = [i.strip('][').split(', ') for i in df_actions_list]
```

```

# flatten it
flat_list = [item for sublist in df_actions_list for item in sublist]

actions_dict = {}
for i in flat_list:
    if i in actions_dict:
        actions_dict[i] += 1
    else:
        actions_dict[i] = 1

keys = list(actions_dict.keys())
for i in range(len(keys)):
    if actions_dict[keys[i]] <= 10:
        count_tens += 1
        actions_dict.pop(keys[i])

with open("actions.json", "w") as outfile:
    json.dump(actions_dict, outfile)
```

Listing 4.8: Analyzing workflow actions

In the result, final regexes shown in Listing 4.9.

```
    any_test_regex = re.compile('.*test.*')
testing_frameworks_regex = re.compile('.*(pytest|brownie|tox|unittest|doctest|
    nox|nbmake).*')
any_linter_regex = re.compile(
    '.*(ruff|flake8|isort|lint|black|eslint|pyupgrade|pylint|yapf|pycodestyle|
    mypy|anchore/scan-action).*')
any_coverage_regex = re.compile('.*(coverage|codecov|coveralls).*')
any_doc_regex = re.compile('.*(codespell|pandoc|freeze|graphviz|mkdocs|sphinx|
    markdown).*')
any_security_regex = re.compile('.*(safety|bandit|codeql).*')
```

Listing 4.9: Regexes for binary metrics

III Analyzing dataset

A. Preprocessing

Before we start applying Machine Learning methods, we need to prepare the dataset. First, we remove all entries which do not have all required metrics. Then, we scale each column with z-score normalisation, to reduce the bias of some columns having much bigger or much smaller value than other columns.

B. Clustering

For clustering we use HDBSCAN method. This method is great for working with dataset of such dimensionality.

```python
def hdbscan_labels(df):
clusterer = hdbscan.HDBSCAN(
    algorithm='best', alpha=1.0,
    approx_min_span_tree=True, gen_min_span_tree=False,
    leaf_size=40, metric='euclidean', min_cluster_size=120, min_samples
=17, p=None).fit(df)

return clusterer.labels_
```

Listing 4.10: HDBSCAN clustering

C. Visualisation

For visualisation we used tSNE algorithm.

```python
def tsne_visualisation(df):
X_embedded = TSNE(n_components=2, learning_rate='auto', init='random',
perplexity=58,
                    early_exaggeration=12, n_iter=5000,
n_iter_without_progress=1000).fit_transform(df)

```

```
    return X_embedded[:, 0], X_embedded[:, 1]
```

Listing 4.11: tSNE visualisation

Chapter 5

Results

I Data retrieval

At the time of the research, around 8000 repositories are meeting our criteria for retrieving the repository. From it, we managed to retrieve 7267 repositories.

Due to simplicity and consistency reasons, we decided to retrieve the only default branch of each repository. We assume that this is the stable version of the project.

From these 7267 repositories only 3561 had any workflows. So, only 49 percent of repositories in our sample has any workflow.

While 3561 repositories have a history of workflows, only 2864 have actual pipeline configurations. However, it is possible that these repositories have some pipeline in their other branches.

To sum up, our final dataset, which we are planning to analyze with machine learning methods, contains 2864 samples. All samples have a full set of features.

In the Fig. 1 we show the histogram of stars in each repository. We excluded some high starred repositories for better visualization. From this graph we can see that most of repositories have less than 1500 stars.

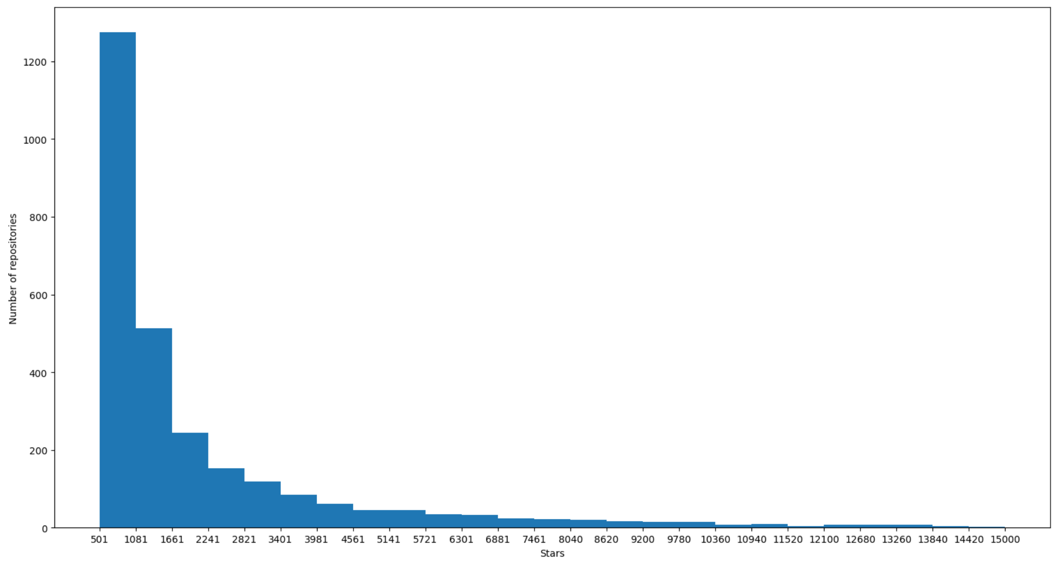

Fig. 1. Stars in each repository.

After analyzing workflows, we identified the most popular tools for testing, linting, coverage measuring, document testing and security frameworks among our dataset.

TABLE V
Most popular tools in our dataset

Tool category	Tools
Testing frameworks	pytest, brownie, tox, unittest, doctest, nox, nbmake
Linters and formatters	ruff, flake8, isort, black, eslint, pyupgrade, pylint, yapf, pycodestyle, mypy
Code coverage tools	coverage, codecov, coveralls
Documentation tools	codespell, pandoc, freeze, graphviz, mkdocs, sphinx
Security tools	safety, bandit, codeql

II Dataset analysis

In Fig. 2 we show resulted clusters. They resulted to be relatively homogeneous, with one bigger than others.

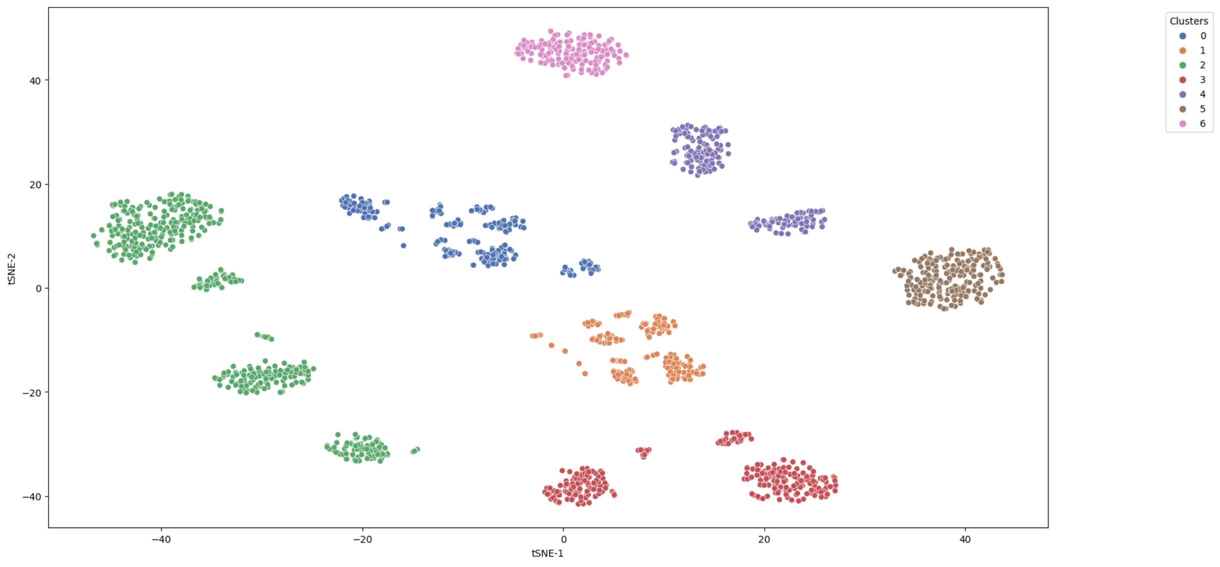

Fig. 2. Resulted clusters.

To better interpret the results, we will take a look at SHAP beeswarm graphs. These graphs show the contribution of each feature value to the clustering result. Color shows the value, and position on x axis shows contribution.

Also, we include a figure with histogram of some repository metrics.

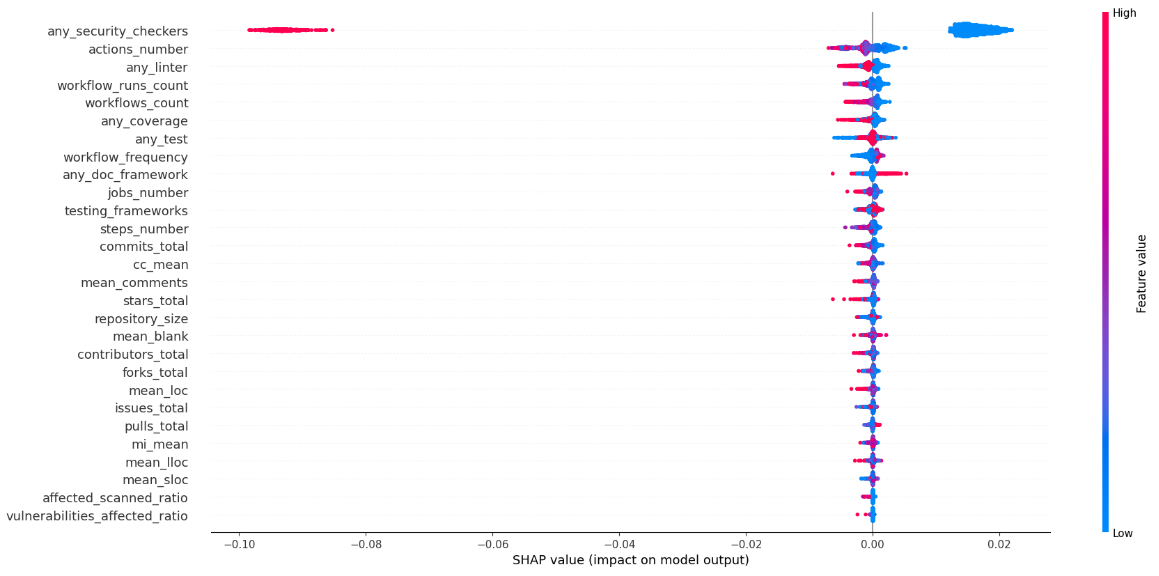

Fig. 3. Beeswarm graph of cluster 1.

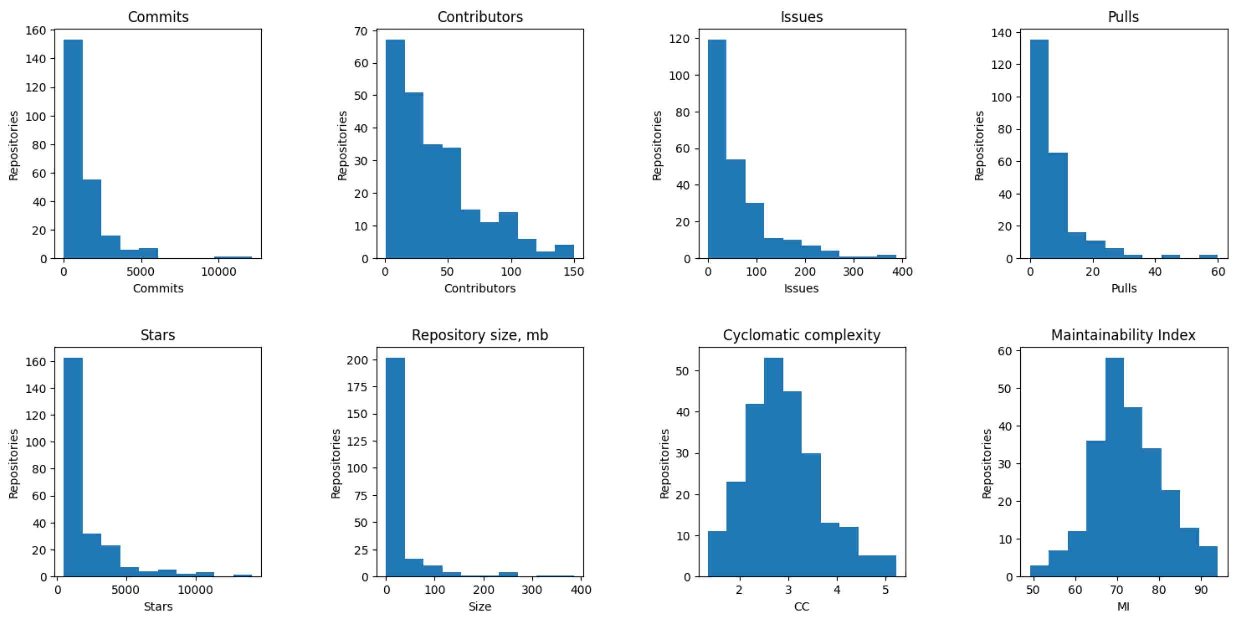

Fig. 4. Histograms of some cluster 1 metrics.

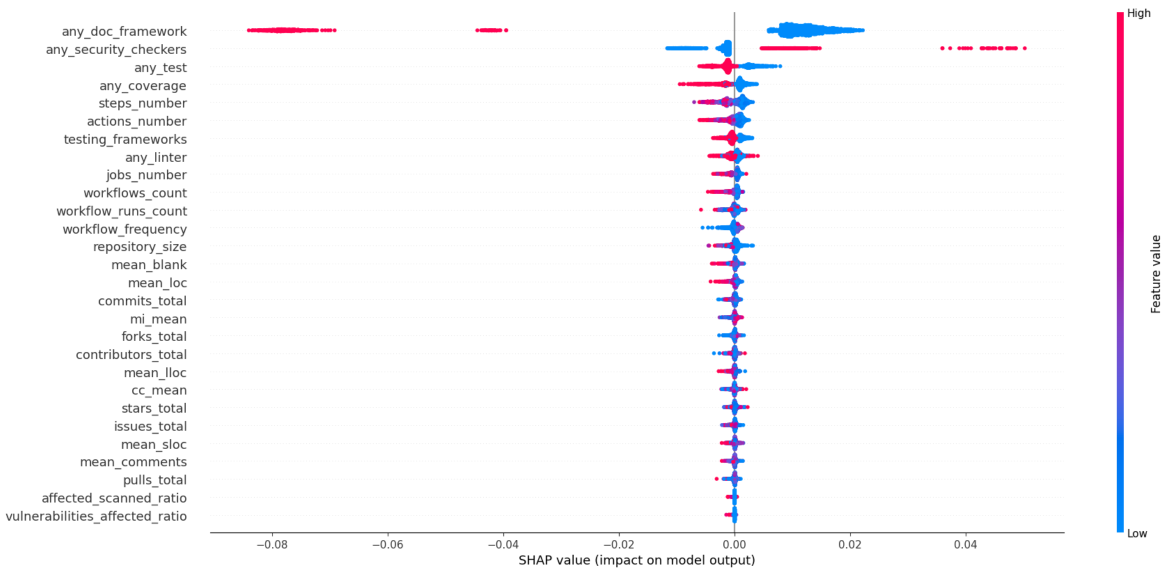

Fig. 5. Beeswarm graph of cluster 2.

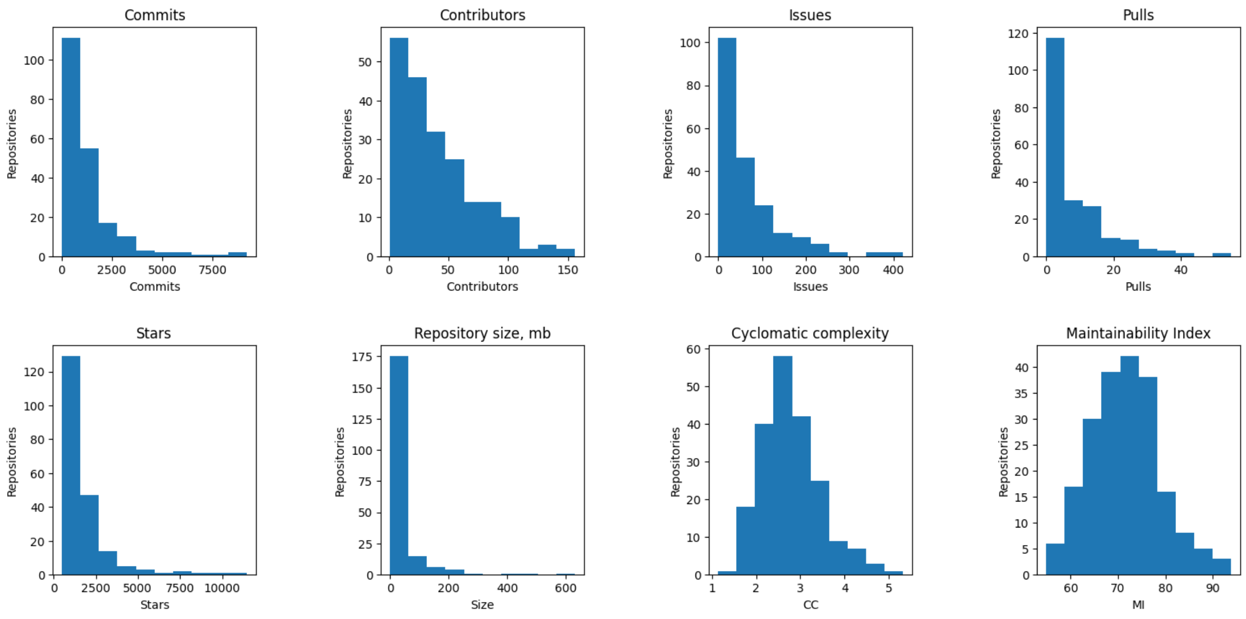

Fig. 6. Histograms of some cluster 2 metrics.

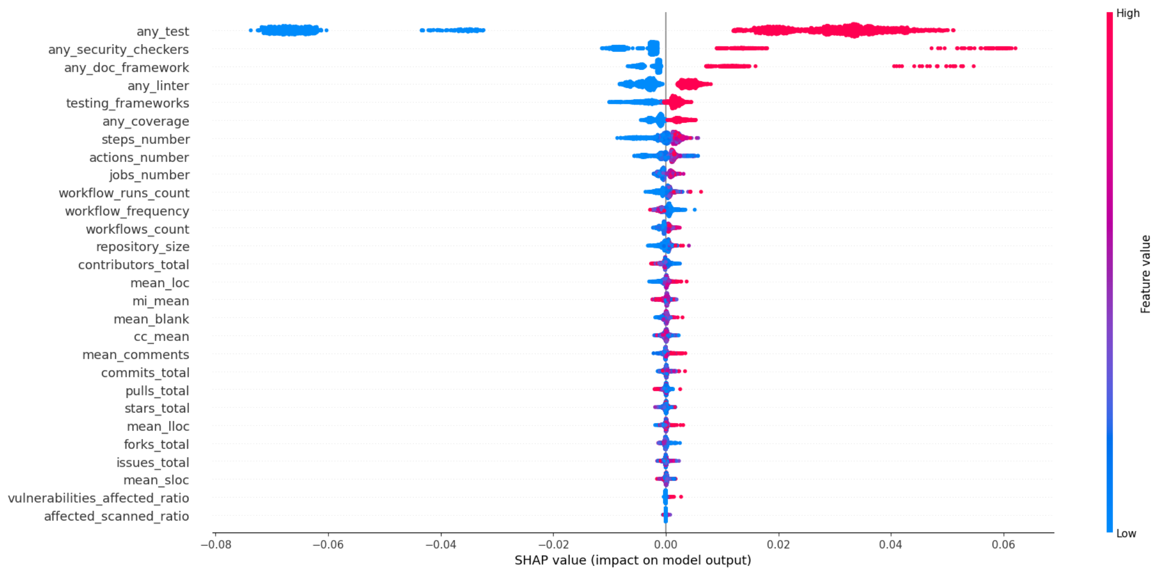

Fig. 7. Beeswarm graph of cluster 3.

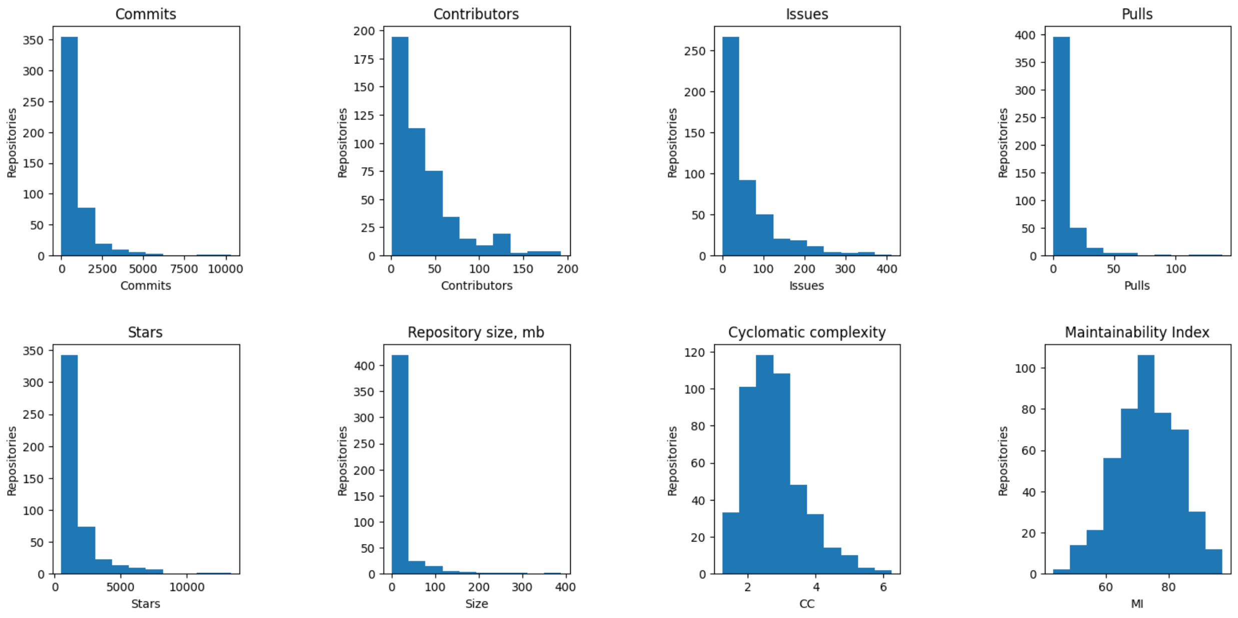

Fig. 8. Histograms of some cluster 3 metrics.

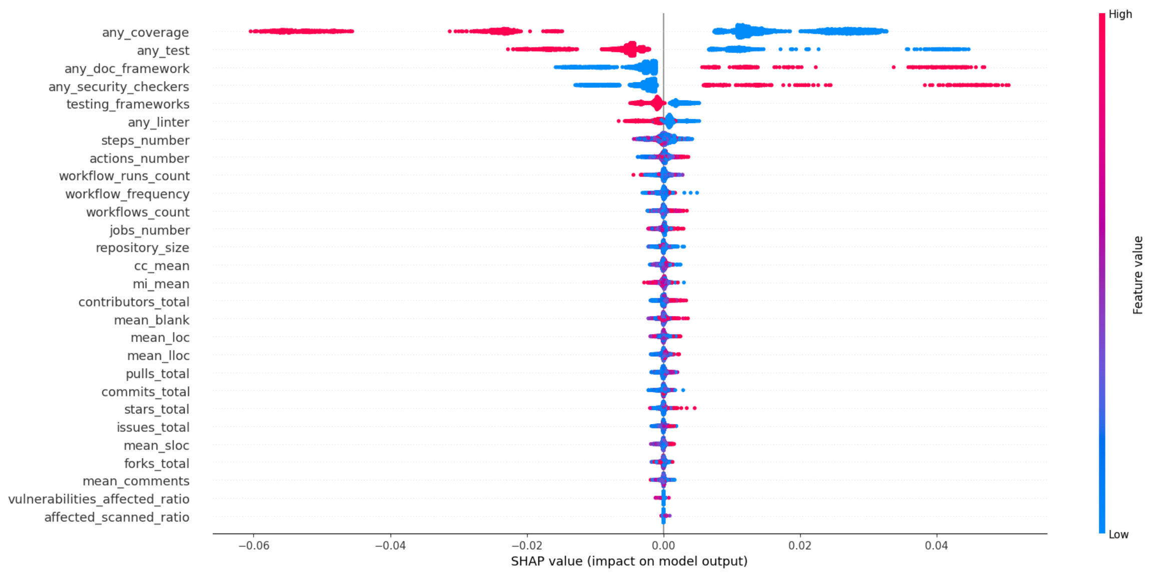

Fig. 9. Beeswarm graph of cluster 4.

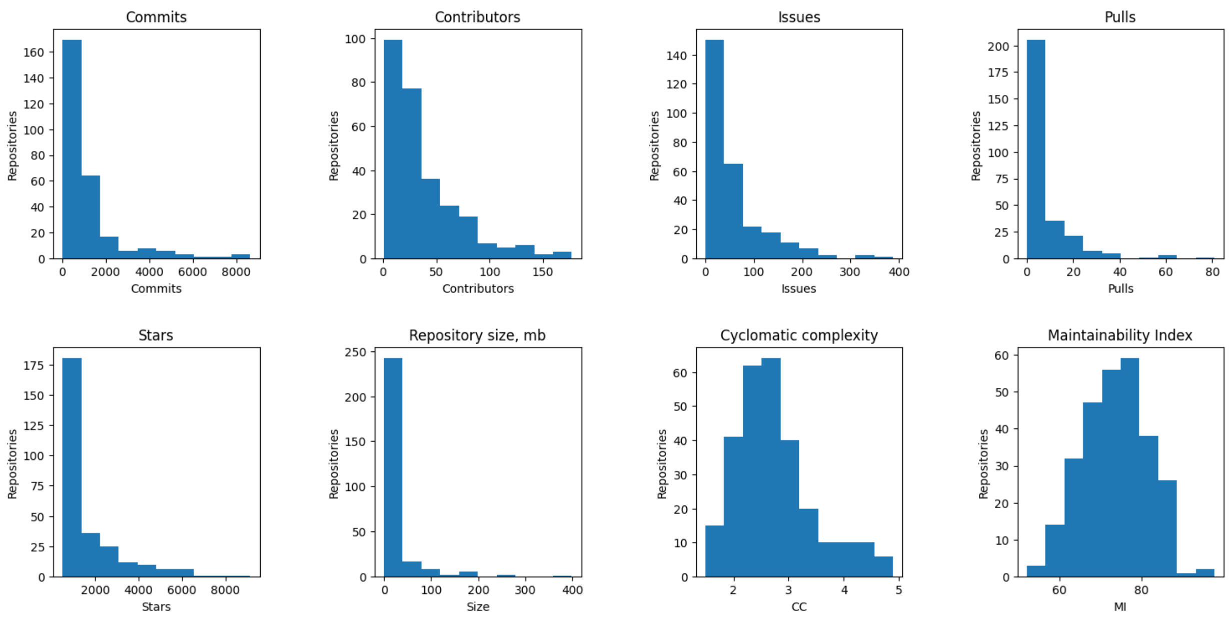

Fig. 10. Histograms of some cluster 4 metrics.

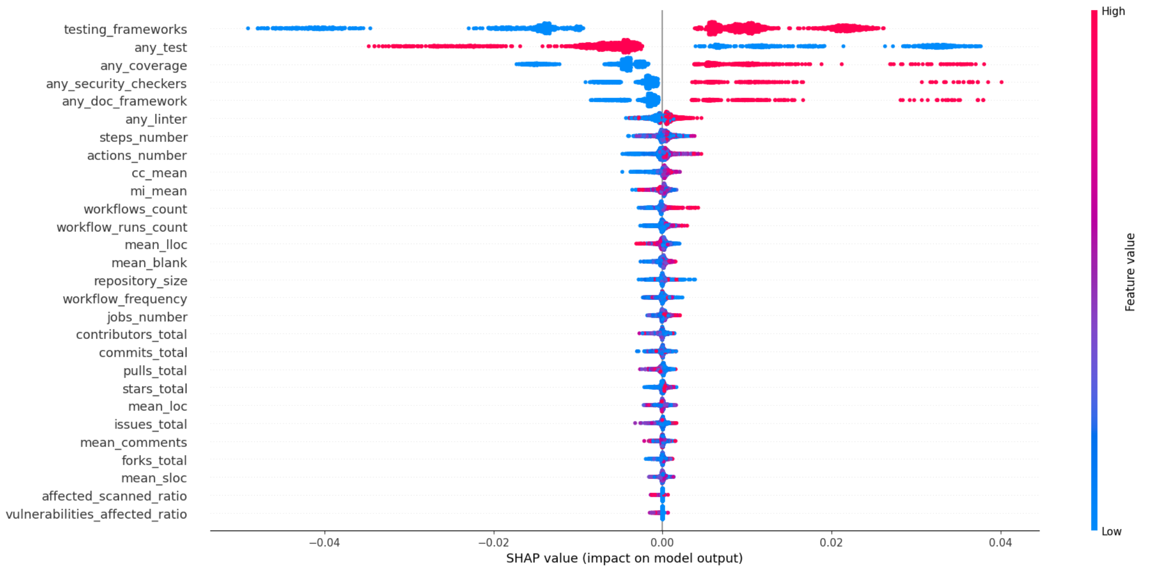

Fig. 11. Beeswarm graph of cluster 5.

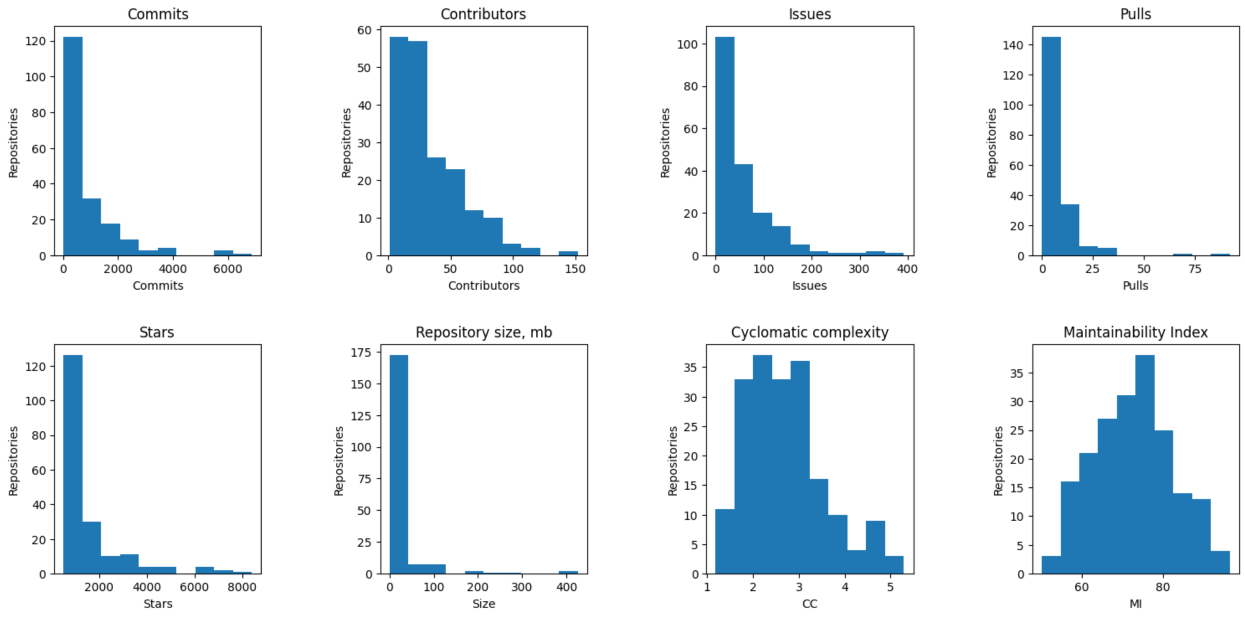

Fig. 12. Histograms of some cluster 5 metrics.

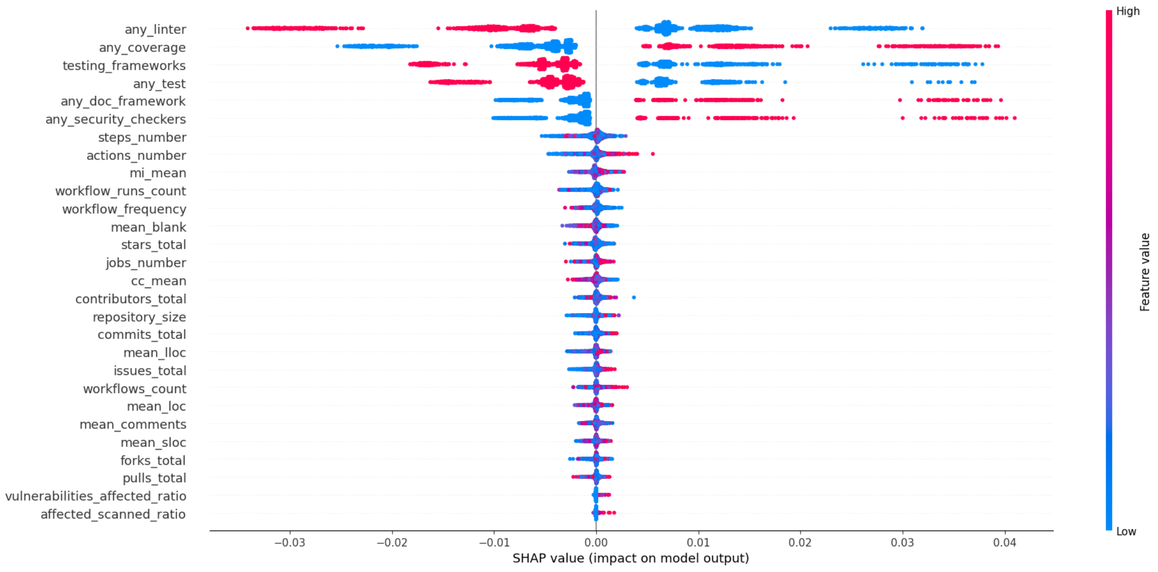

Fig. 13. Beeswarm graph of cluster 6.

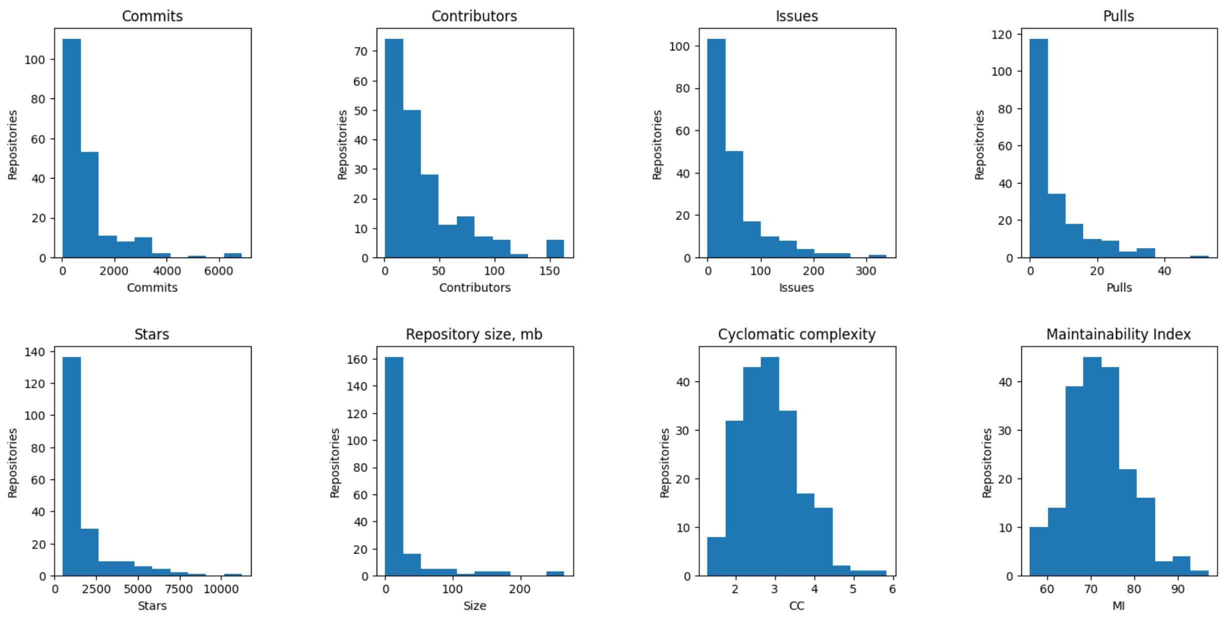

Fig. 14. Histograms of some cluster 6 metrics.

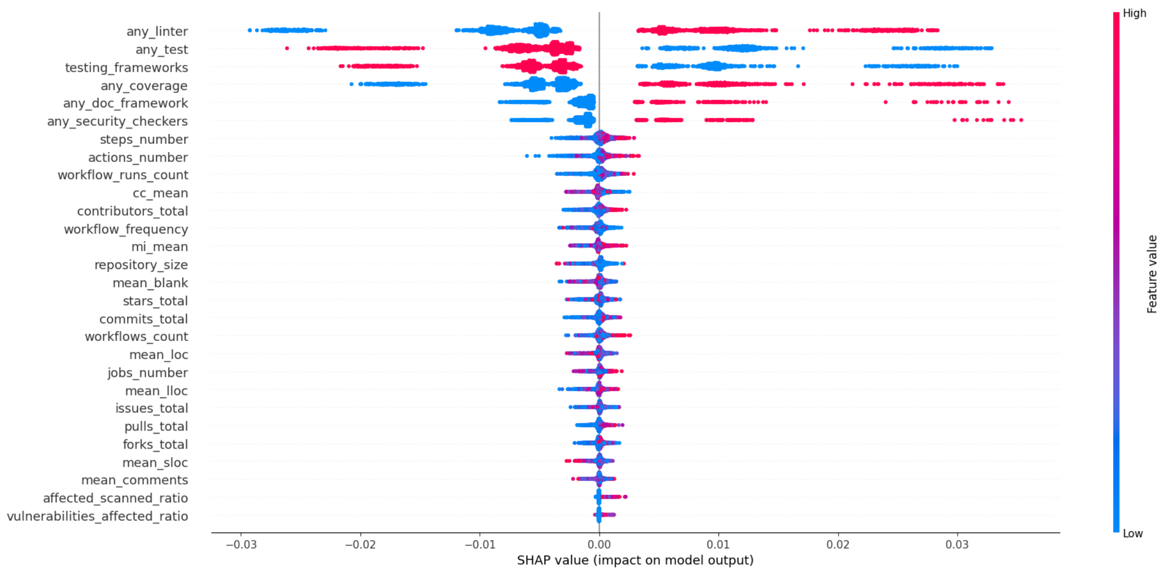

Fig. 15. Beeswarm graph of cluster 7.

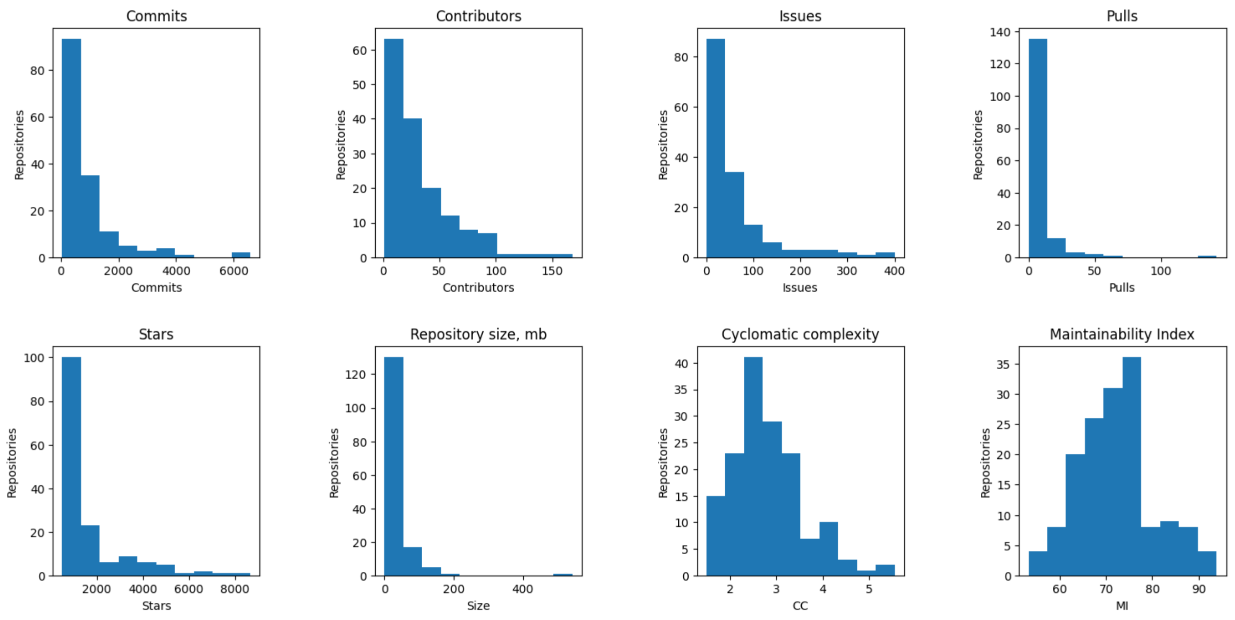

Fig. 16. Histograms of some cluster 7 metrics.

Finally, we include a table with mean values of the selected metrics:

TABLE VI
Some metrics results, cluster average

Cluster	1	2	3	4	5	6	7
Commits	1332.59	1274.4	834.80	1123.04	916.96	959.03	871.42
Contributors	40.35	40.9	37.06	38.03	33.04	36.08	32.11
Issues	61.53	68.49	57.85	56.97	55.86	49.01	59.15
Pulls	6.85	8.03	7.68	7.53	7.03	7.01	7.55
Stars	2025.6	1699.67	1640.77	1597.68	1504.75	1734.54	1619.47
Repository size, mb	24.89	34.56	15.17	19.99	20.89	20.15	15.34
CC	2.9	2.78	2.78	2.74	2.72	2.89	2.82
MI	72.81	71.55	73.27	73.63	72.91	71.92	72.4

Chapter 6

Discussion

I Data retrieved

We discovered that even among popular repositories only about half use GitHub's built-in pipeline tool. Note that it does not necessarily mean that these repositories do not use any sort of pipeline at all - it is possible that they use other services (GitLab, GitBucket, etc.) for the pipeline. But, as these pipelines may have different format due to other tools used, or they may be not accessible. Therefore, we assume that these repositories do not have a CI\CD pipeline.

Our clustering results show that among the most impactful metrics are the tool presence metrics. The reason for it may be in boolean type of these metrics: they may outweigh other numerical metrics, even with z-score normalization.

On the other hand, maintainability metrics did not contribute much to the clustering result. The most contributed metrics are Maintainability Index and Cyclomatic complexity, but even they differ marginally among clusters.

Metrics regarding dependency vulnerabilities also do not contribute much to the result. The reason for this may be too sparse data - vast majority of repositories either do not contain any vulnerable dependencies, or we did not find any dependencies at all.

To apply this model to our program, we need to classify the projects into several clearer categories. To do this, we need to look deeper into each cluster.

For cluster 1, we determined that it uses mostly only documentation checking tools, so it has fewer workflows and fewer workflow configurations. However, this cluster has the most commits, contributors, and stars of any cluster. Although these repositories use fewer tools in their workflows, they seem more mature.

In the second cluster, we determined that the repositories in it use the workflow mostly for security checks and occasionally for linking. It also has a lot of commits and contributors, but the number of stars is average. But it has the largest average repository size.

Since Cluster 1 and Cluster 2 have common features (high popularity and low usage of quality assurance tools), we decided to classify them into the same group of "mature" projects.

Cluster 3 differs significantly from the previous two. It has the lowest average number of commits and repository size, while using all the tools we identified. Similar to cluster 3 is cluster 7: they use almost all the tools we identified. Thus, we will classify them as the second group of "young" projects.

All other clusters we assign to the third "mixed" group. They use different tools and have different results in terms of commits, stars, and project size.

So, such three groups defined:

a) "Mature" group - Clusters 1 and 2 b) "Young" group - Clusters 3 and 7 c) "Mixed" group - Clusters 4, 5 and 6

These groups will be used in our program to classify input repository. [1]

[1]This program and the instruction to use it stored in GitHub repository: https://github.com/Ziucay/thesis-project

II Limitations

Firstly, we want to discuss repository selection of our work. Some of the metrics in our dataset (maintainability code metrics, some data from GitHub API) do not differ a lot, thus not contributing significantly to the clustering. We think that inserting projects of lower popularity can lead to more interesting results.

Secondly, a collection of information regarding repository dependencies could be done better. The issue with requirements file is there is no standard for naming it: most projects have it in the 'requirements.txt' file, meanwhile some have multiple such files for various reasons, and some do not have any. For those repositories that do not have (or we have failed to find) requirements file we assume that they have no dependencies at all. Following our assumption, they therefore do not have vulnerabilities, which is not necessarily true. In the result, we obtained few columns of data with mostly zeros. For future works, it is important to find better ways of scraping such data, or simply use technology stack with more standardized dependency standards (for example, package.json in node).

Lastly, our way of scraping data from workflows is far from ideal, for a few reasons. The first reason is that in our work we do not check whether identified tools work or not, we simply assume that if we have found the keyword, then the project uses these tools in some way. For a dataset with rather popular and active projects this approach is unlikely to cause any significant problems, but a tool with similar implementation may easily lead to false positives. The second reason is that our data may be biased towards false negatives. There can be numerous ways to use the tools we identified while not being found by our code. For example, a project may have a script using some programs, or it can have custom GitHub action, or even custom tools.

III Future work

Taking into account points from the previous section, we propose some improvements of our work to future ones.

Firstly, more data can be used. For example, older or less popular projects can be added to the repository list. Our data contains rather noisy data, and less restrictive criteria can lead to more interesting results.

Secondly, as we used only portion of the data contained in workflow configuration files, workflows can be analyzed much deeper: retrieve platform used for pipeline execution, automatic deploy capabilities, branches on which pipelines are triggered, and so on.

Lastly, in this work we only used HDBSCAN to cluster our dataset, we did not attempt to make predictions based on the model we resulted in. This model can be used in tools for scoring or ranking projects automatically.

With growing popularity of DevOps practices, we believe that analysing pipelines can improve quality of the project, and proposed metrics can be used for new quality measuring tools.

Chapter 7

Conclusion

As a result, we determined which quality characteristics could be measured in open source repositories. We focused on measuring deployability, security, and maintainability of projects. For each of these characteristics, we explored which metrics could be used to measure them. We also investigated for which metrics we already have tools in place and which are yet to be implemented. For several of the most popular metrics, we successfully found existing tools and implemented our own way of calculating workflow metrics.

We then applied the HDBSCAN clustering algorithm to find out how these metrics relate to each other on real data. Next, we used SHAP values to interpret and explain the results of our clustering. This gave us information about how these clusters were constructed and why they differed from each other.

This allowed us to make a program that could assign any Python repository to the appropriate group and provide the user with different information about the required project. In the end, we pinpointed the weaknesses of our research and suggested several ways to fix them.

Bibliography cited

[1] ISO/IEC 25010, *ISO/IEC 25010:2011, systems and software engineering systems and software quality requirements and evaluation (square) system and software quality models*, 2011.

[2] C. Peters, D. Farley, D. Villalba, *et al.*, “2022 accelerate state of devops report,” Tech. Rep., 2022. [Online]. Available: `https://cloud.google.com/devops/state-of-devops`.

[3] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*, 1st. IT Revolution Press, 2018, ISBN: 1942788339.

[4] I. Chowdhury, B. Chan, and M. Zulkernine, “Security metrics for source code structures,” in *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, ser. SESS ’08, Leipzig, Germany: Association for Computing Machinery, 2008, pp. 57–64, ISBN: 9781605580425. DOI: `10.1145/1370905.1370913`. [Online]. Available: `https://doi.org/10.1145/1370905.1370913`.

[5] D. Mellado, E. Fernández-Medina, and M. Piattini, “A comparison of software design security metrics,” in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ser. ECSA

'10, Copenhagen, Denmark: Association for Computing Machinery, 2010, pp. 236–242, ISBN: 9781450301794. DOI: 10.1145/1842752.1842797. [Online]. Available: https://doi.org/10.1145/1842752.1842797.

[6] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 1–7, Sep. 2011, ISSN: 0163-5948. DOI: 10.1145/2020976.2020983. [Online]. Available: https://doi.org/10.1145/2020976.2020983.

[7] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[8] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 131–142, ISBN: 9781605580500. DOI: 10.1145/1390630.1390648. [Online]. Available: https://doi.org/10.1145/1390630.1390648.

[9] M. Dagpinar and J. Weber, "Predicting maintainability with object-oriented metrics - an empirical comparison," Dec. 2003, pp. 155–164, ISBN: 0-7695-2027-8. DOI: 10.1109/WCRE.2003.1287246.

[10] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings Conference on Software Maintenance 1992*, 1992, pp. 337–344. DOI: 10.1109/ICSM.1992.242525.

[11] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining*, J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–172, ISBN: 978-3-642-37456-2.

[12] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *kdd*, vol. 96, 1996, pp. 226–231.

[13] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," *SIGMOD Rec.*, vol. 28, no. 2, pp. 49–60, Jun. 1999, ISSN: 0163-5808. DOI: `10.1145/304181.304187`. [Online]. Available: `https://doi.org/10.1145/304181.304187`.

[14] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: `http://jmlr.org/papers/v9/vandermaaten08a.html`.

[15] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.