Analysis of Software Projects to Detect Bad Smells

[Project Report]

Alok Kucheria
Department of Computer Science
NC State University
akucher@ncsu.edu

Raman Preet Singh
Department of Computer Science
NC State University
rpsingh2@ncsu.edu

ABSTRACT
This report looks into the intricacies of Software Engineering as a human activity. We analyse GitHub data from three projects and based on certain features, try to predict bad smells that are likely to give us an insight into software engineering practices.

General Terms
Software Engineering, Software Development, GitHub, Software Life Cycle, Code Smells

Keywords
Software Development, Patterns, Feature Extraction, Code Smells

1. INTRODUCTION
Software Engineering is defined as the application of engineering to the design, development, testing, implementation and maintenance of software in a systematic manner.

Any software project is generally a collaboration between engineers, designers, managers, clients and various other entities involved from the beginning to the end. And where there are groups of people working together, there are bound to be problems. Be it in design, development or another phase.

In trying to identify bad smells, we are trying to understand if and when a software project faces challenges, visible or otherwise. The idea is to identify such smells beforehand to learn from them and improve software engineering practices in the future.

To identify bad smells, we need data. Based on the data extracted from various projects(3 in our case), we create certain features. Using those features, we identify what could potentially be bad smells that should be avoided in the future.

1.1 Code Smells
Code smells, by definition means detecting symptoms in the source code that can detect deeper problems. In our case study for this report however, we aren't analyzing source code as much as patterns of activity on GitHub for the three projects. We will not be pointing out particular patterns in the source codes as such. What we are trying to create patterns from to extract relevant information will be GitHub activity for a certain repository including issues, commits, user contribution and milestones. We expand on this in Section 2.

2. DATA COLLECTION
We chose three projects among the ones completed in-class as part of CSC 510 - Software Engineering coursework for Spring '16. We decided on 3 out of 14 based on the amount of data we could gather thus giving us sufficient information to create meaningful patterns.

2.1 Attributes
We used GitHub APIs to extract data from each of the three repositories and store them in 5 tables as follows:

- Issues - Issue ID, issue name, creation time, actions (label changes, milestone changes, assignment activity etc), user responsible for said action
- Comments - User, Issue commented on, time stamp, comment text
- Milestones - Milestone ID, description, time of creation, due and close, user
- Commits - User, time stamp, commit message

2.2 Method
GitHub APIs allow for data to be extracted easily. The API provides most of the required information. For something which is not available directly, we can create it using the information extracted before.

We extracted the data using a modified gitable.py file that is the gitable-sql.py used from the model project for Spring 2015. We made minor modifications to the code to extract the complete data and store it into an SQL database. This would be important for creating relevant queries to create relevant features.

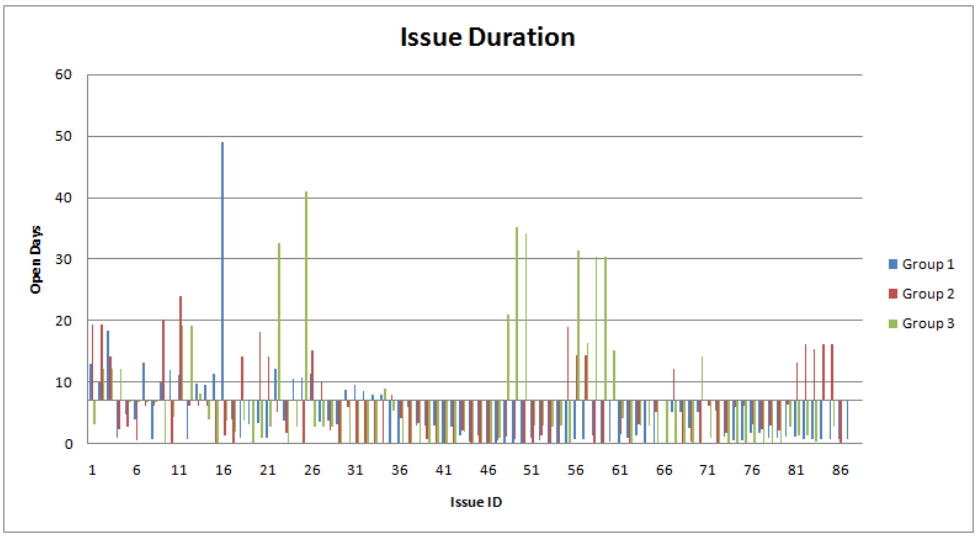

Figure 1: Long Open Issues

2.3 Anonymization

Anonymization of the data was also taken care of in the same script. This was important for two reasons: to protect the privacy of the engineers working on the project and to critically examine the data without any bias.

Both group names and user names were anonymized. Each group was assigned a number based on input parameters. Each user within a group was given a unique id on the first activity detection. Hence, we had data which was readable in a form anon2/user4 instead of project name/user name.

3. FEATURE DETECTION

Gathering all the data is only the beginning. All the data remains just information which needs to be converted to meaningful features to be understood better. The idea of creating features is to start converting all the information at our disposal to knowledge.

We decided to create several features based on categories. The data we have and the way it is visible on GitHub, it is convenient to make such a classification. We created 16 features from which to analyse a project and detect bad smells. Note that this is not based on any classification rules. We can always categorize them as per the delivery requirement.

3.1 Freshness Score

We will assign freshness score to each feature for every group based on our decided threshold. Every feature will have differing thresholds based on their properties. We can have three score levels of 0, 0.5 and 1. The reason we prefer this over a binary classification is that if a project is very close to the threshold, we can give it 0.5 instead of a 0 or 1 based on small deviations in value.

3.2 Issues

The most intuitive way to work through a project and keep the discussion going is through updating issues and using the diverse functionality available at our disposal. From labels to milestones, comments to time stamps, issues can tell a lot about the health of a project.

1. Long Open Issues
   This is a fairly obvious metric to measure the progress of a project. The more long pending issues there are, the more work piles up thereby creating a lot of saturation close to milestones.

   A cut-off of 7 days was set. This was based on the assumption that a major number of issues in a project should be completed within a period of 7 days. There would be instances of issues being long open but they should be few and ideally less than 20% of the total number of issues.

   For Group 1, 20 issues out of their total of 86 issues or 23.25% of issues were open for more than 7 days. Group 2 had 22 out of 87 or 25.82% issues long open and Group 3, with 18 out of their 85 issues had 21.18% of long open issues.

   Since Group 2 had a feature value more than the cut-off value but within the deviation limit, it gets a Freshness Value of 0.5. The other 2 groups were over and beyond the cut-off value for the feature and therefore get a Freshness Value of 0.

2. Orphan Issues
   An issue should be as well classified as possible. Here, we try to understand if an issue has been set a milestone, whether it has been assigned a label. This helps prioritize work as the project grows in size.

   Orphan issues simply should not exist in a system - issues which are neither labelled nor they are part of any milestone. But there are chances of special cases and after considering them but still tagging them as a bad software engineering practice, we decided that not more than 5% of issues should be orphan.

   Group 1 had 21 issues which were orphan. This is an alarming value given that they had 86 issues in total

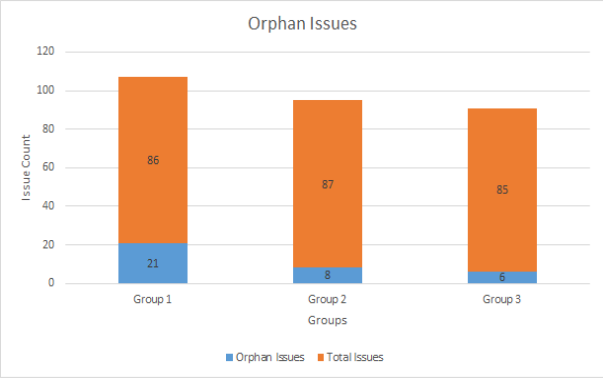

Figure 2: Orphan Issues

and with almost a quarter of issues as orphan, Group 1 gets a 0 for the Freshness Value. Group 2 and Group 3 had 10% and 7% issues as orphan respectively with 8 issues out of 87 and 6 out of 85 total issues. So therefore all 3 groups get a 0 as Freshness Value.

3. Empty Issues
   An issue is not a stand-alone code pointer. It is an issue because it needs to be resolved after a certain discussion, effort, time consideration etc. We look for issues which have no comments. This is an indicator of communication gap.

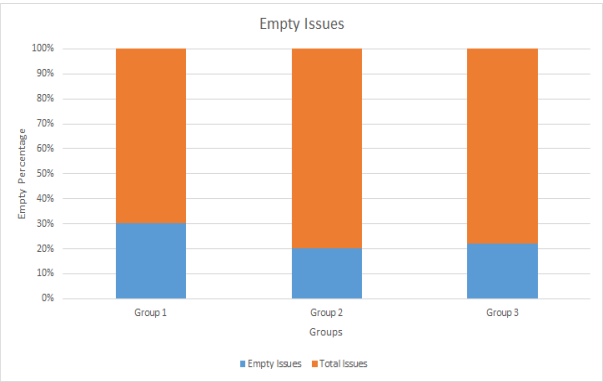

Figure 3: Empty Issues

Many groups had created issues just to increase the issue count and left them with just one or in many cases no comments. Group 1 for instance had 37 issues out of their 86 which is a staggering 43% issues with one or no comment. Group 2 and Group3 were better in comparison to Group 1 with 26% and 29% respectively of empty issues. Issues serve purpose of helping groups coordinate and while they can act a reminders of an activity to be taken care of, they cannot be more than 10% of your total issues. All groups exceed the cut-off by a huge difference and thus get a Freshness Value of 0.

4. Comments per Issue
   As seen above, we need to communicate over why something is an issue and how it should be resolved. This parameter shows us how actively an issue is being discussed within the team.

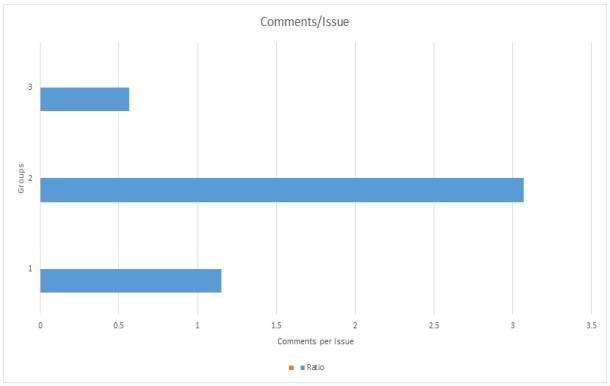

Figure 4: Empty Issues

Every issue should be well discussed and given the size of the class project, issues on an average with 2.5 comments indicate a good practice.
For the groups under consideration only Group 2, with an average of 3 comments per issue, displayed a good software engineering practice of interaction amongst group members. Group 1 just had about 1.1 comment per issue. Group 3 on the other hand just had this ratio as 0.48. This shows minimal group interaction along with creation of just placeholder issues instead of actual issues.
Therefore Group 2 get a Freshness Value of 1 whereas Group 1 and Group 3 get a Value of 0.

5. Pending Issues
   Here, we look at issues which may or may not be long pending, but have missed the milestone deadline. This is a certain red-flag specially in case of project with frequent deliverables.

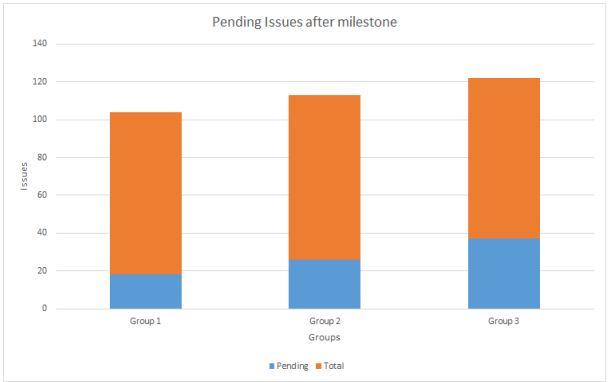

Figure 5: Pending Issues

We see from the graph that **Group 1** has only 20% pending issues whereas both **Group 2 and 3** have over 25% pending issues at 30% and 45% respectively. At a threshold of 25%, we get scores of **1, 0 and 0** respectively.

6. Unassigned Issues This feature is to indicate 2 different sets of trends viz. 1) Issues not being assigned to

anyone 2) Changing the assignees for an issue. While on one hand, not assigning issues to anyone completely defeats the whole purpose of having the issue feature, frequent change of assignees indicates bad management and planning skills.

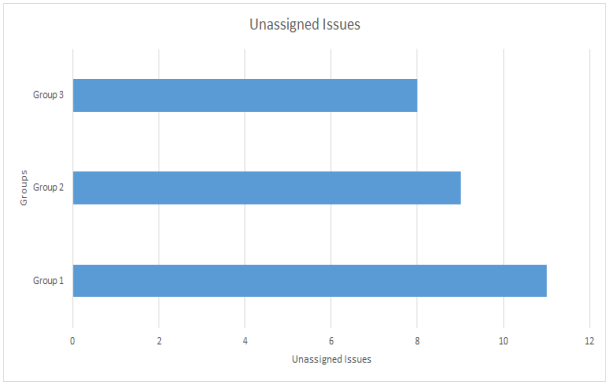

Figure 6: Unassigned Issues

In Group 1 for instance, there were a total of 11 unassigned issues, out of which 5 were for user 1 and 4 for user 2. A quick look at the Github repository for the group indicated that whenever the particular user was assigned an issue, it used to stay with that user for sometime without any activity and then assigned to some other user.
This practice thus indicates one or all of the following things :

(a) Incompetence of the said user.
(b) Over-dependence on a single user.
(c) Dictator and Absent user situation.

But having said that, at times this can also indicate when an issue has been worked upon and assigned to some other user for assistance. The cut-off has been set to 10% for the complete project and individual user as well.
On group level Group 2 and Group 3 have close to 10% of unassigned issues whereas Group 1 had 13On user level also the users of Group 1 perform the worst with 5 and 4 issues out of 11 being unassigned to user 1 and user 4 respectively. Barring user 1 of Group 2 who had 5 issues unassigned out of 9 for the group, the overall group performed well and within the set limit. For Group 3, all the users were well below the threshold value.
Therefore Group 1 gets a Freshness Value of 0, Group 2 gets 0.5 and Group 3 gets 1.

7. Label Usage
Going over label usage, we can understand whether sufficient time is being devoted to every aspect of a project be it bugs, enhancements or requirements. While this is not necessarily a smell detector, given that users can use custom labels to suit project needs, it can still act as a window into how a project doing on all phases in a general sense.

As can be seen from the Fig. 7 above, Group 1 and Group 3 primarily had issues related to enhancements. This is a bad software engineering practice as it is important to report bugs, ask for help, add features etc. Basically for a group project it was clearly lacking group interaction and ability to identify bugs and duplicates. For a software engineering project to not have bugs is simply impossible. This shows incompetency of the group members. Group 2 on the other hand focused on a wider and well distributed sets of labels.
Group 2 therefore gets 1 as the Freshness Value and Group 1 and Group 3 get a 0.

3.3 Users

The users/engineers/developers are the most critical component of this entire study. Software Engineering,as we all know, is first and foremost, a human activity. Trying to find patterns in human activity can give us some of the most insightful indicators into what makes a good project or a bad smell. These parameters can shed light into how every member works in a group.

1. Commits per user
Quite simply, commits per user over total project commits. This shows a users' contribution to the project.

   We will only look at one group for now. This is one of the features we use to build on the user contribution. Commits being the most vital contribution, we give it a **weight of 0.4** in our user contribution feature.

2. Issues closed per user
This shows us how active a user is in tackling issues being faced in the projects. Once again, it shows us their overall contribution.

   Again, we look at a sample from Group 1. We assign a **wieght of 0.3** to issues. This shows user contribution as an approximation.

3. Comments per user
Committing code is important. Just as important is being actively involved in the discussion about the issues being faced in the group. Here, we can judge how interactive the group is and whether there is any communication gap. This is another bright red-flag for identifying a bad smell.

   Finally, we look at the sample from Group 1. We assign a **wieght of 0.3** to comments too as we believe that discussion about the problem is just important as raising a problem in itself.. This shows user contribution as an approximation.

4. User contribution to Group
This isn't a feature in itself but once created using the above three. Using the above three metrics, we can create an overall picture of a users' contribution to the project as a team member rather than as an individual.

   As explained above, we calculate the total contribution to a project based on a simple formula:

$$contribution = 0.4*\#commits+0.3*\#issues+0.3*\#comments \quad (1)$$

   In plain text, 40% weightage is given to commits, 30% to issues and 30% to comments. Based on these parameters, we get an approximation of work done by

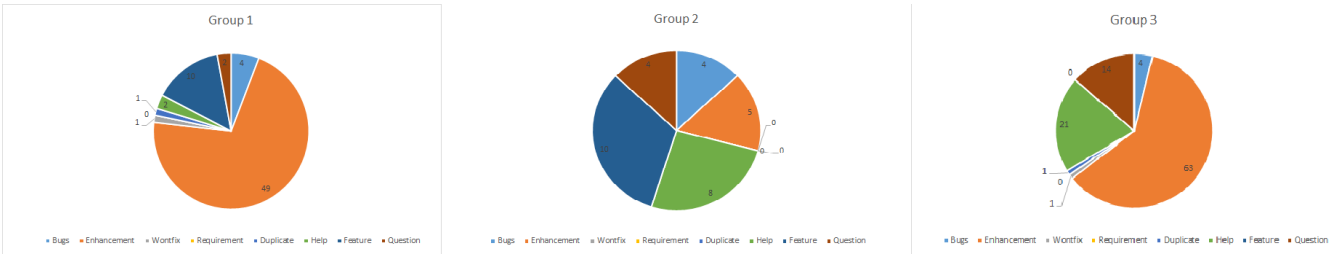

Figure 7: Group wise Label Usage

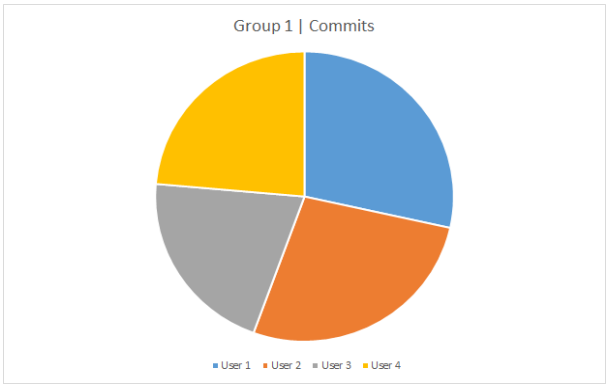

Figure 8: Commits by User

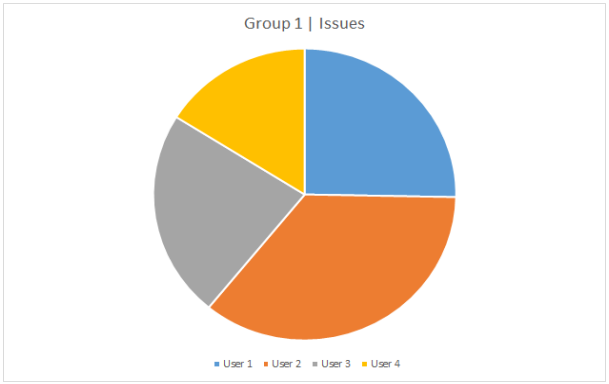

Figure 9: Issues by User

every member. Figure 11 shows this as a simple pie-chart.

Looking at the figure, it is obvious that **Group 1** worked systematically dividing work properly whereas **Group 3** work was mostly done by users 1 and 2. We give them freshness scores of **1 and 0 respectively**. **Group 2** isn't obvious on the first look. On deeper inspection, we see that user 4 hasn't contributed much thereby making user 1 and 2 work a lot more. User 3 too, did less work than expected. Had user 3 helped cover for 4, we could have considered this a case of one bad apple. But given that half the team worked and half did not, we go with a freshness score of **0**.

3.4 Commits

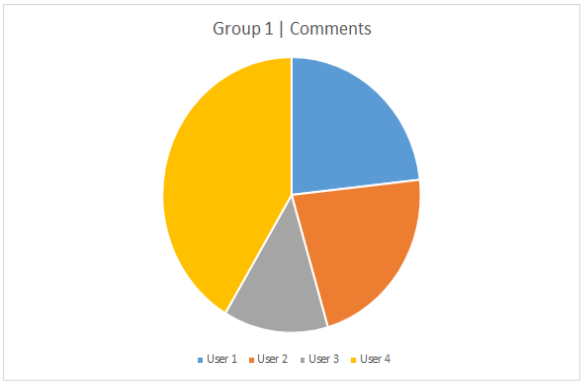

Figure 10: Comments by Users

A project moves forward with every commit. Or so it the ideal scenario. Hence, the quality of every commit matters. These can be measured both subjectively, in terms of quality of code, and objectively using performance metrics. We are primarily focused on the objective aspects as stated in Section 1.1.

1. Commit Regularity
   This metric shows us how regularly the work is being done. This can shed some light on whether agile methodologies are being followed or not. A sudden surge in the graph followed by a lull period shows irregularity.

   We took this data directly from GitHub Graphs to avoid rework. We made sure that anonymity was maintained by redacting user names and repository details.

   At first glance, all three groups have fairly regular commits. **Group 2 and 3** have continuous graphs have we will give them a freshness score of **1**. **Group 1** on the other hand have patchy work, i.e. although they have done a lot of work close to milestones, other times the work hasn't progressed. Hence we give them a rating of **0.5**.

2. Commit Lines
   This one is a difficult parameter to use as a metric towards effort estimation. Some times, committing code along with libraries can add thousands of lines which do not justify the effort. Nevertheless, one measure we can derive is by looking at the additions and deletions. This shows us whether new code is being maintained or not.

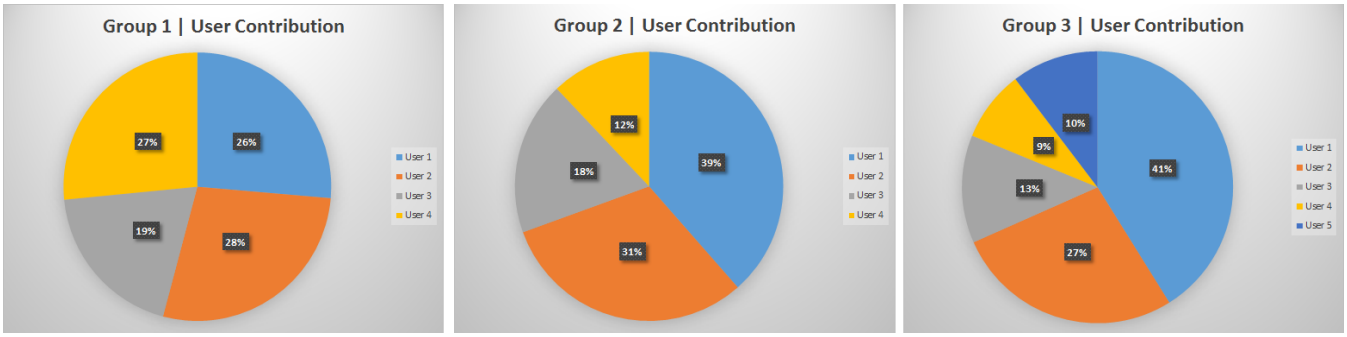

Figure 11: User Contribution

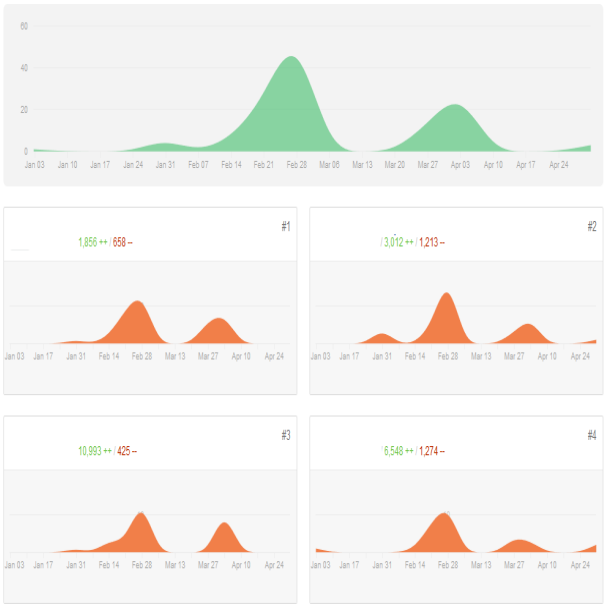

Figure 12: Commit Data Group 1

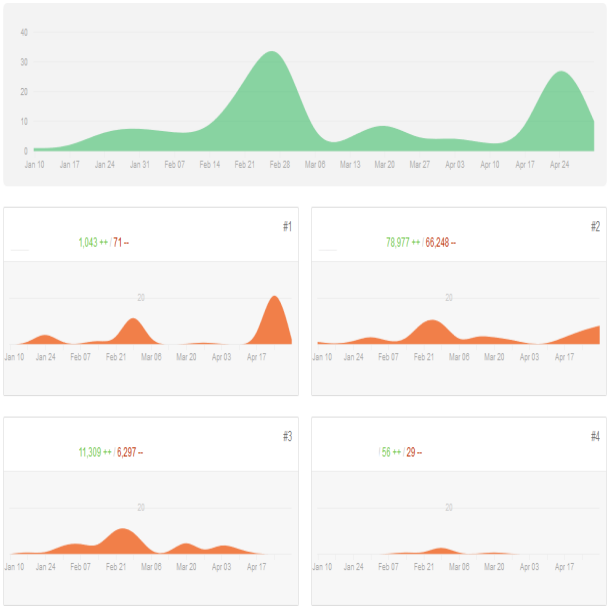

Figure 13: Commit Data Group 2

We took this data directly from GitHub Graphs to avoid rework. We made sure that anonimity was maintained by redacting user names and repository details.

Group 1 not only have decent lines of code but sufficient deletions with additions showing that they constantly tried to improve on the source code. We rate them **1**. **Group 2** also seem to have a similar pattern albeit with probable library commits since 80,000 lines of code in such a short period seems unlikely. We give them a rating of **1. Group 3** is an anomaly in itself. We can be certain that over a million lines of code was not written. Hence, it is a library addition and deletion. For their modifications, actual code modifications seem very low and we give them a **0**.

3. Commit Message
   Commit messages, like label usage, is another subjective parameter. Commits like updating README files can skew the commit values. Secondly, insufficient commit messages can indicate how much of a black box the project is to someone trying to understand the work.

We filtered two specific commit messages as irrelevant:

- Added files via upload
- Updated README.md

Comments like these are possible once in a while. But anything more than 5 and we possibly looking at improper commits like direct uploads without messages or fake commits just to bump up commit numbers by updating the README multiple times.

Based on our data from Figure 15, only **Group 3** manages a score of **1**. Both **Group 1 and 2** get a **0** on this feature.

3.5 Milestones

Milestones, as the word implies, signifies an important event. In case of project management, we use milestones to gauge the progress of a project in quantitative terms. They keep the project in constant motion and keep the engineers motivated, specially for long-term projects, where tangible results take time to deliver.

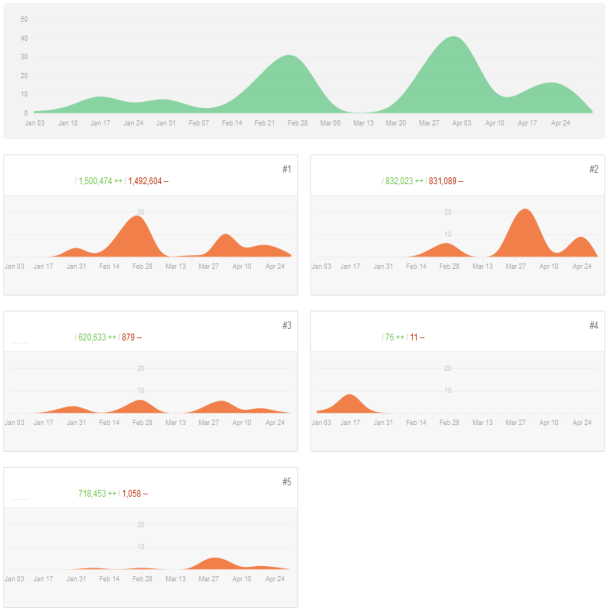

Figure 14: Commit Data Group 3

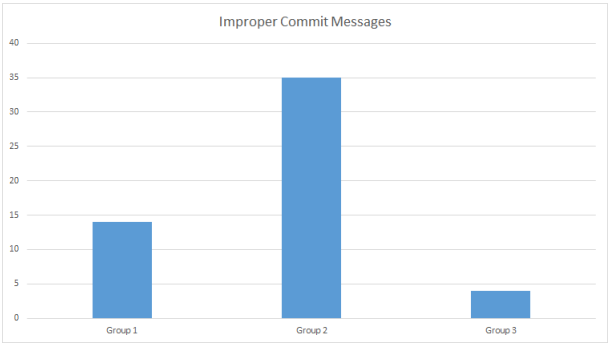

Figure 15: Irrelevant Commit Messages

1. Milestone Closure
   The difference between when a milestone is due and when it is completed is a very strong measure of the progress and delays in a project.

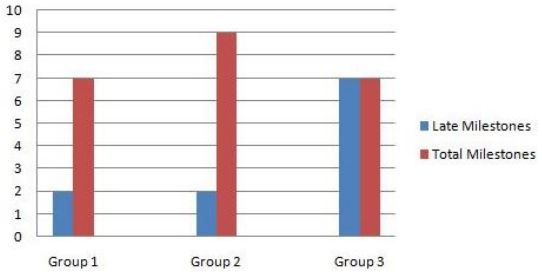

Figure 16: Milestones closed 48 hours after due date.

Fig. 16 will give a clearer picture of this scenario. Looking at the figure it can be seen that Group 3 had setup 7 milestones all of which they closed well after the due date. But at the same time Group 1 and Group 2 were well aware of their milestones and barring 2 for each group, they closed their 7 and 9 milestones on time.
Cut-off for milestone this feature was set at 25% and therefore Group 1 gets 0.5 on the Freshness Value whereas Group 2 and Group 3 get 1 and 0 respectively.

2. Milestone Time Frame
   This is the difference between when a milestone was created and when it is due. It is to make sure that every milestone has a sufficient time frame in which its goals are to be achieved.

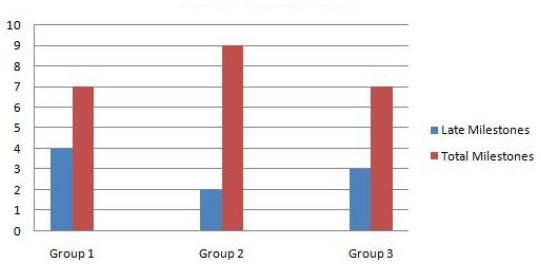

Figure 17: Milestones created atleast a week before the due date.

Fig. 17 indicates how pro-active the users were and not starting off with the milestone a day before the project. Thsi aprameter ensures qualty as the milestones done in a rush will be an indication of a non-serious attitude and thus a compromised project. This is one of the crucial features and barring Group 2 , who created just 2 of their 9 milestones within a week of milestone due date, the other two groups have created half of their milestones a week before the due date.
Cut-off for milestone this feature was set at 25% and therefore Group 2 gets 1 on the Freshness Value whereas Group 1 and Group 3 both get 0.

3. Events per milestone
   We measure the activity for each milestone here. Multiple teams have multiple milestones hence the graph can look skewed. Nevertheless, we get a rough idea of the milestone progress using this graph.

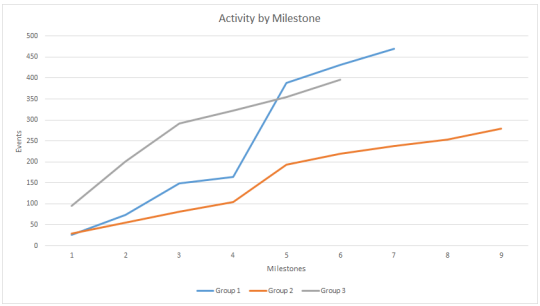

Figure 18: Events per Milestone

Looking at Figure 18, we see that all the groups have regular activity. Besides a slight dip for Group 2, all

graphs are increasing properly. Hence we assign the freshness score of **1 to all groups**.

4. BAD SMELLS

Bad smells, according to Martin Fowler, are a surface indication that usually corresponds to a deeper problem in the system. A 2015 study of commits to 200 open source projects found that most bad smells are already present since creation. This in our features could mean missing labels, missing milestones etc.

We can combine a variety of the features extracted above to create ways to detect bad smells. **The lower the score, the worse the smell.**

4.1 Poor Communication

Communication is of utmost importance in a team project. Using empty issues, comments per issue, comments per user and commit milestones, we can understand how good or poor communication was.

Feature	Group 1	Group 2	Group 3
Empty	0	0	0
com/issue	0	1	0
com/user	1	0	0
Unassigned	0	0.5	1
Overall	1	1.5	1

As we can see from the table above, only Group 2 has a score above 1 out of a possible score of 4. Even at 1.5, it is not flattering meaning **all groups needed to have better communication.**

4.2 Poor Milestone Usage

Usage of milestones is necessary in any project. Looking at milestone features like issues per milestone, commits per milestone, pending issues, milestone time frame and closure can help us detect bad smells in the project.

Feature	Group 1	Group 2	Group 3
Closure	0.5	1	0
Timeframe	0	1	0
Activity	1	1	1
Overall	1.5	3	1

Group 2 does a good job at working with milestones. Group 3, however has lagged behind Group 1. Out of a possible score of 3, only Group 2 performs well. **Both Group 1 and 3** need to detect and change strategy here.

4.3 Absent Member

A project flourishes when all members of a team work together towards a common goal. We look at individual member contributions using commits per user, issues per user, comments per user and their overall contribution metric. This can help detect members who are not contributing to the project. At the same time, it can also show us if only a few member are dictating terms in a metric sense.

We will take the freshness scores assigned individually here. The overall graph we saw above was only to get a clearer visual idea of the contributions.

Feature	Group 1	Group 2	Group 3
Comments	1	0	0
Issues	1	0	0
Commits	1	0.5	0
Overall	3	0.5	0

On a possible maximum of 3, Group 1 performed very well on this criteria. The same cannot be said of the other groups. Group 3 got a 0 score which is a bad indicator of the team ethic. **Both Group 2 and 3 fail on this criteria.**

4.4 Poor Planning

Planning of milestones, including opening, due and closing dates, long open, orphan, pending issues, commit regularity are some of the features which will detect the planning strength or weakness of a software project.

Feature	Group 1	Group 2	Group 3
CommitGraph	0.5	1	1
Pending	1	0	0
Orphan	0	0	1
Long Open	0	0.5	0
Overall	1.5	1.5	2

Almost all three groups performed about average on this category. Out of a possible score of 4, two teams managed 1.5 whereas Group 3 managed a 2. We can say that **all groups were average planners**.

4.5 Poor Standards

Usage of labels, commit lines are some of the subjective parameters which tell us whether a project is of a certain standard or not. Although not quantitative, even qualitative properties like these matter in the long run for any project and a good engineer must always strive towards them.

Feature	Group 1	Group 2	Group 3
Message	0	0	1
Lines	1	1	0
Labels	0	1	0
Overall	1	2	1

On a possible score of 3, Group 2 excelled with the only problem being commit messages. The other two groups, **1 and 3 need to catch up on this**.

5. RESULTS

We summarize the results in a simple graph where all freshness scores are summed up. **Note here that we are using freshness scores, hence, higher the values, better the result. The smell names are kept on graph instead of their nature, good or bad.**

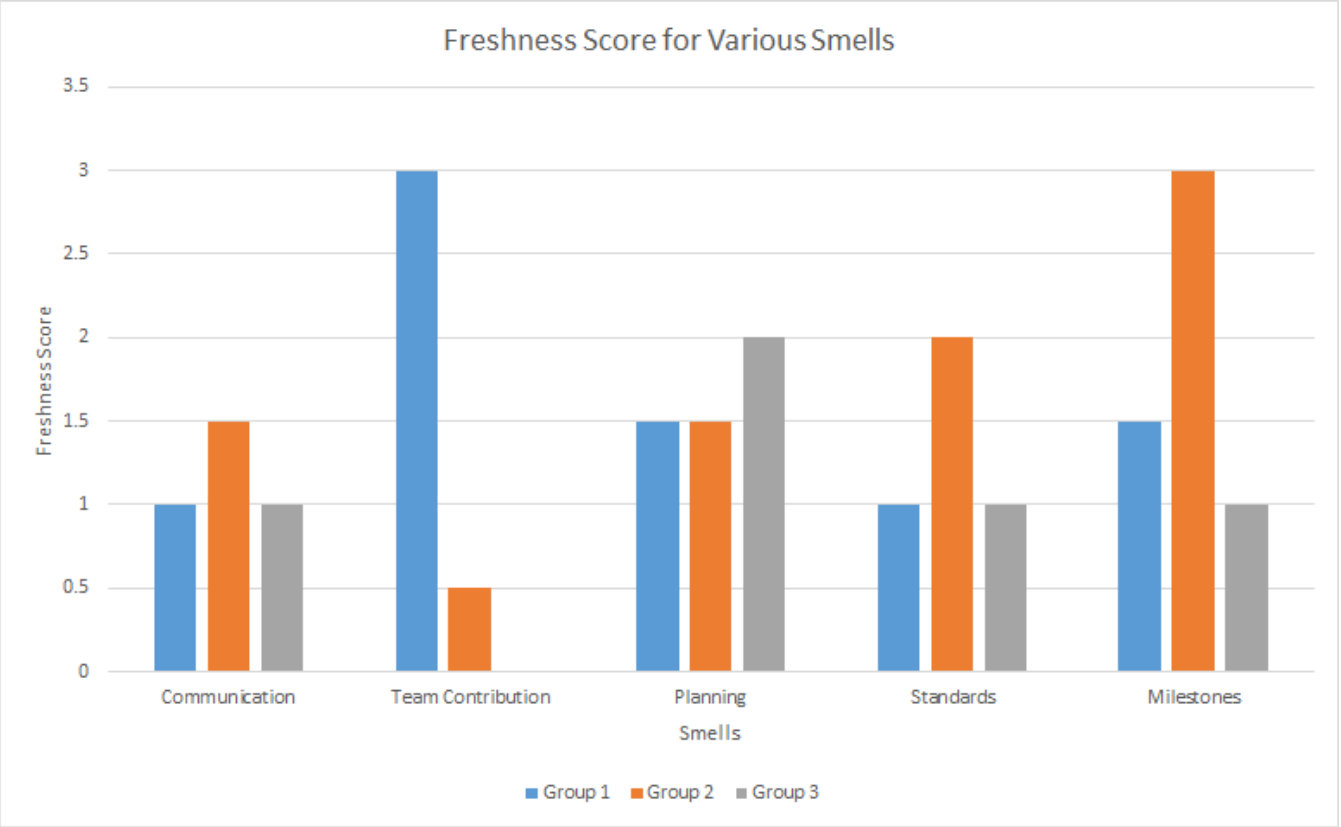

Figure 19: Smells Vs Freshness Scores

At a total of 8.5, Group 2 has the maximum freshness. At a score of 5, Group 3 had the most bad smells. These results are based solely on our features. One stark observation, as we computed above is the 0 score for Group 3 on Team Contribution due to absent members.

These are only some of the smells from our own features. One can create more features and find more bad smells using data that is publicly available on GitHub for each repository.

6. EARLY WARNING

An early warning sign can be gained from the effort estimation of the group. This can be gauged using the milestones data available at our disposal.

The **differences between creation, due and completion date of a milestone** can tell a lot about how accurate a team is with its effort estimation and how much it may need to scale up its efforts.

We can clearly see that all groups had planned their milestones well in advance, whether or not, they were followed. This is a good sign and one that should raise an alarm if missing.

Another important indicator is the **commit graph** we have. Regularity in work in directly visible using the graph and any abnormalities in it should be cause for concern.

Pending issues are also a sign of work piling up. Only one group could hold up to it and it shows in the results as well.

We try to judge a project based on these early signs and assign them scores as we have done above. The early detection part was taken care of in the smell detection itself when we gave each feature a score to compute the totals.

7. REFERENCES

1. CSC-510-2015-Axtitron. Project 2 ”https://github.com/CSC510-2015-Axitron/project2”
2. Martin Fowler on Code Smells http://martinfowler.com/bliki/CodeSmell.html
3. GitHub API - https://developer.github.com/v3/