RepoRank - A Novel Index for Open Source Evaluation

by: Regev Golan

Supervisor: Dr. Uri Globus

Thesis submitted in partial fulfillment of the requirement for an MSc degree in Computer Science

The Academic College of Tel-Aviv Yaffo, Israel

School of Computer Science

1st of November, 2020

Acknowledgments

I would like to thank my dear wife Limor, for her support, encouragement and patience during this thesis preparation.
Also, I would like to thank Dr. Uri Globus for his guidance, mentorship, advice and everything I could learn from his rich experience in innovation, machine learning, data mining, research skills, software design and project management.
Finally, I would like to acknowledge my appreciation and thanks to HiredScore Inc, my workplace during the preparation of the thesis, for their support, encouragement and guidance.

Contents

1 Introduction

Open source is a domain that has been researched for dozens of years. Furthermore, due to its enormous community (over 40M users on GitHub), traction, business, risks and popularity there are endless articles, theses, reports, researches and systems about open source software.
In this paper, I have taken on the audacious task of condensing every open source project on GitHub into a single number- A global **ranking** of all projects, regardless of their content, based solely on the activity and **people's** actions using a **graph** structure.
The importance of a single open source project may be an inherently subjective matter, that depends on the user's interests, knowledge and attitudes. But there is still much that can be said objectively about the relative importance of open source contributions.
This paper describes a research on the domain of evaluating open source projects and suggests an algorithm named **RepoRank** based on PageRank by Sergey Brin and Larry Page (1998)[19] to rate open source users and projects objectively and mechanically, effectively measuring the human interest and attention devoted to them.

1.1 Definitions

#	**Phrase or Acronym**	**Definition**
1	OSS	Open Source Software
2	GitHub	Collaborative Version Control Platform. The world's largest collection of open source software
3	PR	Pull Request- One notifies on changes they want to make to a repository on GitHub
4	git	Version control system with enhanced features for collaboration
5	Repository	A git repository is the basic entity of git- a project with an owner, commits and collaborators
6	Repository Owner	Each repository in GitHub belongs to a single organization or a person.
7	Committer	A git user who committed to a repository
8	Stars (Repository)	In GitHub, users can "star" (i.e. stargazers) certain repositories they are interested in or to to manifest their satisfaction from it.

2 Problem Description

What is the definitive way to assess OSS? How can one know the probability of a repository to contain **security vulnerabilities**? A **community** strong enough to prevent deprecation and fix bugs? A **quality** code base that lives up to its promises, reliable, clear and easily extensible?
Many companies and researchers are devoted to creating systems to solve those questions by a huge human defined rules system, classic algorithms or Artificial Intelligence (training a model based on a huge tagged dataset to predict measurements or other methods from the domain of Machine or Deep Learning).

2.1 Importance of Open Source

In 1983, Richard Stallman launched the GNU Project to write a complete operating system free from constraints on the use of its source code. This was the kickoff for the GNU open source licence that was one of the most popular licenses in use today (right behind Apache and MIT as of 2018). Ever since Linus Torvalds released Linux to the world back in 1991, people have been using open source for more than 35 years now with a community current size of **40 million users** only on Github. Those users are working on more than **28 million public repositories**.
One could be tempted to say that since OSS is now mainstream across all layers of software development, with even its most fierce former opponents turning into fervent adopters, it has reached its maturity phase and there is no longer a need for specialized forums dedicated to studying it. Nothing could be farther from the truth: With the huge number of newcomers that now embrace OSS without having contributed to its evolution, and knowing very little of its values and inner workings, it is now more essential than ever to study, understand, and

explain fundamental issues related to the business models, organizational structures, decision-making processes, quality metrics, and the evolution of the free and open source software ecosystems in general.

OSS has been recognized, over the past years, as a promising socio-technical approach to deliver high quality technology in spite of being usually developed by a diverse, often decentralized, and geographically distributed community.

2.2 History of Open Source

Open source is mistakenly being considered to be born with the birth of GitHub in 2008 (based on the git source control released in 2005). As mentioned above, the roots of OSS go back to the early 80s but in other industries, the idea of Open Source goes back even further. It’s important to understand the history of open source to be able to solve its current challenges.

The concept of free sharing of technological information existed long before computers. For example, in the early years of automobile development, one enterprise owned the rights to a 2-cycle gasoline engine patent originally filed by George B. Selden. By controlling this patent, they were able to monopolize the industry and force car manufacturers to adhere to their demands, or risk a lawsuit. In 1911, independent automaker **Henry Ford** assisted in creating a new association instituted a cross-licensing agreement among all US auto manufacturers: although each company would develop technology and file patents, these patents were shared openly and without the exchange of money between all the manufacturers.

But indeed it was the **booming of the internet** and world wide web that pushed OSS to the hands of untrained professionals, commonly with a lacking ability to assess all properties and predict their value over time.

2.3 Issues with Using OSS

How would one define an OSS project as “popular”? Github’s stars? Forks? contributors? Pull Requests count? Latest commit date? Insights? Committers count? How do people talk about it on Stack-overflow? Twitter? and what about security? Is it safe to use? Is the license suitable? How clean is its code base to be able to understand it deeper? How well was it documented?
All these are only part of the questions people tend to investigate before choosing to use an open source project. Unlike closed source projects where you focus on functionality and API, when examining open source there are dozens more factors to consider.

2.3.1 Quality

Mostly determined by rule based systems. It includes the following sub-problems and solutions:

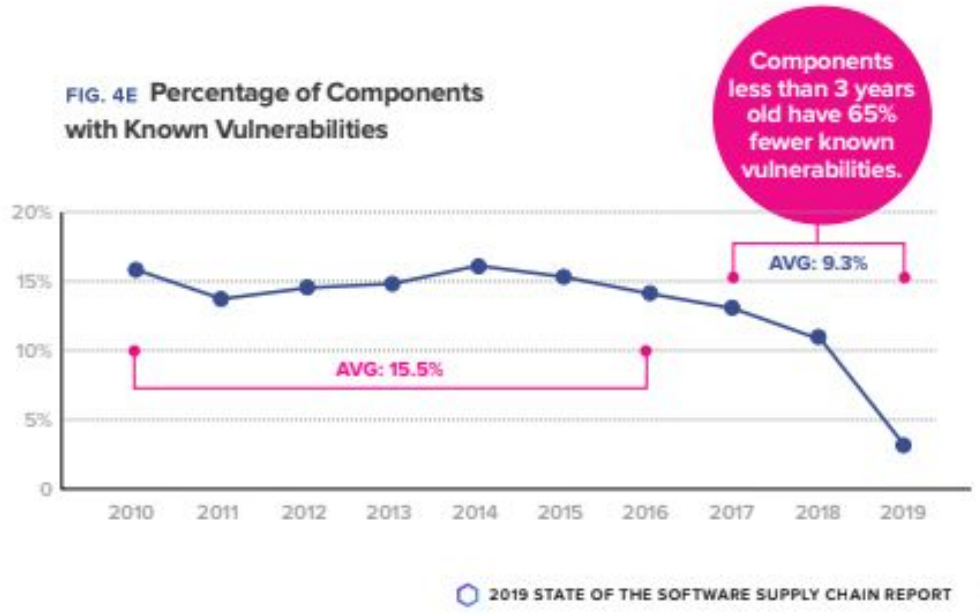
1. Style, language anti-patterns, bugs- measured by **static code analysis** and linters. They exist and are highly used in almost all modern programming languages. They work as compilers do, understanding the issues behind the codebase.
2. Build and test tools to try to prove functionality verification and **test coverage**.

2.3.2 Security

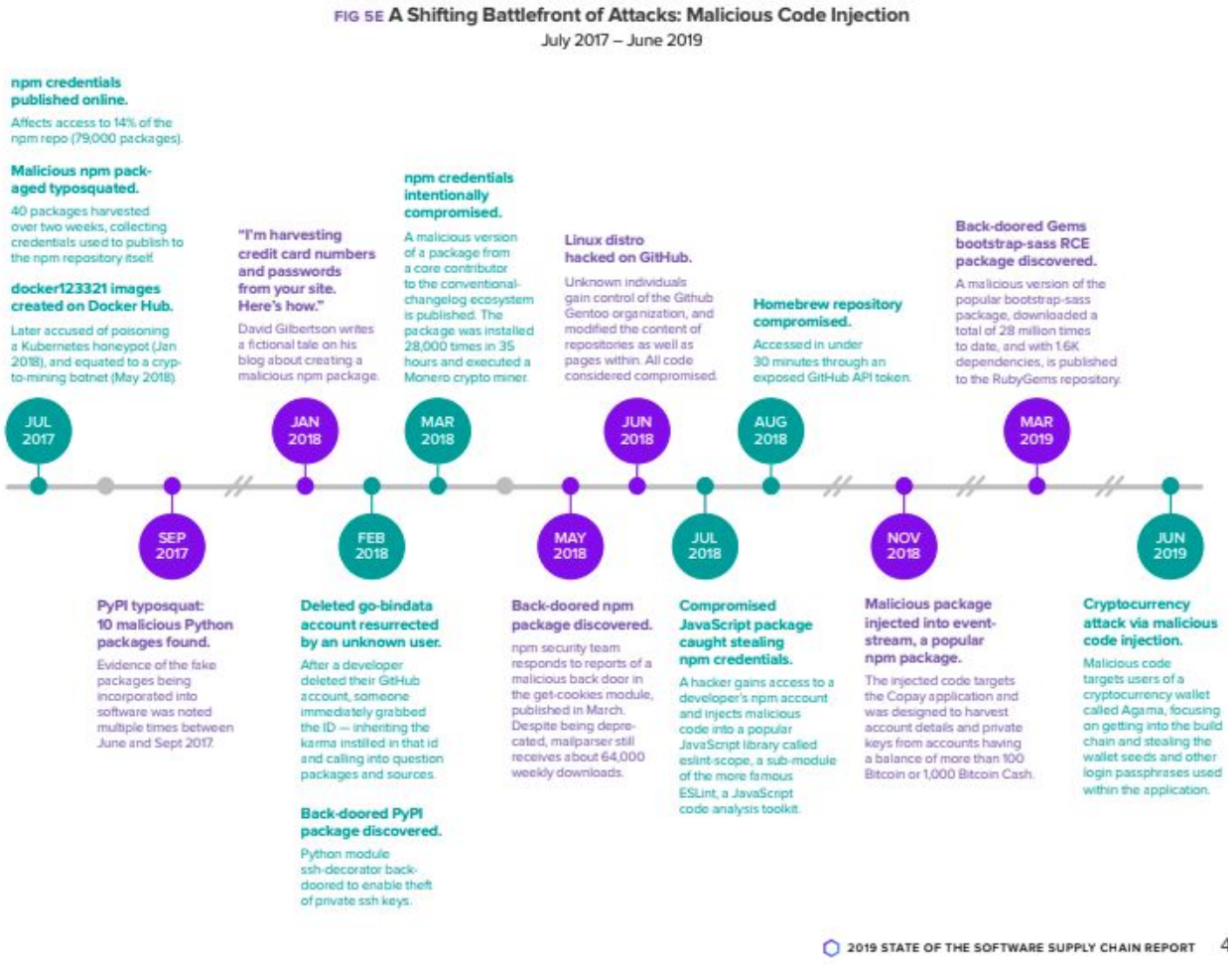
Mostly determined by rule based systems. It includes the following sub-problems and solutions:

1. Known security vulnerabilities **databases**, and code **scanning patterns**- measured by static code analysis and linters. They exist and commonly used in most of the modern programming languages. They also work as compilers do, understanding the issues behind the codebase.

2. **Dependencies-** OSS is usually composed of hundreds of "**tree based**" dependencies. As seen in the graph below, many times they hold a **root vulnerable** dependency that causes a vulnerability along all paths in the tree that depends on it.



3. Malicious activity that can be tracked only by **professionals** trying to watch or accidentally spot it. For example:

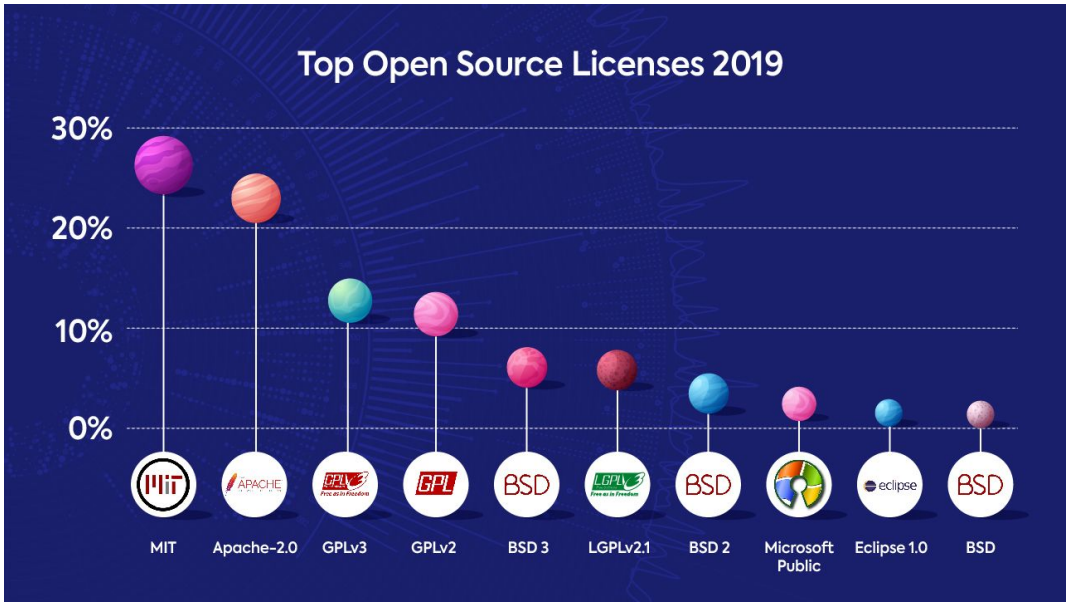


2.3.3 Community

Commonly determined by **rule based** guidelines (as demonstrated in the following literature review). It includes features by GitHub such as **stargazers** but also common practices looking at the **amount of contributors**, and **last commit** date. Those who know and are able to invest more on that section look also on issues, PRs and profiles of the maintainers (backed by the following questionnaires section). Examples are: How many issues or PRs are open for months without a fix or even an answer? What other repositories do the maintainers own? Where do they work? Do they have an impressive Linkedin account?

2.4.4 Legality

This problem may sound simple- a license file is a legal well defined document and only a limited number of options are really common in OSS:



https://resources.whitesourcesoftware.com/blog-whitesource/open-source-licenses-trends-and-predictions

There are numerous software companies trying to solve that area with making it simple to understand but still most of the common software engineers, even those who work for small-medium companies overlook this domain so if no such solution was involved they can cause **serious damage** to companies not fulfilling their **entire stack** of OSS usage license. One known difficulty is the issue of **dependencies**, as in security, their license **may change over time**, hiding that fact from the users who are still **obligated to honour it**.

2.4 Existing Solutions and State of the Art Domain Tools

Traditionally, research on software quality attributes was either kept under wraps within the organization that performed it, or it was

carried out by outsiders using narrow, black-box techniques. The emergence of OSS has changed this picture by allowing us to examine **both the software products and the processes that yield them**. Assets, such as the software source code, the associated data stored in the version control system, the issue-tracking databases, the mailing lists, and the wikis, allow us to evaluate quality in a transparent way.

There exist numerous tools to examine the different aspects of evaluation of OSS. They are usually focused on a single or **few aspects** of the properties as partially described in the following list.

2.4.1 Market Products Examples for OSS Evaluation

1. **Fossa** is an open source management for enterprise team. It is a Scalable, end-to-end management for third-party code, license compliance and vulnerabilities. It tries to create a culture of open source while mitigating open source risk.
2. **WhiteSource** aspires to be the simplest way to secure and manage open source components in your software. It identifies every open source component in your software, including dependencies. It then secures you from vulnerabilities and enforces license policies throughout the software development lifecycle. The result should be a faster, smoother development without compromising on security.
3. **CheckMarx** is a provider of state-of-the-art application security solutions: **static code analysis** software, seamlessly integrated into the development process. It delivers a perfect platform for DevOps and **CI** environments by redefining security's role in the **SDLC** while operating at the speed of DevOps. The fast feedback loop makes security testing of new or edited code fragments quick with speedy remediation by developers. This significantly reduces costs and eliminates the problem of having to deal with many security vulnerabilities close to release. Ultimately, by

enabling developers to test their own code for security issues thus allowing them to get instant results and remediate the issues on the spot, everyone wins.

4. **Semmle** believes security is a shared responsibility. Their mission is to secure all software by bringing the security and development communities together. Combining expertise in the fields of databases, programming languages, data science and security, Semmle is making software truly searchable, allowing deep meaningful questions to be answered, and insights to be shared. Semmle has a **code analysis** platform for finding zero-days and automating **variant analysis**. It has 2 main products: (a) **CodeQL-** a code analysis engine for product security teams to quickly find zero-days and variants of critical vulnerabilities (b) **LGTM-** a code analysis platform for **development** teams to identify **vulnerabilities** early and prevent them from reaching production.

There exists **at least 13 more companies** I found with services or products in the domain of OSS evaluation. They are all looking at different aspects of the code and OSS common metrics. None reveals insights on the people (owners and contributors) behind the open source projects.

2.4.2 Evaluating the Contributors Behind the Project

Evaluation of the users behind the code is overlooked- as found in the research. There are very little tools and even a common practice to look not only on the code and the repository properties, but also on the committers and owners of the code. Is one of them malicious? How experienced are they and trust worthy for the task the repository is trying to fulfill?

For example, under the GitHub repository there exists an Insights tab. For users, the indicators are versatile to include activity metrics and basic resume fields.

2.5 Literature Survey

Selecting appropriate OSS for a given problem or a set of requirements can be very challenging. Some of the difficulties are due to the fact that there is not a generally accepted set of criteria to use in evaluation, and that there are usually many OSS projects available to solve a particular problem. Ahmad Norita et al (2011)[1] proposed a **set of criteria** and a **methodology** for assessing candidate OSS for fitness of purpose using both functional and non-functional factors:

	Criterion	Descriptions
Functional Features		
1	Functionality (m_1)	Used to indicate whether the software possesses the required features. By using the same standard to evaluate multiple products some objective comparisons can be drawn.
2	Product Evolution (m_2)	Used to indicate if the community has developed clear thoughts and plans about which features will be changed or added in the future.
Non-functional Features		
3	Licensing (m_3)	Used to indicate the appropriateness of the software's license for the intended usage.
4	Longevity/Pedigree (m_4)	Used as a quality measure of the authors, patrons, and lineage of the project. Organization and individuals who have produced quality software in the past are more likely to produce quality software.
5	Community (m_5)	Used as a quality measure of the community participating in an open source project.
6	Market Penetration (m_6)	Used to indicate the market penetration of the source software.
7	Documentation Quality (m_7)	Used as a measure of the quality of the documentation
8	Support (m_8)	Used as a measure of the quality of the support, both commercial and community, available for the software.
9	Code Quality (m_9)	Used indicate the quality of the source code.

They then used these criteria in an improved solution to the decision problem using the well-developed **Analytical Hierarchy Process**. In order to validate the proposed model, they applied it at a technology management company in the United Arab Emirates, which integrates many

OSS solutions into its Information Technology infrastructure. The contribution of their work is to help decision makers to better identify an appropriate OSS solution using a **systematic approach** without the need for intensive performance testing.

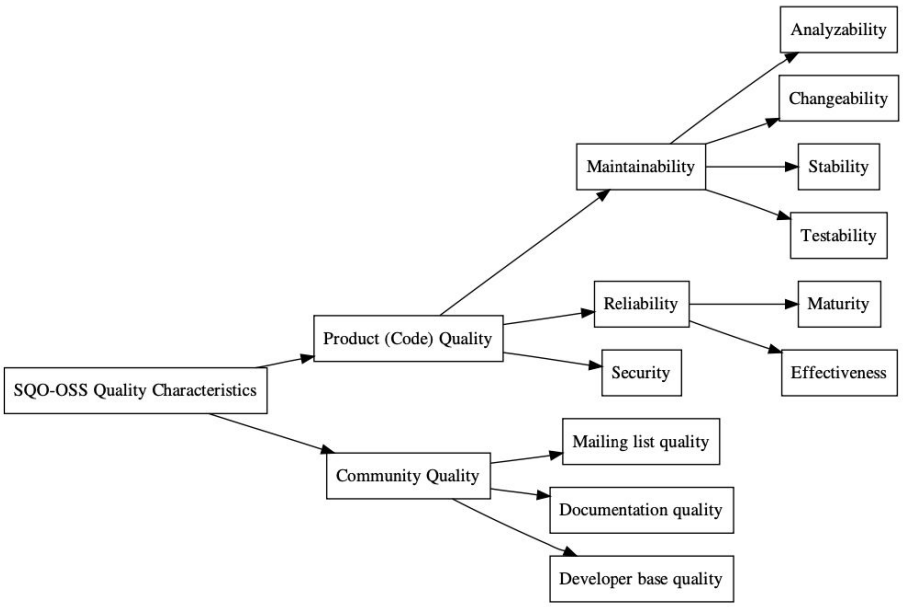
Diomidis et al (2009)[8] presents a technical and research overview of **SQO-OSS**, a cooperative research effort aiming to establish a software quality observatory for OSS. One indicator they presented references agility, the **Mean Developer Engagement** (MDE) metric is introduced and tested through the analysis of public project data. MDE metric for measuring engagement is defined as "the ability, on average, over the lifetime of a Free Software project, for that project to make use of its developer resources." Mathematically this can be described as:

$$\bar{de} = \frac{\sum_{i=1}^{n} \left(\frac{\mathrm{dev}(active)}{\mathrm{dev}(total)} \right)_i}{n} \quad (1)$$

Where:

- $\mathrm{dev}(active)$ is the number of (distinct) developers active in time period i.
- $\mathrm{dev}(total)$ is the total number of developers involved with the project in the periods $0 \ldots i$.
- n is the number of time periods over which the project has been evaluated. For this research these were periods of a week.

They have applied MDE to the entire history of **40** Open Source projects. They have also utilized SQO-OSS quality model that is described as:



A unique algorithm for discovering community patterns in open-source was developed by Tamburri et al (2018)[27], named **YOSHI** (Yielding Open-Source Health Information). It is a tool able to map open-source communities onto community patterns, sets of known organisational and social structure types and characteristics with measurable core attributes. Their **mapping** is beneficial since it allows, for example, (a) further investigation of community health measuring established characteristics from organisations research, (b) reuse of pattern-specific best-practices from the same literature, and (c) diagnosis of anti-patterns specific to open-source, if any. The tool is evaluated in a quantitative empirical study involving **25 open-source** communities from GitHub, finding that the tool offers a valuable basis to monitor key community traits.

The leading factor and influencer when evaluating OSS is its popularity and the community behind it. Borges et al (2016)[5] researched and identified four main patterns of popularity growth measured by their impact on the **stargazers** counter. Their dataset included **2500 public repositories** and they have interviewed developers

of popular repositories to extract **statistical indications** of the impact on popularity.

The idea of using **PageRank** on predictions relating GitHub repositories turned out to be quite popular with different researches such as Caetano et al (2018)[6] where they tried to **predict software releases** of OSS projects using the **links** referenced between GitHub **issues**.

Reliability of OSS was researched by Shelbi Joseph (2014)[22] using a **statistical algorithm** based on past failures of the components in open source repositories. He determined that bug arrivals of most OSS projects will stabilize at a very low level, and the stabilizing point can be viewed as the mature point for adoption consideration. The general **Weibull distribution** offers a possible way to establish the reliability model. An algorithm for software reliability calculation was presented and validated from an OSS dataset.

Common practice for companies and research on open source is to examine the recursive dependencies analysis for the purpose of security and performance evaluation. Bloemen Remco (2012)[4] used a method of **dependency graph analysis** to study OSS development on the grandest scale.

Scoring the owners behind the OSS projects can be inspired by the notorious Social Credit System (2017)[14] where citizens of China are ranked by different parameters but in most areas they use blacklists and whitelists rather than score comparison. Same goes for the research here where **whitelisting or blacklisting** specific repositories using high or low algorithm results can have better precision/recall rather than comparing absolute grades.

2.6 Research Question and Crowd Interviews

Many questions can be asked to answer the gap of evaluating open source projects. As seen in Section 2, the focus in question is the ability of the **average software professional** to weigh-in all aspects of OSS.

I needed to **understand what people care about** when facing OSS. What's the most important question among the following:

1. Does this project fit my needs?
2. Is it safe to use? Does it contain vulnerabilities?
3. How many issues and bugs are hidden? Will the functionality work?
4. How strong is the community of the project? Will I have support when facing issues? Will the project be maintained? Are the maintainers of the project professional for this kind work?
5. Is it legal for me to use it? Are there any licensing issues for me with it?
6. What is the code quality? How hard will it be for me to find and fix bugs on my own? To add features? Did they maintain coding and engineering standards?

All those questions are answering one bigger question for the common software engineer: "Should I use this project?". Of course there are different weights for the questions above to summarize this decision.

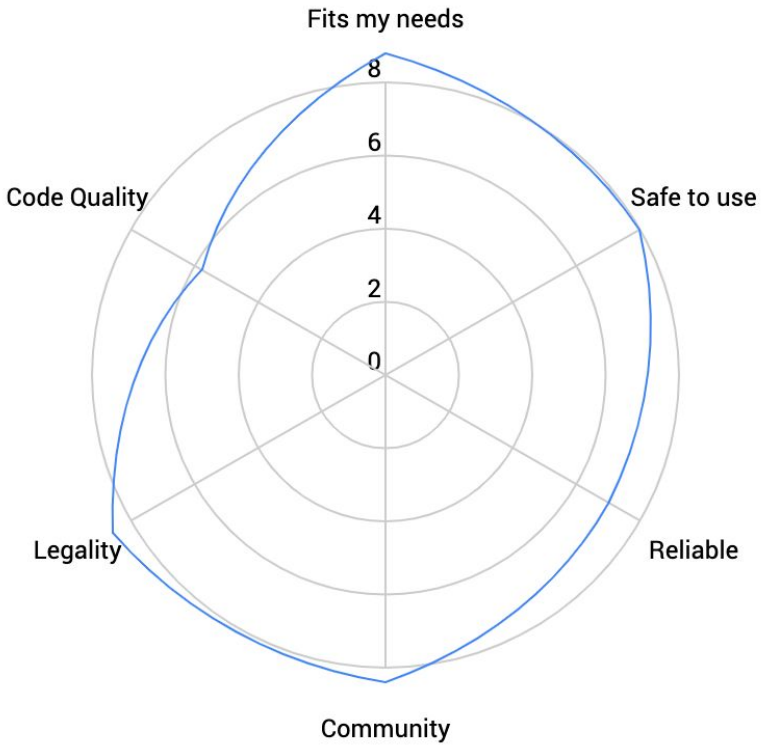
2.6.1 Crowd Interviews Results

The following tables summarize the interviews conducted with senior software engineers (#12), OSS experts (#5), CTOs (#3), CISOs (#4) and Architects (#3) of software companies in Israel and the UK. I've aggregated the grades for each common group.

Q: *What are the weights you give for every aspect of evaluating an open source project (10- highly important, 1- not important, ignoring)?*

Role	Fits my needs	Safe to use	Reliable	Community	Legality	Code Quality
Senior software engineer	10	3	4	6	3	5
OSS expert	8	9	7	10	10	6
CTO	8	9	9	9	10	6
CISO	9	10	8	9	10	5
Architect	9	9	7	8	10	7

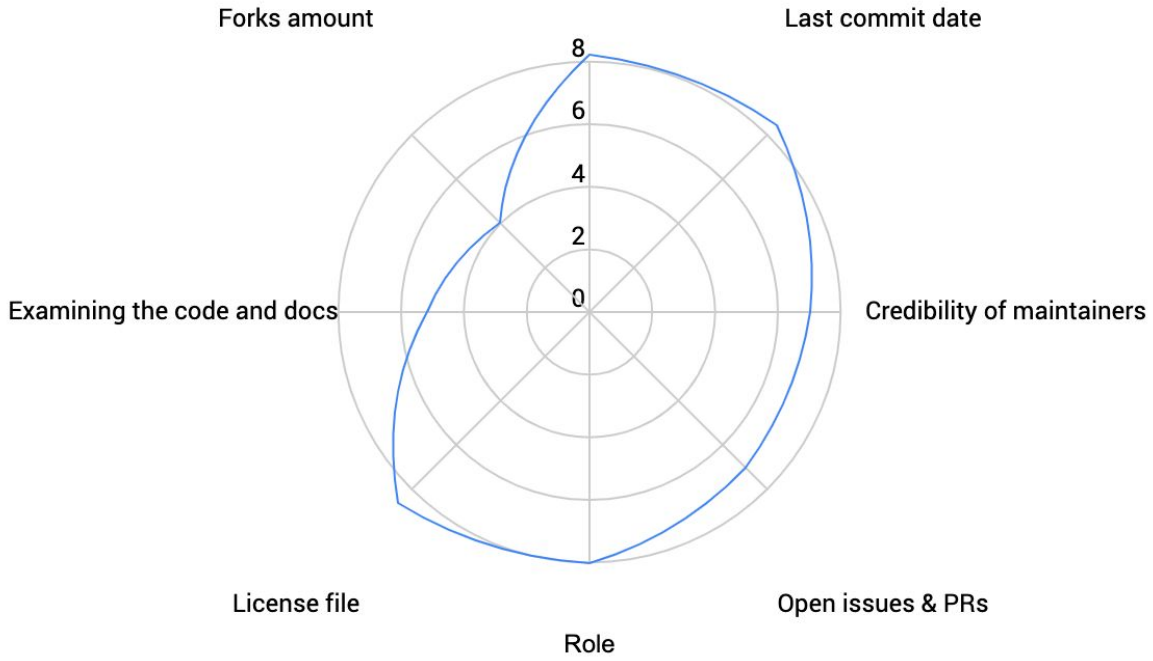
Visualizing the averages:



Q: *What are the features you look at to evaluate the aspects above (10- Most important 1- Not important, overlooking)?*

Role	**Commiters amount**	**Last commit date**	**Credibility of maintainers**	**Open issues & PRs**	**Stars**	**License file**	**Examining the code and docs**	**Forks amount**
Senior software engineer	10	9	2	3	9	3	4	2
OSS expert	8	9	10	8	7	9	7	5
CTO	8	8	7	9	8	10	6	6
CISO	8	9	9	7	7	10	3	3
Architect	7	7	8	8	9	10	6	4

Visualizing the averages:



2.6.2 Finalized Research Question

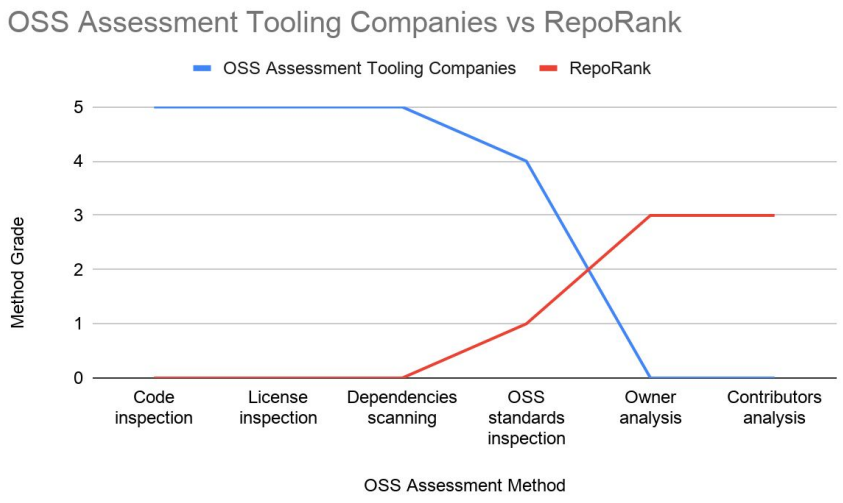
It's apparent from the results of the interviews that there is no consistency between the different job roles relevant to OSS evaluation. I tried to challenge the need to focus on a specific role so to apply to all, I generalized the research question: "**Should I Use This OSS?**". The developed system and algorithms should learn and adapt to the perfect **combination** of the features to prove it fits for all cases. The fact that certain job roles overlook specific features and aspects, is a matter of skills and prioritization of their time, so one automatic system should be of great value. All interviews were consistent in that proof of need.

3 Results and Achievements

This paper presents an algorithm that grades GitHub repositories and **contributors** to indicate their ranking so it can answer the question: “Should I Use This Project?”. The algorithm is comprehensive to track all **commits** and other actions within the huge public GitHub platform. If hosted on a production grade backend it can assist millions of people and organizations trying to evaluate the security, quality and community of the OSS they use. As mentioned this is a market worth billions of dollars.

During the research process I have overcome many engineering and algorithmic challenges in the way of finding a proper dataset, defining a non-biased benchmark, building and refining over many iterations a working definitive algorithm and finally proving its value.

Using the following Strategic Canvas one can immediately see the value of the developed algorithm, RepoRank, and its innovation adding a missing piece in evaluating OSS:



I hope my results may provide valuable insights to developers and maintainers, helping them in building and evolving systems in a competitive software market.

4 Research and Development Process

4.1 Solution Requirements

Following are the requirements the solution of the research question should answer. **Input** is a **GitHub repository**.

1. **Definitive-** A real number between 0-1 to mark the total grade of the repository. A number that will also allow looking at the top 90% to tag as “best performant” or lowest 10% to disqualify a repository. For that a **normal distribution** is ideal.

   $$X \sim N(\mu, \sigma^2)$$

2. **Scalable-** For the algorithm to work on the real world of the entire GitHub traffic, it must handle the same response time for processing actions even when holding millions of data points.
3. **Accurate-** To be practical it’s better and more common to benchmark to an identification of “good and popular” repositories. Given a set of this kind, it’s expected that the algorithms’ best percentile results will outperform the benchmark results.

4.2 Open Pluggable Feature-Based System

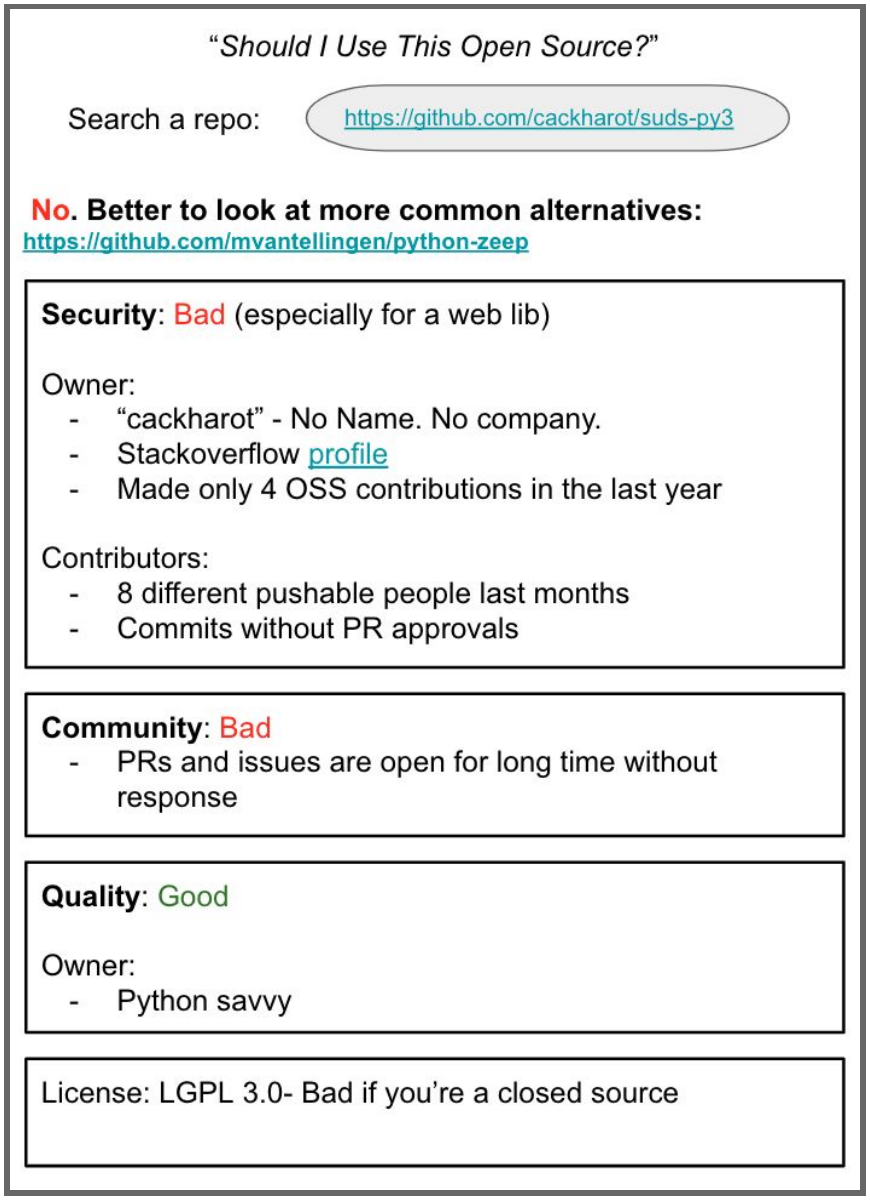
Facing the research problem, the most obvious engineering solution would be to grade each feature and to summarize using adjustable weights to a definitive threshold:

$$g(x, w) = \sum_{i=1}^{n} x(i) * w(i)$$

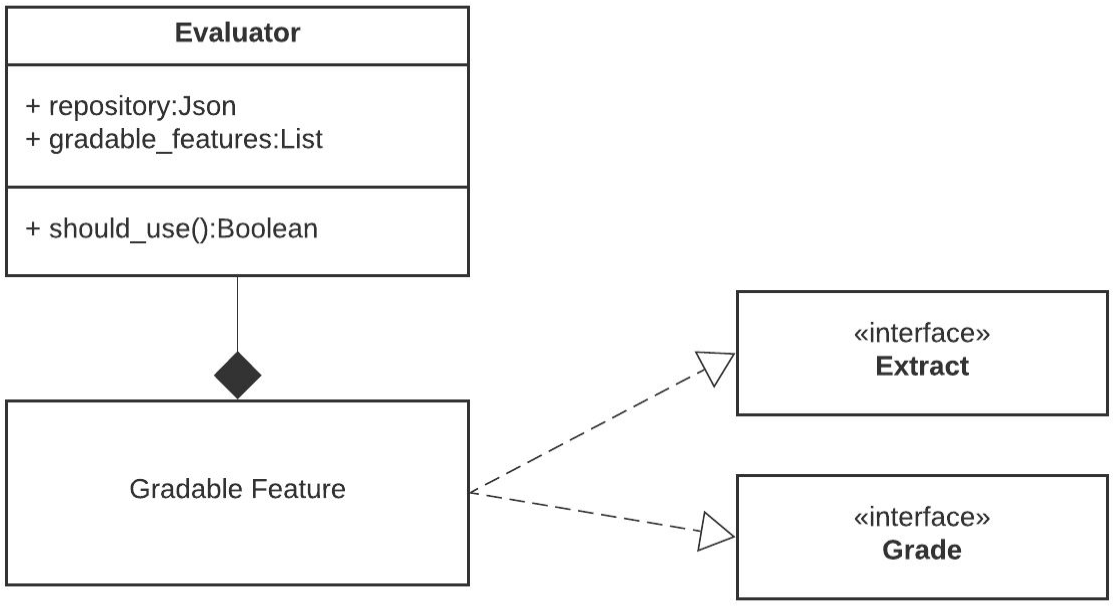
$$f(x, w, threshold) = \{g(x, w) > threshold \Rightarrow True; Otherwise \Rightarrow False\}$$

The system should be pluggable to support dozens of features that can be calculated by an automatic system.

A user interface to visualize an example for such system would be:



A high level Class Diagram:



Downsides of this solution is the lack of ability to **learn** from examples. It's **dependent** on the engineering decisions of the Grade function. Looking at a specific feature it lacks the context to determine its precise grade.

4.3 Dataset Exploration- Github

A dataset is needed to be able to benchmark a solution and implement Machine Learning algorithms to be trained on. After much exploration of open datasets to avoid creating one myself, the one from Carnegie Mellon University by Choudhary Samridhi (2020)[7], Modeling Productivity in Open Source GitHub Projects, was the most useful.

It Contains scraped data from GitHub:

- 6,135,306 commits
- 98,989 unique contributors
- 9073 owners
- 16M issues
- 16,126 unique (and non-forked) Python repositories

- Commits ranging from 1997-02-11 to 2019-03-22

That dataset was collected for the purpose of bursts of activity using markov models, but for the goal of this thesis the raw data is also suitable. It holds the following (sub)scheme per commit:

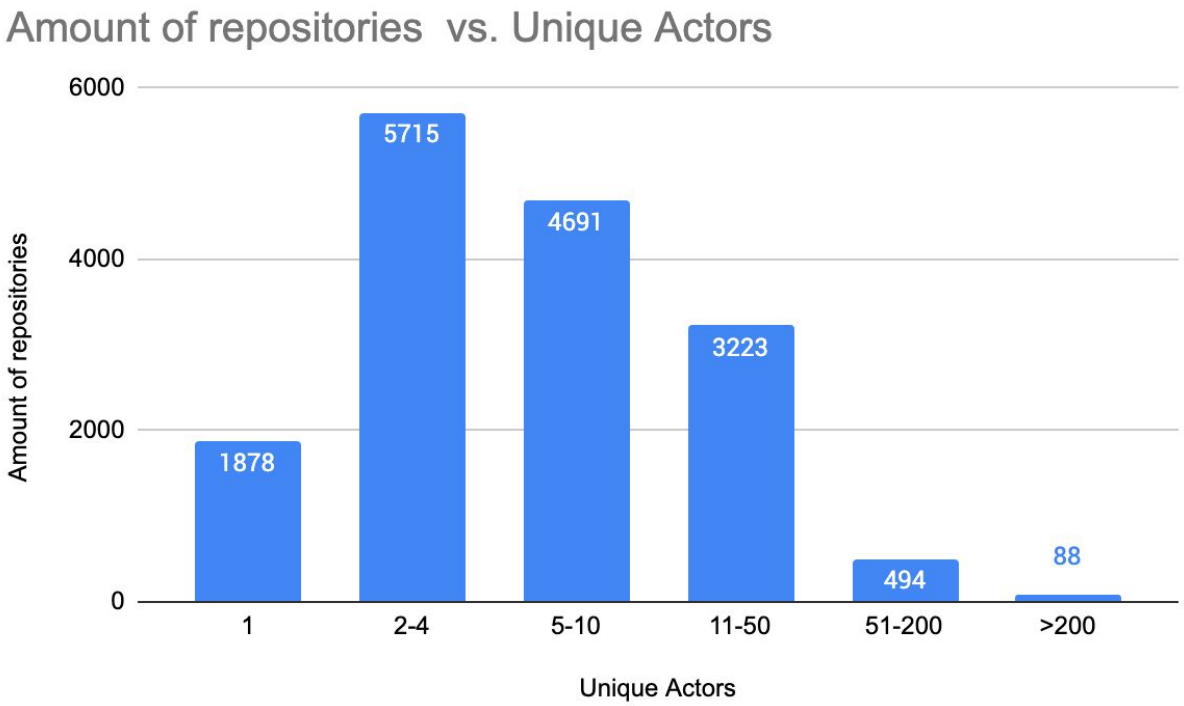
Field in dataset	**Meaning**
Project Owner	Repository owner
Project Name	Repository name
Actor	The contributor who committed
Time	Timestamp of the commit creation
Text	The commit message

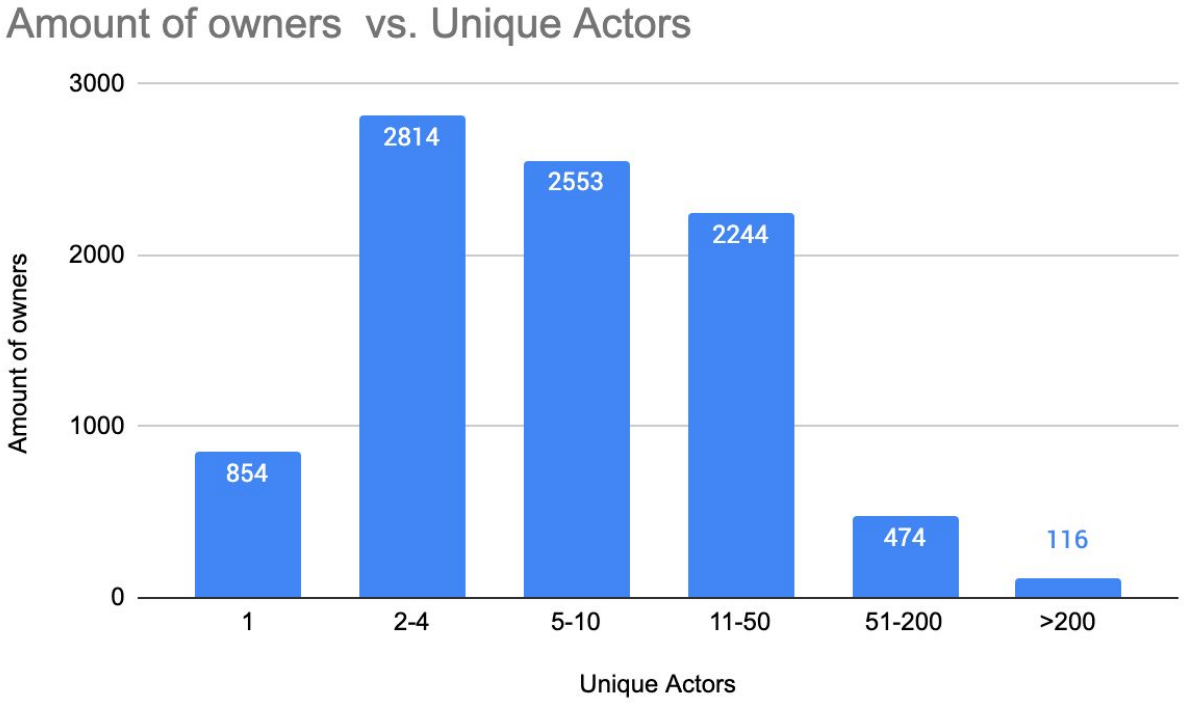
4.3.1 Loading and Exploration

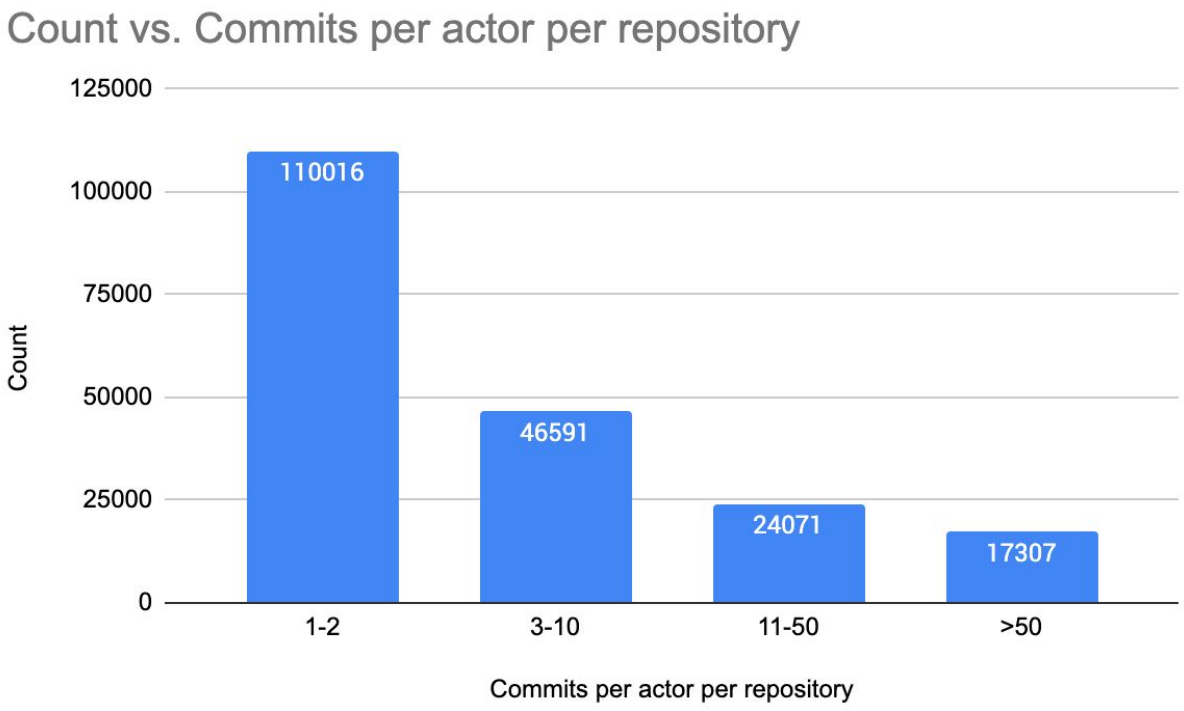
The dataset is spread to thousands of csv files. To be able to explore the data I've created the csv schema and necessary indices inside a local **PostgreSQL** database (running on a 2020 Mac Pro with 32GB of RAM) and then imported it using the following bash script (due to the amount of files it must be in a loop):

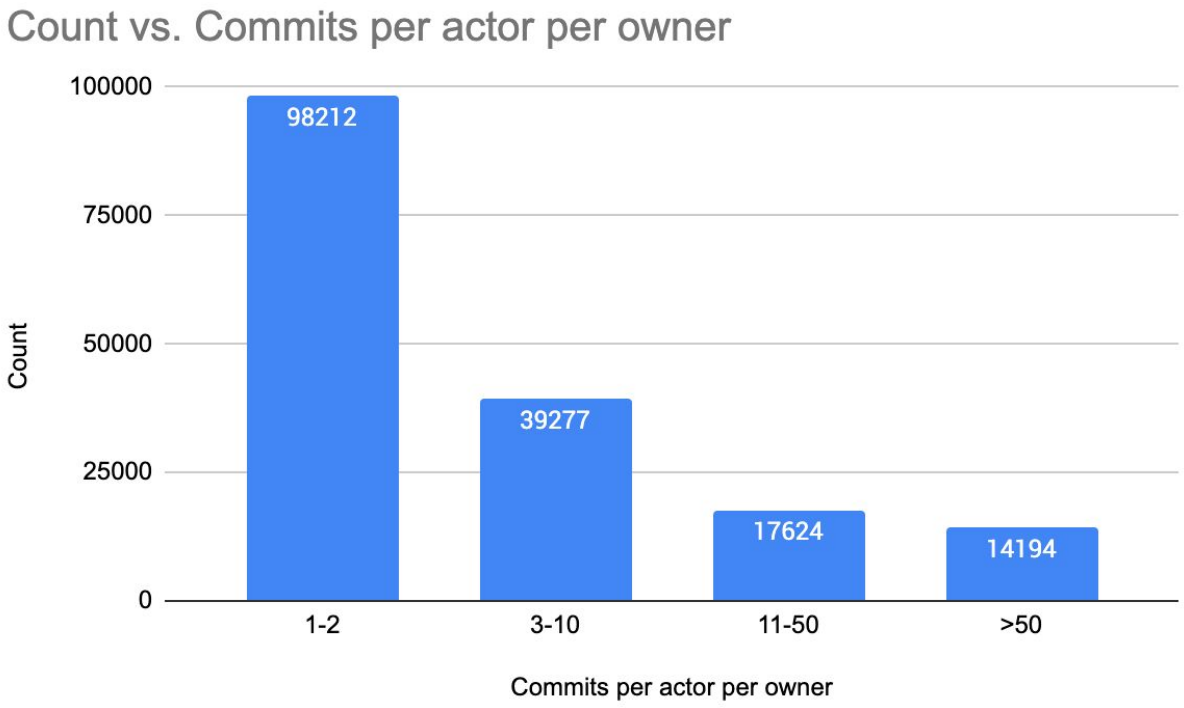
```
for f in *.csv; do
  cat $f | \
  psql -h localhost github_db -c "
    COPY commits(project_name, actor, time, text)
    FROM stdin
    DELIMITER ','
    CSV HEADER"
done
```

Following are dataset analytics to be relevant in next sections:



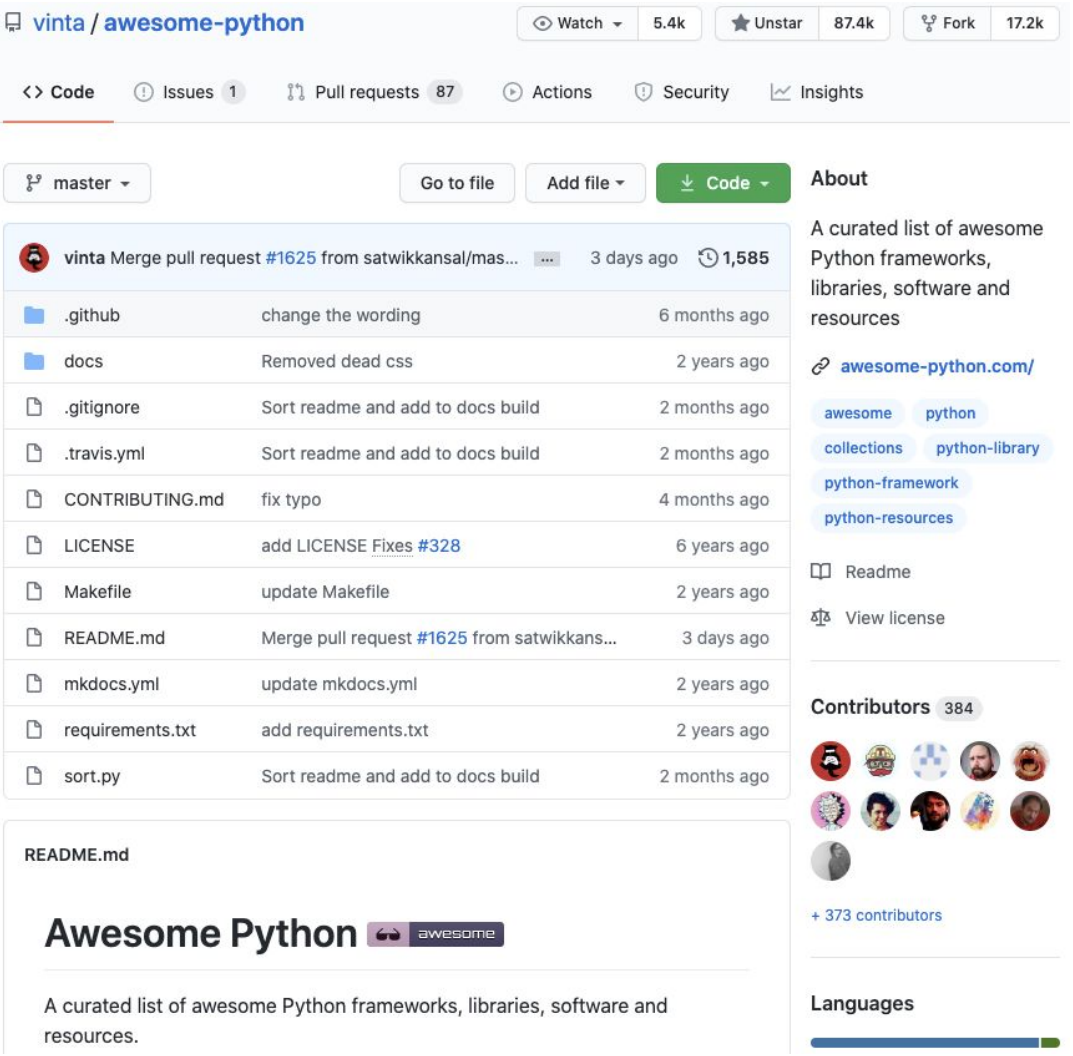



Count vs. Commits per actor per repository
125000
100000
75000
50000
25000
0
Count
110016
46591
24071
17307
1-2
3-10
11-50
>50
Commits per actor per repository


Count vs. Commits per actor per owner
100000
75000
50000
25000
0
Count
98212
39277
17624
14194
1-2
3-10
11-50
>50
Commits per actor per owner

4.3.2 Test Set & Benchmark

Assuming a test group of “good” repositories to measure the precision results is not a straightforward task. This is exactly what the research is trying to accomplish. Luckily, utilizing the “awesome” list will be perfect for the job. In OSS the trend of aggregating and maintaining community lists of repositories references has emerged since 2014. It’s usually moderated by a single owner but requires dozens more votes and popularity indicators to qualify. Considering the above dataset is from the python ecosystem, the awesome-python (https://github.com/vinta/awesome-python, August 2020) list was chosen:

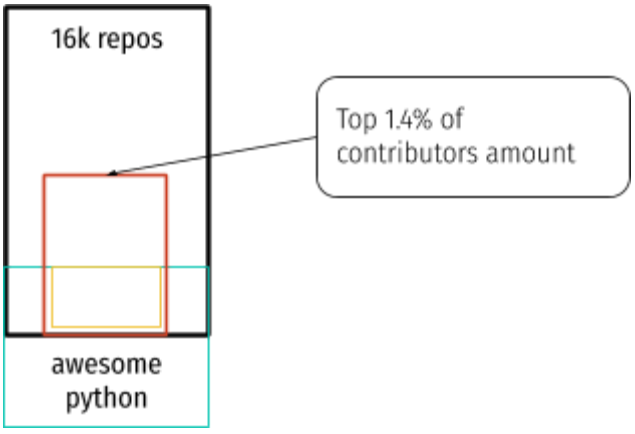


The following table describes some numbers around those datasets.

	awesome-python[25] repositories list	Carnegie Mellon Github commits dataset[1]	Found ratio from Carnegie Mellon as oppose to awesome-python
Unique repositories	504	16,126	228 (1.4%)
Unique owners	441	9,073	269 (2.9%)

Setting a benchmark for this research will be the most common and absolute approach to evaluate OSS: The **amount of contributors**. The common belief is that the more contributors, the better the quality, less chance of security incidents and the better the community. Taking the dataset and aggregating by the top most 1.4% (as the ratio of the test-set to the dataset) of repositories with the highest contributors will result with:

- precision = 28.51% (red area bellow)
- recall = 23.05% (yellow area bellow)



This benchmark of 23.05% to classify correctly a repository as the high end “awesome-python” doesn’t look impressive at first but considering 98.6% of the repositories are not there it is positively surprising that even the simple metric of top contributors has this high correlation.

```
select project_name as repo
from (
    select project_name, actor
    from commits
    group by (project_name, actor)
) dis
group by project_name
order by count(*) desc
```

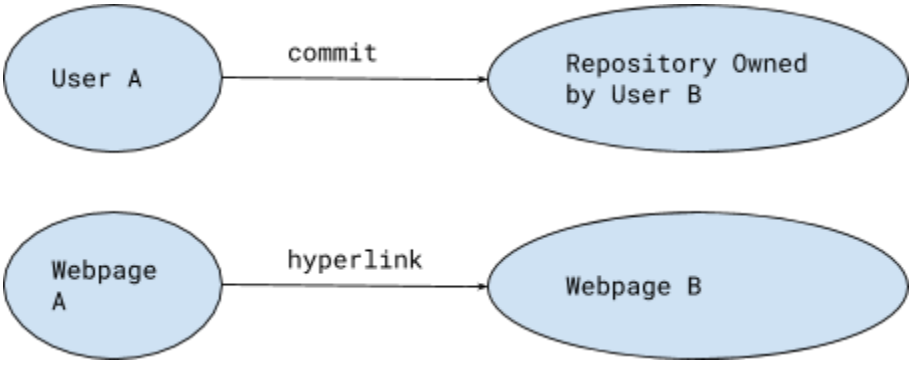
Benchmark is “simple”- using this short SQL to get it

4.4 Focus on People and Model in a Graph

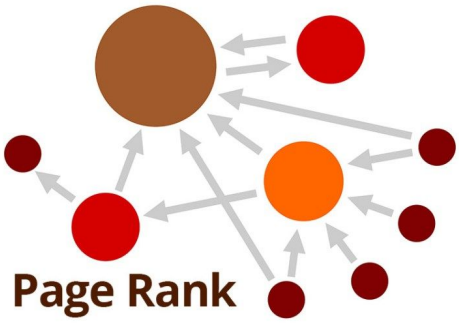
How can we prepare an algorithm that will **learn based on commits** data and metadata of the repository (stars, etc)?
The approach follows the same vector taken in **fraud analysis** or marketing. **Looking on the people behind the repositories-** Would you use a python repository made by a python guru with many years of OSS involvement? Most probably yes, but would the answer be the same if this is someone who has just created his user on GitHub with 0 reputation? Most probably not.

4.4.1 Commits to Hyperlinks Analogy

When someone is making a commit in another person’s repository, they mark high interest in that repository. Same goes for stars, issues, PRs, etc. The analogy rises: commit from a user A to a repository owned by a user B is the same as a hyperlink from a webpage A to another webpage B. A simple diagram for that:



One popular way today to model popularity and credibility of web pages is to model them in a **graph** where the edges are links between 2 web pages. Then the popular **PageRank** can quickly and definitely can determine the score of each vertex. Next step is to try and model the dataset into a graph.



https://kcmarketingagencyllc.com/wp-content/uploads/2020/01/Pagerank.jpg

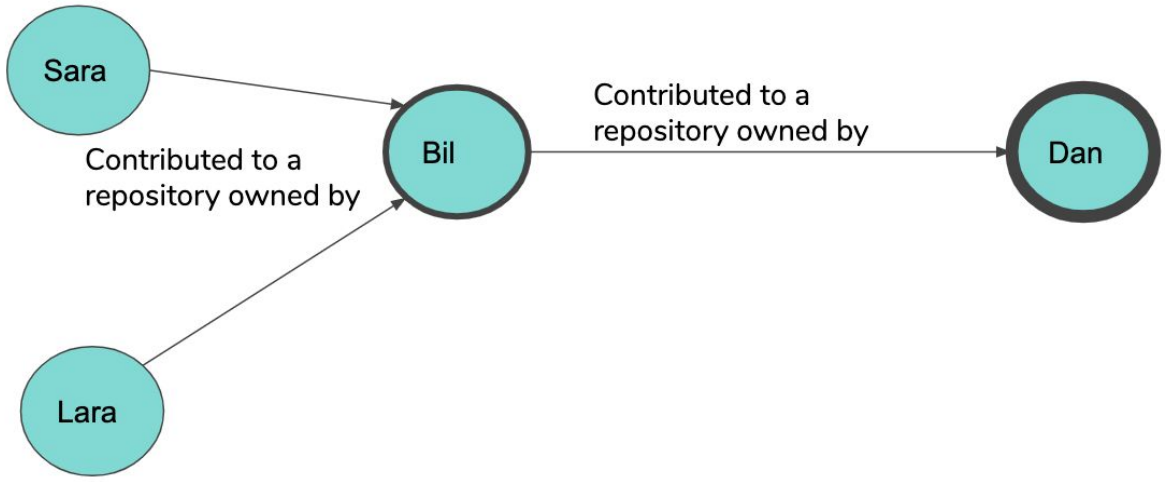
4.4.2 Modeling The Dataset as a Graph

Graph definition:

V = {v for every actor in dataset}

G(v1,v2,w) = {Directed edge v1 -> v2 with weight w as the amount of commits in total that actor v1 contributed to repositories owned by v2}

An example:



Importing the dataset from PostgreSQL into a python networkx graph:

```
from networkx import DiGraph
cursor = psycopg2.connect().cursor()
graph = DiGraph()  # "Di" means directed
cursor.execute("""select project_owner, actor, count(*)
                  from commits
                  group by project_owner, actor""")
for (owner, actor, commits_count) in cursor:
   graph.add_edge(actor, owner, weight=commits_count)
```

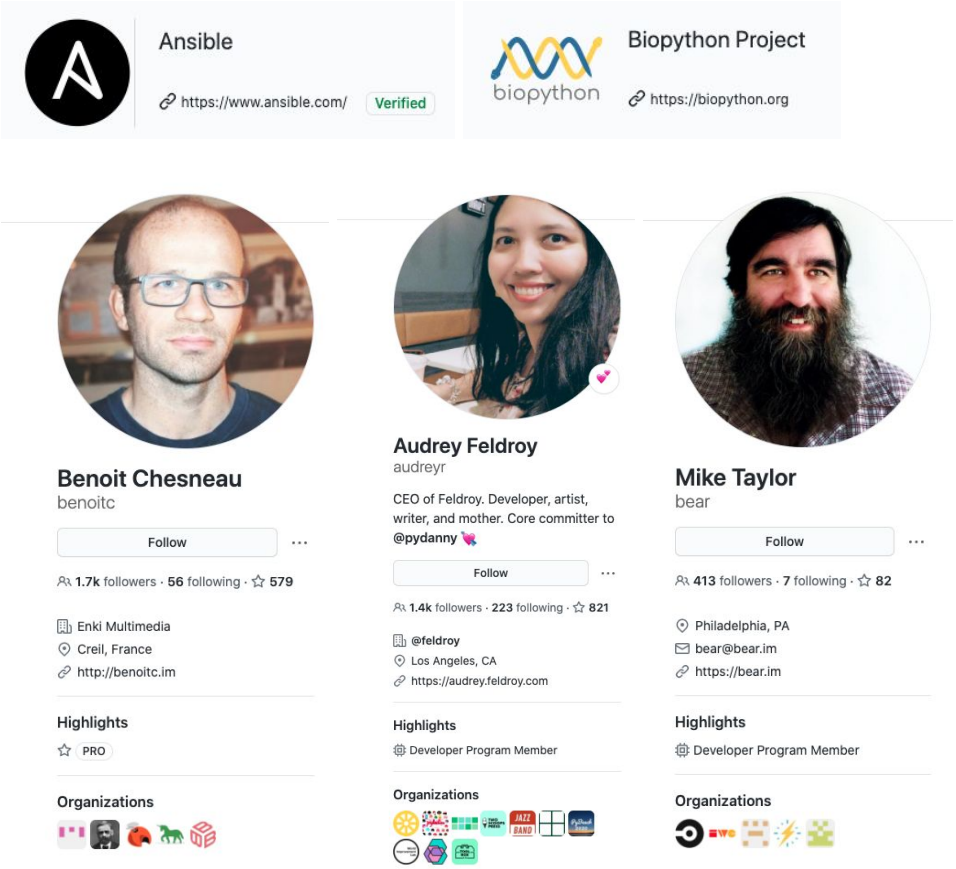
This code running on the dataset db has resulted with a graph with 169,307 edges and 100,392 nodes.

4.4.3 Using PageRank to Grade Users- Early Results

As suggested earlier, to get a "grade" for each owner that can later determine an indicator for an owner's repository evaluation, PageRank is needed. Using the NetworkX library from python with the default alpha=0.85 (the recommended value):

```
from networkx import pagerank
VertexesPageRankGrades = pagerank(graph, weight="weight")
```

Iterating this python dictionary result from the highest grade descending demonstrates that in terms of results sanity, the algorithm came out with reasonable outcomes. The top 5 rated users were in-fact very active OSS advocates from the python community at that time:



Adding the weight to the pagerank algorithm is not the default behaviour of pagerank. One web page holding multiple links to the same destination should not affect the score to avoid abuse but for considering commits however, most likely that the more the commits the more the effort and time a person has invested into a repository so without a doubt this was taken into account from the start.

4.4.4 Reasoning Behind PageRank

The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will

arrive at any particular page. The PageRank computations require several passes, called "iterations", through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

A probability is expressed as a numeric value between 0 and 1. A document with a PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to said document.

The problem of evaluating open source repositories is **the same-** An owner of a repository with a PageRank of 0.5 can mean that there is a 50% chance that a person searching for an open source repository to fit his needs will find the said repository as the most suitable one.

4.4.4.1 Simplified PageRank algorithm on commits data set

Assume a small universe of four repositories all owned by different owners: v_1, v_2, v_3, and v_4. Commits from a repository's owner to itself are ignored. Multiple outbound commits from one repository to another are treated as a single commit. PageRank is initialized to the same value for all owners. PageRank assumes a probability distribution between 0 and 1. Hence the initial value for each owner in this example is 0.25.

The PageRank transferred from a given actor to the targets of its outbound repositories upon the next iteration is divided equally among all outbound repositories.

If the only commits in the system were from actors v_2, v_3, and v_4 to v_1, each commit would transfer 0.25 PageRank to v_1 upon the next iteration, for a total of 0.75.

$$PR(v_1) = \sum_{i=2}^{4} PR(v_i)$$

Suppose instead that actor v_2 had a commit to owner v_3 and v_1, actor v_3 had a commit to repository v_1, and actor v_4 had commits to all three other repositories. Thus, upon the first iteration, owner v_2 would transfer half of its existing value, or 0.125, to owner v_1 and the

other half, or 0.125, to owner v_3. Actor v_3 would transfer all of its existing value, 0.25, to the only repository it links to, v_1. Since v_4 had three outbound commits, it would transfer one third of its existing value, or approximately 0.083, to v_1. At the completion of this iteration, owner v_1 will have a PageRank of approximately 0.458.
$PR(v_1) = PR(v_2)/2 + PR(v_3)/1 + PR(v_4)/3$

In other words, the PageRank conferred by a commit is equal to the owner's own PageRank score divided by the number of commits L().
And in general for any owner u:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

4.4.4.2 Damping factor

The PageRank theory holds that an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor d. Various studies have tested different damping factors, but it is generally assumed that the damping factor will be set around 0.85.

The damping factor is subtracted from 1 (and in some variations of the algorithm, the result is divided by the number of documents (N) in the collection) and this term is then added to the product of the damping factor and the sum of the incoming PageRank scores. That is,

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots\right).$$

So any page's PageRank is derived in large part from the PageRanks of other pages. The damping factor adjusts the derived value downward.
The formula uses a model of a random surfer who reaches their target site after several clicks, then switches to a random page. The PageRank value of a page reflects the chance that the random surfer

will land on that page by clicking on a link. It can be understood as a Markov chain in which the states are pages, and the transitions are the links between pages – all of which are all equally probable.
If a page has no links to other pages, it becomes a sink and therefore terminates the random surfing process. If the random surfer arrives at a sink page, it picks another URL at random and continues surfing again.
Back to commits and repositories - there are much less possibilities of repositories that fit one's needs than the average reader who traverses a lot of documents on the World Wide Web so when taking this into account for the dumping factor it makes sense to lower the default dumping factor of 0.85 to a much lower 0.3. Tests on the dataset however have shown no difference in the relevancy outcome so it was kept with the default of 0.85.

When calculating PageRank, pages with no outbound links are assumed to link out to all other pages in the collection. Their PageRank scores are therefore divided evenly among all other pages. In other words, to be fair with pages that are not sinks, these random transitions are added to all nodes in the Web. Same can fit to actors who don't commit to any other owner.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where p_i are the owners, $M(p_i)$ is the set of actors that commit to p_i, $L(p_j)$ is the number of commits to owner p_j, and N is the total number of actors.
The PageRank values are the entries of the dominant right eigenvector of the modified adjacency matrix rescaled so that each column adds up to one. This makes PageRank a particularly elegant metric: the eigenvector is

$$\mathbf{R} = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_N) \end{bmatrix}$$

where R is the solution of the equation:

$$\mathbf{R} = \begin{bmatrix} (1-d)/N \\ (1-d)/N \\ \vdots \\ (1-d)/N \end{bmatrix} + d \begin{bmatrix} \ell(p_1,p_1) & \ell(p_1,p_2) & \cdots & \ell(p_1,p_N) \\ \ell(p_2,p_1) & \ddots & & \vdots \\ \vdots & & \ell(p_i,p_j) & \\ \ell(p_N,p_1) & \cdots & & \ell(p_N,p_N) \end{bmatrix} \mathbf{R}$$

where the adjacency function $l(p_i,p_j)$ is the ratio between the number of links outbound from page j to page i to the total number of outbound links of page j.

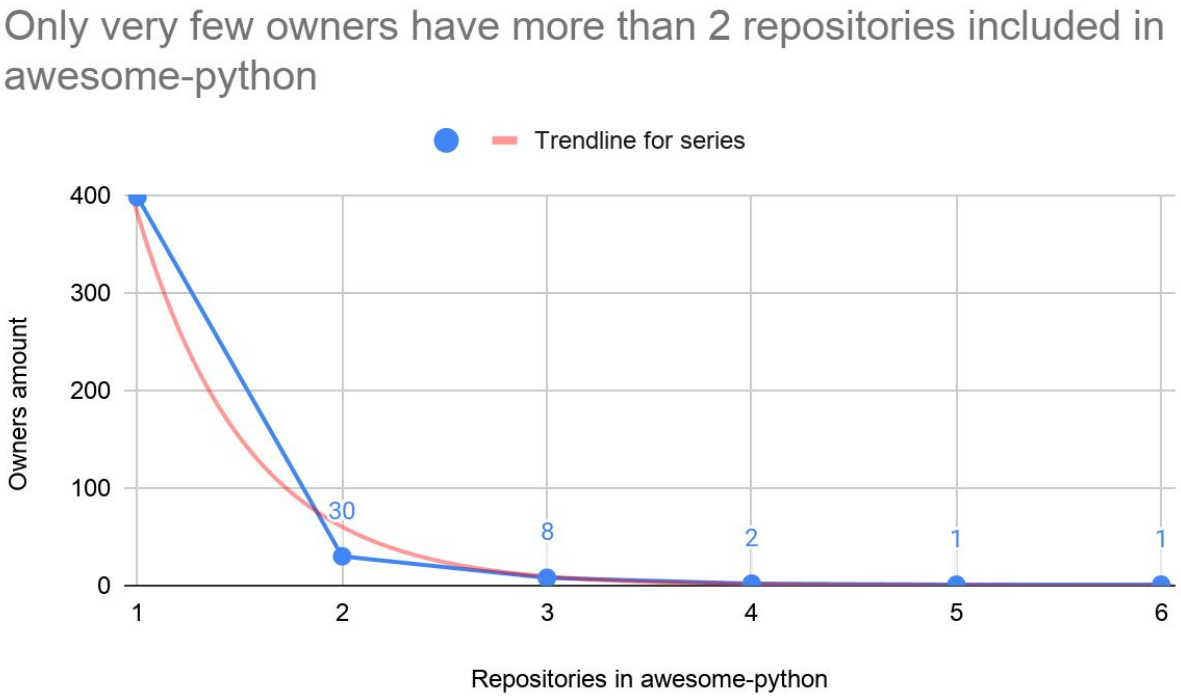
It's known that PageRank converges very fast (e.g. within a tolerable limit of a few dozens of iterations for a network with hundreds of millions of edges). The scaling factor for extremely large networks would be roughly linear in log(n), where n is the size of the network. This is suitable to represent the vast network of GitHub community.

One main disadvantage of PageRank is that it favors older pages. A new page, even a very good one, will not have many links unless it is part of an existing site. Same goes for our case of OSS in which a great, devoted and experienced developer can create a GitHub profile for the first time and upload repositories to there without any collaboration from others for a long time. The algorithm won't be effective in this area. It assumes an owner is "worthy" for high ranking only after gaining traction from other people.

4.5 Development Milestones

4.5.1 Top Owners' Repositories

First idea that came to mind to utilize the graph of owners and contributors with their PageRank grade, was to mark all of the top owners' repositories as positive classification. That led to results several percent **under** the benchmark (high recall, but small precision). It was too "noisy" for high end owners to have repositories which are obsolete or "under-cooked" to be considered as top rated.



4.5.2 Weighted Contributors Grade

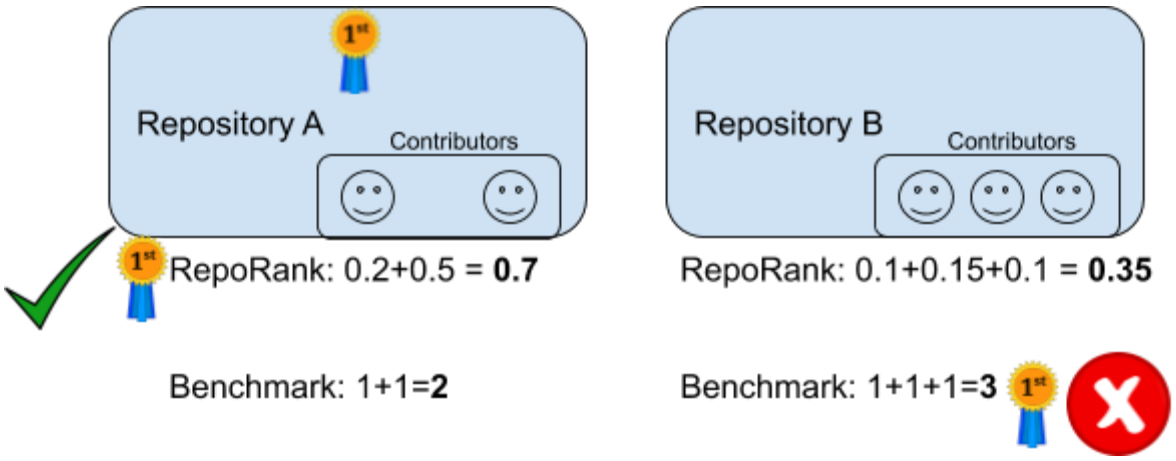
Getting closer to the benchmark idea, simply counting contributors to grade a repository doesn't sound optimal once thinking on the fact that not every contributor is "equal" in terms of their experience,

security risk and commitment to open source. A python guru for instance being involved with a project shouldn't get the same "participation" grade as a novice python programmer. With that idea in mind, **instead of counting the contributors** per repository to grade it, I have **summed up the grades of those contributors** from the PageRank grade of each one. The guru-novice example will be solved by this method thanks to PageRank giving high grades to vertexes getting a lot of accumulated rankings.

Formal definition of a repository grade for a given input of n contributors each marked by V_i for i=0..n and the calculated graph after PageRank marked with PR:

$$RepoGrade(V\,i) = \sum_{i=0}^{n} PR(V\,i)$$

An illustration with example to compare a common case in which weighted contributors grade is better than a simple count for when a smaller count of contributors hold higher end users:



This pivot in the algorithm has greatly improved the results getting closer to the benchmark threshold to beat.

4.5.3 Data Cleansing

Even though the dataset was already used in another research, it was not 100% "suitable" for correct processing and learning. Several improvements were needed:

1. lowering all owners and actors names to prevent changes in capital letters between the data set and the test set.
2. Removing a couple of corrupted commits
3. Removing the owner as the second unique identifier for a repository- A repository name isn’t unique in GitHub. Only the pair of [owner, repository name] is unique. However, in the dataset each repository was chosen once. No forks were taken. Same goes for the test set. There were several repositories in awesome-python that changed their owner with time- moving to be under an official organization rather than under their owning founder. Using only the repository name improved results for both the benchmark and the RepoRank.

4.5.4 Replacing Self Edges

It is known from the PageRank algorithm that no self-edges should exist in the input graph. However in the dataset and in GitHub it is very common to have many commits of users under the repositories they are owning. To prevent those edges, and the bad impact on performance and results, a simple condition was added to keep this activity under a new copied actor name. That enabled the originating owner to still have the right to accumulate ranking with the open source activity being done, but without hurting PageRank.

4.6 Final Results

After solving all the challenges described above and more, the **RepoRank algorithm achieved better results** than the benchmark:

- Precision: 30.26% (**improved by 1.75%**)
- Recall: 24.47% (**improved by 1.42%**)

Those numbers are small but in terms of **relative increase, it's equal to 6% increase**. As mentioned, the market of open source is worth billions so such improvement can translate into millions of dollars of savings for commercial companies on better decisions about the OSS they use.

To get the feeling of what those statistics mean, it's worth to note that RepoRank repositories found from awesome-python were 69 in total as opposed to the benchmark's 65 repositories. There were 51 repositories they both found but RepoRank had 18 repositories unique as opposed to only 14 unique for the benchmark.

The resulting algorithm is simple, fast, requires low resources, and was proven to be better than the common pattern of summing the contributors amount. A weighted grade for the people behind open source is a better feature than the common count in predicting success of an open source repository.

5 Future Work

The impact of basing on PageRank for relative ranking of contributors in the network empowers better evaluation by placing a smarter focus on the developers behind the repository. The algorithm introduced, RepoRank, has the ability to **better evaluate contributors** and with that impact repository evaluation. The overlooked area of evaluating the people behind the code has a lot of potential in making better decisions with OSS assessments.

The GitHub dataset was big but focused only on the python community and from limited years so it’s not as big as the entire GitHub community. To advance the RepoRank graph and it will be beneficial to work on a bigger dataset.

Lastly, the algorithm is also scalable in its features. It can work on any relation factor between people. The stargazer common feature for instance will be also very relevant- A star from a random unrecognized person probably worth a lot less than a star from an OSS guru. Using the same weighted sum of stargazers from the graph instead of the common count of stars is expected to improve this feature as it improved the contributors count.

Adding even more features should give more value as well both on building a better contributors graph and on weighted sum calculations. Examples for features that add more edges to give credit to the owning repository:

1. Author vs Committer - Many repositories are owned by an organization account, not a private user. In that case the committer is not the person who contributed the commit- who is called in GitHub “Author”, while the person that did the pull request review, approval and merge deserves credit as well from the author.
2. Releasers
3. Pull Request

4. Issue
5. Sponsorship

Examples for features that can be more useful as weighted sum instead of the common count:

1. Watchers
2. Follow

As Google uses many other grading mechanisms and not only PageRank, it’s best to use several other approaches existing today to also determine the quality factors of the codebase alongside the weighted grade of the contributors.

In terms of data sources it might be important to analyze network data from more sources. Developers live not only on GitHub but also on twitter, stackoverflow, facebook, linkedin, etc. Adding to the algorithm linking insights from those sources as well can have an important impact (followers on twitter for example). With common APIs those social networks allow, it won’t be a complicated task.

The focus of the thesis was to prove a solution to extract features to grade the people and by that determine the repository’s rating. However to get this mechanism to a product that works with high accuracy and precision-recall, it’ll be best to fuse together the above tools as well.

6 Bibliography

[1]Ahmad, Norita & Laplante, Phillip. (2011). A Systematic Approach to Evaluating Open Source Software. IJSITA. 2. 48-67. 10.4018/jsita.2011010104.

[2]Ahmed, I., Ghorashi, S., & Jensen, C. (2014). An exploration of code quality in FOSS projects. Open Source Software: Mobile Open Source Technologies, 181-190. https://doi.org/10.1007/978-3-642-55128-4_26

[3]Balaguer, F., Cosmo, R. D., Garrido, A., Kon, F., Robles, G., & Zacchiroli, S. (2017). Open source systems: Towards robust practices: 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, proceedings. Springer

[4]Bloemen, Remco (2012): Innovation dynamics in open source software. University of Twente

[5]Borges, H., Hora, A., & Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). https://doi.org/10.1109/icsme.2016.31

[6]Caetano, A., Leite, L., Meirelles, P., Neri, H., Kon, F., & Travassos, G. H. (2018). Using PageRank to reveal relevant issues to support decision-making on open source projects. IFIP Advances in Information and Communication Technology, 102-113. https://doi.org/10.1007/978-3-319-92375-8_9

[7]Choudhary, Samridhi; Bogart, Christopher; Rose, Carolyn; Herbsleb, James (2020): Modeling Productivity in Open Source GitHub Projects: A Dataset and Codebase. Carnegie Mellon University. Dataset: https://doi.org/10.1184/R1/6397013.v1

[8]Diomidis Spinellis, Georgios Gousios, Vassilios Karakoidas, Panagiotis Louridas, Paul J. Adams, Ioannis Samoladas, Ioannis Stamelos, Evaluating the Quality of Open Source Software, Electronic Notes in Theoretical Computer Science, Volume 233, 2009, Pages 5-28, ISSN 1571-0661, https://doi.org/10.1016/j.entcs.2009.02.058

[9]Fagerholm, Fabian (2007): Measuring and tracking quality factors in Free and Open Source Software projects. University of Helsinki

[10] Gupta, Shweta (2018): Software Development Productivity Metrics, Measurements and Implications. University of Oregon- Department of Computer and Information Science

[11] Kahani, Avital; Gutman, Chen (2020): Python Open Source Repositories. Shenkar College of Engineering and Design, Israel

[12] Ke, Weiling; Zhang, Ping (2011): Effects of Empowerment on Performance in Open-Source Software Projects. Engineering Management, IEEE Transactions on. 58. 334 - 346. 10.1109/TEM.2010.2096510

[13] Leitner, David (2017): A Model for Measuring Maintainability Based on Source Code Characteristics. University of Applied Sciences Technikum Wien, Vienna

[14] Maurtvedt, Martin (2017): The Chinese Social Credit System, Surveillance and Social Manipulation: A Solution to 'Moral Decay'?. University of Oslo

[15] McHugh, W. Carter (2018): Open Source Building Performance. Conestoga College Institute of Technology and Advanced Learning, Ontario, Canada

[16] Misra, Sanjay; Adewumi, Adewole; Omoregbe, Nicholas; Crawford, Broderick (2016): A systematic literature review of open source software quality assessment models. SpringerPlus. 1936. 10.1186/s40064-016-3612-4

[17] Nielsen, B. Michael (2017): Quality and IT Security assessment of Open Source Software projects. Technical University of Denmark

[18] Orucevic-Alagic, A. (2016): Understanding Software Development in an Open Source Context: Network Analysis of Source Code Repositories. LTH Tryckeriet E-huset.

[19] Page, Larry; Brin, Sergey (1998): The PageRank Citation Ranking: Bringing Order to the Web. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[20] Rajanen, Mikko (2011): Applying usability cost- benefit analysis - explorations in commercial and open source software development contexts. University of Oulu

[21] Raymond, E. (2001). The cathedral & the bazaar: Musings on Linux and open source by an accidental revolutionary. O'Reilly Media.

[22] Shelbi, Joseph (2014): Reliability Estimation of Open Source Software based Computational Systems. Cochin University of Science and Technology, Kochi, India

[23] Sivonen, Juho (2014): Identifying and Controlling Legal Risks of Open Source Software in a Software Company. Department of Commercial Law, Hanken School of Economics, Helsinki

[24] Sonatype Inc.; Galios Inc.; IT revolution Inc. (2019): State of the Software Supply Chain, the 5th annual report on global open source software development

[25] S. Suzuki, H. Aman, S. Amasaki, T. Yokogawa and M. Kawahara (2017): An Application of the PageRank Algorithm to Commit Evaluation on Git Repository. 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Vienna, pp. 380-383, doi: 10.1109/SEAA.2017.24.

[26] Tahir, Muhammad; Tariq, Aleem (2008): Quality of the Open Source Software. Blekinge Institute of Technology, Sweden

[27] Tamburri, D.A., Palomba, F., Serebrenik, A. et al. Discovering community patterns in open-source: a systematic approach and its evaluation. Empir Software Eng 24, 1369–1417 (2019). https://doi.org/10.1007/s10664-018-9659-9

[28] Wasserman A.I., Guo X., McMillian B., Qian K., Wei MY., Xu Q. (2017) OSSpal: Finding and Evaluating Open Source Software. In: Balaguer F., Di Cosmo R., Garrido A., Kon F., Robles G., Zacchiroli S. (eds) Open Source Systems: Towards Robust Practices. OSS 2017. IFIP Advances in Information and Communication Technology, vol 496. Springer, Cham. https://doi.org/10.1007/978-3-319-57735-7_18

7 Appendices

7.1 Published Article on Medium Kicking of Research- “Should I Use This Open Source”

See https://medium.com/@Look4regev/should-i-use-this-open-source-m-sc-cs-thesis-7549403962ce

7.2 Published Project Homepage

See https://vettopensource.com/

7.3 Research Codebase

See https://github.com/look4regev/oss-research

7.4 The Academic College of Tel Aviv–Yaffo Research Day- 2nd Prize Winner

Poster:

Certificate:

וזאת לתעודה כי
הפוסטר של עבודת המחקר
בנושא

האם כדאי להשתמש בקוד הפתוח הזה?

שהוצג ביום המחקר תש"ף

של

רגב גולן

זכה במקום השני

ד"ר ורד הולצמן
ראש רשות המחקר

פרופ' אילון ועדיה
סגן הנשיא לעניינים אקדמיים

האקדמית תל-אביב-יפו