Curating GitHub for Engineered Software Projects

https://reporeapers.github.io/

Nuthan Munaiah

Steven Kroh

Craig Cabrey

Mei Nagappan

R·I·T

Software Engineering Mix Volume 2: Large-scale Data Analysis of Software Repositories

Register

Overview

Region: North America

Date: July 15 2016 - July 15 2016

Time: 8:30 A.M. – 3:30 P.M.

About

Software Engineering Mix (SE-MIX) provides a forum for our colleagues from academia to interact directly with Microsoft engineers. The program will feature talks from academics: highlights of published research that is highly relevant for Microsoft and blue sky talks summarizing emerging research areas. In addition, practitioners will give presentations about theoretical and pragmatic engineering challenges they face, perhaps soliciting help from academia. A coffee round table setting will be used to facilitate discussions. This session builds on the success of SEIF Days, which provided a discussion forum about the future of software

Why is this topic interesting today?

Why is this topic interesting today?

Access to Data

Why is this topic interesting today?

Access to Data

Computing Power

What can we do with this data?

Data Driven Decision Support for Software Stakeholders

Developer

Maintainer

Operator

Manager

Build Engineer

A Large Scale Study of Programming Languages and Code Quality in Github

728

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, Premkumar Devanbu
{bairay@, dpposnett@, filkov@cs., devanbu@cs.}ucdavis.edu
Department of Computer Science, University of California, Davis, CA, 95616, USA

ABSTRACT

What is the effect of programming languages on software quality? This question has been a topic of much debate for a very long time. In this study, we gather a very large data set from GitHub (728 projects, 63 Million SLOC, 29,000 authors, 1.5 million commits, in 17 languages) in an attempt to shed some empirical light on this question. This reasonably large sample size allows us to use a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, to study the effect of lan-

1. INTRODUCTION

A variety of debates ensue during discussions whether a given programming language is "the right tool for the job". While some of these debates may appear to be tinged with an almost religious fervor, most people would agree that a programming language can impact not only the coding process, but also the properties of the resulting artifact.

Advocates of strong static typing argue that type inference will catch software bugs early. Advocates of dynamic typing may argue

Towards building a universal defect prediction model with rank transformed predictors

Feng Zhang[1] · **Audris Mockus**[2] · **Iman Keivanloo**[3] · **Ying Zou**[3]

1,385

Abstract Software defects can lead to undesired results. Correcting defects costs 50 % to 75 % of the total software development budgets. To predict defective files, a prediction model must be built with predictors (e.g., software metrics) obtained from either a project itself (within-project) or from other projects (cross-project). A universal defect prediction

Quality and Productivity Outcomes Relating to Continuous Integration in GitHub

1,884

Bogdan Vasilescu[†*], Yue Yu[‡†*], Huaimin Wang[‡], Premkumar Devanbu[†], Vladimir Filkov[†]

[†]Department of Computer Science
University of California, Davis
Davis, CA 95616, USA
{vasilescu, ptdevanbu, vfilkov}@ucdavis.edu

[‡]College of Computer
National University of Defense Technology
Changsha, 410073, China
{yuyue, hmwang}@nudt.edu.cn

ABSTRACT

Software processes comprise many steps; coding is followed by building, integration testing, system testing, deployment, operations, among others. Software process integration and automation have been areas of key concern in software engineering, ever since the pioneering work of Osterweil: market

1. INTRODUCTION

Innovations in software technology are central to economic growth. People place ever-increasing demands on software, in terms of features, security, reliability, cost, and ubiquity; and these demands come at an increasingly faster rate. As the appetites grow for ever more powerful software, the hu-

A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance

Shane McIntosh · Meiyappan Nagappan · Bram Adams · Audris Mockus · Ahmed E. Hassan

177,039

Author pre-print copy. The final publication is available at Springer via:
http://dx.doi.org/10.1007/s10664-014-9324-x

Abstract Build systems specify how source code is translated into deliverables. They require continual maintenance as the system they build evolves. This build maintenance can become so burdensome that projects switch build technologies, potentially having to rewrite thousands of lines of build code. We aim to understand the prevalence of different build technologies and the relationship between build technology and build maintenance by analyzing version histories in a corpus of 177,039 repositories spread across four software forges, three software ecosystems, and four large-

An Empirical Study of Goto in C Code from GitHub Repositories

11,627

Mei Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, Ahmed E. Hassan

R·I·T

KYUSHU UNIVERSITY

McGill

Queen's UNIVERSITY

246,657 out of the 2,150,387 files
(11.47%)

Extent of use of goto statements by
Developers is non-trivial

246,657 out of the 2,150,387 files (11.47%)

Extent of use of goto statements by Developers is non-trivial

Most goto usage is for error handling and cleanup

```
1  int fun (int x)
2  {
3    code...
4    if(error)
5      goto err_label;
6    code...
7    err_label:
8      print(error);
9      cleanup(mem);
10     return 0;
11 }
```

Error Handling = 80%

Cleanup = 40%

246,657 out of the 2,150,387 files (11.47%)

Extent of use of goto statements by Developers is non-trivial

Most goto usage is for error handling and cleanup

```
1  int fun (int x)
2  {
3    code...
4    if(error)
5      goto err_label;
6    code...
7    err_label:
8      print(error);
9      cleanup(mem);
10     return 0;
11 }
```

Error Handling = 80%

Cleanup = 40%

Spaghetti code is uncommon

```
1  int fun (int x)
2  {
3    //Spaghetti code due to goto
4    code...
5    if(error)
6      goto err_label1;
7    code...
8
9    err_label1:
10     print(error)
11     code...
12     if(another_error)
13       goto err_label2;
14     code... /* The above code block is skipped
                  when another_error evaluates to true.
                  Thus there is now spaghetti code due to a
                   goto inside the code block of a label.*/
15   err_label2:
16     print(another_error)
17     return 0;
18 }
```

6%

246,657 out of the 2,150,387 files (11.47%)

Extent of use of goto statements by Developers is non-trivial

Most goto usage is for error handling and cleanup

```
1  int fun (int x)
2  {
3    code...
4    if(error)
5      goto err_label;
6    code...
7    err_label:
8      print(error);
9      cleanup(mem);
10     return 0;
11 }
```

Error Handling = 80%

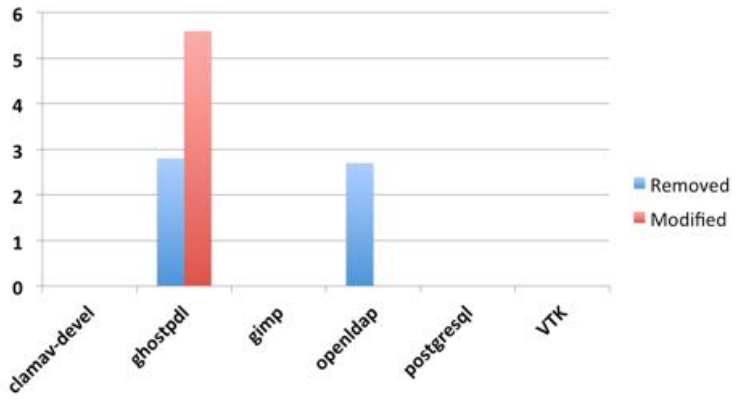
Cleanup = 40%

Spaghetti code is uncommon

```
1  int fun (int x)
2  {
3    //Spaghetti code due to goto
4    code...
5    if(error)
6      goto err_label1;
7    code...
8
9    err_label1:
10     print(error)
11     code...
12     if(another_error)
13       goto err_label2;
14     code... /* The above code block is skipped
                  when another_error evaluates to true.
                  Thus there is now spaghetti code due to a
                   goto inside the code block of a label.*/
15   err_label2:
16     print(another_error)
17     return 0;
18 }
```

6%

Even fewer Gotos are removed/ modified in the post-release bug fixes



However, there is a lurking issue

What are these projects on Github?

85 ± 5% files are system or networking files

Noise

- Student projects
- Tutorial projects
- Personal projects
- Forked projects

We need to choose engineered software projects

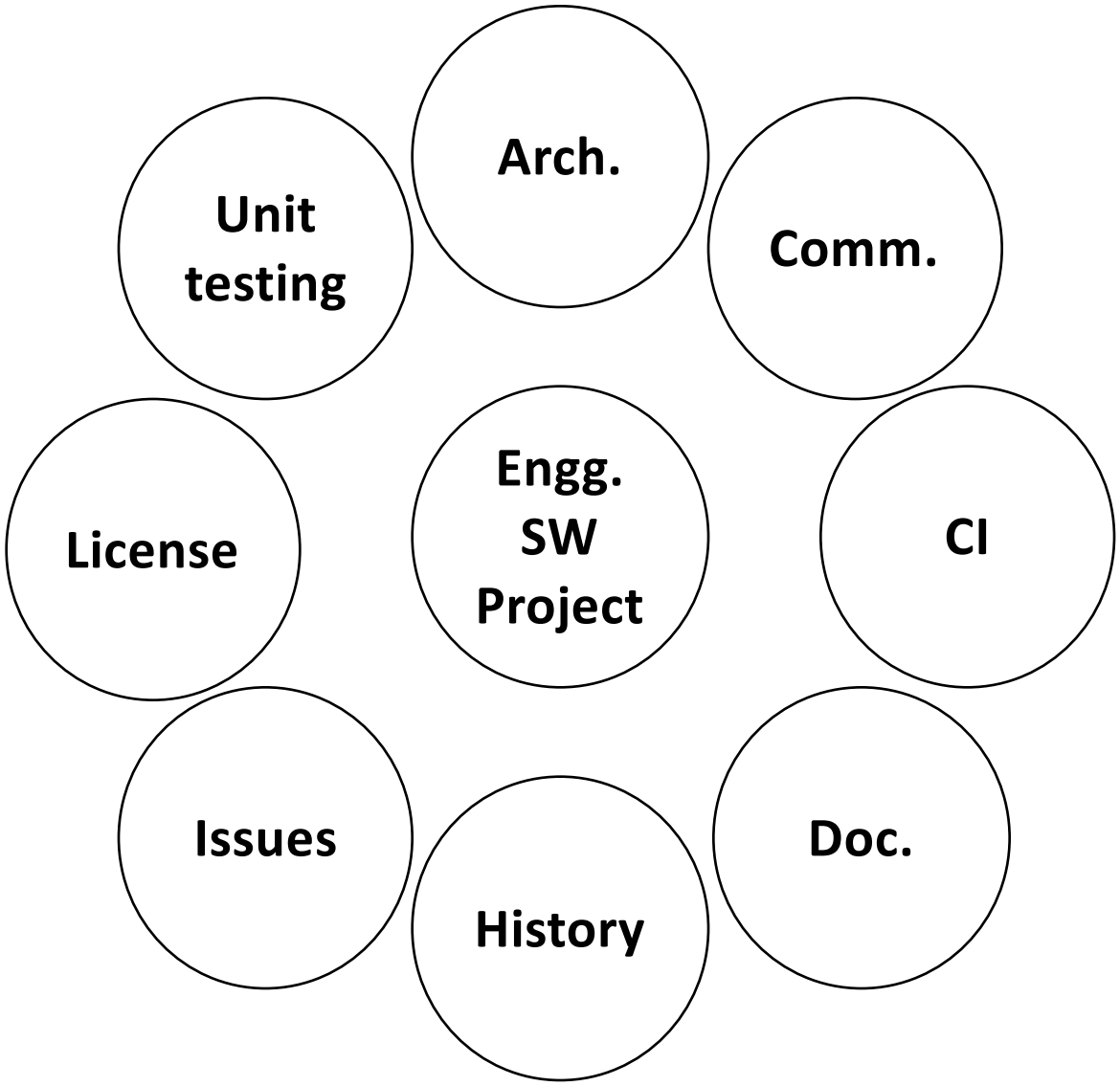
So how are we finding engineered software projects?

Stargazers/Watchers/Forks

Can we do better?

Curate Github to find the engineered software projects

How do we define an engineered software project?


Arch.
Unit testing
Comm.
License
Engg. SW Project
CI
Issues
History
Doc.

$$\text{score}(r) = \sum_{d \in D} h_d(M_d, t_d) \times w_d$$

Metric

Threshold

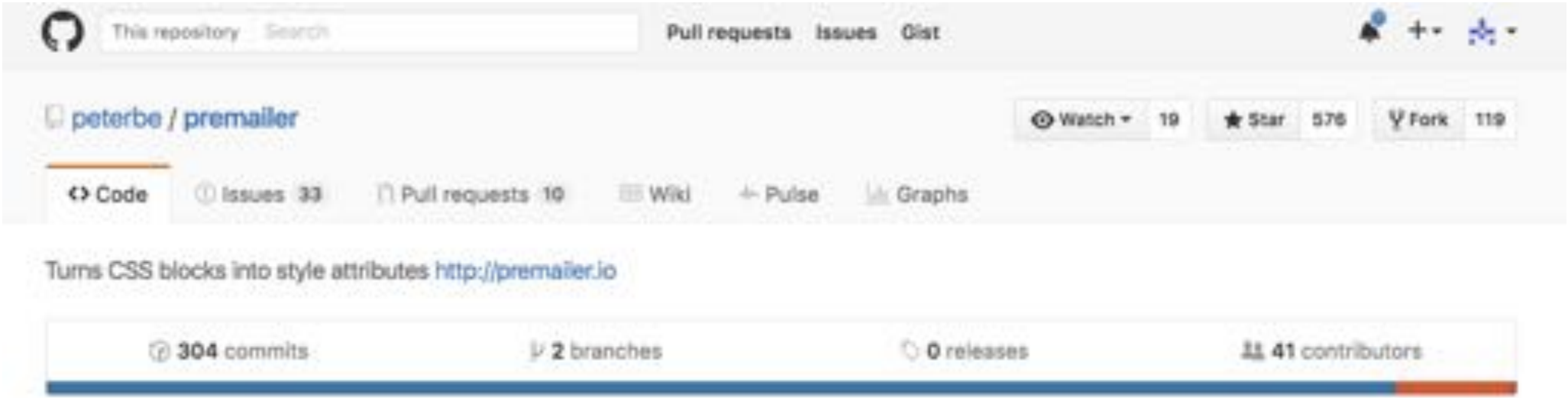
Weight

Thresholds – 150 Github Projects

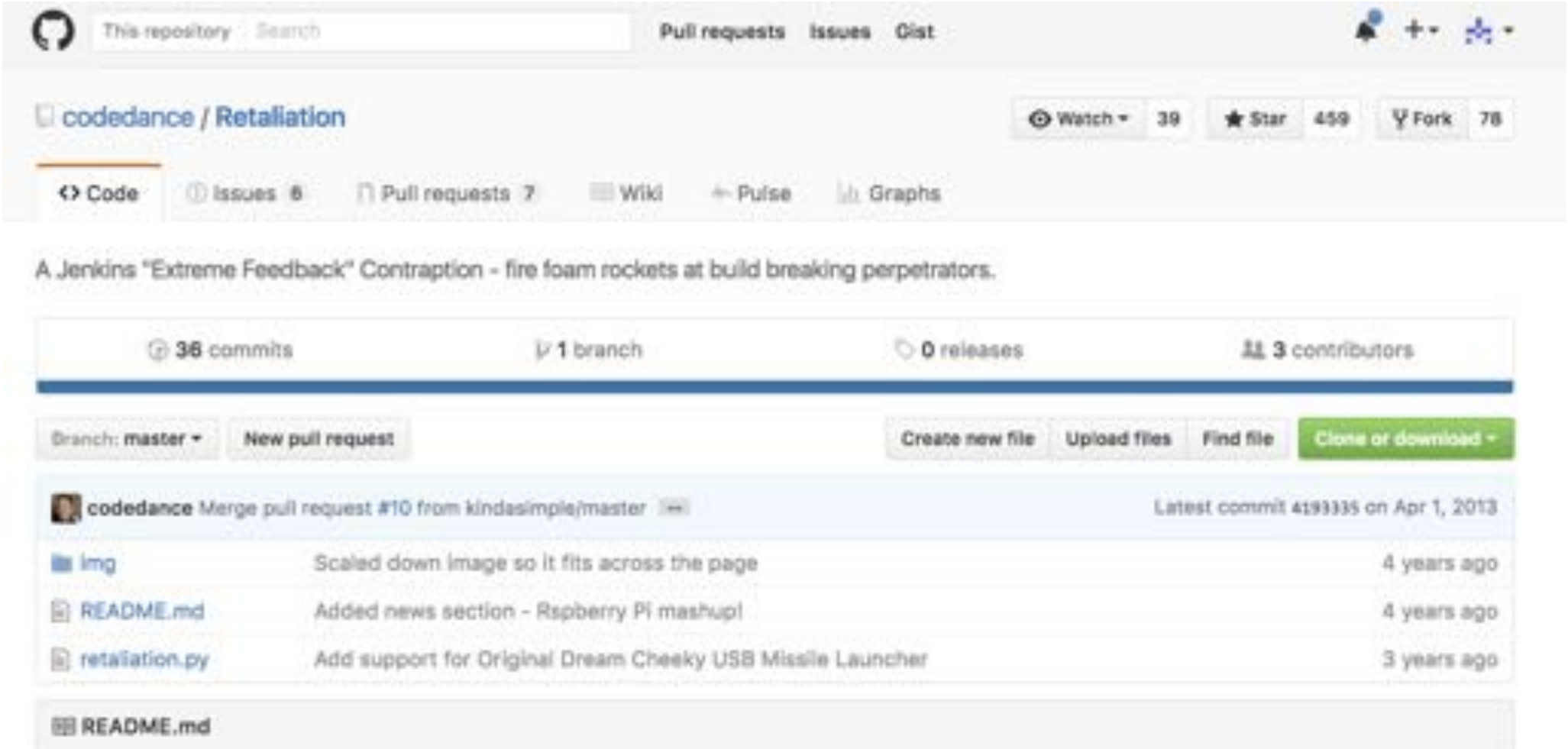
Table 1 Dimensions and their corresponding weights, metrics, and thresholds

Dimension (d)	Weight (w_d)	Metric (M_d)	Threshold (t_d)
Architecture	20	Monolithicity	0.649123
Interpretation: At least 64.9123% of source files must be connected to one another in the largest subsystem.			
Community	20	Core Contributors	2
Interpretation: At least 2 contributors whose commits account for 80% of the total commits.			
Continuous Integration	5	Evidence of CI	1
Interpretation: Evidence of continuous integration usage must be present.			
Documentation	20	Comment Ratio	0.001866
Interpretation: At least 0.1866% of the source lines must be comments.			
History	20	Commit Frequency	2.089552
Interpretation: At least 2.089552 commits per month.			
Issues	5	Issue Frequency	0.022989
Interpretation: At least 0.022989 issues per month.			
License	0	Evidence of License	1
Interpretation: Evidence of a license usage must be present.			
Unit Test	10	Test Ratio	0.001016
Interpretation: At least 0.1016% of source lines must be unit test code.			

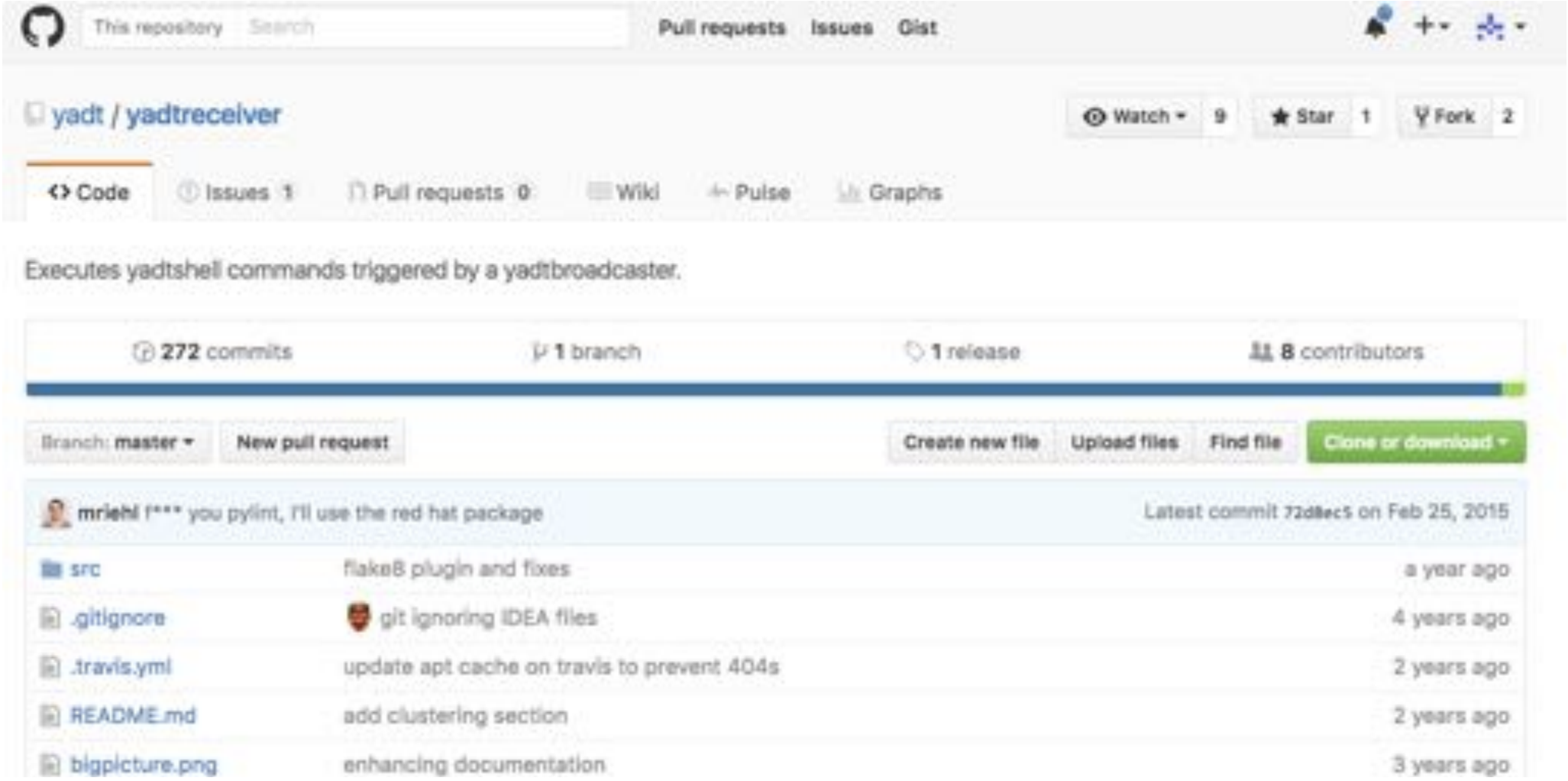
High Stars, High Score (100)



High Stars, but very low score (25)



Low Stars, but very high score (100)



Evaluation – Ground truth data

384 Github repos were manually analyzed

Evaluation – 384 Github repos were manually analyzed

Performance of **reaper**-based classification scheme against the ground truth

FPR	FNR	Precision	Recall	F-measure
26.87%	32.10%	76.56%	67.90%	71.97%

Evaluation – 384 Github repos were manually analyzed

Performance of **reaper**-based classification scheme against the ground truth

FPR	FNR	Precision	Recall	F-measure
26.87%	32.10%	76.56%	67.90%	71.97%

Performance of stargazers-based classification scheme against the ground truth

FPR	FNR	Precision	Recall	F-measure
0.00%	99.31%	100.00%	0.69 %	1.38%

Perfect precision, but very low recall

Performance of **reaper**-based classification scheme against the ground truth

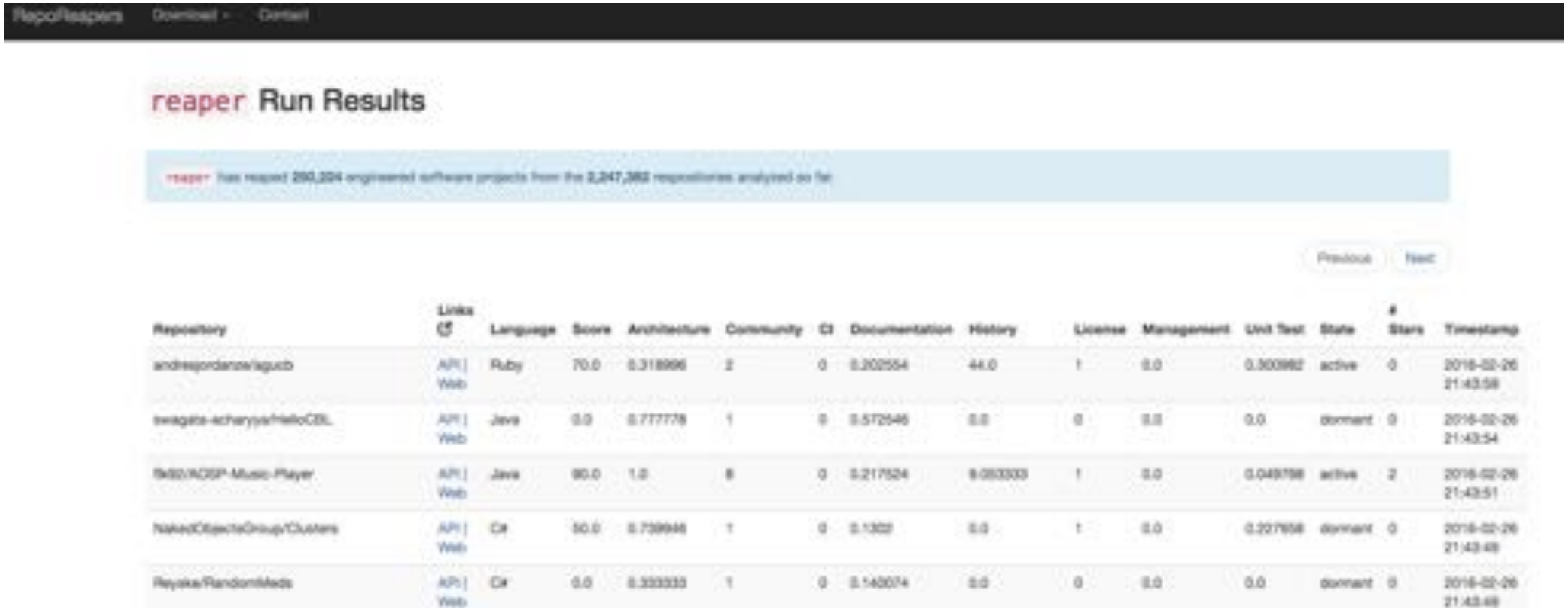
FPR	FNR	Precision	Recall	F-measure
26.87%	32.10%	76.56%	67.90%	71.97%

Performance of stargazers-based classification scheme against the ground truth

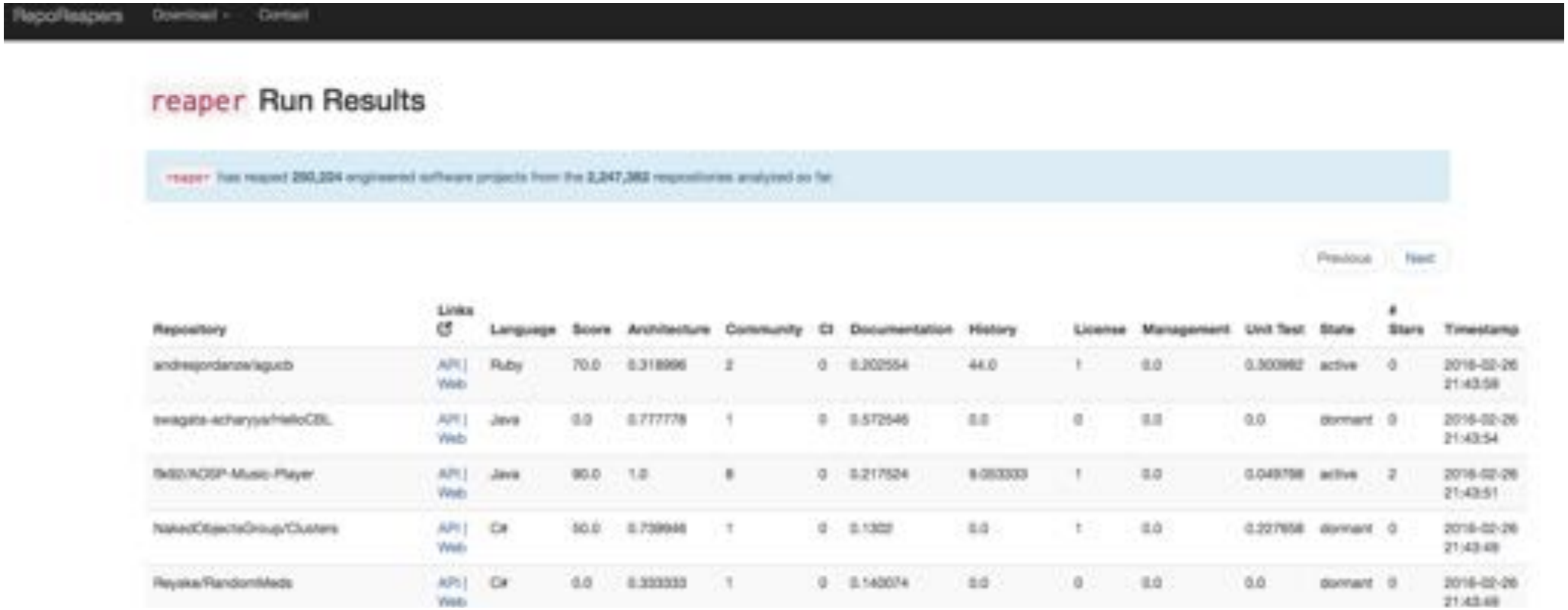
FPR	FNR	Precision	Recall	F-measure
0.00%	99.31%	100.00%	0.69 %	1.38%

Data

https://reporeapers.github.io/



Data https://reporeapers.github.io/



Source Code https://github.com/RepoReapers/reaper