GitHub Heuristics Engine

Yujia Chen
School of Computer Science
North Carolina State University
Raleigh, NC 27695
Email: ychen71@ncsu.edu

Kaustubh Gondhalekar
School of Computer Science
North Carolina State University
Raleigh, NC 27695
Email: kgondha@ncsu.edu

Siddharth Sharma
School of Computer Science
North Carolina State University
Raleigh, NC 27695
Email: ssharm24@ncsu.edu

***Abstract*—In this paper, we propose a GitHub Heuristics Engine to analyze a GitHub project and determine if the project is in bad shape. We do this by building the Heuristics engine on top of git and GitHub statistics engines. We define heuristics and rules that test relevant stats from Git and GitHub, and failing tests signify a bad smell in the project. These Heuristics can be tweaked for better accuracy by using domain knowledge of the project, like project type, deadlines, contributor roles etc. We finally use this engine to analyze 3 software projects from the CSC 510 Software Engineering course, with varying features to derive insights about them. We analyze the problems with the engine and propose further work and improvements.**

I. Introduction

In any software project today, source code management tools like git, mercurial, svn, are integral parts for developers to work and collaborate. A rich web ecosystem has evolved around these tools by providing repository hosting, and more features. Web Services including GitHub, Bitbucket, SourceForge, GitLab, provide tools on top of the SCM(git), like issues, pull requests, milestones, releases to further manage and distribute a project effectively. Using these services provides a great opportunity to view the state of a software project at any given time, and analyze it. By carefully applying heuristics based on software bad practices or "bad smells" on this data, we can determine if a project is in bad shape. This Heuristics Engine can be used like a warning system in an organization, to alert project leaders, or computer science professors teaching a project-oriented class, to know if the student projects are on the right track.

Ideally, such an engine would be agnostic of the SCM(git/mercurial) and the service(GitHub/Bitbucket) used. This is because certain features have been standard across these tools, like commits in SCM, and issues, milestones in repository hosting services. We explore setting up a framework upon which the "Bad smells" test runs on, with flexible core implementations. Some of the factors in deciding the test heuristics are discussed.

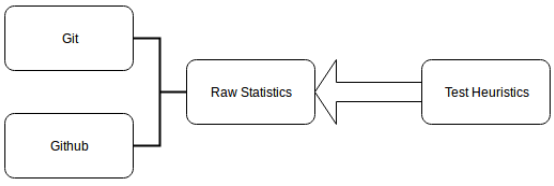

Fig. 1. Heuristics Engine

A. Software Variety

Software comes in large variety with many varying factors. A hardware driver written in C will have a very different repository structure, than a software tool written in a functional language. Today, in web, desktop and mobile applications, programming is usually done on top of a framework, which provides all the bootstrapping code (also called Scaffolding) to get started. Therefore our heuristics need to be flexible to accommodate this variety. Static rules on number of commits, or lines of code would not accurately reflect on the state of a project.

B. Developer Roles

With changing software, developer roles have also become quite varied. Apart from writing code, Designing, Documentation, Testing are now essential parts of the software development process. Therefore lines of code is just one factor, where a successful project needs good tests and documentation. Considering bug reporting, bug fixing and invites public contributors if the project is public also becomes an important part. Open source software is particularly know for the Bazaar model of software development, where a large number of contributors collaborate in small parts. There is also the Cathedral model of development, in which one or two core developers shape the project and contribute the most. These models of development need to be considered before applying the heuristics. The domain knowledge of the project needs to be used, as it gives insight into how to effectively analyze it.

C. Multiple Repositories

A project may span into multiple repositories, distributed among separate core components. In which case, it is better to analyze them together, to get an better idea of the development. GitHub Organization is such a feature, where an Organization can create and manage repositories under it. Analyzing closely coupled repositories together also makes sense if the same developers are working across them. It helps fill the gaps, when a particular repository might see a dip in activity, because of work on another related repository.

These are some of the concerns with heuristics, which need to be considered. Our analysis in the paper is divided into 2 sections, Data Collection and Heuristics. Each is divided into subsections, Git - Source Code Management, and GitHub - Project Management. We believe the separation provides a clearer view of these two aspects.

II. Data Collection

For building the Git and GitHub stat engines, we use a Python script to extract necessary data using the GitHub API. We clean the data, anonymize it and then store them in separate CSV files. The heuristic engines are detailed below.

A. Git Engine

To analyze if a project is going on the right track, we need information about the codebase. We need to look at how frequently are people committing/adding code in a project, who and how much are they adding. Since a project has several milestones/deadlines along the way, we expect to see peaks when approaching a deadline, and a dip of activity in between them.

We use a Python script to communicate and pull data from GitHub via their APIs. GitHub gives you granular data about commits, but we decide to analyze weekly data for the codebase. We think this is appropriate as a project might takes several months to build; also deadlines are approximately setup for every month, therefore weekly data provides us with good insights. We found some good open source statistic tools for repositories like Gitstats[2] , Pepper[1] (which supports git and subversion). For report generation we found GitInspector[3], which is a nifty tool to quickly generate a single page graphical report of a git repository. We currently are not using, but they are robust and can be used in the future to improve, to collect more granular data. Our git statistics revolve mostly around commits as they form the core of the repository. We collect the following stats.

1) commits over time
2) commits per member
3) lines of code over time
4) lines of code per member
5) number of branches

B. GitHub Engine

Issues forms the core part of collaboration on GitHub, and therefore our stats revolve around issues, and activity inside them. In relation to issues, we analyze labels, milestones, comments and pull requests. These give us an insight into project planning and how well the project is being managed. GitHub API provides detailed events of the repository and we clean, anonymize, sort and then visualize it to detect patterns. For each issue posted on the GitHub repository of the project, we collected the following statistics.

1) state of the issue (open / closed)
2) is this a pull request (yes / no)
3) opened and closed timestamps
4) open duration in hours
5) who created this issue (eg. user1)
6) number of comments per issue
7) list of assignees per issue
8) milestone assigned
9) number of labels attached
10) labeled bug or not

III. Bad Smell Heuristics

A. General Heuristics

In order to determine what should constitute as a heuristic for a bad smell within a project, we fist looked at the best practices and in general what constitutes a good development cycle. 'The Manifesto for Agile Software Development' lists twelve golden principles for ensuring good, working software built efficiently and optimally. We also looked at GitHub and Git best practices and the general consensus around them. From these we gathered the following insight about good software projects.

1) Frequent communication: This is perhaps the most important principle about good software projects. The members of the team should communicate their ideas effectively. As a result each member knows where the project is heading and what the others are working on. To judge this aspect of the project we decided to come up with heuristics on the data generated by issues in a GitHub repository. Every issue can be labeled and commented upon for the member to discuss it. Thus the traction an issue received can be judged by the number of comments it has. Also which users post these comments will help us look at the member-participation in discussing these issues.

2) Delivering Working software frequently: Frequent releases is perhaps the most revered best practice in the agile world. We recognize that it is a good practice to distribute the work load over time in a big project, and an ideal codebase would not have sudden peaks and troughs. The metrics measured for this heuristic would be changes in codebase including commit frequency, branches and branch merges, GitHub Releases, and agile practices like Continuous Integration and Testing. Many projects today use free Continuous Integration services like Travis-CI or Circle-CI with their GitHub repositories. If we take the build data, and track tests and build failures over time, this provides us with more opportunities to estimate progress of the project. These practices help in early detection of something wrong with the project. A good way to judge this aspect is to look at how much code actually gets committed in the repository and how frequently. How frequent are the branch merges in the repository, if the developers are working on their own branches or feature branches. We could also potentially check for build failures if the repository is integrated with a Build Engine (e.g Travis CI) for every commit.

3) Ideal Member Contribution: For a project to succeed, it is important for every member in the group to contribute. In a code repository, contribution percentages of members are quite representative of effective coordination between them. If they are skewed, like when a very small percentage of individuals are doing most of the changes, it indicates a problem. A Project may include minor roles and small contributors, which we are not taking into consideration. This metric is only suitable for the core contributors of the project.We define ideal member contribution when, the sum of contributions of any 2 members is greater than the third member. This derives inspiration from the triangle property, where sum of lengths of any 2 sides is greater than length of the third side. This ensures that we don't have a very big skew, while flexible enough to account for minor differences.

B. Git Heuristics

We searched the literature and the Internet for common sense Git best practices[4][6][5]. We found git bad practices that should be avoided[7]. These are common workflow and recommended patterns, which are summarized into our heuristics. We designed them according to a small 2-3 months project, with monthly or bi-weekly deadlines, and how an ideal workflow might looks like. These can be adjusted for different project sizes. We tried to keep it flexible for different models of development, by setting modest thresholds below which we think there is definitely a bad smell. Every heuristic can test multiple statistics to decide whether it meets the minimum threshold. For coming up with the stink score, we calculate for each heuristic how much the deviation is from the minimum threshold and then quantify it.

1) Commit Activity: The most popular best practice for using version control is "Commit early and often". This has many advantages including quicker integration in the codebase, better code reviews, committing related changes. Also in a project, we expect the commits to hit a peak in the middle, and near the deadlines/milestones. If the commit activity is too concentrated in a region over the duration of the project, then we consider it to be a bad smell. We chose 3 as our ideal commit/week number because that seems like a reasonable count for a small project, with 2-3 developers working on it.

Stink score The ratio of stinky weeks vs total weeks. stinky weeks are the number of weeks where commits are less than 3.

2) Member Contribution: As we discussed in the ideal member contribution section in General Heuristics, we expect to have similar contribution levels between developers. The rule is, contribution of any two developers should be greater than of the third one. This tells us that code contributions weren't too skewed. For calculating stink score, we measure how much the rule is violated and then quantify it out of 1.

Stink score The ratio of stinky pairs vs total pairs. The stinky developer pairs are those where the sum of contribution of 2 developers is less than the third one. For a 3 member team, total pairs would be 3.

3) LOC and commit size: Lines of code depend on the language and the framework being used to work on the project. Now in the start of the project, a framework might generate lots of bootstrap or scaffolding code, which we take into consideration. In general, commit sizes should be consistent, and when we see a very unusual commit size, it's a bad smell. It indicates that lots of changes were done without intermediate commits, violating the "commit early often" principle.

Stink score The ratio of very unusual sized commits vs total repository size. A commit which contributes 50% of total code in repository will have 0.5 stink score.

4) Branches: When there are Multiple developers, working on a single master branch is a bad practice. This usually results in conflicts, and poor workflow. Ideally each developer should have their own branch. When working on multiple features, then a branch for each self contained feature is recommended. After the work on feature is complete, the feature branch is then merged with the stable branch, and usually deleted so that it doesn't pollute the commit history. Therefore in our heuristic we look for adequate branching (according to number of developers), and the absence indicates bad smell.

Stink score If 1-(total branches/total developers). The lesser the number of branches the more the stink score.

C. GitHub Heuristics

We found general recommended workflows for small teams when working on GitHub[8] or Bitbucket[9]. Issues are at the heart of project management, with milestones, labels, comments to micro-manage them. We expect the issues to have milestone to indicate delivery, good labeling[10], relevant people assigned, and some traction in comments. We summarize our heuristics for detecting bad smells in GitHub below.

1) Issue Activity: This heuristic tracks the number of issues over the project duration, and type and traction in each issue. We first look at the issue distribution over time, and expect them to be distributed almost evenly. If there is a very large concentration of issues near deadlines, then it indicates a bad smell. Because this usually indicates late discovery of issues in the repository when closing to the delivery deadline.

Next we look at Issues activities like comments, label and milestones. We know that using GitHub to completely manage a project is a new experience for students and teams, so we came up with a heuristic which says that the number of comments on an issue varies linearly with the number of hours the issue was open. For issues opened and closed per week, ideally a team should close and open 2 issues.

Stink score Comments per issue -
ratio of comments below the regression line to total comments. We felt that considering all the comments data in a single linear regression didn't make much sense as the range of open issues varied from about 1 hour to about 1200 hours. We divided the comments data into two graphs: comments on issues open for less than 48 hours and comments on issues open more than 48 hours and applied linear regression separately on these. This we felt should better show any stinks if present in the group.

total issues opened per week -
(stinky weeks/total weeks)

total issues closed per week -
(stinky weeks/total weeks)
We considered the total duration to be equal to 10 weeks.

2) All Issues: This heuristic is geared to look at the level of engagement in all the issues as a whole. Since GitHub provides features like milestones, labels, issue-assignees to increase productivity and help in managing issues. Our heuristics try to look at how well are the teams using these features to manage their project.

Stink score percent of issues associated with milestones - percent issues not associated with milestones

Heuristic	Statistic	Ideal
Commit Activity	commits per week	>= 3 commits/week
Author Contribution	LOC changed per author commits per author	Ideal Member Contribution Ideal Member Contribution
Commit size	LOC per commit	Linear Regression Model
Branches	branches per member	>= 1

TABLE I. GIT HEURISTICS

Heuristic	Statistic	Ideal
Issue Activity	comments per issue total issues opened per week total issues closed per week	linear regression model >= 2 >= 2
All Issues	percent of issues associated with milestones percent of issues labeled as bugs percent of issues labeled number of issues assigned per user	>= 75% of Issues >= 20% of Issues >= 75% of Issues ideal member contribution

TABLE II. GITHUB HEURISTICS

percent of issues labeled as bugs - >= 20% with error rate of 10% is considered to be stink score = 0. Otherwise stink score = 1

number of issues assigned per user - stinky pairs/total pairs

IV. CS510 GROUP 1 PROJECT ANALYSIS

Note: Group 0 and Group 1, in the graphs below are two repositories for the same team. The stink scores depict a score for the combined groups.

Commit Activity

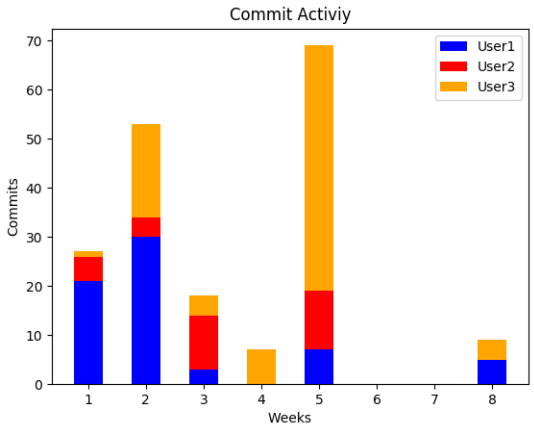

Fig. 2. Group0 Commits per week

The commits are fairly even at the start and peak in the middle. Distribution of commits indicate that the initial part was worked upon by User1 and the later part on User2 and User3. We see a slight skew in number of commits with User2 being considerably low than commits by User1 and User3. Weeks 6 and 7 see no activity at all which is a bad smell.

Stink score = 2/8 = 0.25

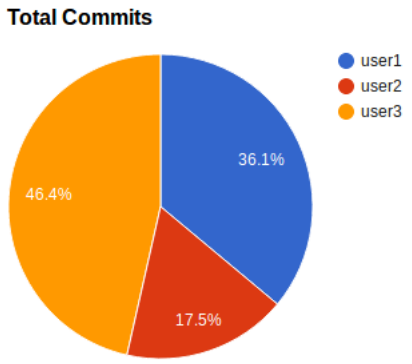

Fig. 3. Group0 Commit Percentages by Users

Although contribution of User2 is comparatively lower, but the percentages of commits are fairly even with no developer completely dominating the other 2 developers. This satisfies our "Ideal member contribution" criteria.

Stink score = 0

Repository Activity in Lines of Code

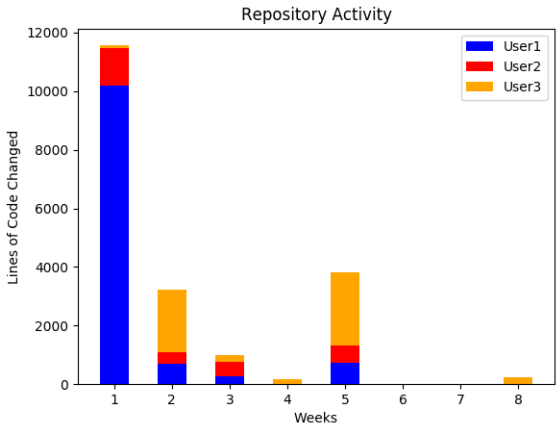

Fig. 4. Group0 Lines Of Code Changed per week

The Lines of Code contributed in the beginning indicate

committing automatic code generated by a framework. Therefore leaving the first week the commit sizes look consistent.

Stink score = 0

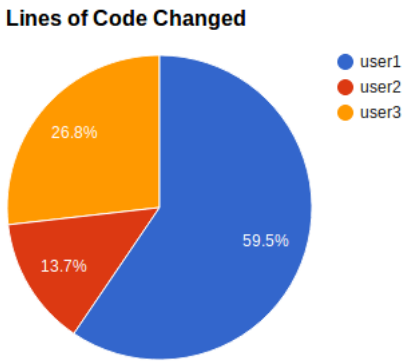

Fig. 5. Group0 LOC Percentages by Users

The code contribution by User 1 is the largest, almost 60%. This is a bad smell.

Stink score = 2/3 = 0.66

Branching

Since there were 2 repositories

Repository 1:
Developer Branches : 4
Feature Branches : 0

Repository 1:
Developer Branches : 2
Feature Branches : 3

Stink score = 0

Issue Activity

Comments per issue

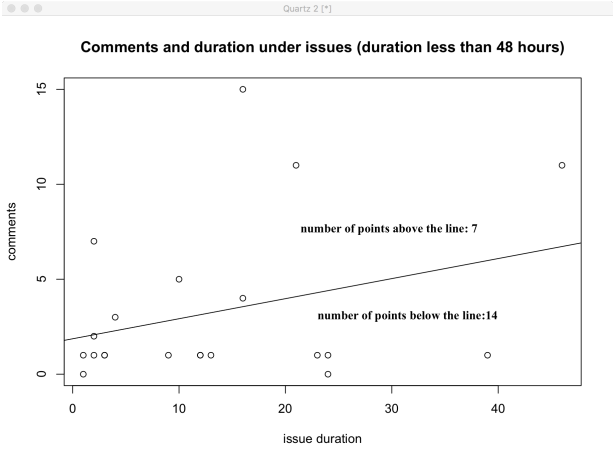

Fig. 6. Group0 Linear Regression on comments on issues lasting less than 48 hours

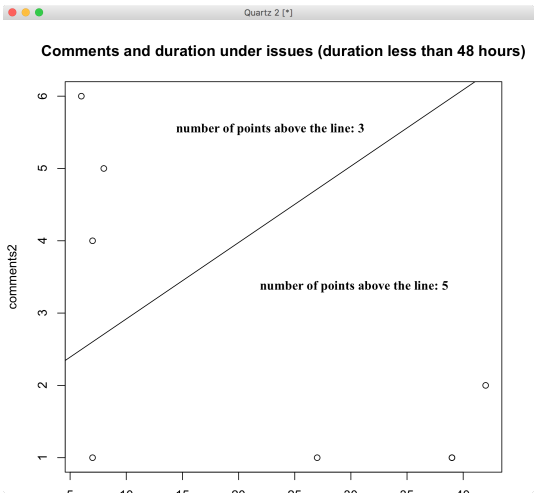

Fig. 7. Group1 Linear Regression on comments on issues lasting less than 48 hours

We see that for issues lasting less than 48 hours, the combined ratio for comments below the regression line to total comments is (14+5)/(21+8) = 0.62

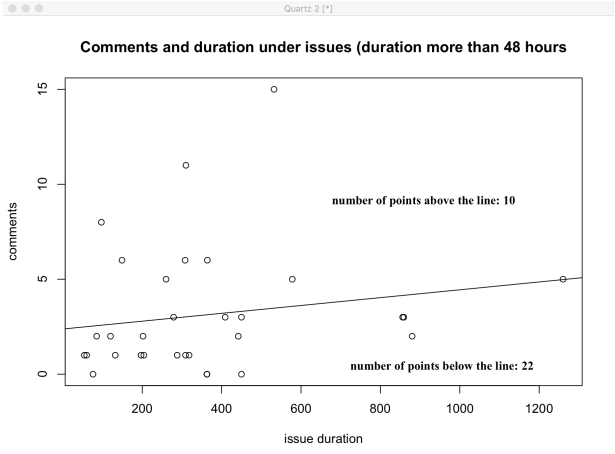

Fig. 8. Group0 Linear Regression on comments on issues lasting more than 48 hours

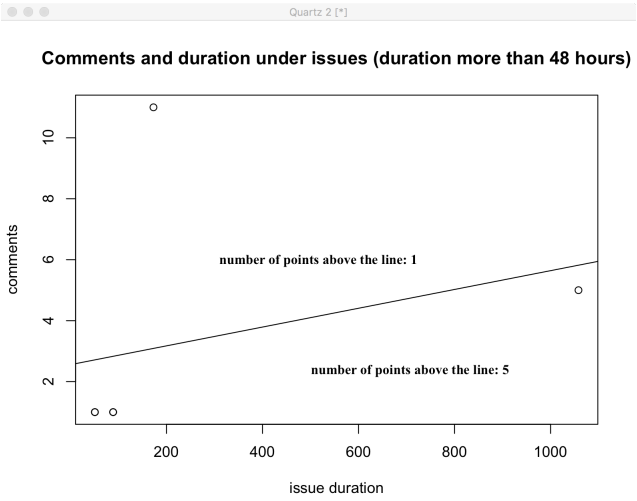

Fig. 9. Group1 Linear Regression on comments on issues lasting more than 48 hours

For comments on issues lasting more than 48 hours, the ratio is (22+5)/(32+6) = 0.71

Stink Score = (0.62 + 0.71)/2 = 0.67

Issues opened per week

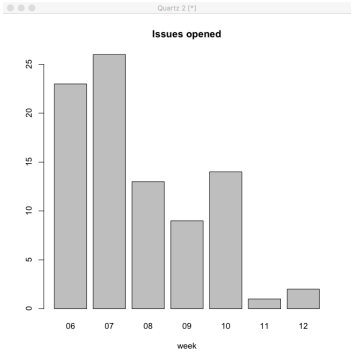

Fig. 10. Group0 Issues opened per week

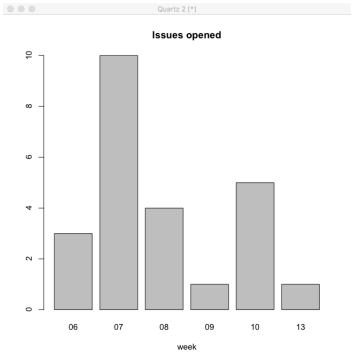

Fig. 11. Group1 Issues opened per week

We see that the combined issues opened per week are always greater than equal to 2, hence the stink score is equal to zero.

Stink score = 0

Issues closed per week

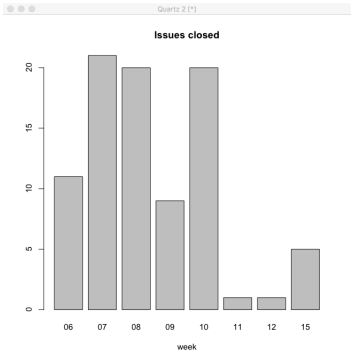

Fig. 12. Group0 Issues closed per week

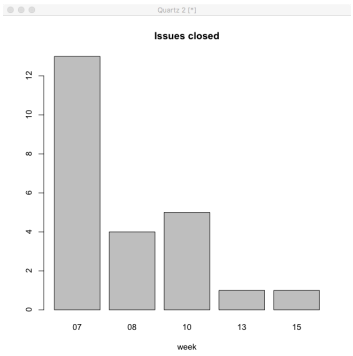

Fig. 13. Group1 Issues closed per week

Stink score = 0

We see that the combined issues closed per week are always greater than equal to 2, hence the stink score is equal to zero.

Total stink score for this heuristic: $0.67 + 0 + 0 = 0.67$

All Issues

percent of issues associated with milestones

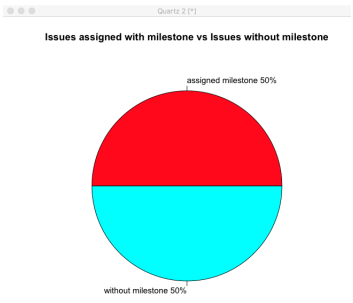

Fig. 14. Group0 Issue Milestone pie chart

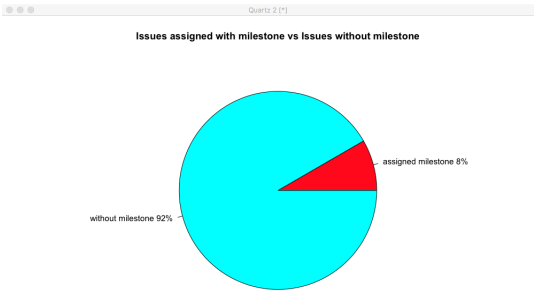

Fig. 15. Group1 Issue Milestone pie chart

Combined issues associated without milestones = 71 %.

Stink score = 0.71

percent of issues labelled as bugs

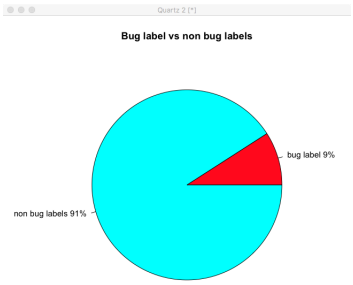

Fig. 16. Group0 Bug Label percentage

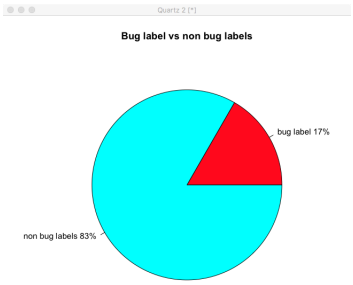

Fig. 17. Group1 Bug Label percentage

The combined issues labelled as bugs is equal to 13 percent which falls within our error limits of 10%. Thus the stink score for this metric is equal to zero.

Stink score = 0

number of issues assigned per user

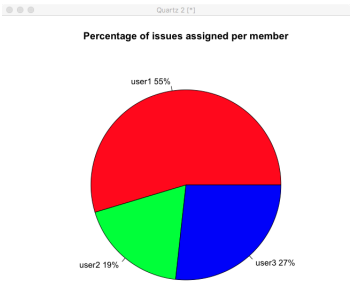

Fig. 18. Group0 Issue assignee percentage

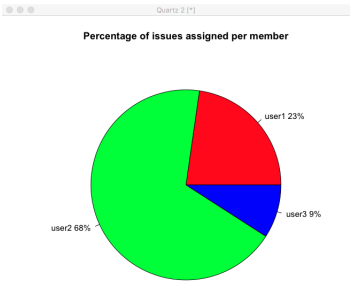

Fig. 19. Group1 Issue assignee percentage

We see that the issues assigned don't conform to the uniform user contribution (a + b ¿ c) and thus this metric has a stink score of 1.

Stink score = 1

Total stink score for this heuristic = $0.71 + 0 + 1 = 1.71$

1) Group 2: **Commit Activity**

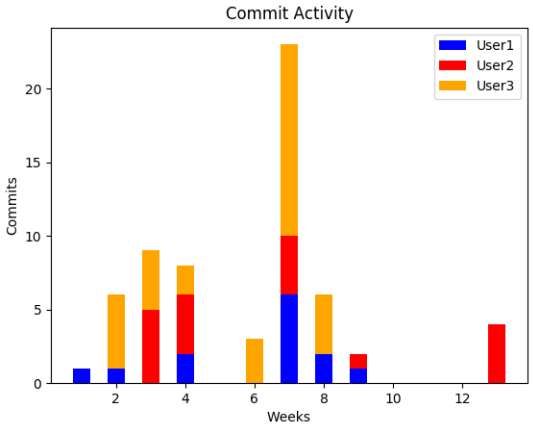

Fig. 20. Group2 Commits per week

Commit activity looks fairly even and distributed with a peak in the middle. User 3 dominates the commit activity in

the middle. There are 4 weeks without any commits and 2 weeks of low activity.

Stink Score = 6/12 = 0.5

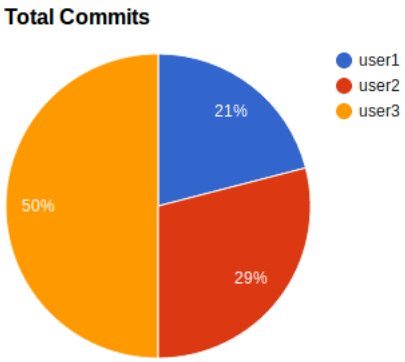

Fig. 21. Group2 Commit Percentages by Users

User 3 contributed 50% of the total commits. Therefore it just manages to cross our minimum contribution rule, and no developer dominates any other pair.

Stink Score = 0

Repository Activity in Lines of Code

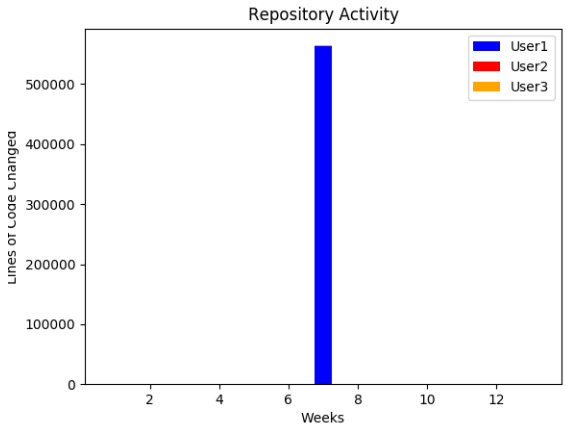

Fig. 22. Group2 Lines Of Code Changed per week

User 1 commits more than 95% of code in week 7. While analyzing the repo we observed that User 3 committed a massive amount of code and then deleted it. When considering the lines of code changed, this doubles the adding and deleting the code. Here this heuristic might be less accurate because this massive code does not remain in the final codebase for long. But still it was a costly mistake on the part of the developer.

Stink Score = 0.95

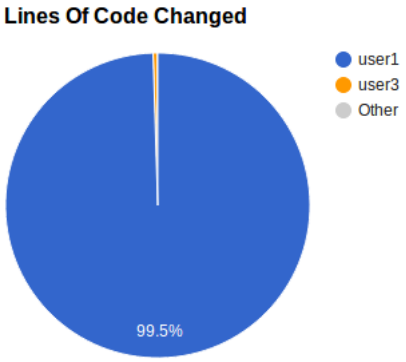

Fig. 23. Group2 LOC Percentages by Users

The LOC contributed because of the massive code committed skews heavily in favor of User1. This is a bad smell.

Stink Score = 0.95

Branching Branches : 3

Many branches were deleted after pull request so 3 is not the total branches created during the project, but it cleares our minimum criteria.

Stink Score = 0

Issue Activity

Comments per issue

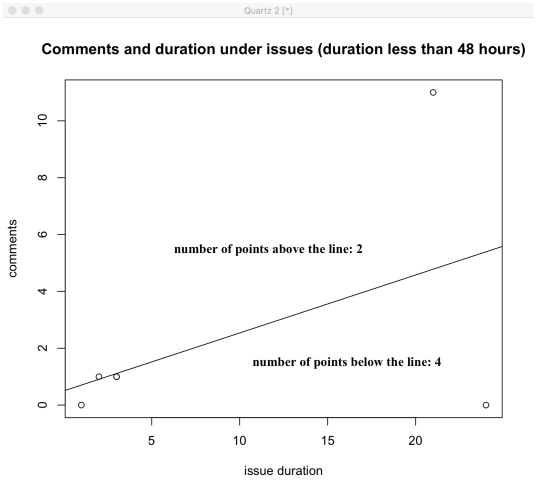

Fig. 24. Group2 Linear Regression on comments on issues lasting less than 48 hours

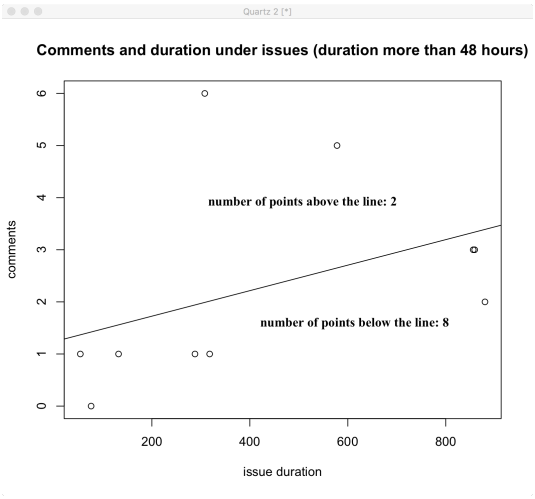

Fig. 25. Group2 Linear Regression on comments on issues lasting more than 48 hours

Stink score = 0.75

Issues opened per week

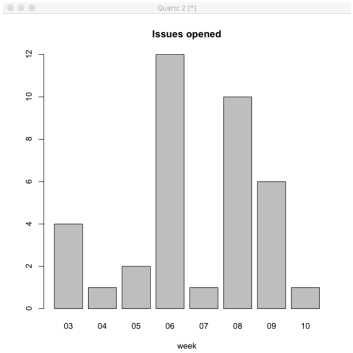

Fig. 26. Group2 Issues opened per week

We see that weeks 4, 7 and 10 fall short of our minimum of 2 issue per week threshold. Thus the stink score is 0.3.

Stink score = 0.3

Issues closed per week

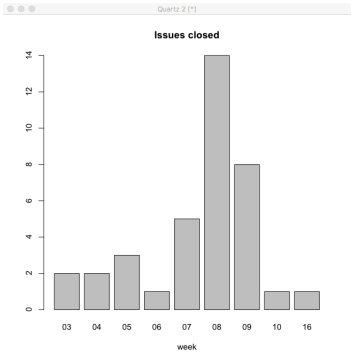

Fig. 27. Group2 Issues closed per week

We see that weeks 6, 10 and 16 fall short of our minimum of 2 issue per week threshold. Thus the stink score is 0.3.

Stink score = 0.3

Total stink score for this heuristic = $0.75 + 0.3 + 0.3 =$ 1.35

All Issues

percent of issues associated with milestones

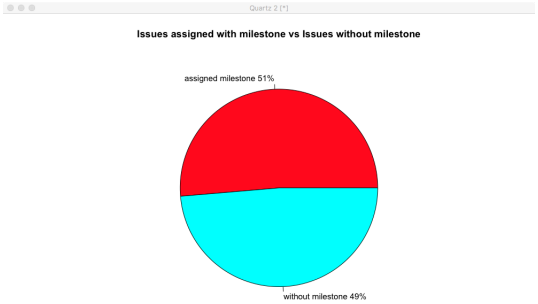

Fig. 28. Group2 Issue Milestone pie chart

Stink score = 0.49

percent of issues labelled as bugs

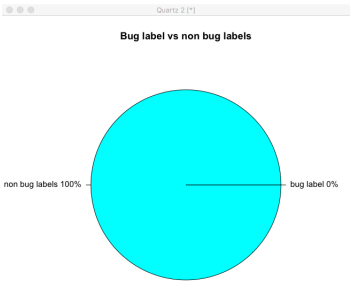

Fig. 29. Group2 Bug Label percentage

We see that no issues were labelled as bugs which seems unlikely in a reasonably sized project and thus rings of bad-smell. Since it falls outside of our error limits, the stink score is equal to 1.

Stink score = 1

number of issues assigned per user

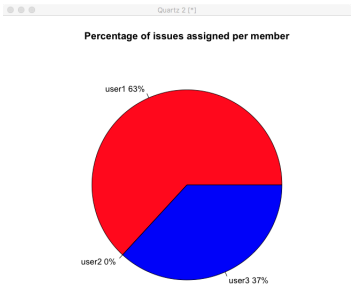

Fig. 30. Group2 Issue assignee percentage

Here user2 has a 0 % contribution which breaks our rule of equal user contribution (a + b ¿ c). Hence the stink score is equal to 1.

Stink score = 1

Total stink score for this heuristic = $0.49 + 1 + 1$ **=** 2.49

2) Group 3: **Commit Activity**

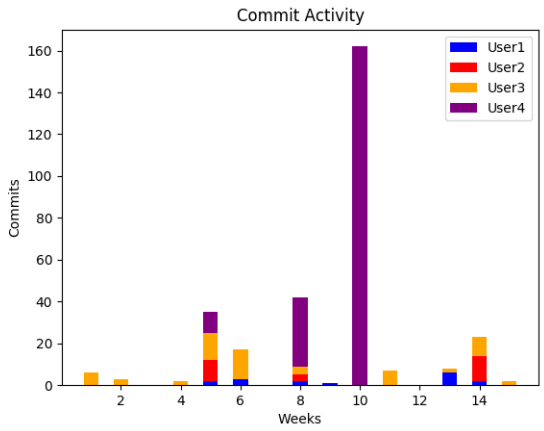

Fig. 31. Group3 Commits per week

The commit graph looks fairly distributed except week 10. Week 10 saw the most commits, an unusual amount from user 4. We have 3 weeks of no activity, and 3 weeks of very low activity.

Stink Score = 6/15 = 0.40

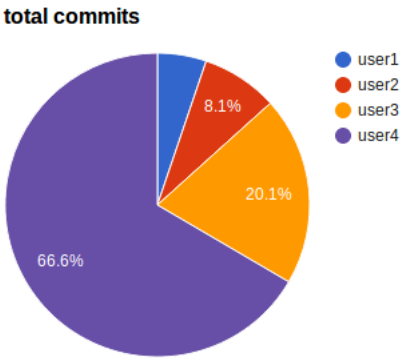

Fig. 32. Group3 Commit Percentages by Users

User 4 dominates the group with more than 60% of the activity. User 3 and User 4 did most of the commits, and therefore we see there are dominating developers violating our member contribution rule. We find 4 stinky pairs who do not contribute more than a dominating developer.

Stink Score = 4/12 = 0.33

Repository Activity in Lines of Code

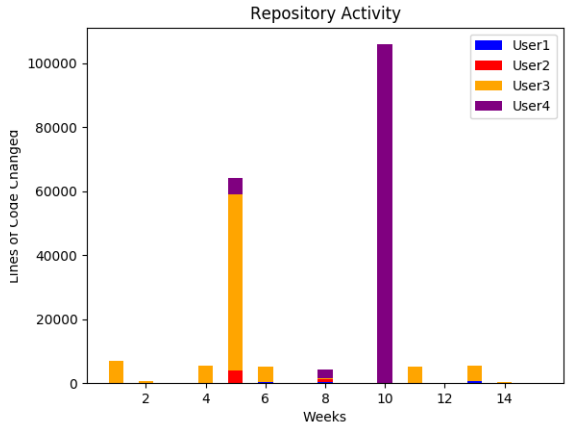

Fig. 33. Group3 Lines Of Code Changed per week

User 4 adds a massive amount of code in week 10, and user 3 in week 5. This skews the graph. We have 2 stinky weeks, where a very large percent (80%)of the whole codebase was committed.

Stink Score = 0.8

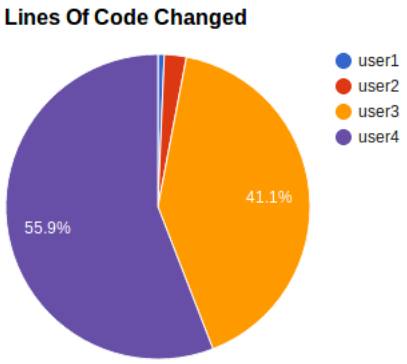

Fig. 34. Group3 LOC Percentages by Users

User 3 and 4 dominate the lines of code changed metric for the repository. Therefore we find stinky pairs forming, and bad smell.

Stink Score = 6/12 = 0.5

Branching

Branches : 5

The total branches are more than the total developers.

Stink Score = 0

Issue Activity

Comments per issue

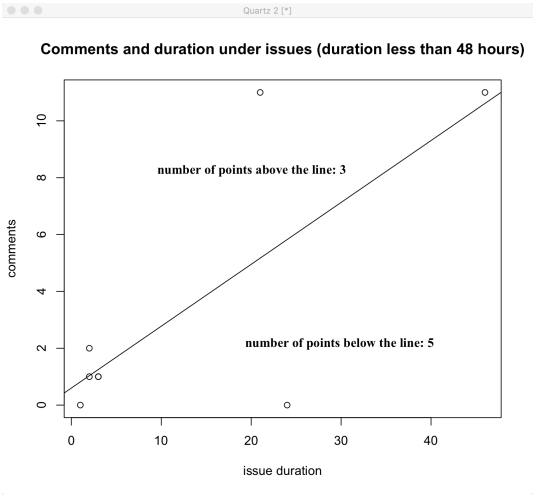

Fig. 35. Group3 Linear Regression on comments on issues lasting less than 48 hours

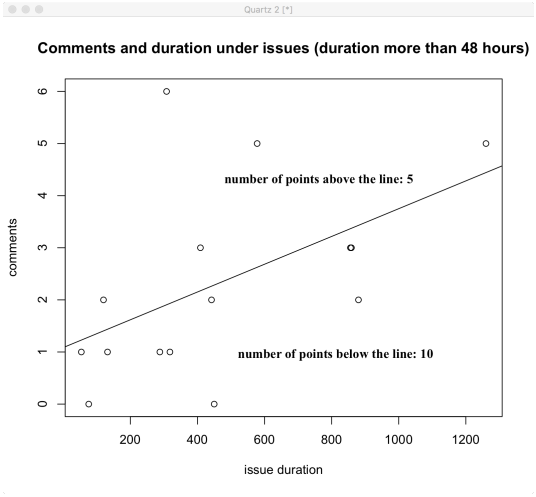

Fig. 36. Group3 Linear Regression on comments on issues lasting more than 48 hours

Stink score = 0.65

Issues opened per week

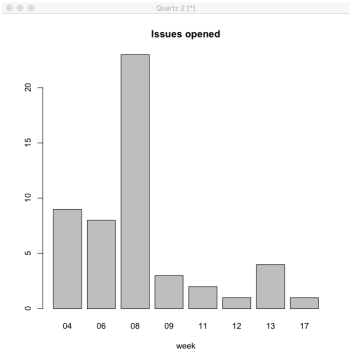

Fig. 37. Group3 Issues opened per week

Weeks 12 and 17 don't meet the thresholds. Hence stink score is 0.2.

Stink score = 0.2

Issues closed per week

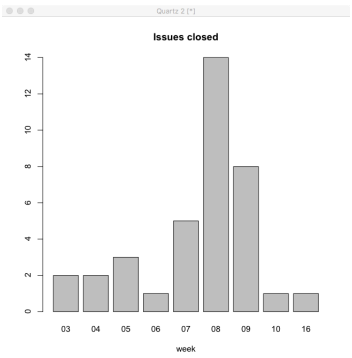

Fig. 38. Group3 Issues closed per week

Weeks 6, 10 and 16 don't meet the thresholds. Hence stink score is 0.3.

Stink score = 0.3

Total stink score for this heuristic = $0.65 + 0.2 + 0.3$ **=** 1.15

All Issues

percent of issues associated with milestones

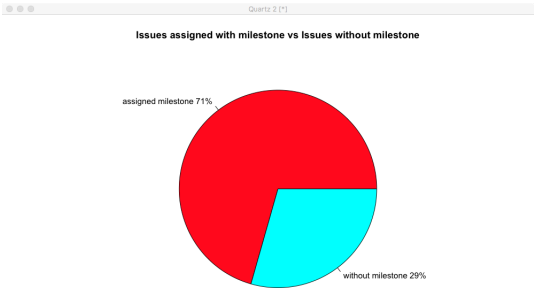

Fig. 39. Group3 Issue Milestone pie chart

Stink score = 0.29

percent of issues labelled as bugs

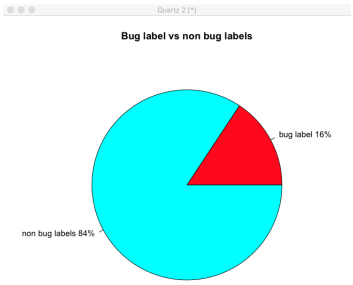

Fig. 40. Group3 Bug Label percentage

This falls within our recommended range of 20% with and error of 10%, thus the stink score is equal to zero.

Stink score = 0

number of issues assigned per user

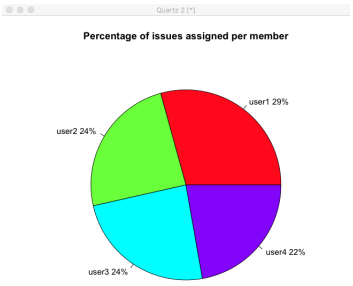

Fig. 41. Group3 Issue assignee percentage

All the users conform to our heuristic of uniform user contribution (a + b + c ¿ d) and thus the stink score is zero.

Stink score = 0

Total stink score for this heuristic = $0.29 + 0 + 0 = 0.29$

GitHub Stink Scores

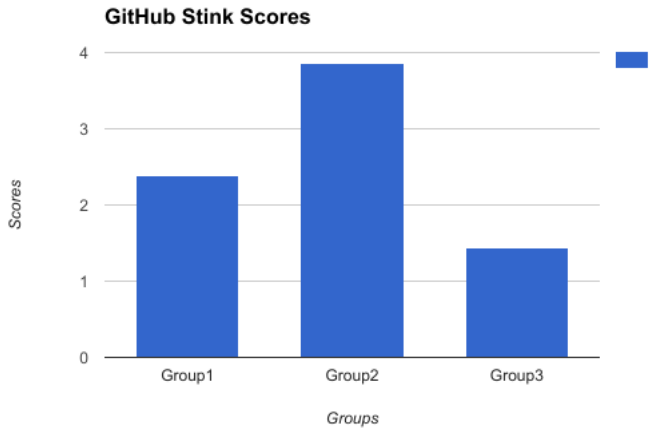

Fig. 42. GitHub stink scores

We see that Group2 has the highest stink score and thus is a candidate for bad-smells under project management.

Git Stink Scores

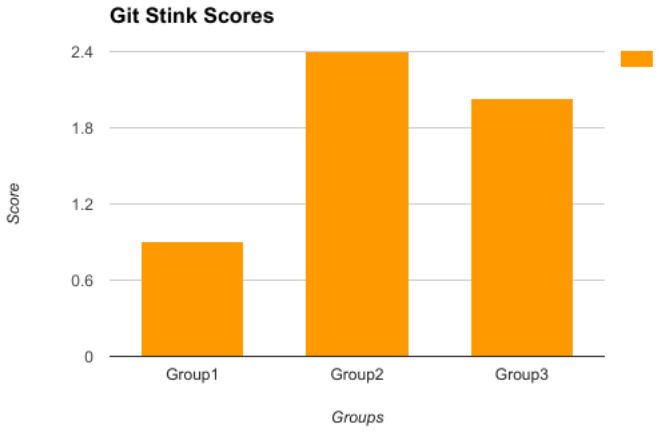

Fig. 43. Git stink scores

Here too Group2's stink scores are higher than the other two, again making it the candidate for bad-smells under source code management.

Total Stink Scores

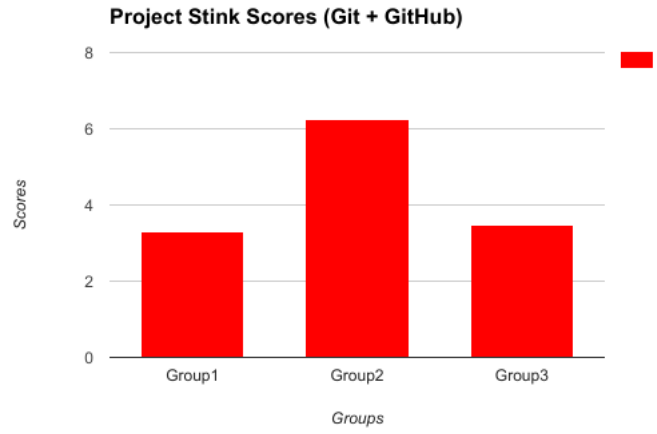

Fig. 44. Cummulative stink scores

We saw that Group2 very likely has bad smells as it had the highest stink scores on both fronts of project management and source code management.

V. Early Detector

Detecting if there are bad smells in the project early in its development cycle is always beneficial. It helps address the pain points and re-aligns the project back to the right track. We looked at some of the heuristics we identified and picked some that could be used as early detectors. The justification of choosing the following heuristics is that they don't depend on the 'total duration' of the project and hence can be good indicators of bad smells at any point during the timeline of the project.

A. User contribution should not skew

It is important for all of the team members to contribute, and for a software project, it's crucial to divide up the workload onto each team member, so that the team members can keep the pace and the software building process can be smooth. We propose this early detector, where during the development process, if one of the team member has more commit than others combined for two weeks in a row, then we raise a flag, showing that there might be a potential bad smell along the way.

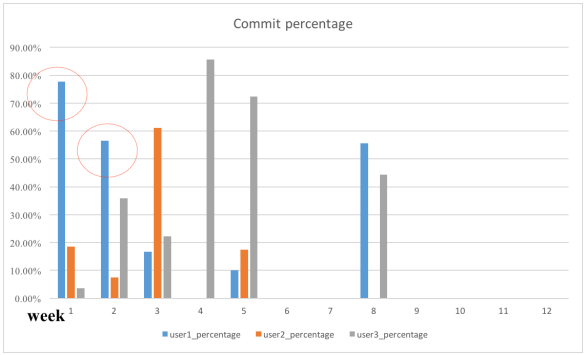

Fig. 45. Group0 Commit Predictor

For example, in group0, we gather the commit made by three members, and calculate the percentage of commits made by them. If we see there's one member that makes more than 50% commits during that week and the week after that, we will call this a potential bad smell, because once this happened, that means the other two members are not keeping up with the development process, and something needs to be done the adjust it.

B. Issue Remain Open Predictor

Issues are a big factor for controlling development process on GitHub. During the development process, if there are more and more issues that are open along the way, something needs to change so that the team can get back on track. For this predictor, we will take the graph of the issues that are remain open during the development period, and see if we can find a continuously grow for 2 weeks in a row. If there is, we will flag it and see it as a potential bad smell.

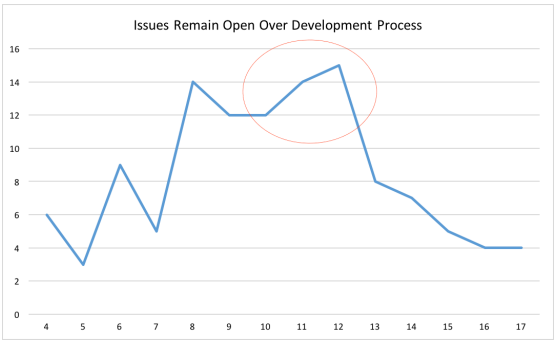

Fig. 46. Group3 Issue Remain Open Predictor

Heuristic	Group1	Group2	Group3
Commit Activity	0.25	0.5	0.4
Commit Contribution	0	0	0.33
Commit Size	0	0.95	0.8
LOC Contribution	0.66	0.95	0.5
Branches	0	0	0
Total(out of 5)	0.91	2.4	2.03

TABLE III. GIT STINK SCORES

Heuristic	Group1	Group2	Group3
Issue Activity	0.67	1.37	1.15
All Issues	1.71	2.49	0.29
Total (out of 6)	2.38	3.86	1.44

TABLE IV. GITHUB STINK SCORES

Fig 46 showed the issues remained open over the development process from week 4 to week 17. As we can see from the plot, during week 10 through 11, the issues that are still open continue to grow, and we deem it might be a bad smell along the way. As we predicted, week 12 has even more issues that remained open, and by the end of the development process, which is week 17, there are still 4 issues that are open.

C. Comments Predictor

Comments under issues are a great way for team members to exchange ideas and contribute thoughts on other team members' tasks, so if there's a lack of commutation been going on for a while, there will be a bad smell on the way.

The predictor we purposed mainly focus on issues that has a duration less than 48 hours in order to predict bad smells as soon as possible. We believe that the longer the issue remains open, the more communication it needs between team members. For this, we generated a linear model to describe the relationship between comments under the issue and the duration on the issue.

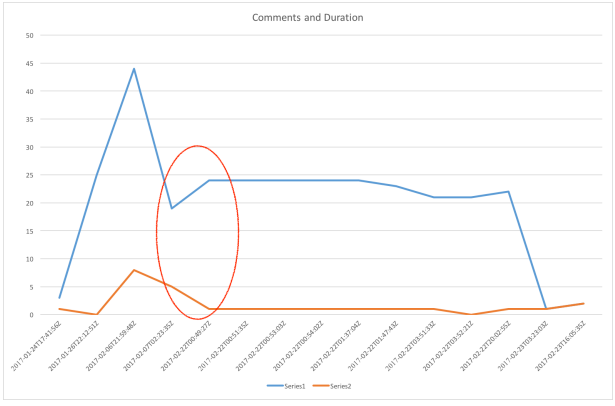

Fig. 47. Group3 Comments Predictor

As shown in Fig 47, during the time around the red circle, the duration of the issue went up but there's no more comments have been committed under that issue, which we think will contain a bad smell along the way. As we already shown in Fig 36, a lot of long lasting issues have little comments under them, by which we deem contains a bad smell.

D. Stink Score Alert

In the previous sections we generate a system that calculates the stink scores based on different aspects of the git and GitHub, such as the number of comments under issues, commit frequency, commit size, etc. This predictor, or what we call Stink Score Alert, is a mechanism which would send an alert if the stink score crosses a threshold value.

Since there are two three parts of the heuristics for the stink score calculator, what we propose is a system which will send out alert once it senses an continuous growth on either of those three heuristics, so the team can have a better idea on what went wrong and make changes to fix the bad smell.

VI. IMPROVEMENT FOR GITHUB

Communication is one of the key factors that affect the development of software. Team members need to exchange ideas, report status and collaborate with each other to get the project going on a good and steady pace. We have analyzed three of the groups' activities on GitHub, creating projects for the 2017 Software Engineering course. What we discovered from the GitHub data is that team members fare poorly on communication.

To explain this result, we hypothesize that they must be using approaches such as slack or having face to face meetings. That's because that is what we also did though we tried to stick to keeping all of our project management activity inside GitHub. A potential reason for this is that GitHub doesn't provide a primary feature for discussions, logging meeting minutes or providing status updates. Sure there are issues, but their main purpose is to raise issues about the code. This is something we believe can be improved on GitHub. Currently, if you want to add a meeting note, you have to create a GitHub issue, and team members input their meeting notes in the comment section, maybe put a label saying "meeting note" for that issue and that's it. But in reality, documenting the preparation is a important part, and "issue" is just to broad for the job. A separate entity for discussion of software one level higher up the abstraction (e.g general direction of the project, meeting minutes, status updates, design discussions etc.) would be great.

VII. CONCLUSION

In this report, we analyze three sets of data from the groups in 2017 Spring Software Engineering course from GitHub.

Git and GitHub provide us with many useful tools which help teams developing software with ease, but there are bad practices using those tools. We talk about how we analysis those data, figuring out what should the ”bad smell” be and determine at which point did those bad smell appear, then we come up with three predictors which can help us foresee the bad smell before it actually happens. There's a lot more that can be done in the future, for example, those predictors would be more accurate if we have sufficient data to generate a predict model using machine learning techniques such as KNN or Neural Network. For this analysis, we realize that all teams are different, they have different approach to develop software and use different tools, which makes it even harder to generalize a unified predictor that works on all projects. a lot more work needs to be done, and we look forward to see other predictors from different teams.

REFERENCES

[1] Repository statistics and report tool. https://jgehring.github.io/pepper/

[2] GitStats is a git repository statistics generator. https://github.com/tomgi/git_stats

[3] The statistical analysis tool for git repositories. https://github.com/ejwa/gitinspector

[4] Pro-Git, Official git book written by Scott Chacon and Ben Straub. https://git-scm.com/book

[5] Git Best Practices by Seth Robertson. https://sethrobertson.github.io/GitBestPractices/

[6] Developers answer what are the Git best practices on Quora. https://www.quora.com/What-are-the-best-practices-of-git

[7] Developers answer what are the Git worst practices on Quora https://www.quora.com/What-are-some-of-the-worst-practices-while-using-Git

[8] Github Flow: branch based workflow. https://guides.github.com/introduction/flow/

[9] Git Branching and Forking workflow. https://www.atlassian.com/git/tutorials/comparing-workflows

[10] Organizing Issues: Stylesheet guide. https://robinpowered.com/blog/best-practice-system-for-organizing-and-tagging-github-issues/