May1 Report: Bad Smell Detection and Analysis

Moharnab Saikia
North Carolina State University
Raleigh,NC
msaikia@ncsu.edu

Pranjal Deka
North Carolina State University
Raleigh,NC
pdeka@ncsu.edu

Rashmi Sandilya
North Carolina State University
Raleigh,NC
rsandil@ncsu.edu

Sindhu Balakrishnan
North Carolina State University
Raleigh,NC
sbalakr2@ncsu.edu

ABSTRACT
The paper focusses on detecting problems and issues in the Github repositories of the teams based on their activities in the first project. The data collected from the repositories are queried to extract different features and the observed results are presented in this report through various charts and graphs. The data collected, feature detectors, the way the features are extracted and the results and analysis are discussed in various sections of the paper.

Keywords
Issues, milestones, commits, comments, open, bugs, enhancements, closed

1. INTRODUCTION
The paper reflects data collected from Github and throws light on what activities other teams performed and how they performed. For data collection, we ran a python script called gitable.py on 6 github repositories including ours. We have made sure that our data set protects the users' and the groups' identity. To achieve data anonymity, we followed a set of procedures which are described in detail later in the paper. We filtered the anonymized data and put into various database tables. We also ran some other scripts written by our team to find out bad smells from the collected data set.

For identifying the bad smell, we have taken certain set of features into consideration. These features includes Too few/too many issues per milestone, Commit with very short message, Lesser number of bugs etc. Each of these features have been explained briefly in the paper and the supporting data for each of these features have been provided in visual data format like table, bar graphs, images etc. The most important part is the analysis of occurrence of particular behavior(good/bad) for each of the identified feature. Analysis is followed by overall conclusion and early warning.

2. DATA COLLECTION
Data collection plays a key role in any kind of analysis. We have run some scripts to collect the data from the repositories and store the data in the database which will be easier for us to run queries on the database. The data collection method employed and the types of data collected are discussed in the sections below.

2.1 Data collection method
In order to collect the initial set of data from the Github repositories we ran a script gitable.py. We made some modifications to the intial script provided to us and collected the data given in section 2.2. We input the target repository and the group name (anonymized later) and run the script that collects the data and dumps it into various tables in the SQL database. Using SQL database will help us to run queries in the future to extract the features.

2.2 Types of data collected
The various types of data collected from the repositories are given below:

1. Milestones
   (a) Milestone ID
   (b) Description
   (c) Created Time
   (d) Due Time
   (e) Created by
   (f) Closed Time
2. Issues
   (a) Issue ID
   (b) Issue Name
   (c) Created Time
   (d) Closed Time
   (e) Created by
   (f) Closed by
3. Commits
   (a) User
   (b) Commit Time
   (c) Commit message
4. Comments

(a) User

(b) Comment text

(c) Related Issue

(d) Timestamp

3. ENTITY ANONYMIZATION

To protect the user/group identity, we provided entity anony -mization. Each group was assigned a group number which ranged between 1 and total number of groups. Similar thing was done for the user. This randomization was achieved using a random function generator rand(), the output of which was divided by the total number of groups or users to get the random value in the range. The code snippet for anonymization process is as follows:

```
def anonymize(user):
  if anonymizer:
    if user == None : return ''
    idx = user_list.index(user) if user in user_list
         else -1
    if idx == -1:
      user_list.append(user)
      return "user"+str(len(user_list) - 1)
    else:
      return "user"+str(idx)
  else:
    return user
```

4. DATA SAMPLES

We ran the gitable.py script and have created a database consisting of various tables with the fetched data. The various tables, their schema and the table format are shown in the section below.

4.1 Tables

The various kinds of tables created for the purpose of storing the data and extracting information for feature analysis are discussed in Figure 1

5. FEATURES EXTRACTORS

The feature extractors are the indicators of the effective usage of the Github by any team. We chose the following feature extractors which we thought will be useful for analysis of the bad smells and also give early indicators of the problems teams faced during the development.

5.1 Too few/too many issues per milestone

The number of issues per milestone is an indicator of how wisely the team has planned their work. Milestones with either too few or too large number of issues indicates an improper planning of the project by the team. As milestones are placed at constant intervals of due date, the number of issues in a milestone should be more or less the same and sufficient to accomplish within the given timeframe.

```
sqlite3 -csv gitExtract.db "select count(id) as
    Num_Issues, milestone_identifier, group_id from
    issue group by milestone_identifier order by
    group_id" > num_milestone.csv
```

5.2 Commit frequency

The commit frequency is a measure of the commits made by the team over an extended period of time. Sometimes there will be no commits for extended periods of time while all the code gets committed in a very short interval. This is an indicator of improper planning of the project by the team and has been computed using the data from the sql database.

```
sqlite3 -csv gitExtract.db "select G.user, G.group_id,
    C.sha, C.time from commits C, user_group G where
    C.user = G.user order by G.group_id" >
    user_commits.csv
```

5.3 Commit with very short message

It is very important that each commit should carry the proper message so that other users of the repository can understand the functionality of the code that has been checked in and rework can be saved.

```
sqlite3 -csv gitExtract.db "select A.group_id,B.user,
    B.sha, B.time, B.message from user_group A,
    commits B where A.user = B.user order by
    A.group_id " > small_commit.csv
```

5.4 Issues open for very short time

Teams should plan issues that consume considerable amount of time. Some teams may create many issues even for small tasks that take few minutes in order to increase the Github activity. Hence tracking the issues that have very less open time will indicate if the team has properly created the tasks.

```
sqlite3 -csv gitExtract.db "select A.group_id, A.id,
    (cl.time - op.time) as openTime from issue A,
    event cl, (select id_grp, min(time) as time from
    event group by id_grp) op where cl.action ==
    'closed' AND cl.id_grp == op.id_grp and A.id_grp =
    cl.id_grp order by A.group_id" >
    issue_close_time_spent.csv
```

5.5 Issues with no comments

Github provides users with the feature to comment on each issue which provides a means of communication for the team about a particular issue. Tracking the number of comments per issue and detecting the number of issues that has no comments will indicate us how effectively the team has communicated during the development process. The sql query is as follows:

```
sqlite3 -csv gitExtract.db "select A.id, A.group_id,
    B.count from issue A left outer join (select
    id_grp, count(*) as count from comment group by
    id_grp) B on A.id_grp = B.id_grp order by
    A.group_id" > comments_per_issue.csv
```

5.6 Uniformity of user commits

Code commits in the Github should not be always from a particular contributor or a set of contributors, rather it should be more or less uniformly distributed. Non uniformity of commit presents bad planning, less team effort and poor cooperation.

User_group:

user	group_id
user_14	group_1

Issue:

id_grp	id	name	milestone_identifier	group_id
4_group_1	1	Seek approval for the three proposed solutions	1568411	group_1

Milestone:

id	name	description	created_at	due_at	closed_at	identifier	group_id
4	Exploring how to upload images to amazon cloud	Exploring APIS to upload images	1455575325	1455771600	1456004895	1584399	group_1

Event:

id_grp	time	action	label	user	identifier
3_group_2	1452808201	milestoned	Feb 1	user_41	515531065

Comment:

id_grp	user	created_at	updated_at	text	identifier
2_group_5	user_29	1453158073	1453605857	Suggestions could be like , facebook events, office outlook appointments, personal events, etc.	172676377

Commits:

id	time	sha	user	message
4	1459898113	86ddfe3dd480eab5dc0e441cd6b27bec17392d07	user_34	minor chnages in formatting

Figure 1: Schema of different tables in the database

```
sqlite3 -csv gitExtract.db "select a.group_id,
    b.commit_count, b.user from user_group a, (select
    user, count(id) as commit_count from commits group
    by user) b where a.user = b.user" >
    commit_per_user.csv
```

5.7 Addition vs Deletion Ratio of Code

As teams get into the agile methodology during the March phase there should be fewer code additions and more deletions. Adding more code means that the product is not optimized well and adding features does not necessarily mean that the product is improved. The data for this feature has been computed mathematically from Github data.

5.8 Issues assigned per user

Ideally each user in a group should able to identify the issues and contribute to the list of issues. The distribution of issues per user should be more or less uniform. Absence of uniformity of issues per user reflects that issues were not discussed properly and individual contribution was not uniform which can affect not only the creativity but also effect the of achievement of goal for the team.

```
sqlite3 -csv gitExtract.db "select C.label, G.group_id,
    C.count from user_group G, (select label, count(*)
    as count from (select id_grp, label from event e1
    where action = 'assigned' and time >= (select
    max(time) from event where e1.id_grp = id_grp and
    action = 'assigned')) group by label) C where
    C.label = G.user order by G.group_id" >
    issues_assigned_per_user.csv
```

5.9 Lesser number of bugs

The number of bugs reported by the team is an indicator of how effectively the team has tested their product. Teams who have not reported any bugs are not effectively testing the features in the product and it is very unlikely that a team has developed a product with no bugs. Hence we used this feature to detect the effective testing carried out by the team. The sql query is as follows:

```
sqlite3 -csv gitExtract.db "select A.identifier,
    A.user, B.group_id from (select * from event where
    label="bug") A, user_group B where A.user=B.user"
    > bug_label_count.csv
```

5.10 Dangling Issues

Dangling issues are the issues without milestones. This type of issue is not the desirable feature as every issue should belong to a milestone. We have used the same query used in the Section 5.1.

5.11 Distribution of Issues created per user

The distribution of issues posted per user should be more or less uniform. It is important to know the identifier or creator of the issue. Uniformity of such feature reflects the team effort.

```
sqlite3 -csv gitExtract.db "select a.group_id, b.user,
    b.issue_count from user_group a, (select user,
    count(*) as issue_count from event e1 where time
    <= (select min(time) from event where id_grp =
    e1.id_grp) group by user) b where a.user = b.user
    order by a.group_id" > issue_created_per_user.csv
```

5.12 Comments per user

The number of comments per user will give us an idea of the effective participation in communication of each member of the team. It is calculated as a ratio of the number comments per user to the total number of comments by the entire team.

```
sqlite3 -csv gitExtract.db "select A.group_id, A.id,
    B.count from issue A left outer join (select
    id_grp, count(distinct user) as count from comment
    group by id_grp) B on A.id_grp = B.id_grp order by
    A.group_id" > distinct_users_comment_per_issue.csv
```

5.13 Issues closed after Milestone due date

Each issue should be closed before the due date of milestone. If the issue is not closed before the due date of milestone, this reflects poor project management, lack of on time completion of task and lack of productivity.The sql query for this feature is as follows:

```
select ev.id_grp, G.group_id, ev.time-M.due_at as
    secondsAfter from (select id_grp, user, time from
    event ev1 where action = 'closed' and time >=
    (select max(time) from event where id_grp =
    ev1.id_grp and action = 'closed')) ev, milestone
    M, issue I , user_group G where I.id_grp =
    ev.id_grp and I.milestone_identifier =
    M.identifier and G.user = ev.user order by
    G.group_id;
```

5.14 Percentage of duplicate issues

Team members may create the same kind of issues independently and later mark them as duplicate. Though duplicate issues are marked and closed, creating a lot of them shows that the team members are not effectively communicating and more than one person tend to work on similar work which is a waste of effort. The sql query used for this feature is:

```
sqlite3 -csv gitExtract.db "select A.identifier,
    A.user, B.group_id from (select * from event where
    label like '%duplicate%') A, user_group B where
    A.user=B.user"
```

6. RESULTS OF FEATURE EXTRACTORS

The results of feature extractors will help us gain useful insight into the effects of the bad smells observed during the software development process.

6.1 Too few/too many issues per milestone

Figure 2 shows the number of issues in each milestone of all the groups. It is calculated by aggregating the issues under each milestone for each group.

6.2 Commit frequency

The commit frequency is shown in the Figure 3. It calculates the commits performed over a period of time by the members of the groups.

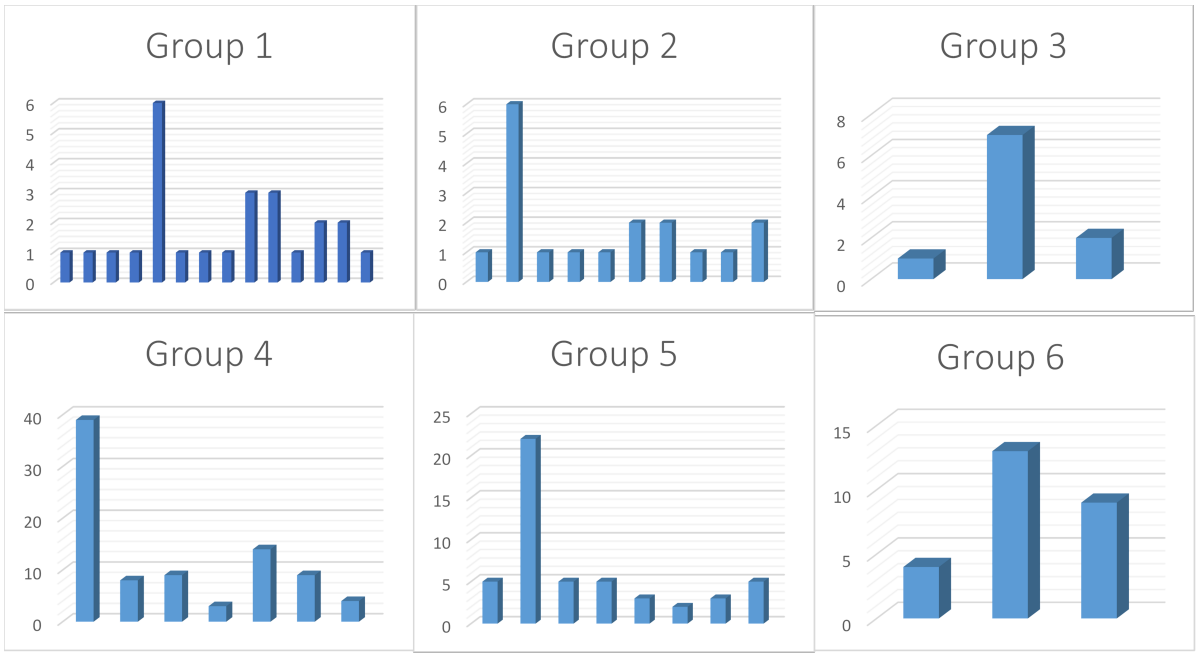

Figure 2: Number of issues per milestone for all six groups

Figure 3: The number of commits made every week by all the groups

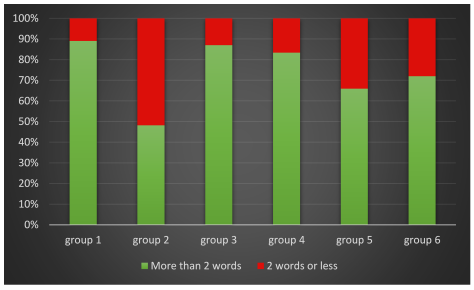

Figure 4: Ratio of the length of shorter vs longer comments

6.3 Commit with very short message

This section takes an aggregate of the number of commits made in a group with very short commit messages. A message is identified short if the number of words in the comment is lesser than 3. The ratio of the shorter and the longer messages is shown in the Figure 4.

6.4 Issues open for very short time

Figure shows the sum of the issues that are open for a very short time. It is calculated from the difference of the timestamp values of issue created time and the issue closed time. After that calculation, the issues with very short time span are taken into account.

6.5 Issues with no comments

The number of issues without any comments is calculated from the number of comments per issue and taking into account the issues whose comments count is zero. The statistics is shown in the Figure.

6.6 Uniformity of user commits

Figure 5 shows the percentage of commits made by each user in all the six groups. It is calculated from the commits made by each user to the total number of commits in a group.

6.7 Addition vs Deletion Ratio of Code

The addition and the deletion ratio of the code is calculated from the number of lines of code that are added or removed over a period of time and is shown in the Figure.

6.8 Issues assigned per user

Figure 6 shows the number of issues assigned to each user and it is calculated by summing up the issues assigned for each user in each group.

6.9 Lesser number of bugs

The number of bugs reported in each group is determined by summing up the tasks with the tag as bug. It is shown in the Figure 7.

6.10 Dangling Issues

The number of issues without milestones is calculated by summing up all the issues that are not linked to any milestone and is shown in the Figure 8.

6.11 Distribution of Issues created per user

The number of issues created per user is shown in the Figure 6. It shows the issues per user in all the groups by fetching the direct sum of the issue count.

6.12 Comments per user

The number of comments per user in each group in shown in the Figure 9. It is calculated by adding all the comments of a single user in all the tasks.

6.13 Issues closed after Milestone due date

The issues closed after milestone due date are calculated with the timestamp values and the ratio is shown in the Figure 10.

6.14 Percentage of duplicate issues

The number of duplicate issues is identified by the issues with duplicate tags. The values are summed and shown in the Figure.

7. BAD SMELLS DETECTOR

This section describes the various bad smell detectors using combination of feature extractors. These bad smell detectors are listed as follows:

7.1 Lack of Communication

Lack of communication is one of the bad smell detectors. When communication is not clear, it is not understood. Understanding is the point of information transfer. Lack of communication leads to lost production. Any time during a project, if there is need to have to re-explain or re-communicate or re-anything because it was not properly communicated in the first time, is a waste of time. This type of bad smell detector utilizes the following feature extractors:

- Issues with no comments
- Distribution of issue created per user
- Comments per user
- Commits with very short message

7.2 Improper Planning

Planning is the foundation on which project execution is based. Good planning ensures that the resources are ideally utilized. Improper planning results in the assignment of redundant tasks, thus increasing the cost and time taken to complete a project. This type of bad smell detector utilises following feature extractors:

- Too few/too many issues per milestone
- Commit frequency
- Issues open for very short time
- Uniformity of user commits

7.3 Poorly defined roles and responsibilities

The distribution of tasks among the members of the team determine how effectively the team has divided the labour among them. The scope of a task should also be taken into account when they divide the work among themselves. The feature extractors that result due to this bad smell are listed below.

- Issues assigned per user
- Distribution of Issues created per user
- Uniformity of user commits

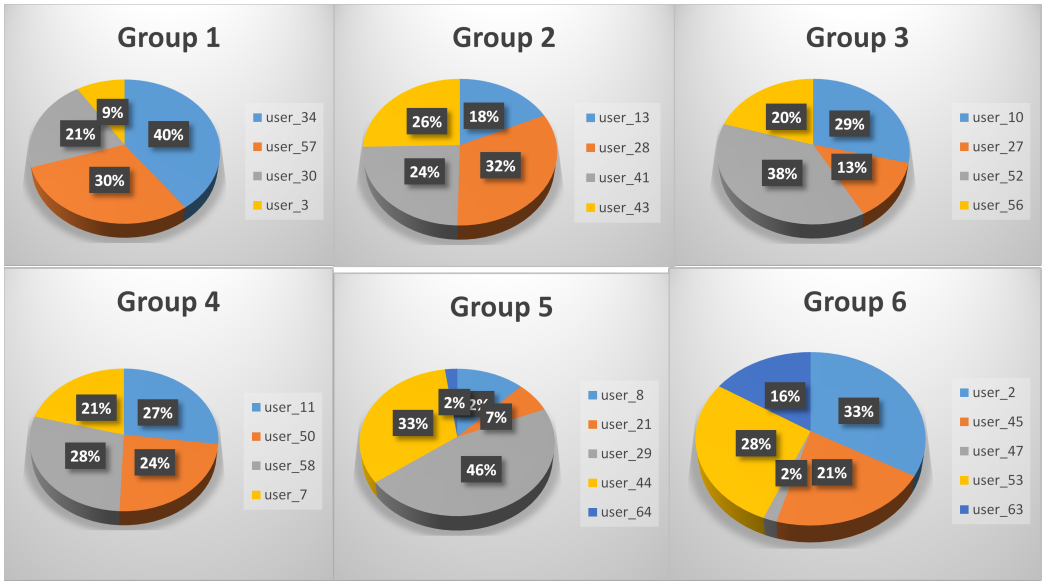

Figure 5: Percentage of commits per user in all six groups

Figure 6: Distribution of Issues created per user

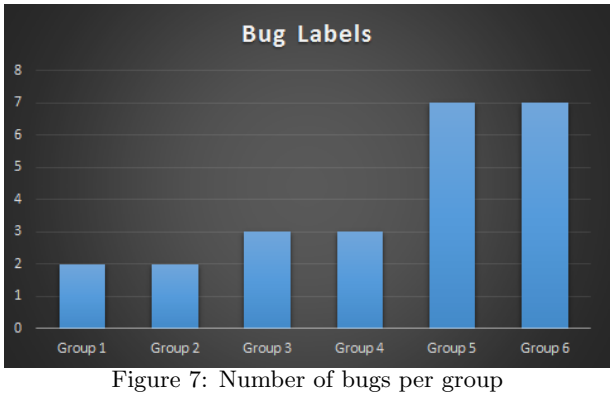

Figure 7: Number of bugs per group

Figure 8: Number of issues without milestone in each group

7.4 Inadequate testing processes

Any product developed should undergo adequate testing so that it gets better over a period of time. Testing will lead to finding and fixing bugs and enhancements and even lead to new feature ideas. The feature extractors that results from inadequate testing process include the ones listed below.

- Lesser number of bugs
- Addition vs Deletion Ratio of Code

7.5 Scope Creep

Often the scope of the project is not realised by the team during the early stage which leads to poor planning and poor division of tasks. Some tasks may take longer while the others might take shorter time. In such cases the team should be able to divide the bigger task into multiple smaller tasks so that the workload will be equally distributed among the members of the team.

- Percentage of duplicate issues
- Issues closed after Milestone due date
- Dangling issues

7.6 Non uniform division of task

This bad smell leads to few people contributing more while other contributing very less to the project.This behavior can be extracted from the following features.

- Issues assigned per user
- Uniformity of user commits

8. BAD SMELL OUTSET

To detect the stink score of each team, we have calculated the bad smell score on each feature we analysed. For convenience we have marked the features with codewords f1,f2,etc. The mapping is given in the table below.

F1	Too few/too many issues per milestone
F2	Commit frequency
F3	Commit with very short message
F4	Issues open for very short time
F5	Issues with no comments
F6	Uniformity of user commits
F7	Addition vs Deletion Ratio of Code
F8	Issues assigned per user
F9	Lesser number of bugs
F10	Dangling Issues
F11	Distribution of Issues created per user
F12	Comments per user
F13	Issues closed after Milestone due date
F14	Percentage of duplicate issues

The bad smell scoring has been done as per following scheme for each of the feature:

- F1- We give a bad smell score of +1, if number of issues per milestone is below or above the threshold range($> 20\%$ and $< 80\%$). Threshold range has been calculated, taking the mean of issues per milestone. We recognize if the number of issues for more than 20% milestone is greater than the threshold range, it is bad smell of type "Too many issues" per milestone. Similarly if the number of issues for 20% milestone is less than threshold range, it is of bad smell of type "Too few issues" per milestone and the group is given a score of +1.
- F2- Commit frequency is given a bad smell of +1 if maximum commit has been done in the last week. We have identified the threshold as 50%. So if more than 50% of code has been committed in the last week of project duration the group gets an score of +1.
- F3- We feel that if the commit message length is less than 3 words, it may not convey the information adequately. We have given a bad smell score of +1 if more than 50% of the commit messages have the length of less than 3 words.
- F4 - For the issues in open state for a very short duration we have given a bad smell of +1. The threshold to determine if an issue has been closed in a very short time is calculated as the average lifetime of of all the issues. This gives the average velocity of the group and determine if an issue is closed in a very short interval.
- F5- This is fairly simple. A group gets a bad smell of +1 if more than 25% of the issues have no comments in their lifetime and has been closed.
- F6 - We give a stink score of +1 if the number of commits made by any user is less than the average number of commits made by all the users in the group. First we

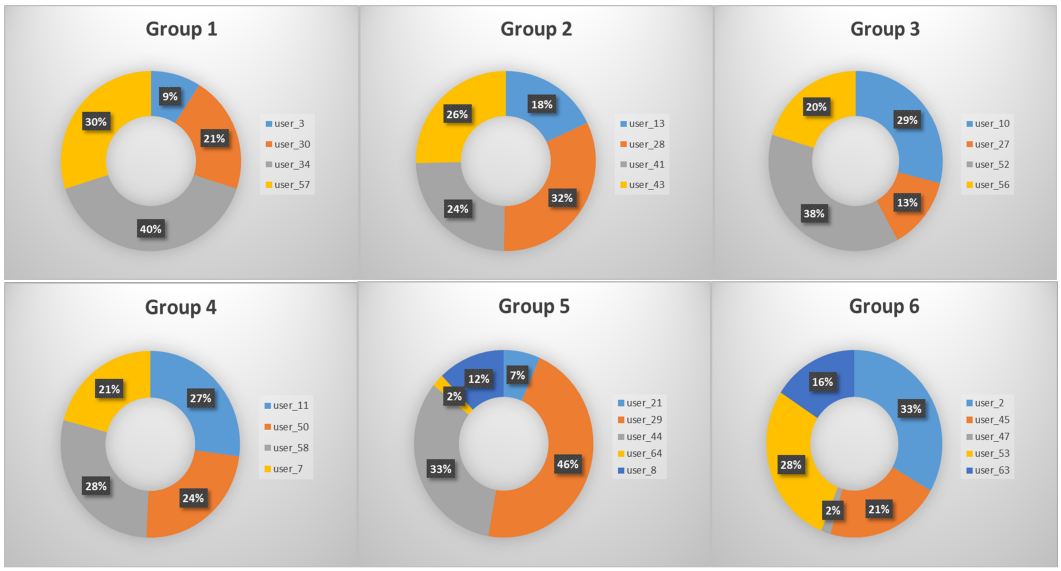

Figure 9: Percentage of comments per user in all six groups

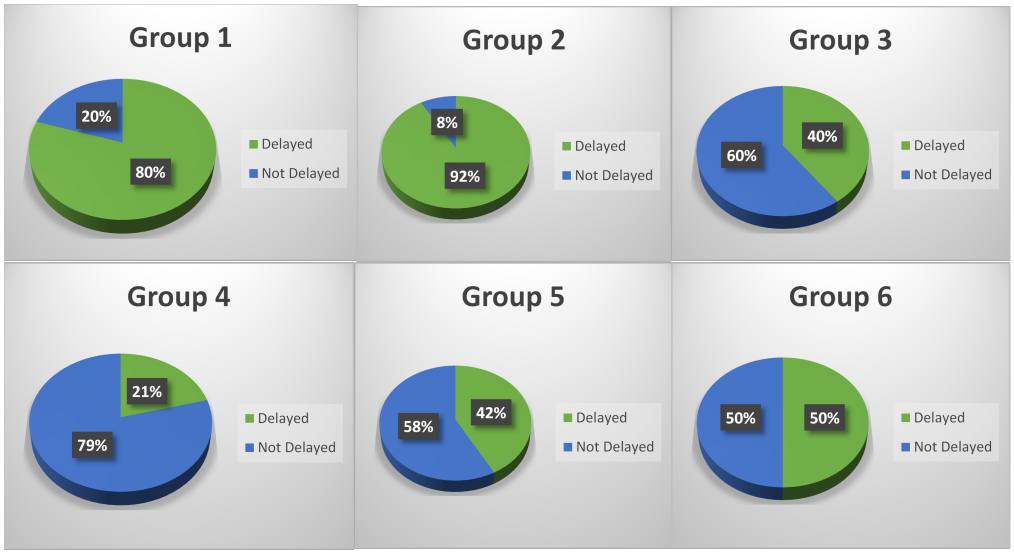

Figure 10: Ratio of issues closed before and after milestone due date

calculate the average number of commits made by the team and any user committing lesser than this value will earn a stink score of +1 to the entire team's score.

- F7- We feel that for a better outcome there should not be only addition of code but also deletion of extraneous code or bad code is required. Ideally the addition and deletion should be between 50:50 to 75:25. Any group having this ratio above or below this gets a bad smell of +1.
- F8 - If the number of issues assigned to a single member of the team is less than the weighted average number of issues per member, we assign a stink score of +1 to the team. The weighted average number of issues per member is calculated by the total number of issues divided by the number of members and an adjustment factor of 10
- F9 - We assign a stink score of +1 to the team if the team has reported bugs lesser than a certain threshold. The threshold is calculated as 20% of the number of issues or tasks the team worked on.
- F10 - Dangling issues are the issues which does not belong to any milestone. So if a group has significant number of dangling issues, the group gets a bad smell score of +1. The term 'significant' has been attributed to a value of 25%.
- F11 - We assign a stink score of +1 to the team if the number of issues created by any user is less than the average number of issues per user with an adjustment factor of 20%.
- F12 - If number of comments from any user of the group is below the average number of comments per user for the group then the group gets a bad smell score of +1.
- F13- No issue is expected to be delayed beyond the milestone. If more than 25% of the issue belonging to a group has been delayed beyond the milestone due date, then the group gets a bad smell score of +1.
- F14- The group gets a bad smell score of +1 if the percentage of duplicate issues is more than 20% of total number of issues

  The bad smell score table for each of the detectors has been given in Figure 11.

	G1	G2	G3	G4	G5	G6
F1	1	1	1	0	0	1
F2	0	1	0	1	0	1
F3	0	1	0	0	0	0
F4	0	1	0	0	1	1
F5	0	1	1	1	0	0
F6	0	1	0	0	1	1
F7	0	1	1	0	0	1
F8	0	0	1	0	0	1
F9	1	1	1	1	1	1
F10	0	0	0	0	0	1
F11	1	0	1	1	1	0
F12	1	1	0	1	1	1
F13	1	1	1	0	1	1
F14	1	1	0	0	1	1

Figure 11: Stink score of each team for each feature

In the above Figure 11, we can see the scores of each team for each of the feature based on the description given in the previous section. Now the scores from the different features are aggregated according to the bad smell detectors we observed in Section 7. This calculation can be seen in the Figure 12.

In Figure 12, the overall stink score is calculated for all the six groups based on the features responsible for each bad smell detector. The stink score of each feature is summed up to arrive at the final stink score of a specific bad smell detector. The stink score chart is given in Figure 13

Bad Smell Detectors	Features	G1	G2	G3	G4	G5	G6
Lack of Communication	F3 + F5 + F11 + F12	2	3	2	3	2	2
Improper Planning	F1 + F2 + F4 + F6	1	4	1	1	2	4
Poorly defined rules and responsibilities	F6 + F8 + F11	1	1	2	1	2	2
Inadequate testing processes	F7 + F9	1	2	2	1	1	2
Scope Creep	F10 + F13 + F14	2	2	1	0	2	3
Non uniform division of task	F6 + F8	0	1	1	0	1	2

Figure 12: Stink score of each team for each bad smell detector

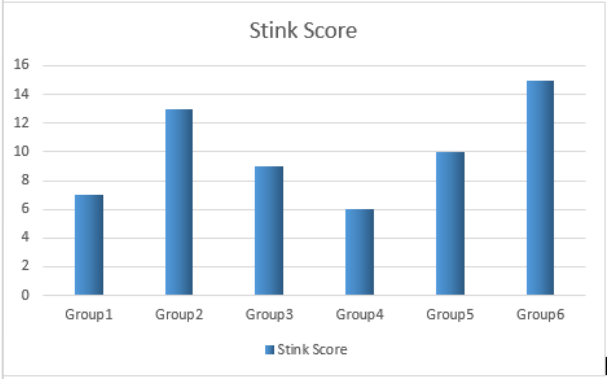

Figure 13: Stink score of each team for each bad smell detector

9. EARLY WARNING

We chose our feature 6 - Uniformity of user commits as early warning detector due to the following reasons:

- It will make work load distribution even in the team. If uniformity of commit is not there, it will lead to declination of team productivity.
- It will make the project delivery on time.
- It will lead to proper testing in each of the development phase.
- Teams will be able to deliver the best product since multiple members can have diverse dimension of thought and thus think about pros and cons in all the direction.

11. CONCLUSION

The analysis of the bad smell detectors on the data collected from the GitHub repositories of various projects has given us useful insights into the effectiveness of the team work in software development. We have analysed various features that will be responsible in determining the effectiveness of the team performance.The causes of the features are analysed as bad smell detectors which lead to such inefficiencies in the team performance. In order to evaluate the stink score of each team, we assigned various metrics to score the teams based on each feature. Finally we calculated the final stink score of each team for each bad smell detector by summing up the stink scores of the features responsible for the bad smell detector. As a result, we have given an efficient analysis of six teams on the basis of the above discussed factors and also gave early signs of the bad smells across the teams.

Group Number	Total Stink Score	Score for early warning detector (Number of commits per user)
1	7	0
2	13	1
3	9	0
4	6	0
5	10	1
6	15	1

Figure 14: Early warning detector results

10. EARLY WARNING RESULTS

The early warning results for the chosen feature is shown in Figure 14. It indicates that the early warning score clearly detects the issues in the progress of the project earlier. This is also backed up by the overall stink score of the groups. We can see that the final stink score of the Groups 2, 5 and 6 to be very high indicating that there were some bad smells in the development process of the groups. The early warning indicator shows a boolean indicator of 1 for these groups which means it is able to detect earlier that the projects of these groups are experiencing some bad smells.