Data Reflection

Karan Jadhav
Masters of Computer Science
North Carolina State University
Raleigh, North Carolina
Email: kjhadav@ncsu.edu

Samantha Scoggins
Masters of Computer Science
North Carolina State University
Raleigh, North Carolina
Email: smscoggi@ncsu.edu

Qaiss Alokozai
Masters of Computer Science
North Carolina State University
Raleigh, North Carolina
Email: qalokoz@ncsu.edu

***Abstract*—Software Engineering team projects can vary greatly. Project teams hope to avoid "bad smells" to avoid inefficient project management. This paper compares the data from three different teams to analyze and reflect on software engineering project management and processes.**

I. Introduction

Over the course of the semester in Spring 2017, 10 teams worked on researching, building, and testing software solutions to everyday problems. Software engineering projects can vary greatly in terms of pacing, goals, and contribution distribution. Previous papers have discussed our teams development of three separate solutions that could help with Attendance Management. This paper's goal is to compare the data we produced from planning, organizing, and contributing to our project to two other teams for the purposes of learning and reflecting on how Software Engineering projects can be more effective and avoid "bad smells."

This paper will first discuss data collection and anonymization. Then it will go on to compare specific data arranged in data sets determined by feature extractors. Based on the feature extractor results, "bad smells" can be revealed. These "bad smells" reveal team practices that could be improved for a productive and efficient software engineering project.

II. Collection

A. Data Mining Script

We made use of the gitable.py[1] script available on github. It's a small project management tool that does data mining from Github by pulling data using its REST based API from specific repositories. It is meant to serve as a starting point and a tutorial. This script pulls issue data in the form of events. Events are basically occurrences in a repository in terms of actions performed on issues or the changing of their state. So one issue can have multiple events associated with it. The script pulls basic data associated with each issue. This includes the issue ID, time stamp, action performed, user and the associated milestone. The figure below shows the code snippet that does this.

```
for event in w:
  issue_id = event['issue']['number']
  if not event.get('label'): continue
  created_at = secs(event['created_at'])
  action = event['event']
  label_name = event['label']['name']
  user = event['actor']['login']
  milestone = event['issue']['milestone']
```

We extended this script to capture additional data associated with the other two main features that Github provides, milestones and commits. We pulled milestone related data from within issue data which in turn is part of an event. We could have pulled data from the "/milestones" API service, but that would have led to segregation of milestone data from issue data. And ultimately issues are associated with milestones, so we would have needed a way to connect the two. Plus it seemed simpler to simple extend the already wriiten code. We pulled milestone name, description, user, creation time, due time, close time and status data for each milestone. The figure below shows the code snippet that pulls milestone related data.

```
milestone = event['issue']['milestone']
m = False
if milestone != None :
    m = True
    ms_id = milestone['number']
    ms_desc = milestone['description']
    ms_title = milestone['title']
    ms_user = milestone['creator']['login']
    ms_ctime = secs(milestone['created_at'])
    ms_duetime = secs(milestone['due_on'])
    ms_status = milestone['state']
    if ms_status == "closed":
        ms_closetime = secs(milestone['closed_at'])
    else:
        ms_closetime = ""

user = event['issue']['user']['login']
actor = event['actor']['login']
```

In terms of commits related data we stuck to, what we felt were simple data fields. These were the author, the timestamp and the message associated with each commit. We chose to do this since we wished to focus on the number of commits by each user and also the frequency of commits by each user over the course of the project. The corresponding code snippet is shown below.

```
if e['author'] is None:
    continue
a = e['author']['login']
t = secs(e['commit']['author']['date'])
m = e['commit']['message']
```

Ultimately we exported our data into .csv files. The code snippet below shows an example.

```
filename = group_id+".csv"
with open(filename, 'w', newline='') as outputFile:
    outputWriter = csv.writer(outputFile)
    outputWriter.writerow(["issue_id", "issue_name",
    for issue, events in issues.items():
        for event in events:
            if event.m:
                outputWriter.writerow([issue, event.i
            else:
                outputWriter.writerow([issue, event.i
```

One point regarding our data that we'd like to state is that it wasn't well structured or normalized data that we ultimately had at the end of the script run. Issue and milestone data particularly weren't well structured and there was a lot of repetitive data. This structure of the data was the original format as obtained from the Github API, where milestone data comes nested within issue data. We felt it best to not change this structure and we felt writing complex queries as per our requirements to extract succinct, meaningful data from our unstructured, non-normalized data would be a good approach. A subsequent section describes in detail how we imported this data to a different platform and the queries we wrote to extract said data.

B. Anonymization

It is vital, as part of any project that utilizes and showcases individual related data on a public platform, to anonymize said data. The same was the case with our project. Since we were dissecting and analysing the repository data of our fellow coursemates, it was imperative that we anonymize the data. We made use of the anonymization techniques implemented by one of our fellow classmates[2]. They anonymized the repository name as "group¡number¿" and the individual users under each group as "user¡number¿". Following this they exported all the anonymous user/group mappings to a .csv file and used that to ultimately export data for all the groups and users. The code snippets fore this are below.

Figure 1. Anonymized groups and mappings

```
group_id = "group{}".format(index+1)
with open("private_mappings.csv", 'a', newline='') as fi
    outputWriter = csv.writer(file)
    outputWriter.writerow([reponame, group_id])
    for username, user_id in mapping.items():
        outputWriter.writerow([username, user_id])
```

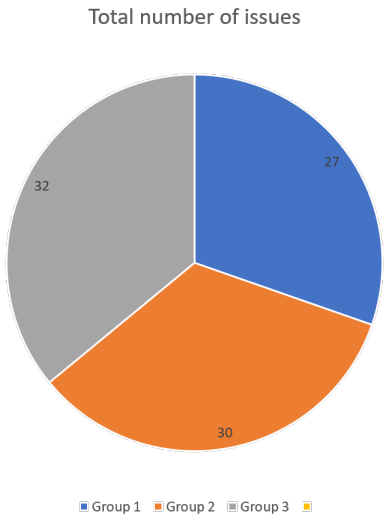
Figure 2. Anonymizing users

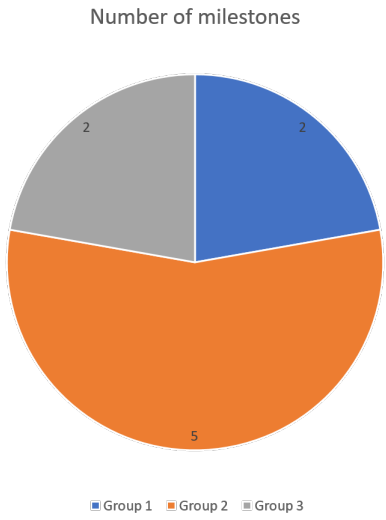
```
if not mapping.get(user):
    mapping[user] = "user{}".format(len(mapping)+1)
user = mapping[user]
```

III. Summary

A. Description of Entire Data Set

Data was taken from three groups. One of which was our own. The other two groups were chosen at random. The following totals were found when comparing basic data between groups:





number of commits:

As shown for Distinct Features, there are a similar number of issues per group, the mean being 29.66 and median 30. This can indicate that all groups decided on a similar amount of tasks and goals to be completed for their project. However, the number of milestones between groups differ. So at least one group organized their tasks and goals differently. Its important to not that milestones can have different meaning between groups. As in Milestones could be ordered steps with preferably no overlap for a project, or they could be overlapping sections of a project to be completed in parallel.

B. Feature Extractors

The following feature extractors separate and compare data so as useful information can be pulled. All data except commit data was contained in one table for each group. Each csv file created with that table was placed into Access. So, one group will have one table with the following column headings each row containing an auto-id as the key. Simple Queries were created to help extract and simplify data for visualization and future use. Relevant Queries used will be listed.

Issues per User

The first feature our team was interested in looking at was the distribution of issues assigned to each user. An equal distribution could mean good planning, while an unequal distribution paired with other factors could be indicative for unhealthy work distribution plans.

Query used:

```
SELECT [g1- distinct issues].user, count([g1- distinct issues].group1_issues) as issues
FROM [g1- distinct issues]
GROUP BY [g1- distinct issues].user
```

Labeled vs. Unlabeled Issues

The second feature we were interested in was whether an issue was labeled or unlabeled. While our group did not use labels extensively, it would be useful to compare the use of labels to see if that made a difference in organization.

Query used:

```
SELECT Count([g1- distinct issues].group1_issues) AS Expr1
FROM [g1- distinct issues], [group1-new]
WHERE ((([group1-new].action)="labeled")) and [group1-new].issue_id = [g1- distinct issues].group1_issues;
```

Issues with vs. without Milestones

We also wanted to look at the distribution of issues with and with out milestones compared to the total. These extra issues outside of milestones could also be indicative of unorganized planning. However, if the number of issues with out milestones is low, they could have been random extra issues that came later in time, or may not fit in with the goals of the already set milestones.

Query used:

```
SELECT Count([g1-dis_issue_mile].group1_issues) AS NumIssues_w_milestones
FROM [g1-dis_issue_mile]
WHERE ((([g1-dis_issue_mile].milestone_id) Like "*"));
```

Closed issues to Assigned Issues per user

Issues assigned per user may show planning efficiency,however if we compared the number of issues closed by a user to the number that user was assigned to, it would reflect upon the actual distribution of work in relation to issues.

Query used:

```
SELECT [group1-new].actor, Count([group1-new].action) AS Closed
FROM [group1-new]
WHERE ((([group1-new].action)="closed"))
GROUP BY [group1-new].actor;
```

Issues closed by assignee

Another way to look at issue related work distribution is to look at whether an issue was actually closed by its assignee. Though users can close each others' issues to make up for the ones he/she did not close to keep the ratio described before (Closed Issues to Assigned Issues per user), this will reveal whether users are supporting each other or doing all the work.

Query used:

```
SELECT [group1-new].actor, Count([group1-new].action) AS ['# closed assignments']
FROM [group1-new]
WHERE ((([group1-new].action)="closed") and [group1-new].actor = [group1-new].user)
GROUP BY [group1-new].actor;
```

Issue finish time

Taking the time it took to finish an issue after the first action was performed on it, which we took as that was when the issue was being worked on, would give insightful information as to the norm of issue times.

No Query was used.

Open Milestones compared to Total Milestones

The ratio of Milestones left open to the total number of Milestones in a group could show the percentage of the project that group felt they accomplished, as well as a bad planing indicator.

Query used:

```
SELECT DISTINCT ([group1-new].milestone_id), [group1-new].ctime, [group1-new].closetime
FROM [group1-new], [g1-distinctMilestones]
WHERE [group1-new].milestone_id = [g1-distinctMilestones].milestone_id;
```

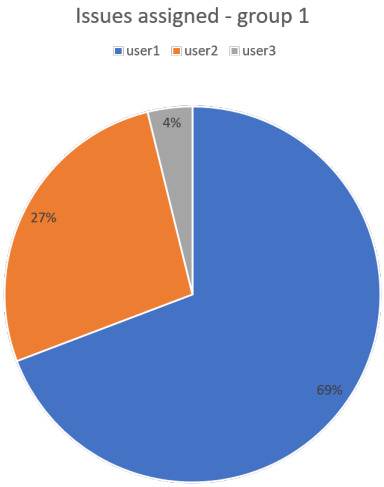
Commits over time

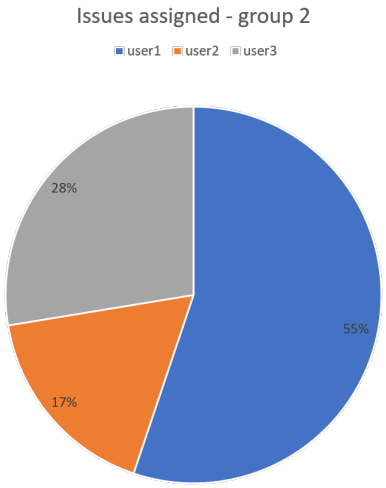
This is the most important feature extractor. Viewing when users committed to Github will show how the project progressed over time. It would reveal procrastinating users, as well as spikes of when contributions to the project were added.

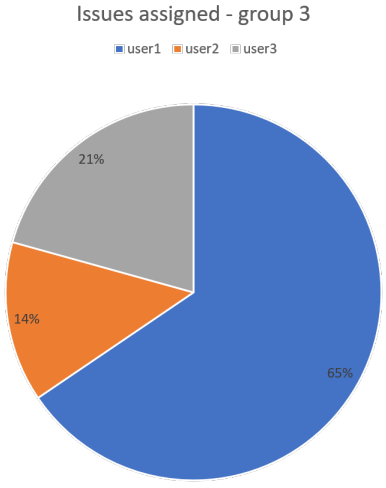
No Query used.

IV. Results

A. Issues per User

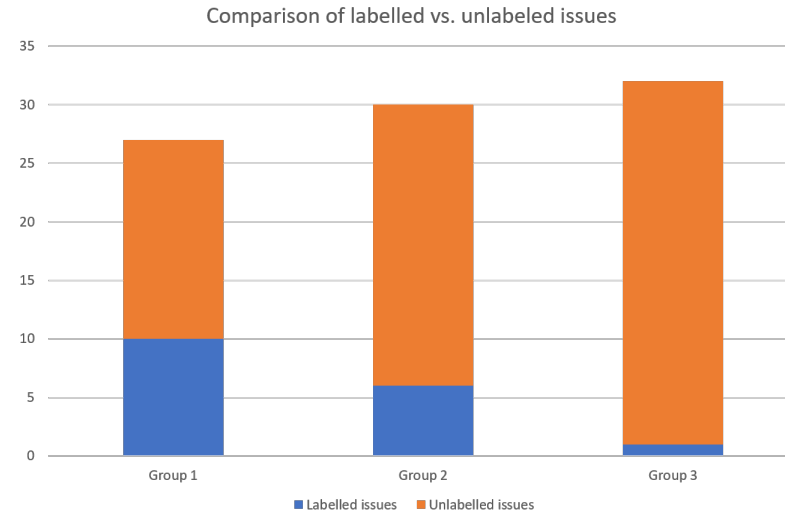






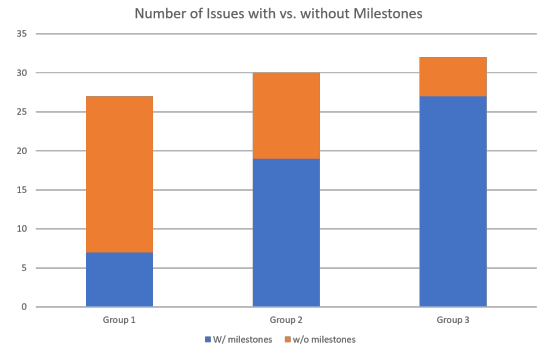
Among all the three groups we can see one user having a majority of issues assigned to them. This ranges from 55-69 percent. This clearly points to an unbalanced distribution of work among the users of each group and is clearly a bad smell. Unbalanced distribution of work leads to slower progress of the project not to mention the added burden on a few specific users. There is no one clear outlier in this case, with all groups exhibiting this bad smell.

B. Labeled vs. Unlabeled Issues



In all the three groups we see a much higher number of unlabeled issues as compared to labelled issues. These account for 62-96 percent of the total issues. Unlabeled issues are an indicator of a lack of proper segregation of tasks into the right categories. This leads to sporadic, not well thought out, scattered tasks, not focussed on one particular area of the project, for example, enhancements, bug fixes etc. While all the groups exhibit this bad smell, group 3 is the clear outlier with almost all issues unlabelled.

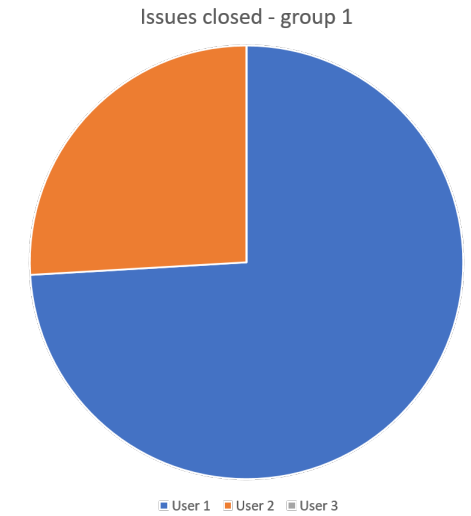
C. Issues with vs. without Milestones

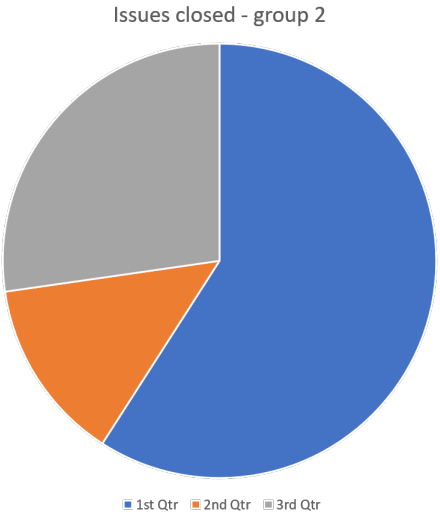


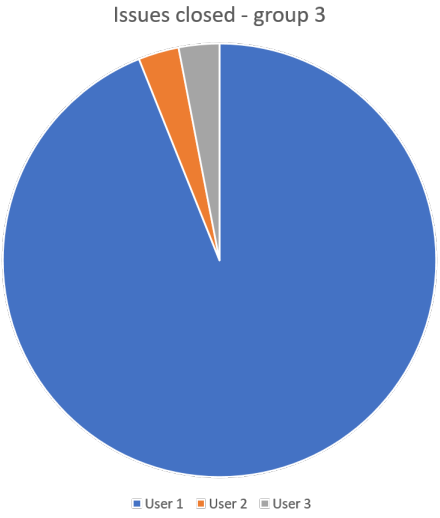
As we can see from the figures, two groups have a relatively low number of standalone issues, i.e., issues that are not associated with a particular milestone. However group 1 has a high number of non-milestone issues clearly indicating a bad smell. Issues not associated with milestones are detrimental to project progress because they are an indicator of a lack of a

bigger goal. Issues need to make up a milestone so that small steps can lead to the completion of a bigger phase. Otherwise it leads to stand alone, fragmented work that does not achieve progress in one particular area of the project.

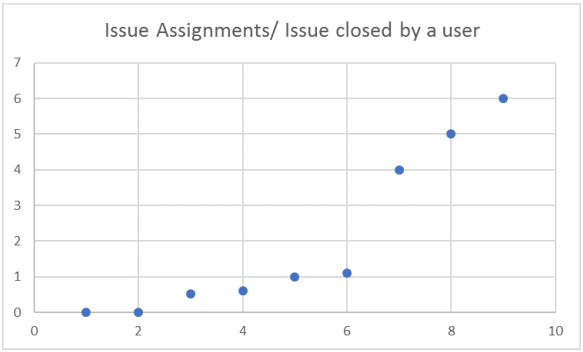
D. Closed issues to Assigned Issues per user





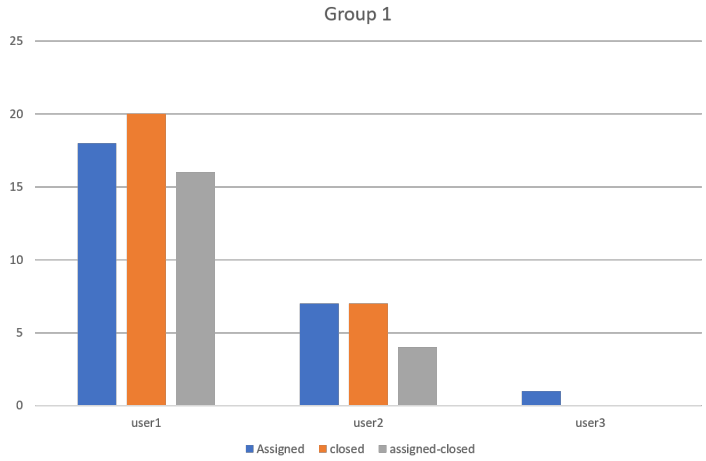


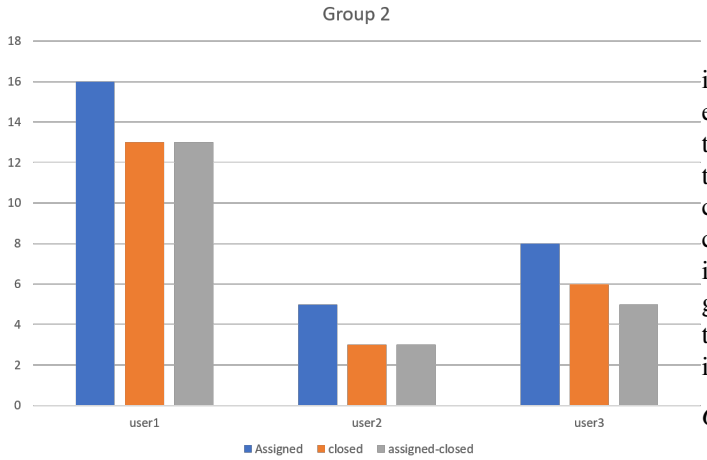
This is a relatively straightforward observation of user actions. We can see that in all three groups one user was responsible for closing between 60-95 percent of the issues. This is similar to the issue distribution observation that we talked about in the first part. An unbalanced number of issues being closed among users of a group points to few specific users finishing a majority of the tasks. This leads to slower progress on the project and an added effort on part of certain users. All three groups exhibit the bad smell in this case.

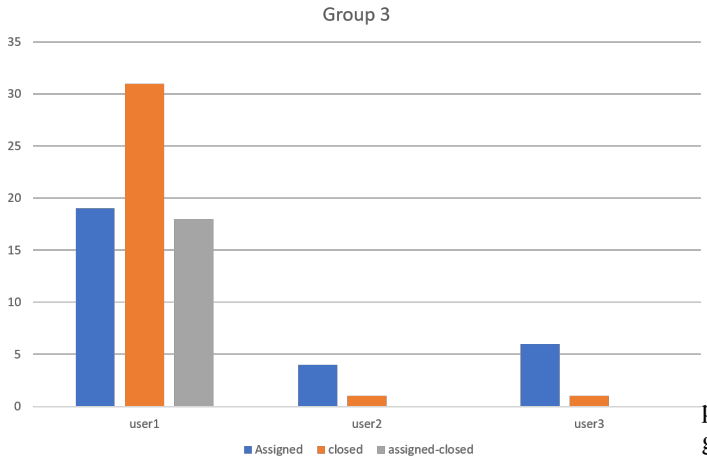


This figure is a ratio of the number of assigned issues to the number of closed issues for each user of each group. The lower the ratio, the better, since it shows that the user closed more issues than he/she was assigned. The first six users all have ratios equal to or under 1. This indicates them having closed as many or more issues than they were assigned. The last three users are the clear outliers who closed fewer than 25 percent of the issues they were originally assigned.

E. Issues closed by assignee

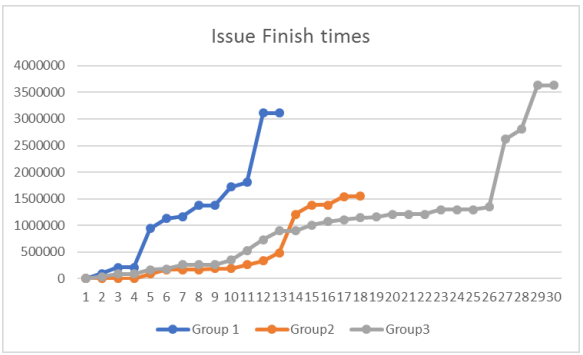






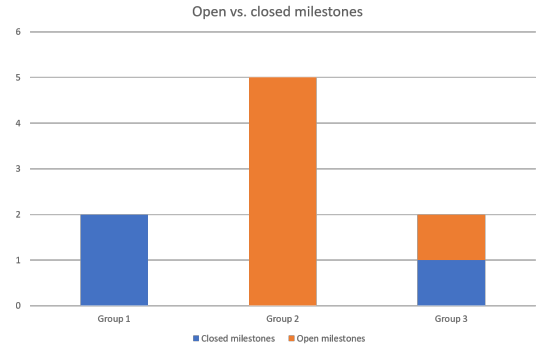
This observation is similar to the one just before. Here we make a comparison between the number of issues assigned to a user, the number of issues closed by the user and the the number of issues closed by a user that were assigned to them. The first two groups have a relatively satisfactory graph with regard to this criteria. However group 3 seems to be the outlier in this case. User 1 is closing a much higher number of issues than what he/she was originally assigned. A little under half these issues are ones that were assigned to someone else. This is clearly a bad smell since it points to other users not finishing their assigned tasks and the burden ultimately coming onto a few particular users.

F. *Issue finish time*



The general trend of time taken by a group to finish their issues relative to other groups is a good indicator of the general efficiency of the group as a whole. If we look at the figure in this case, we see that while group 2 has lesser issues compared to group 3, their finish times for the issues until the last common issue number is about the same. Group 1 are the clear outlier in this case. Not only do they have quite fewer issues than the other two groups, about 40 percent of those of group 3, they still took a comparable amount of time to finish them. This is a bad smell as the time taken by them per issue is much higher compared to the other two groups.

G. *Open Milestones compared to Total Milestones*

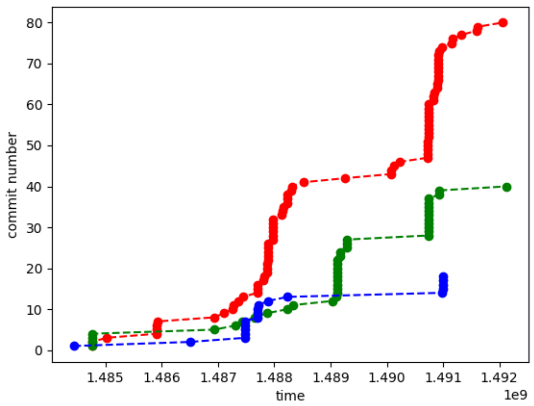


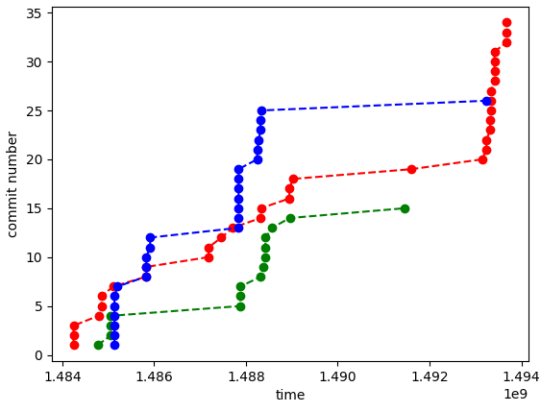
Closed milestones are an indicator of the progress of the project as a whole. This is because milestones are smaller goals that follow a sequential order. In the three groups we see a great variation in this criteria. While one group has closed all their milestones indicating perfect progress on their project, another group has closed only half their milestones. Group two is the outlier here, not having closed any of their milestones. This is clearly a bad smell since this points to non-sequential and unplanned goals running in parallel. This shows that no phased of the project was accomplished, end to end.

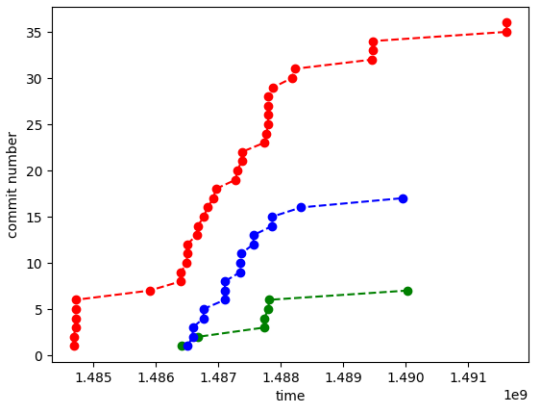
H. *Commits over time*

Commits over time will show the progress each group has made. As the below graphs show, in each group there is at least one user who does not commit as much nor as consistent as the other two users. All graphs show a spike in commits in the median of the time stamps, which would reflect implementation iterations and data that would be normal around February. Over all the expected derivative of the regressed line that would come from all commits in a group would ideally show as being equal to one, or the slope of the commits over time should be one. This would show a groups consistency throughout a project. The graphs below do reflect that. However, one graph (group3) shows one user as the only user committing in the beginning, while the other two do not. Also, with Group three, while the other two show spikes of commits near the end of their time lines, group three does not. Though group three also has more commits near the beginning of their time line. It's possible that the bad smell here is the other two groups did not commit

as much in the beggining and so needed to commit more near the end. Group 3 most likely had extensive planning, so no procrastination commits was reflected. Though there was a user that had generally more commits in group3, combined with the possible extensive planning, this does not mean the other users were not contributing to their project.







V. Conclusion

Software engineering projects require tedious amount of work with a focus on minute details as well as focus on ideas and techniques for better productivity. Secondly the increase in effectiveness of Software engineering projects requires extensive data mining. In this project analysis report we focused on number of important extractors from Github repositories of the groups involved in SE17 software engineering projects. We mainly focused on issues, milestones, commits and compared among various groups (group1, group2 and group3). We used python script to extract data form GitHub and utilized the power of SQL queries to clean the data. Our analysis results shows various bad smell detectors. The data shows bad smells such as issues which are open, unorganized assigned issues, and variation in number of commits among groups. We observed larger variation in issues, commits and other factors among users which are indication of lack of planning, unorganized approach to project planning as well as time management. We also observed total amount of work allocated to each user and how much contribution they have made. In conclusion our data analysis of these three groups shows unorganized assigning of issues is possible because of lack of planning within group which bring multiple factors to impact the overall path of the project. When issues are closed randomly, they indicate that the team is not organized. However, Group 3 showed promise in having generally less bad smells and showed higher planning and organization. If bad smells like procrastination, and late planning can be avoided, Software engineering projects would go much smoother.

[**1**]https://tinyurl.com/m2gzg4h

[**2**]https://tinyurl.com/mzhay6l