Masaryk University
Faculty of Informatics

Risk Assessment Model for Open Source Software Projects in GitHub

Master's Thesis

Samuel Macko

Brno, Spring 2021

Masaryk University
Faculty of Informatics

Risk Assessment Model for Open Source Software Projects in GitHub

Master's Thesis

Samuel Macko

Brno, Spring 2021

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Samuel Macko

Advisor: RNDr. Josef Spurný

Acknowledgements

I would like to thank my supervisor RNDr. Josef Spurný for his help. I would further like to thank doc. Mouzhi Ge, Ph.D. for his guidance and tips, mostly in the earlier stages of the thesis development. Huge thanks also belongs to Ing. Fridolín Pokorný for his supervision regarding mostly technical parts of the thesis.

Abstract

Today, more and more technologies depend on open-source projects. It is increasingly important to assess the level of future development of such dependencies. Goal of this thesis is to develop a machine learning model to estimate a probability that a repository will continue being actively developed.

I develop a set of tools used for compiling a dataset used for training and testing of machine learning models. I proposed a set of features used to evaluate projects, as well as sample labeling strategy.

In the last part I train and evaluate multiple machine learning models and analyze the results.

Keywords

classification, GitHub, machine learning, open-source

Contents

Introduction **1**

1 Previous work **3**
1.1 Identifying Unmaintained Projects in GitHub 3
1.2 Dataset 3
1.3 Features 4
1.4 Models 4
1.5 Results 5

2 Features **7**
2.1 Popularity 7
2.2 Community growth 9
2.3 Community activity 11
2.4 Management 12
2.5 Maintenance 13
2.6 Maturity 15
2.7 Development rate 16
2.7.1 Fork and Pull Request (PR) workflow 17
2.7.2 Clone and Push workflow 18
2.7.3 Features regarding a contribution workflow . . 18

3 Implementation **21**
3.1 Data collection 21
3.1.1 Tools used for data collection 21
3.1.2 Searching for suitable repositories 24
3.1.3 Developed applications 26
3.1.4 Features computer 27
3.1.5 Features of crawler and computer 27
3.2 Dataset 29
3.3 Removing highly correlated features 29
3.4 Dataset preprocessing 29
3.5 Samples labeling 32
3.6 Training and evaluation 33

4 Results **37**

5 Summary **43**

5.1 Future work 44

Bibliography **45**

List of Tables

2.1 Features. 8

3.1 Parameters grid for RandomForest. 35
3.2 Parameters grid for SVM. 36
3.3 Parameters grid for kNN. 36
3.4 Parameters grid for MLP. 36

4.1 Parameters of the best RandomForest model. 39
4.2 Parameters of the best SVM model. 40
4.3 Parameters of the best kNN model. 40
4.4 Parameters of the best MLP model. 41

List of Figures

1.1 *Features used to identify projects' level of maintenance.* 4

2.1 *Magnetism.* 10
2.2 *Stickiness.* 11
2.3 *Shorter development time.* 16
2.4 *Longer development time.* 16
2.5 *Scheme of two types of GitHub workflows.* 17

3.1 *Data gathering architecture.* 22
3.2 *Container architecture.* 22
3.3 *Correlation heatmap.* 30
3.4 *Scheme of preprocessing.* 32
3.5 *Label counts of train data.* 34
3.6 *Label counts of test data.* 34

4.1 *Results.* 38

Introduction

Open source software (OSS) is a type of computer software in which source code is released under a license in which the copyright holder grants users the rights to use, study, change, and distribute the software to anyone and for any purpose [1]. Open source projects have an increasing relevance in modern software development. For example, many critical software systems are currently available under open source licenses, including operating systems, compilers, databases, and web servers. Similarly, it is common nowadays to depend on open source libraries and frameworks when building and evolving proprietary software [2].

GitHub[1] is the world's largest collection of open source software, with more than 56 million users and 100 million projects. It offers the distributed version control and source code management functionality of *Git*[2] and some more features.

Aim

As more and more technologies rely on open source software, it is important to put a lot of attention into choosing such dependencies. If an open source repository ceases to be actively developed and becomes obsolete, it can cause serious issues for all of the other projects that depend on this repository. It would be helpful to be able to determine if a *GitHub* hosted repository has a high chance of being actively developed in the future, or if it will likely die off. The aim of this thesis is precisely that, to develop a machine learning[3] model to estimate a probability of survival of *GitHub* repositories.

This thesis was developed in collaboration with *Red Hat*[4].

Key contributions

I developed applications 3.1.3 for gathering relevant data that can be used to construct an input dataset for machine learning models 3.6. I

1. See `https://github.com/`
2. See `https://git-scm.com/`
3. See `https://en.wikipedia.org/wiki/Machine_learning`
4. See `https://www.redhat.com/en/global/czech-republic`

also proposed and tested a set of features 2 for determining project's survivability.

Contents

Chapter 1 goes over a research paper aiming to solve a similar problem as this thesis. In the chapter 2, I propose features, go over them, and give an explanation of what they measure and why I decided to include them. Chapter 3 describes the process of gathering and preparing data, as well as methods used for training and evaluating machine learning models. Then, the chapter 4 provides a summary and analysis of the achieved results. The final chapter, chapter 5, provides a summary of the thesis.

1 Previous work

While there is a lot of research being done on open source projects, I was able to find only one work whose goal was similar to the goal of this thesis. This chapter goes thorough it, analyses methods its authors used and results they obtained.

1.1 Identifying Unmaintained Projects in GitHub

Coelho et al. [2] proposed an approach to identify GitHub projects that are not actively maintained. Their goal is to detect unmaintained repositories as soon as possible.

Ten models were trained for the identification. The best model was then validated by means of survey with the owners of projects classified as unmaintained.

1.2 Dataset

Initial dataset was the top 10 000 most starred repositories on GitHub. Three strategies were used to filter out not suitable ones:

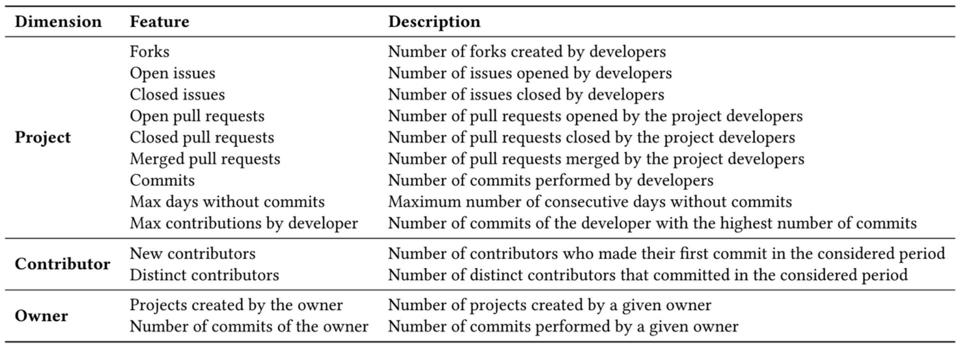
1. Remove repositories with less than 2 years between the first and the last commit. Models need historical data. 2 810 repositories were removed this way.

2. Remove repositories with null size, measured in lines of code. Typically projects implemented in non-programming languages. 331 repositories were removed.

3. Remove 74 non-software repositories. These are identified by searching for the topics *books* and *awesome-lists*.

Stripped down version of the original dataset contains 6 785 repositories. In the next step in a data preparation process, 1 002 repositories were selected from the filtered dataset and split into two categories.

1. Active: 754 projects that have at least one release in the last month.



Dimension	Feature	Description
Project	Forks	Number of forks created by developers
	Open issues	Number of issues opened by developers
	Closed issues	Number of issues closed by developers
	Open pull requests	Number of pull requests opened by the project developers
	Closed pull requests	Number of pull requests closed by the project developers
	Merged pull requests	Number of pull requests merged by the project developers
	Commits	Number of commits performed by developers
	Max days without commits	Maximum number of consecutive days without commits
	Max contributions by developer	Number of commits of the developer with the highest number of commits
Contributor	New contributors	Number of contributors who made their first commit in the considered period
	Distinct contributors	Number of distinct contributors that committed in the considered period
Owner	Projects created by the owner	Number of projects created by a given owner
	Number of commits of the owner	Number of commits performed by a given owner

Figure 1.1: Features used to identify projects' level of maintenance. Source: [2]

2. Unmaintained: 248 projects that were either: (1) explicitly declared as unmaintained by their principal developers or (2) archived.

1.3 Features

Figure 1.1 lists used features. These features do not refer to the whole history of the project, but only to the last *n* months, counting from the last commit. Each feature was collected in the interval of *m* months. Authors experimented with different combinations of *n* and *m*. They call these combinations *scenarios*. For each *scenario*, they used a clustering analysis to remove correlated features.

1.4 Models

Out of ten tested models, *RandomForest* achieved the best results. Two baselines were compared: #1 all projects are predicted as unmaintained and #2 random predictions. Models were validated using a 5-fold cross validation. Six metrics were used to evaluate a performance of classifier:

1. precision

2. recall

3. F-measure
4. accuracy
5. AUC
6. Kappa

1.5 Results

The best results – a precision of 80% and a recall of 96% – were achieved when features were collected for the time period of 2 years in the interval of 3 months. *RandomForest* also produces a measure of the importance of the features. The most important features are:

1. number of commits
2. maximal number of days without commits
3. maximal contributions by a developer
4. number of closed issues

2 Features

GitHub repositories are analyzed based on seven different areas. Each area considers a different aspect of a repository. I combined them and gave models 3.6 an overview of repository qualities across several dimensions.

The idea is that a repository that is lacking in some of the areas can still excel in others. Higher number of analyzed features from different categories might result in a better classification accuracy.

These areas and their corresponding sections are:

- Popularity 2.1
- Community growth 2.2
- Community activity 2.3
- Management 2.4
- Maintenance 2.5
- Maturity 2.6
- Development rate 2.7

List of features can be seen in the table 2.1. All of these features are fed as an input for the learning algorithms, they are only grouped into these areas for the better understandability.

2.1 Popularity

Features measured in this category:

- stargazers_count: number of stars
- watchers_count: number of watchers
- forks_count: number of forks

Table 2.1: Features.

Features
pulls_count_open
pulls_count_closed
issues_count_open
issues_count_closed
commits_count
branches_count
releases_count
owner_type
watchers_count
forks_count
stargazers_count
development_time
owner_account_age
avg_dev_account_age
has_test
has_doc
has_example
has_readme
owner_projects_count
owner_following
owner_followers
devs_followers_avg
devs_following_avg
commits_by_dev_with_most_commits
magnetism
stickiness
wealth
last_commit_age

While not stated explicitly by GitHub, number of stars[1] and watchers[2] can be considered as a measure of a project's popularity.

GitHub *stars* are intended to be used primarily to bookmark a repository[3]. Users can also use it as a form of appreciation for the project, it is used as a proxy for a project popularity in several studies [3, 4, 5, 6], and even by GitHub[4]. Because of an ambiguity in terms of user's intentions to *star* a project, I do not consider sufficient to use it as the only popularity metric.

To *watch* a repository, means that the user is interested in getting notified on changes in that repository. This can indicate a stronger interest than a *star*.

Repository might be *forked* when a developer intends to contribute to it in some way, to add a feature, fix a bug, etc. These might not be the only reasons to fork a repository, however, high number of forks can still indicate high popularity. Developers are also more likely to fork a project in their preferred programming language [7].

2.2 Community growth

Features measured in this category:

- magnetism
- stickiness

Magnet projects are those that attract a large proportion of new contributors. *Magnetism* of a project is the proportion of contributors who made their first contribution in the time period under study who contribute to a given project [8].

In this project, magnetism is computed as a fraction of a number of *new* contributors and a number of *old* contributors.

1. See `https://docs.github.com/en/enterprise-server@2.22/github/getting-started-with-github/github-glossary#star`.
2. See `https://docs.github.com/en/github/getting-started-with-github/be-social#watching-a-repository`.
3. See `https://docs.github.com/en/github/getting-started-with-github/saving-repositories-with-stars`
4. See `https://github.com/trending`

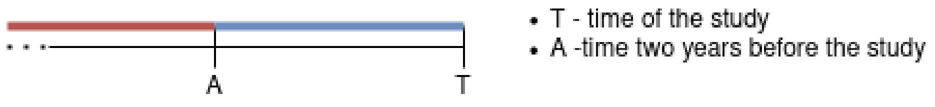


Figure 2.1: Magnetism. Source: own

Old contributors are contributors who committed to the project before a time threshold of two years. New contributors, on the other hand, are contributors who committed to the project only after the time threshold.

Figure 2.1 shows the difference between *new* and *old* contributors, T represents time of the study and A represents time two years before the study. Developers that contributed to the project before A, the red line, are considered *old* contributors. Developers that contributed to the project after A, the blue line, are considered *new* contributors.

Sticky projects are those where a large proportion of the contributors will keep making contributions in the time period the following and under study [8]. *Stickiness* in this project is calculated as a fraction of number of contributors that *sticked*, and a number of *new* contributors.

Contributors are considered *new* if they commit to the project within past two years. Contributors that *sticked* are those *new* contributors, who also committed to the project within the past year.

It is a proportion of contributors who worked on a given project in the period between one to two years before the study, who also continued to make contributions for the past year before the study.

Figure 2.2 shows the difference between *new* and *sticked* contributors, T represents time of the study, B represents time two years before the study and A represents time one year before the study. Developers that contributed to the project between B and T, red and blue lines, are considered *new* contributors. Developers that contributed to the project between A and T, the blue line, are considered *sticked* contributors.

While magnetism metric can indicate how good the project is at attracting new contributors, stickiness can show how good the project is at keeping them.

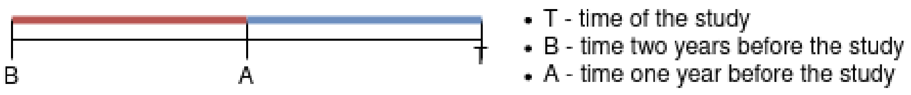

B
A
T
• T - time of the study
• B - time two years before the study
• A - time one year before the study

Figure 2.2: Stickiness. Source: own

2.3 Community activity

Features measured in this category:

- commits_by_dev_with_most_commits: number of commits by a contributor with the highest number of commits
- devs_followers_avg: average number of a repository contributor's followers
- devs_following_avg: average number of repositories followed by a repository contributors

Small minority of the most active project contributors tend to produce the most activity [9]. For this reason, it might be useful to consider the most active contributor. This can be done in several ways. Measures of activity can be: frequency of commits, code addition in commits, message frequency in *Pull requests*[5] and *Issues*[6] and others. In this project, the most active contributor is selected based on the number of commits.

The average number of followers of developers in the project and the average number of developers/projects followed by them can be seen as an approximation of social relations of project members. It turns out, that the project whose developers follow many others are generally more popular. This discovery results in a practical advice for projects: finding developers engaged in the community is good for the projects popularity and can be measured by a simple proxy quantity [10]. Having more popular contributors might improve the projects longevity.

5. See https://docs.github.com/en/enterprise-server@2.22/github/getting-started-with-github/github-glossary#pull-request
6. See https://docs.github.com/en/enterprise-server@2.22/github/getting-started-with-github/github-glossary#issue

Initially, there was supposed to be another feature in this area, *health*. In terms of OSS, it is defined as a indicative of three factors of how community activities are performed in a project:

- workrate of each contributor
- attractiveness of new contributors to a community
- active retention of experienced members

Labor is measured as community contributions within the projects. Only source code changes as contributions are considered (i.e., comments made by contributors are ignored) [11].

Although this metric is interesting and potentially very insightful, its computation requires a considerable amount of GitHub API calls[7] to which I have a limited access. Compared to most of the other features, it is also resource heavy, which is not suitable for the project of this scale. For these reasons, I decided to drop this feature, it could be added in the future work.

2.4 Management

Features measured in this category:

- owner_type: type of the repository owner, can be either *organization* or *personal*
- owner_projects_count: number of repository owner's repositories
- owner_following: number of repositories that the repository owner follows

GitHub offers two types of accounts[8]: *personal* and *organization*. Personal accounts are intended for individual users while organization accounts are for groups of people to collaborate across many projects

7. See `https://docs.github.com/en/rest`
8. See `https://docs.github.com/en/github/getting-started-with-github/types-of-github-accounts`

at once. Developers are more likely to fork a repository owned by an *organization* type owner [7].

Rationale for the number of project owner's other projects is that the the high number of them might negatively affect the amount of attention each of them gets [2].

Number of repositories followed by the project's owner, the out-degree, tends to be linked with more forks of the project [7].

2.5 Maintenance

Features measured in this category:

- has_test: whether a repository contains a dedicated test code
- has_doc: whether a repository contains a documentation
- has_readme: whether a repository contains a README file
- has_example: whether a repository contains an usage examples
- issues_count_open: number of *open* issues
- issues_count_closed: number of *closed* issues

Test folder provides the test code for quality assurance, and may help others to test the code they contribute [12]. This makes contributing easier and thus may attract new developers. Tests are often run periodically in an automated fashion as a part of CI/CD workflow[9]. Such practice greatly improves bug detection and can improve overall project quality. In this project, a file is considered a test file if it is in a folder named *test*, *tests*, *t* or *spec*. File names are the same as in [12]. Presence of a dedicated test folder[10] is shown to be positively correlated with increased number of project forks [12].

It is shown that presence of a proper project documentation is positively associated with the usability improvement of an OSS project [13]. Proper documentation increases understandability and learnability of a software. In this way, it may help with the project maintenance,

9. See `https://www.redhat.com/en/topics/devops/what-is-ci-cd`
10. Example `https://github.com/jekyll/jekyll`

as new developers are able to join existing community more smoothly. Project documentation can be provided in several ways. For smaller projects, it might be sufficient to cover all of the documentation in a *README* file. Larger projects often require several dedicated documents. These documents might be incorporated into the GitHub repository[11], or they can reside outside of the repository, for example on the project's web page[12]. Detecting a documentation living outside of the repository would be difficult to do in an automated manner. This feature looks for a directory named *doc*, *docs*, *document* or *documents*. String are the same as in [12].

As mentioned above, *README*[13] file may contain a project's documentation. That is usually the case for very small projects. It is typically the first thing an user sees in the repository and generally contains information about what the project does, who are its intended users, use cases and so on. Detecting a *README* file is a straight forward process, we look for a file named *README* in any case and with any extension.

Projects may contain an *examples* folder, containing various usage examples[14]. Presence of an examples folder may indicate the desire to attract an external attention [12], as these examples are helpful especially for the new users. In terms of this project, detecting an *examples* folder means searching for the folder named either *example* or *examples*.

The use of these folders suggests a practice of following conventional folder structure in these projects [12].

GitHub issues[15] can be used for bug reporting, requesting features[16], and others. Creating and commenting on an issue is one of the forms of communication between users of the project and its developers. It is also often a starting point for new contributors, especially issues labeled as *good first issues* [14]. Amount of open and closed issues

11. Example `https://github.com/electron/electron`
12. Example `https://github.com/facebook/react`
13. See `https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/about-readmes`
14. Example `https://github.com/kubernetes/client-go`
15. See `https://docs.github.com/en/enterprise-server@2.20/github/getting-started-with-github/github-glossary#issue`
16. See `https://guides.github.com/features/issues/`

might be a good indicator of two things: (a) project's usage and (b) level of involvement of the developer team.

2.6 Maturity

Features measured in this category:

- development_time: time between the first and the latest commit
- avg_dev_account_age: average of a repository contributor's accounts age in days
- owner_account_age: age of repository owner's account in days
- owner_followers: number of a repository owner followers

Repository *development time* is measured in days. The rationale: projects that are being developed over longer periods of time and are still active probably also have higher chance of continuing to be active in the future. It is also an important feature to separate suitable repositories for this study from the rest. Because of many features that require historical data, only projects developed for more than 730 days are being considered.

Note that time between the first and the last commit is not necessarily the same thing as the repository age. Difference can be demonstrated on the example: repository illustrated in the figure 2.3 was being actively developed for less than a year. Other repository, illustrated in the figure 2.4 was being actively developed for almost 2 years, the most recent commit added not very long ago. Red line in the figures represents development of the projects. Ages of both of these repositories are two years, however, second repository 2.4 clearly has a higher chance of continuing to being actively developed, than the first one 2.3.

Having newer developers in the project is positively correlated with its popularity. This phenomenon can probably be explained by the fact that programmers may join GitHub in order to contribute to attractive projects [10]. As mentioned in the section 2.1, gain in popularity can attribute to the project's longevity.

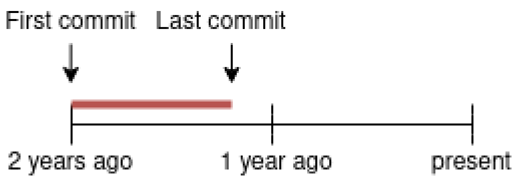


Figure 2.3: Shorter development time. Source: own

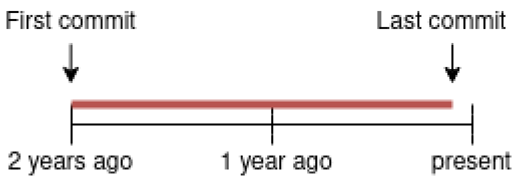


Figure 2.4: Longer development time. Source: own

More popular repository owners tend to have older GitHub accounts [7]. Popularity in this context means that repositories of these owners are being forked more often. Reason for this may be that older project owners (project owners with older GitHub accounts) are more established in the community or own more projects that are potentially successful.

Related to project owner's popularity is also a number of owner's followers. More popular owners usually have more followers [10].

2.7 Development rate

Features measured in this category:

- commits_count: number of commits
- pulls_count_open: number of *open* pull requests
- pulls_count_closed: number of *closed* pull requests
- branches_count: number of branches
- releases_count: number of releases
- wealth: weighted measure based on Pull Requests

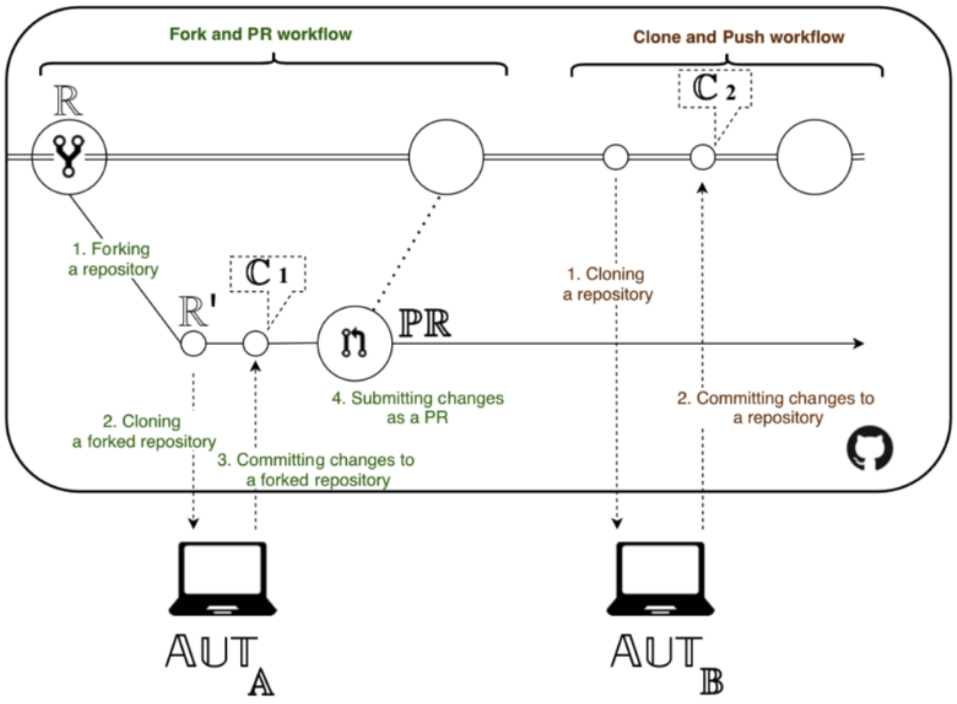

Fork and PR workflow
Clone and Push workflow
ℝ
ℂ 2
1. Forking a repository
ℂ 1
ℝ'
ℙℝ
1. Cloning a repository
4. Submitting changes as a PR
2. Committing changes to a repository
2. Cloning a forked repository
3. Committing changes to a forked repository
𝔸𝕌𝕋 𝔸
𝔸𝕌𝕋 𝔹

Figure 2.5: Scheme of two types of GitHub workflows. Source: [15]

- last_commit_age: age of the last commit in days

Features *number of commits*, *number of open pull requests*, *number of branches* and *age of the last commit* are closely related. To better understand their importance for the study, it will be helpful to explain the standard workflow of contribution to a GitHub repository.

Figure 2.5 illustrates two types of GitHub contribution workflows: a) Fork and PR workflow and b) Clone and Push workflow.

ℝ denotes a repository, ℂ set of commit changes and ℙℝ represents a pull request [15].

2.7.1 Fork and Pull Request (PR) workflow

1. Fork a repository: A fork is a copy of a repository. Forking a repository allows users to freely experiment with changes without affecting the original project. Most commonly, forks are used to either propose changes to an existing repository or

to use that repository as a starting point for a new project[17]. As shown in the figure 2.5, $\mathbb{AUT}_{\mathbb{A}}$ makes a fork of repository $\mathbb{R}$. We now call this repository $\mathbb{R}'$ [15].

2. Clone a repository: Cloning a repository pulls down a full copy of all the repository data that GitHub has at that point in time, including all versions of every file and folder for the project[18]. This creates a local copy of a remote repository, in which the user can make changes. As shown in the figure, $\mathbb{AUT}_{\mathbb{A}}$ clones $\mathbb{R}'$ onto their local computer, become a clone repository $\mathbb{R}'_{\mathbb{C}}$ [15].
3. Commit changes to a cloned repository: user can make arbitrary changes to a local copy of a forked repository. User can then apply a set of these commit changes $\mathbb{C}\mathbb{1}$ to a forked repository $\mathbb{R}'_{\mathbb{C}}$ [15].
4. Create a Pull Request (PR): Pull Request enables the user to express an interest in adding their changes to the project. Project's owner then decides whether accept those changes or not. As shown in the figure 2.5, the pull request $\mathbb{PR}$ contains the set of commit changes $\mathbb{C}$, that will submit to the original repository $\mathbb{R}$, thus completing the workflow [15].

2.7.2 Clone and Push workflow

This is a simpler version of Fork and PR workflow 2.7.1. Users do not fork a repository $\mathbb{R}$, they only clone it to the local machine. After making changes $\mathbb{C}$ in the local version of the repository, users can push them to the original repository $\mathbb{R}$.

2.7.3 Features regarding a contribution workflow

Sections 2.7.1 and 2.7.2 demonstrated two basic GitHub contribution workflows. As we can see, number of commits and Pull Requests closely relates with a development activity of a project.

17. See `https://docs.github.com/en/github/getting-started-with-github/fork-a-repo`
18. See `https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository`

Branches can be created from the base repository branch to safely experiment with changes without a danger of making changes to original files[19]. *Releases* can be created to bundle and deliver iterations of a project to users[20]. High number of *releases* and *branches* can thus be also one of the indicators of the high development activity.

Wealth in terms of OSS can be defined as the number of completed Pull Requests in month m. To add weight on more recent PRs, we use a weighted measure $PR_m(pr)$ to return the number of months that a PR pr took to close. Thus, PRs taking more than a month to complete has lower weight. Wealth is formally defined as:

$$Wealth_m = \sum_{pr \in PRs} \frac{1}{PR_m(pr)} \tag{2.1}$$

19. See https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/managing-branches

20. See https://docs.github.com/en/github/administering-a-repository/about-releases

3 Implementation

3.1 Data collection

Quality of any machine learning model very strongly correlates with quantity and quality of input data. In this project, input data are features 2 computed from the GitHub repositories[1].

Figure 3.1 illustrates the data gathering process. Three containers: *crawler* 3.1.3, *computer - unmaintained* and *computer - maintained* 3.1.4 are running on the *OpenShift* 3.1.1 platform. Containers store and load data from the *S3* storage 3.1.1 and make calls to the *GitHub API*.

3.1.1 Tools used for data collection

Containers

Containers contain a runtime environment which comprises of the software application, its dependencies, libraries, binaries and configuration files. Software application runs in the container and does not depend on the host environment, except for the operating system. A container can contain multiple apps and each app will have its own environment[2]. Containerization thus provides a lightweight, isolated environment that makes apps easier to deploy and manage.

Figure 3.2 shows the schema of a container architecture. Applications *App 1*, *App 2* and *App 3* are bundled with their dependencies and running on the *container runtime*, software responsible for running containers. *Container runtime* runs on the host operating system which runs on the underlying infrastructure.

Running data collection components 3.1.3 and 3.1.4 as containers has allowed me to take advantage of an Red Hat's OpenShift cluster 3.1.1.

1. See `https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/about-repositories`
2. See `https://www.techopedia.com/2/31967/trends/open-source/container-technology-the-next-big-thing`

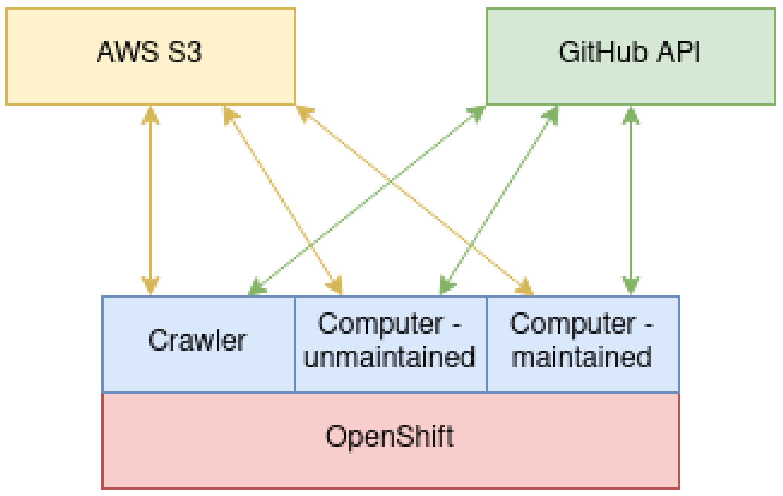


Figure 3.1: Data gathering architecture. Source: own

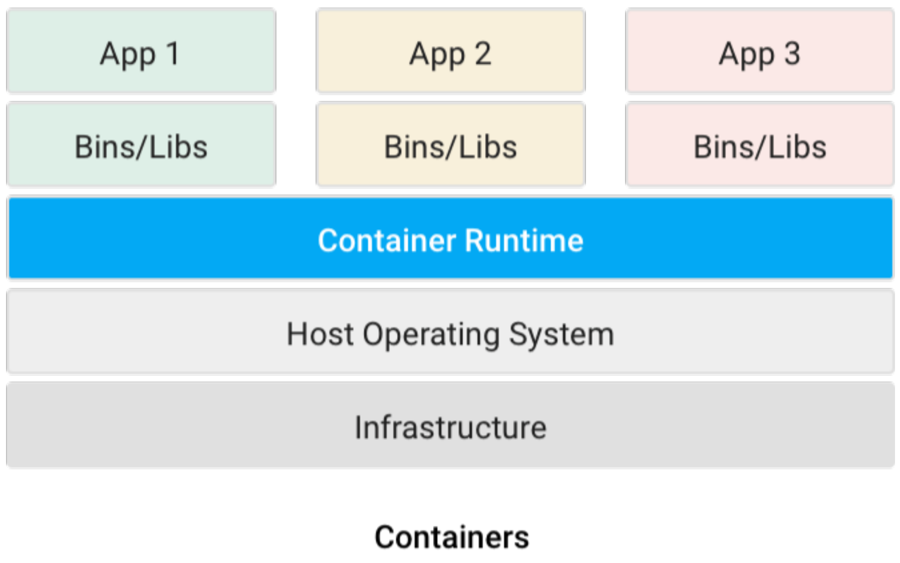


Figure 3.2: Container architecture. Source: `https://cloud.google.com/containers`

OpenShift

Red Hat OpenShift[3] is an enterprise-ready Kubernetes[4] container platform. It functions as a Platform-as-a-Service[5] and a Container Orchestration[6] Engine.

In the simplest of terms, it provides a platform for running and managing containers 3.1.1. OpenShift is focused on a security on every level of a container stack, it also provides a huge set of other functionalities[7] including automated installation, upgrade, lifecycle manager and others.

Amazon S3

Data collection process is carried out by applications running as containers 3.1.1. These applications generate data that need to be stored in a permanent storage outside of those containers. *Amazon S3*[8] provides such storage.

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytic[9].

3. See `https://www.redhat.com/en/technologies/cloud-computing/openshift`
4. See `https://kubernetes.io/`
5. See `https://www.oracle.com/cloud/what-is-paas/`
6. See `https://www.vmware.com/topics/glossary/content/container-orchestration`
7. See `https://www.openshift.com/`
8. See `https://aws.amazon.com/s3/`
9. See `https://aws.amazon.com/s3/`

3.1.2 Searching for suitable repositories

Filtering repositories

In the time of the study, GitHub offers more than 38 million public repositories[10]. Not all of them are suitable for this study, there are three minimal requirements repositories have to meet.

1. *Development time* 2.6 of more than 730 days (two years). Several features 2 require historical data. Idea behind is that estimating a longevity based on couple of weeks of development could be very inaccurate.

2. Repository contains a code in a programming language. Considerable portion of repositories do not contain files in a programming language. Examples are projects providing resource lists, CSS[11] templates or a content in some other markup or data type language.

3. Project started a development using *git* version control system. Some projects on GitHub were initially using some other version control[12] and ported to *git* in a later stage of the development. This devalues the data as the whole development process before the transition is squashed into a single or few commits, which do not realistically represent the full process from the start.

   A project is considered to be started outside of GitHub and then transitioned there later if more than 50% of files were added in less than 20 commits. Same method is used in [16].

Repositories satisfying previous criteria are considered suitable for the study. Such projects are then divided into two categories.

1. Unmaintained: there are three criteria necessary for unmaintained repositories:

10. See `https://github.com/search?q=is%3Apublic&ref=simplesearch`
11. See `https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS`
12. See `https://en.wikipedia.org/wiki/Template:Version_control_software`

(a) Repository is archived[13]. Repository owners can archive their repositories to make it read-only and to let people know that the project will no longer be maintained. This is far better alternative to just deleting the repository, as users can still fork and use the archived projects. It would be very convenient for this study, if every unmaintained project was marked as archived. This is not the reality. While archiving projects that are no longer under active development is considered a good practice, it is not required.

(b) README file of a repository contains at least one of the following strings: *deprecated*, *unmaintained*, *no longer maintained*, *no longer supported*, *no longer under development*, *not maintained*, *not under development*, *obsolete*, *archived*. In some cases, repository owner puts the information about the end of development into the README file. The crawler scans this file and marks a repository unmaintained if any of the keywords is found.

In some instances, such keyword can be used for some other purpose than to let the reader know that the project is no longer being developed. For this reason, I manually checked all of such repositories to confirm if the repository was classified correctly. In the case of misclassification caused by a keyword being used in some other context, said repository was reclassified manually. Process was similar as in [17].

(c) Last criterion is that the last commit was submitted more then a year (104 weeks) ago. No activity for a year is a clear sign that the project's community is no longer continuing the development of the project.

2. Maintained: suitable projects not considered unmaintained are viewed as maintained.

13. See `https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/archiving-a-github-repository`

Searching through repositories and their consecutive classification to not suitable/suitable and unmaintained/maintained is done by a *crawler* 3.1.3.

3.1.3 Developed applications

Repositories crawler

I created a crawler program to go through GitHub repositories and classify 3.1.2 them. Crawler goes through repositories in a randomized fashion. The process is as follows:

1. User selects a range of years for the crawler to search through. Repositories created within this time range will be considered for crawling. Input is added via the configuration file, the default search range and the one used for the study is years *2009 - 2019*. Because of the *suitability* requirement 3.1.2 of two years worth of historical data, the upper bound of this range is set to 2019. Repositories created after 2019 naturally do not have enough data to be fitting for the study. Lower bound is chosen arbitrarily.
2. Time range is then transformed into an ID range. Repository IDs are represented as integers. Values of IDs of newly created repositories are higher than values of IDs of previously created repositories. This is convenient, as it can be exploited to make a random crawling of repositories easier. Transformation from a time range to an ID range is as follows:
   (a) Lower bound of an ID range is an ID of the first repository created in the year given by the lower bound of a time range.
   (b) Upper bound of an ID range is an ID of the first repository created in the year given by the upper bound of a time range.
3. Crawler that chooses an integer from the ID range with the uniform probability[14]. Not all IDs within the range still represent

14. Method used for this step of the process is *random.randrange*, see `https://docs.python.org/3/library/random.html#random.randrange`

an existing repository, so the crawler needs to check the validity of the chosen ID. If the generated integer is a valid ID, process proceeds to the next step, if it is not, another integer is chosen.

4. In the last step, the crawler classifies the repository 3.1.2 and saves the ID into the corresponding file.

3.1.4 Features computer

Once the *crawler* 3.1.3 gathers IDs of suitable repositories, another application, *computer*, can compute the requested features of the repositories. *Computer* will go through the IDs collected by the *crawler*. For every ID, it will compute all of the implemented features specified in the *configs/features.yml* configuration file [18].

3.1.5 Features of crawler and computer

Although responsible for different tasks, both programs, crawler and computer, work in a very similar way. They were both created with the same goals in mind. Here are some notable features.

Extensibility

New dataset features can be added quickly: they need to be implemented in the *data_gathering/repository_data.py* and then referenced in the *configs/features.yml* file.

Extensibility is important, as it allows different users with different use cases to easily shape *crawler* and *computer* to better meet their specific needs.

Scalability

Project contains three *Dockerfiles*[15] for a *crawler*, a *computer* of unmaintained repositories and a *computer* for maintained repositories. These *Dockerfiles* can be used to build containers 3.1.1 which can be run on

15. See `https://docs.docker.com/engine/reference/builder/`

a container platform[16]. Users with sufficient resources can scale up number of these containers and gather the data at a faster rate.

In the case of this project, applications were running as containers in the OpenShift cluster 3.1.1 provided by Red Hat[17].

Flexibility

Data gathering of a sufficient scale takes up hundreds of hours. Whole process does not need to finish in the single run. Applications *crawler* and *computer* are build in such way that their computation can be terminated without the loss of previously computed data.

Both programs periodically and at the end of the execution serialize all of the computed data and upload them to the specified permanent storage. At the beginning of executions, both programs first check for the existing data in the permanent storage. When preceding data is detected, applications will download them, unpack and then carry on with the computation. Frequency of periodic saves and uploads of the data can be adjusted via a configuration file[18]. Solution for the permanent storage used for this project is *AWS S3* 3.1.1.

This alleviates the danger of losing potentially hours worth of computed data due to an unexpected application termination. Containers can also be shut down, rebuild and re-run in order to update the application.

Logging

Both programs, *crawler* and *computer*, make use of detailed logging. Log files are uploaded to the persistent storage with the other data. This is useful in case of Pod[19] crashes which leave no Pod log file available. Logs also inform about the progress of the data gathering and feature computation.

16. See `https://en.wikipedia.org/wiki/OS-level_virtualization#Implementations`
17. See `https://www.redhat.com/en`
18. See `https://github.com/samuelmacko/master_thesis/blob/master/data/configs/gathering.yml`
19. See `https://kubernetes.io/docs/concepts/workloads/pods/`

There are two levels of logging, NORMAL and DEBUG. NORMAL level is mainly for displaying the progress, DEBUG on the other hand, shows more exhaustive progress information and possible errors.

3.2 Dataset

For the thesis, 1 657 suitable 3.1.2 repositories were gathered. Of these 125 were classified as *maintained* and the rest – 1 532 – as *unmaintained* 3.1.2. *Maintained* projects make only about 8% of all of the suitable repositories. This small sample of all of the GitHub repositories might indicate that only a smaller fraction of them are under an active maintenance.

3.3 Removing highly correlated features

A dataset sample consists of 28 features 2. I removed highly correlated features using *Pearson correlation coefficient*[20]. Threshold for the removal is set to 90%, sufficiently correlated – and thus removed – features are:

- issues_count_open
- issues_count_closed
- releases_count
- stargazers_count
- wealth

Correlation heatmap can be seen in the figure 3.3. Final number of features is then 23.

3.4 Dataset preprocessing

Features of the dataset have data of both types, numeric and categorical. Integer and float features all have different means and ranges. For a

20. See `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`

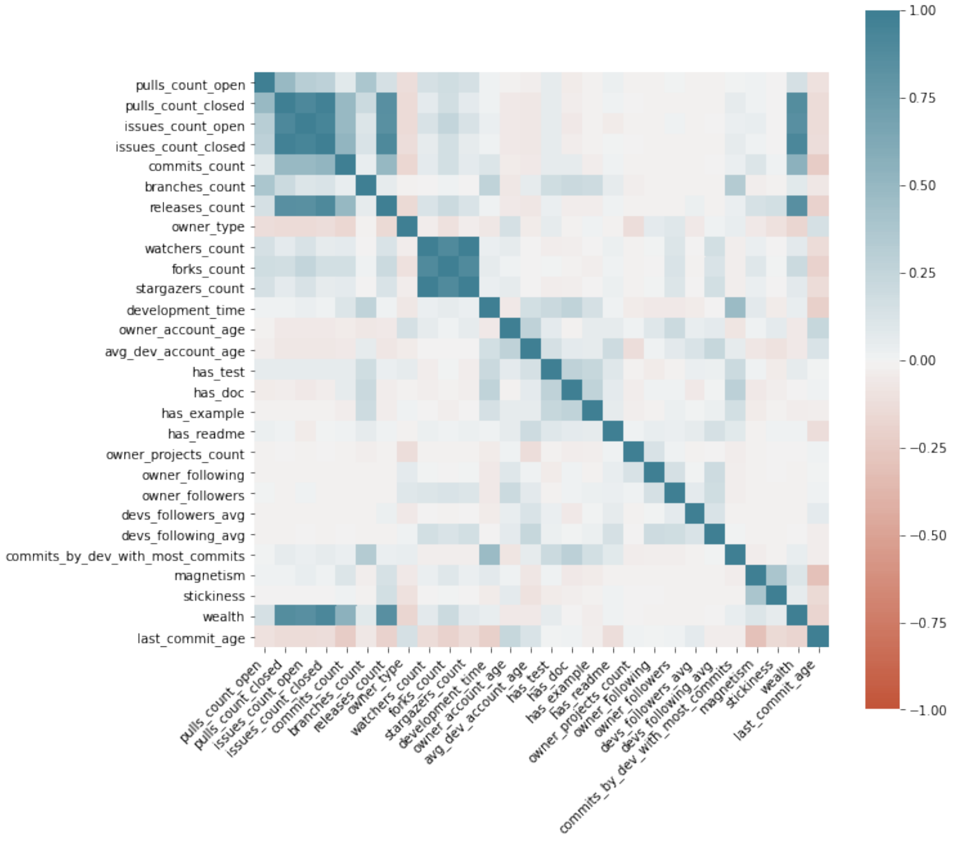


Figure 3.3: Correlation heatmap. Source: own

dataset to be usable as an input for used machine learning algorithms 3.6, the data needs to be processed.

Numeric features are measured in different ranges and in different units (days, counts, etc.). Such input data would not produce a valid results, as features with values in higher ranges would dominate over features with values in lower ranges. Several distance-based machine learning models would attribute higher weight to features with a higher magnitude and thus bias results. To prevent this problem, feature scaling solution *StandardScaler*[21] was used. *StandardScaler* subtracts the mean value from the feature and then divides the result by the feature's standard deviation. This process produces data that have mean value of 0 and standard deviation of 1.

Apart from numeric features, dataset also contains nominal categorical features. While these features are encoded as integers, there is no particular order between categories. Using these integers as input would skew models, as categories with higher value integer would arbitrarily be assigned a higher weight. Encoding such features is done by *OneHotEncoder*[22], it transforms a categorical feature with n possible values into n binary features.

Whole preprocessing can be seen in the figure 3.4. Maintained repositories are not used for training, nor for evaluation, only for computation of the centroid. Centroid is needed for labeling 3.5 of projects classified as unmaintained.

Unmaintained projects are split to two sets, one used for training and the other used for testing. Training set contains 80% of all of the unmaintained repositories. The rest, 20%, is used for testing. Features of these sets are preprocessed separately. This is because *StandardScaler* uses mean and standard deviation of samples for its computation. If both – test and training – sets were preprocessed together, information from the training set would leak[23] into the test set, which is unwanted.

21. See `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`
22. See `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html`
23. See `https://machinelearningmastery.com/data-leakage-machine-learning/`

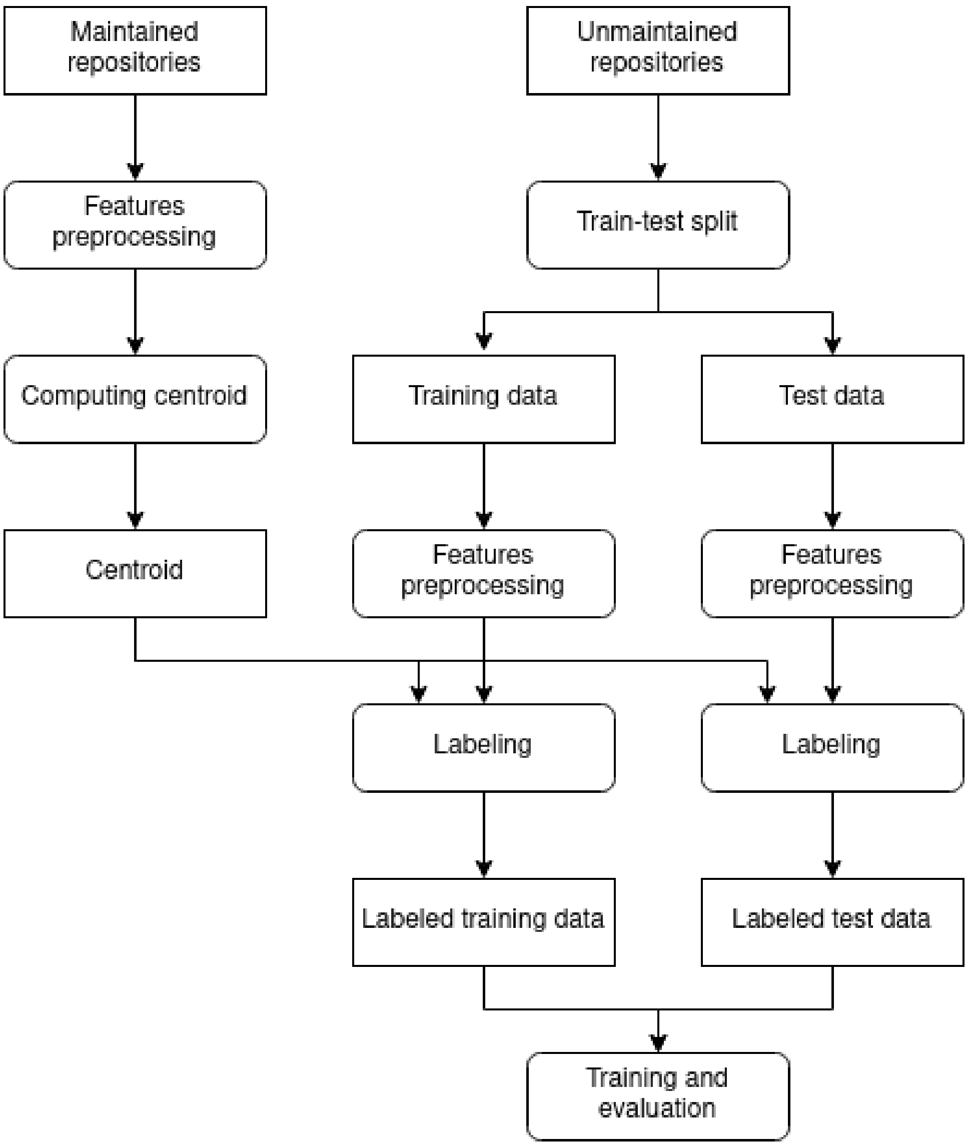


Figure 3.4: Scheme of preprocessing. Source: own

3.5 Samples labeling

Classification models are trained in a supervised manner, this means that the training data need to be labeled. Labeling process is as follows:

1. Gather a sufficient quantity of suitable repositories 3.1.2.

2. Cluster *maintained* repositories and find their centroid using a *KMeans*[24] algorithm. Centroid is central a vector, which may not necessarily be a member of the dataset. Centroid can be thought of as a multi-dimensional average of a cluster.

3. Compute an *l2*[25] (Euclidean) distance of every *unmaintained* repository from the *maintained* repository cluster centroid.

4. Create a subset of *unmaintained* repositories with outliers removed. Outliers are determined using *Interquartile range* approach[26] computed on the Euclidean distances of repositories from the centroid.

5. Based on this subset, 10 equally sized bins are created.

6. Label all repositories based on the bin they belong to. Repositories with the distance greater than the limit of the last bin are put into that last bin, and repositories smaller than the limit of the smallest bin are put into the smallest one.

Labeled train and test data are then used in the training and evaluation process 3.6. Label counts of train and test data can be seen in figures 3.5 and 3.6 respectively.

Labels represent a probability of a repository to die off – to not be maintained in the future. Label *1* means that a repository will die off with a probability of 10%, label *2* represents a probability of 20% and so on.

It is clear from the figures that label distributions are very skewed.

3.6 Training and evaluation

Four machine learning algorithms were tested for the best result, these are:

24. See `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`
25. See `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.euclidean_distances.html`
26. See `https://www.statology.org/find-outliers-with-iqr/`

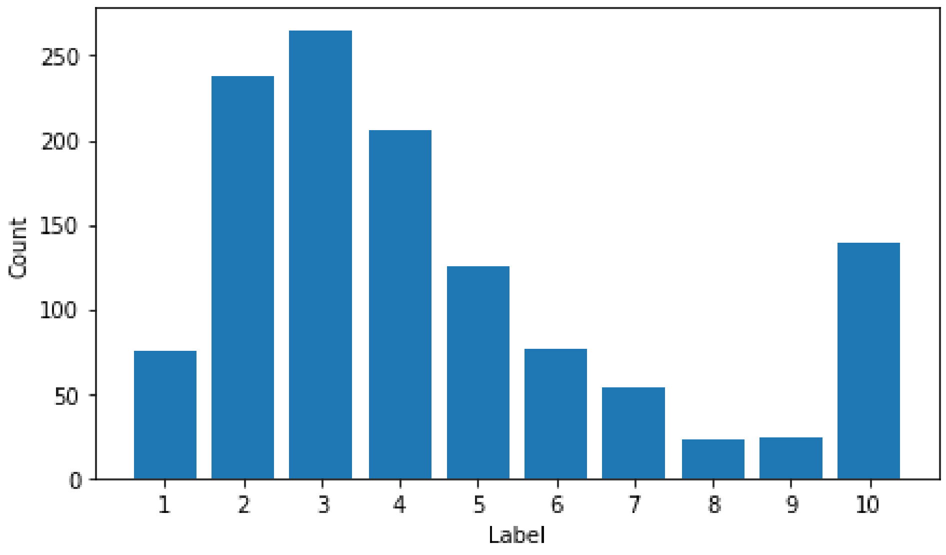


Figure 3.5: Label counts of train data. Source: own

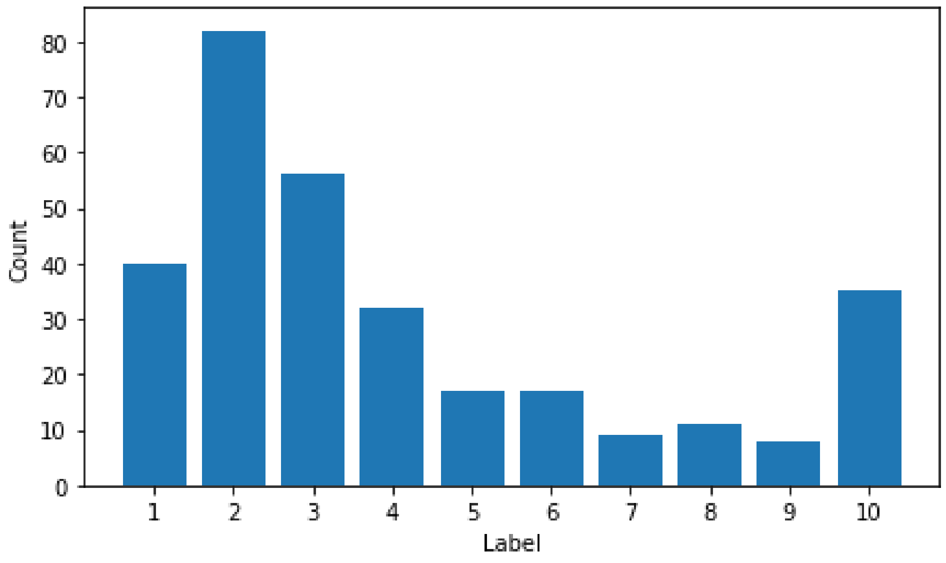


Figure 3.6: Label counts of test data. Source: own

Table 3.1: Parameters grid for RandomForest.

Hyper-parameter	Values
bootstrap	True, False
criterion	gini, entropy
max_depth	10, 20, 30, 40, 50, None
max_features	auto, sqrt, log2
min_samples_leaf	1, 2, 4
min_samples_split	2, 5, 10
n_estimators	50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150

1. RandomForest[27]
2. SVC[28]
3. kNN[29]
4. MLP[30]

Set of hyper-parameters for models 3.6 were tuned using *grid search*[31]. In *grid search* technique, every possible combination of pre-defined hyper-parameters is tried and evaluated in order to find the set with the best performance. Parameter grids used in the thesis can be seen in the tables 3.1, 3.3, 3.2 and 3.4.

Models were evaluated using 5-fold *cross validation*[32].

27. See `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`
28. See `https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`
29. See `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`
30. See `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`
31. See `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV`
32. See `https://en.wikipedia.org/wiki/Cross-validation_(statistics)`

Table 3.2: Parameters grid for SVM.

Hyper-parameter	Values
C	0.1, 1, 10, 100, 1000
decision_function_shape	ovo, ovr
gamma	scale, auto, 1, 0.1, 0.01, 0.001, 0.0001
kernel	linear, poly, rbf, sigmoid
shrinking	True, False

Table 3.3: Parameters grid for kNN.

Hyper-parameter	Values
algorithm	auto, ball_tree, kd_tree, brute
leaf_size	20, 25, 30, 35, 40
metric	minkowski, chebyshev
n_neighbors	1, 2, 3, 4, 5
p	1, 2
weights	uniform, distance

Table 3.4: Parameters grid for MLP.

Hyper-parameter	Values
activation	logistic, relu
alpha	0.001, 0.0001
hidden_layer_sizes	50, 75, 100, 125, 150
learning_rate	constant, adaptive
shuffle	True, False
warm_start	True, False

4 Results

Trained models 3.6 were compared with two baselines[1]:

1. Stratified: generates predictions by respecting the training set's class distribution.
2. MostFrequent: always predicts the most frequent label in the training set.

Model and baseline predictions were evaluated using two scores:

1. Accuracy score - simple fraction of correct predictions and all predictions.
2. (custom) Accuracy score with deviation: very similar to Accuracy score, but prediction label of one more or one less than the true label is still considered a correct prediction. For example, consider the true label of a sample to be 7. If a model predicts 6, 7, or 8, all of these are considered a correct prediction. Number of correct predictions is then divided by the number of all predictions as usual.

   I included this score to compute accuracy with a small deviation, if a user can accept a small error in predictions.

Results can be see in figure 4.1. As mentioned in section 3.6, parameters for each model were determined using *grid search*. Best parameters for each model, based on *Accuracy score* can be seen in tables 4.1, 4.2, 4.3 and 4.4.

As we can see, *Accuracy scores* of trained models are around 50%. *Accuracy scores with deviation* are naturally higher, reaching around 90%. However, all of the models performed significantly better than baselines. Better performing, *MostFrequent* baseline, reaches accuracy of around 21%.

I believe that lower accuracy can be contributed to the combination of two things.

1. See `https://scikit-learn.org/stable/modules/model_evaluation.html#dummy-estimators`

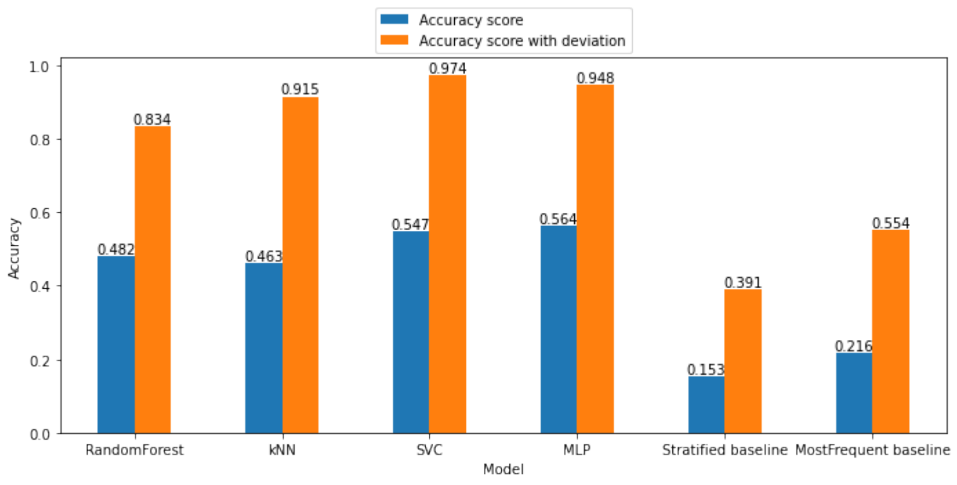


Figure 4.1: Results. Source: own

1. Very skewed distribution of data labels 3.5 and 3.6. In dataset used for training, the most frequent label covers 21.7% of all samples, and three most frequent labels cover around 58% of all samples. On the other end, only around 2% of samples have the least frequent label. To increase classification accuracy, it would be helpful to have more balanced label distribution.

2. Insufficient number of *maintained* repositories. Data collection process is very slow and resource consuming. Out of 1 652 collected repositories, only 125 (around 8%) of them were classified as *maintained* and thus suitable to be used as training and test data.

It is questionable whether models developed in the thesis reach high enough accuracy to be reliably used in most production-level scenarios. It very much depends on the situation. Especially *MLP* could be used in cases where we can afford to tolerate a small inaccuracy. Metric *Accuracy score with deviation* tells us that models are very good in determining at least a general direction a repository is headed. If it is likely to die off or not.

To improve performance of these models, I will continue to work on the thesis to solve not only the two main problems listed above:

Table 4.1: Parameters of the best RandomForest model.

Hyper-parameter	Value
bootstrap	False
ccp_alpha	0.0
class_weight	None
criterion	gini
max_depth	40
max_features	sqrt
max_leaf_nodes	None
max_samples	None
min_impurity_decrease	0.0
min_impurity_split	None
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0.0
n_estimators	150
oob_score	False
random_state	None

unbalanced labels, and not enough maintained repositories in the dataset.

Table 4.2: Parameters of the best SVM model.

Hyper-parameter	Value
C	100
break_ties	False
cache_size	200
class_weight	None
coef0	0.0
decision_function_shape	ovo
degree	3
gamma	auto
kernel	rbf
max_iter	-1
probability	False
random_state	None
shrinking	True
tol	0.001

Table 4.3: Parameters of the best kNN model.

Hyper-parameter	Value
algorithm	ball_tree
leaf_size	35
metric	chebyshev
metric_params	None
n_neighbors	1
p	1
weights	uniform

Table 4.4: Parameters of the best MLP model.

Hyper-parameter	Values
activation	relu
alpha	0.001
batch_size	auto
early_stopping	False
hidden_layer_sizes	150
hidden_layers learning_rate	constant, invscaling, adaptive
learning_rate_init	0.01, 0.001, 0.0001
momentum	0.0, 0.25, 0.5, 0.75
power_t	0.3, 0.4, 0.5
shuffle	True, False
solver	lbfgs, sgd, adam
warm_start	True, False

5 Summary

Aim of this thesis was to develop tools that would help estimate whether a given GitHub repository is likely to be developed in the future or not.

First, I reviewed existing work in the area. I summarized a research paper by Coelho et al. [2], which has a similar aim and execution, in the chapter 1. Other work concerned with evaluating open-source projects was used mainly for developing a set of features, these are examined in chapter 2.

Since I was not able to find a dataset suitable for needs of this thesis, I implemented a set of tools to create it. I compiled a dataset of 1 652 repositories, of which 125 were classified as maintained, and the rest – 1 532 – as unmaintained. Collection process is slowed down mostly by the rate limit imposed by the GitHub API, which data collection tools implemented for the thesis rely on, and computational power. There is also a big disproportion in collected repositories. Majority of crawled projects – around 92% – are classified as unmaintained.

Assemblage of collected repositories were then preprocessed and labeled as described in chapter 3. Finished dataset was than used as an input for the machine learning algorithms. Results for the *Accuracy* metric are around 50% and for *Accuracy with deviation* around 90%. All of the models perform considerably better than baselines, so it is worth to use them. I believe that results can be improved by solving two most obvious problems with the dataset: (a) very skewed distribution of sample labels, and (b) small quantity of maintained repositories in the dataset.

Working on the thesis has provided me with a great deal of hands-on experience. I widened my knowledge on a real-world project of a larger scale. Apart from the area of machine learning, I was also exposed to the area of containerization, as dataset collection tools ran as containerized applications in a cluster.

5.1 Future work

In the follow-up work, more data, especially maintained repositories should be collected. Other labeling strategies should be tested, as well as more classification algorithms.

Bibliography

1. LAURENT, Andrew M. St. *Understanding Open Source and Free Software Licensing*. O'Reilly Media, Inc., 2004. ISBN 0596005814.
2. COELHO, Jailton; VALENTE, Marco Tulio; SILVA, Luciana L.; SHIHAB, Emad. Identifying unmaintained projects in github. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2018. ISBN 9781450358231. Available from DOI: `10.1145/3239235.3240501`.
3. BORGES, Hudson; HORA, Andre; VALENTE, Marco Tulio. Understanding the Factors That Impact the Popularity of GitHub Repositories. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016. ISBN 9781509038060. Available from DOI: `10.1109/icsme.2016.31`.
4. BORGES, Hudson; HORA, Andre; VALENTE, Marco Tulio. Predicting the Popularity of GitHub Repositories. *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2016. ISBN 9781450347723. Available from DOI: `10.1145/2972958.2972966`.
5. AGGARWAL, Karan; HINDLE, Abram; STROULIA, Eleni. Co-Evolution of Project Documentation and Popularity within Github. 2014, pp. 360–363. ISBN 9781450328630. Available from DOI: `10.1145/2597073.2597120`.
6. BORGES, Hudson; VALENTE, Marco Tulio; HORA, Andre; COELHO, Jailton. *On the Popularity of GitHub Applications: A Preliminary Note*. 2017. Available from arXiv: `1507.00604 [cs.SE]`.
7. JIANG, Jing; LO, David; HE, Jiahuan; XIA, Xin; KOCHHAR, Pavneet Singh; ZHANG, Li. Why and How Developers Fork What from Whom in GitHub. *Empirical Softw. Engg.* 2017, vol. 22, no. 1, pp. 547–578. ISSN 1382-3256. Available from DOI: `10.1007/s10664-016-9436-6`.
8. YAMASHITA, Kazuhiro; MCINTOSH, Shane; KAMEI, Yasutaka; UBAYASHI, Naoyasu. Magnet or Sticky? An OSS Project-by-Project Typology. 2014, pp. 344–347. ISBN 9781450328630. Available from DOI: `10.1145/2597073.2597116`.

9. ROBLES, Gregorio; GONZALEZ-BARAHONA, Jesus; HERRAIZ, Israel. Evolution of the core team of developers in libre software projects. *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR 2009*. 2009, pp. 167–170. Available from DOI: 10.1109/MSR.2009.5069497.

10. JARCZYK, Oskar; GRUSZKA, Błażej; JAROSZEWICZ, Szymon; BUKOWSKI, Leszek; WIERZBICKI, Adam. GitHub Projects. Quality Analysis of Open-Source Software. 2014, pp. 80–94. ISBN 978-3-319-13733-9. Available from DOI: 10.1007/978-3-319-13734-6_6.

11. ONOUE, Saya; KULA, Raula; HATA, Hideaki; MATSUMOTO, Kenichi. The Health and Wealth of OSS Projects: Evidence from Community Activities and Product Evolution. 2017.

12. ZHU, Jiaxin; ZHOU, Minghui; MOCKUS, Audris. Patterns of Folder Use and Project Popularity: A Case Study of Github Repositories. 2014. ISBN 9781450327749. Available from DOI: 10.1145/2652524.2652564.

13. RAZA, Arif; CAPRETZ, Luiz. Contributors Preference in Open Source Software Usability: An Empirical Study. *International Journal of Software Engineering Applications*. 2010, vol. 1. Available from DOI: 10.5121/ijsea.2010.1204.

14. REHMAN, Ifraz; WANG, Dong; KULA, Raula Gaikovina; ISHIO, Takashi; MATSUMOTO, Kenichi. Newcomer Candidate: Characterizing Contributions of a Novice Developer to GitHub. 2020. Available from arXiv: 2008.02597 [cs.SE].

15. REHMAN, Ifraz; WANG, Dong; KULA, Raula Gaikovina; ISHIO, Takashi; MATSUMOTO, Kenichi. Newcomer Candidate: Characterizing Contributions of a Novice Developer to GitHub. 2020. Available from arXiv: 2008.02597 [cs.SE].

16. AVELINO, Guilherme; VALENTE, Marco; HORA, Andre. What is the Truck Factor of popular GitHub applications? A first assessment. 2015. Available from DOI: 10.7287/PEERJ.PREPRINTS.1233.

17. COELHO, Jailton; VALENTE, Marco Tulio. Why modern open source projects fail. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017. ISBN 9781450351058. Available from DOI: 10.1145/3106237.3106246.

18. MACKO, Samuel. *Master thesis* [https://github.com/samuelmacko/master_thesis]. GitHub, 2021.