Analysis of Software Projects for Bad Smells

[Analysis Report]

Vivek Gopalakrishnan
vgopala2@ncsu.edu

Anbarasi Manoharan
amanoha2@ncsu.edu

Gautam Jeyaraman
gjeyara@ncsu.edu

Jaithrik Yadav Bollaboina
jbollab@ncsu.edu

ABSTRACT

Collaborative software development projects have become much more easier than ever since the introduction of Github since its introduction in the year 2008. Although it has numerous advantages, there is no proper way to analyze the quality of a project built on Github. Even with the fact that everything on Github is publicly available, the lack of a proper analyzer for the projects is quite surprising. This report describes the work that has been done to evaluate software projects and identify bad smells that occurred during the development phase. Our objective was to find various bad practices that were followed during the development of the projects as part of the CSC 510 course. Also, we tried to develop a bad smell detection system that could predict the future trends of the project according to the detected bad smells earlier in the development phase.

General Terms

Software Development, GitHub, Code Smells

Keywords

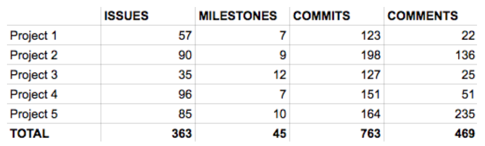
Bad Smeells, Code Smells, Feature Extraction, Data Collection

1. INTRODUCTION

Github is an excellent tool for developing collaborative projects in the field of software engineering. This report consists of the evaluations done on five such projects from the CSC 510 projects list. As per the guidelines, anonymity of the projects is maintained. Various features have been extracted from all of the projects, which were considered to be important in detecting the bad smells during the development phase. From these features, we can study how the groups functioned jointly.

Visual data would give more idea to anyone than written explanation. Hence, we developed visualizations to examine the effects of each of the features. This would allow us to easily compare the work done in various projects. Identifying bad smells is a great way to evaluate the quality of the product developed. Moreover, if detected at an early stage, they can also be used to predict the implications that might arise in the future. Hence we developed a script that could give the user an early warning if any bad practice is detected at any stage.

In the next section, we explain about the data collection process in detail. Section 3 provides details about the bad smell detection process and a discussion about the results obtained. Next, we speak about our early warning procedure and the implementation behind it. Finally, Section 4 concludes.



	ISSUES	MILESTONES	COMMITS	COMMENTS
Project 1	57	7	123	22
Project 2	90	9	198	136
Project 3	35	12	127	25
Project 4	96	7	151	51
Project 5	85	10	164	235
TOTAL	**363**	**45**	**763**	**469**

2. DATA COLLECTION

2.1 What we collected

We downloaded the datasets of five different repositories from Github. Github provides a whole variety of data representing issues, milestones, comments, commits, etc. The data is available in a rest api and we had to make http calls to fetch the data in json format. Also, we fetched the data, processed it and stored it as json files locally so that we did not have to fetch the files every single time. The data contained the following fields:

- Issues

  - Action, What, When, User, Milestone

- Milestones

  -User, Created At, Name, Due At, Closed At, Description, Id, Number

- Comments

  - Text, Created At, Updated At, User, Issue, Id

- Commits

  - Message, Sha, User, Time

All these fields were first fetched, processed and stored as json files for five different teams using the Github api. The structure of the data was generally a list of dictionaries that hold all the fields of each time. The only difference is with the issues, which are stored as dictionaries with their ids being the key and the values being a list of dictionaries that contain all the events in that issue. The number of items in each category was not even and so, many of the metrics had to be calculated as percentages. The distribution of the data on each of the types and projects is shown in the figure.

2.2 How we collected

We started with the given gitable.py as a starting point. From there, we started modifying that code to read configurations from a json file, download and process data from all the repositories specified in the file and store them again as json files. So, the script took each project's github url and performed the same set of actions on them to get the data. Initially the process had only the steps to download and process issued of a project. We added options to download and process commits, comments and milestones as well. So, our current data fetching mechanism follows the following steps:

- Have the target repositories specified in a configuration files
  - In json format
  - Just a dictionary with project id as key and the github url as value
  - Named links.json
- Start gitable.py
  - This is the python file that contains the altered code to download and process the whole dataset.
- The script reads each input project from the configuration
  - Takes each project one by one and starts processing
- It downloads each of the above mentioned data fields
  - Each of the above mentioned fields follow the same steps
  - A http fetch to get the raw json data for each page
  - This loops through all the available pages for each feature
- Process the data
  - Here, we alter each of the downloaded data to the format that we want it to be in
  - Example include changing all date and time to a fixed timestamp format
  - Choosing only the required fields we need for analysis
  - Converting it to a format that we can access easily in
  - We also, anonymize the data to remove references of the project and the user
- Store them as json files
  - We then store the processed data of each of these projects into their respective json files in the output folder
  - These files can then be loaded in other scripts to perform various analytics

Once all these steps are completed, we will have all the data required for the analysis stored as json files in our output folders, ready to be analysed.

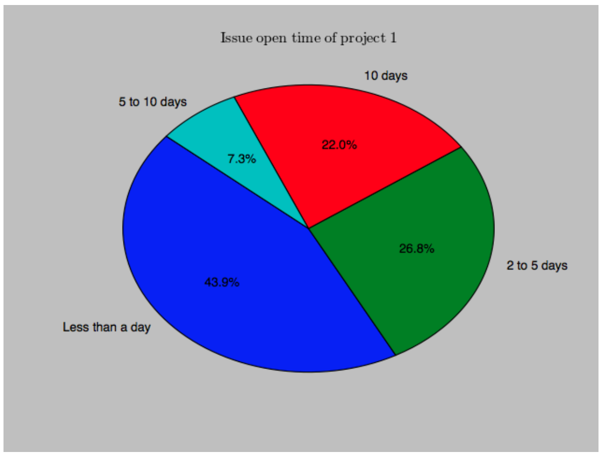


Figure 1: Issue open time period

2.3 Anonymizing the data

The data was then anonymised to make sure that the analysis remains altered or skewed. This includes removing all the details like names of the users, projects, etc. To perform this, we had a function that maintained the state of the already seen users and used a counter to replace their actual names with these numbers. The same logic was used for the project data as well. Thus, the data now contains only users named as user(number) and projects named project(number).

3. FEATURE EXTRACTION

With the data that has been fetched for the projects, we then extracted fourteen different features that will help us in detecting the bad smells of software development. The following are the features that we extracted and their results shown in suitable visualizations. Each section explains what the feature is, how we extracted it, what the results were, visualizations and implications of the feature.

1. Issue open time period:

   The issue open time represents how long an issue was open. This is the difference between the time when the issue was created and the time when the issue was closed. The importance of this feature is that it tells us how long an issue was open and this directly gives us some idea on how effectively git was used. This also gives us an idea on how complex each issue was and how much work the team generally puts in a single issue.

2. Issues closed before milestone due:

   This data represents how early issues were closed before the deadline of the milestone the issue belongs to. For this, we first take each issue, take its closed time and the milestone it is under, compare that with the deadline of the milestone and figure out how early issues are generally closed before the due date of the milestone they are assigned to. This feature helps us see if the users assigned enough issues for a deadline and if they completed it before their deadlines.

3. Issues closed after milestone due:

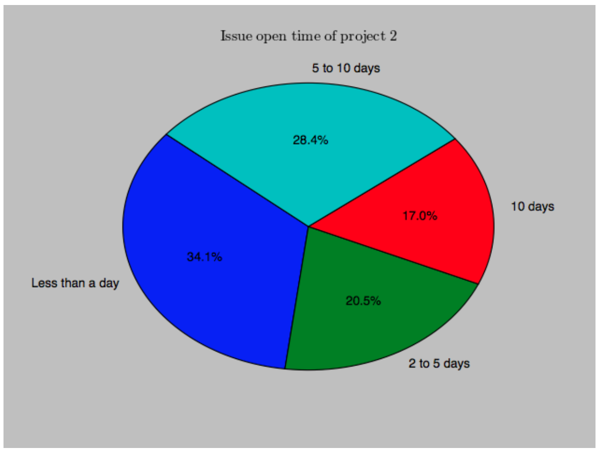


Figure 2: Issue open time period

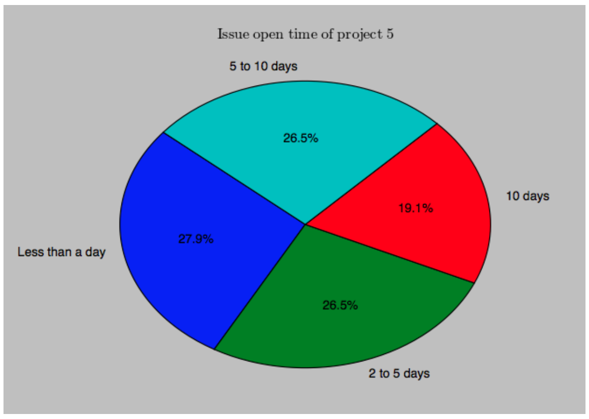


Figure 5: Issue open time period

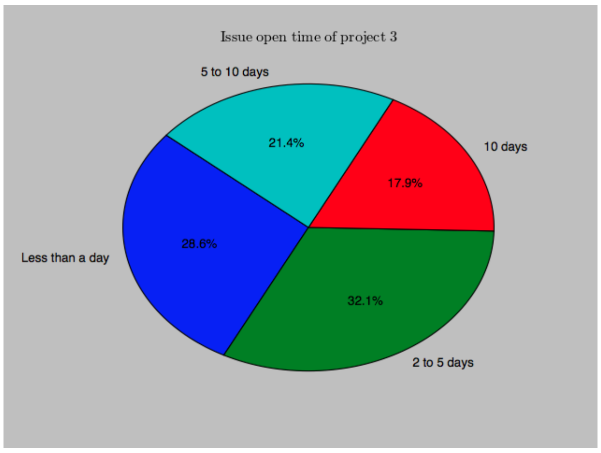


Figure 3: Issue open time period

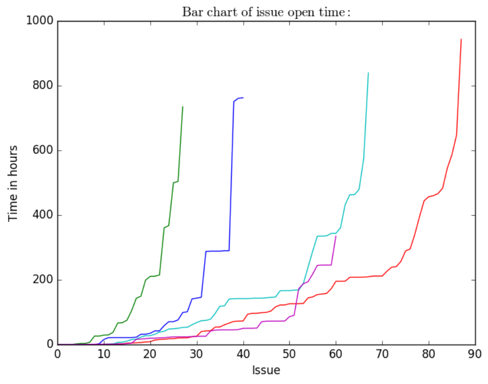


Figure 6: Issue open time period

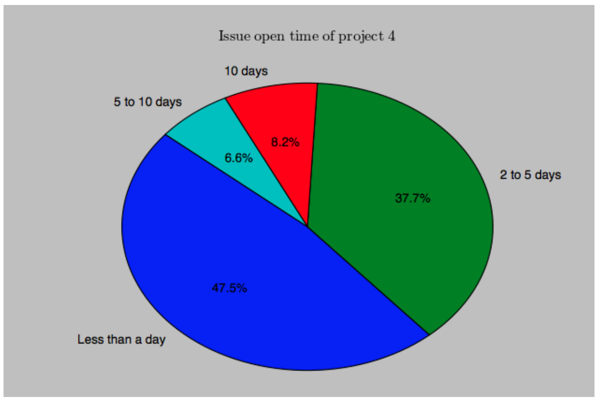


Figure 4: Issue open time period

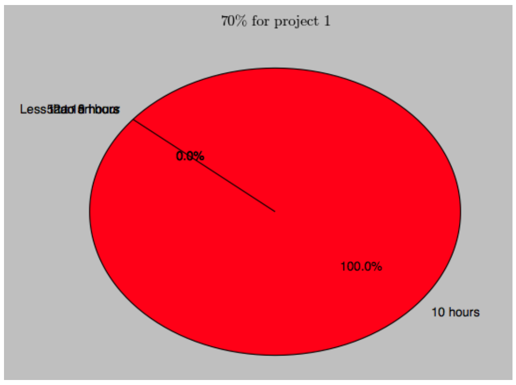


Figure 7: Issues closed before milestone due

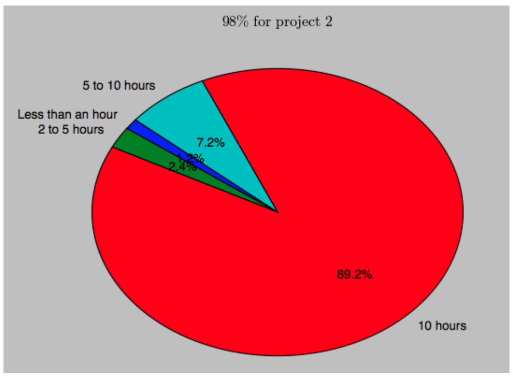


Figure 8: Issues closed before milestone due

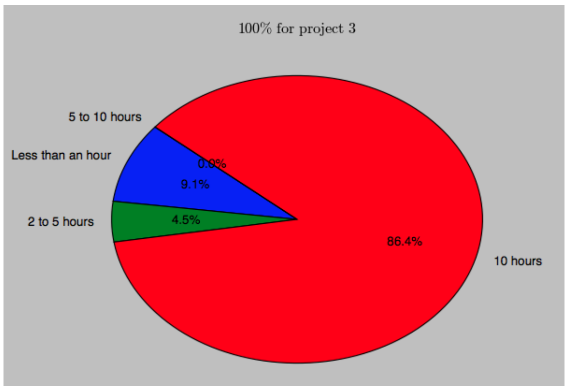


Figure 9: Issues closed before milestone due

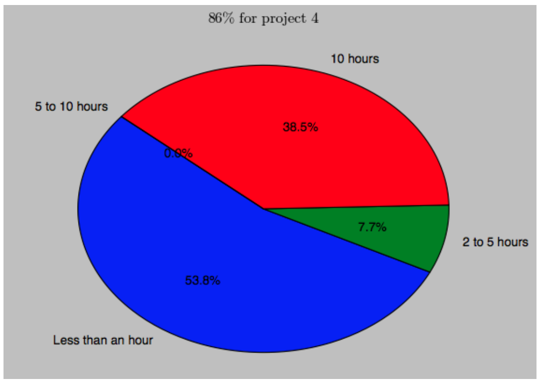


Figure 10: Issues closed before milestone due

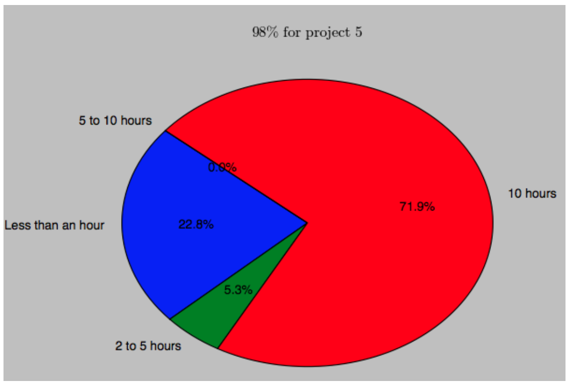


Figure 11: Issues closed before milestone due

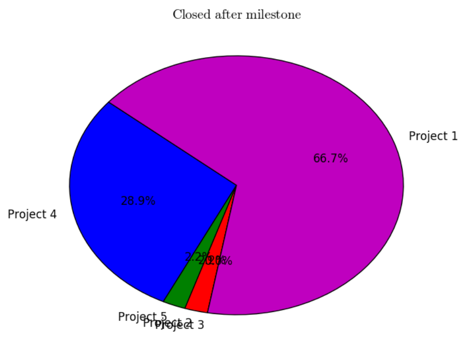


Figure 12: Issues closed after milestone due

This feature is very similar to the previous feature. It shows how many of the issued were closed after the deadline. So, if the issue was closed after the deadline this feature indicates it. The way this is implemented is by taking the percentage of issues closed after the deadline of the milestone each issue is assigned to and then comparing this percentage between the different projects. This feature shows the how properly the deadline was met and if the complexity and workload decided for a milestone was achievable or if it was ambitious.

4. Number of issues assigned to each user:

   This just shows the number of issues assigned to each user in the project. This is done by just count the number of issues assigned to each user. The main advantage of this feature is that it shows what percentage of the issues are handled by each user and so, it helps us see if any users are not contributing enough or if some user was contributing too much. This is one of the main metric that can be used if we want to check the individual contribution to the project.

5. Number of comments on an issue:

   This feature represents the number of comments the users have made on an issue. The way it is implemented is by just counting the number of comments per issue and using these values to check statistics like

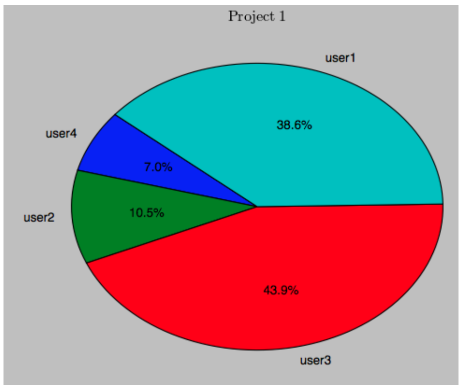


Figure 13: Number of issues assigned to each user

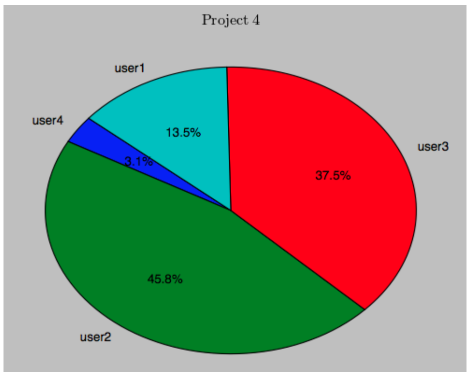


Figure 16: Number of issues assigned to each user

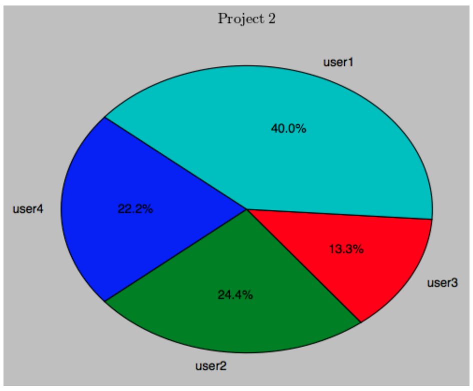


Figure 14: Number of issues assigned to each user

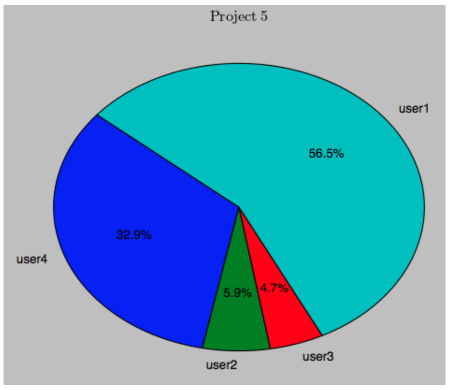


Figure 17: Number of issues assigned to each user

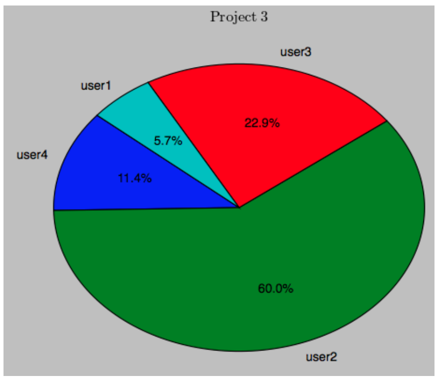
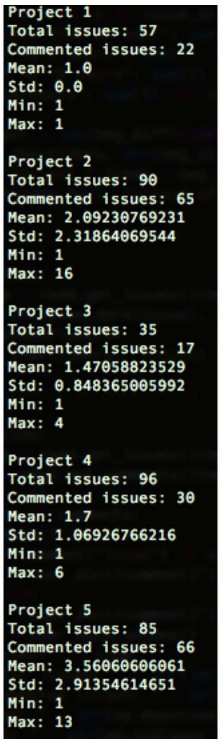


Figure 15: Number of issues assigned to each user



```
Project 1
Total issues: 57
Commented issues: 22
Mean: 1.0
Std: 0.0
Min: 1
Max: 1

Project 2
Total issues: 90
Commented issues: 65
Mean: 2.09230769231
Std: 2.31864069544
Min: 1
Max: 16

Project 3
Total issues: 35
Commented issues: 17
Mean: 1.47058823529
Std: 0.848365005992
Min: 1
Max: 4

Project 4
Total issues: 96
Commented issues: 30
Mean: 1.7
Std: 1.06926766216
Min: 1
Max: 6

Project 5
Total issues: 85
Commented issues: 66
Mean: 3.56060606061
Std: 2.91354614651
Min: 1
Max: 13
```

Figure 18: Number of comments on an issue

the mean and the standard deviation, minimum, maximum, total number of comments and how many issues had comments in the first place. This feature is very helpful in finding out how much the users were communicating among themselves regarding issues in github and also gives us an idea of how long the conversations generally were. So, this feature is useful in figuring out how the users communicated over issues on github.

6. Number of comments by each user:

   This feature shows the total number of comments each user made on the project totally. This feature is also implemented in a similar way by just taking a count on the number of issues each user made in the project and comparing them. This feature gives us an idea of how communicative each user was and how interested they were in discussing about the issues in their project

7. Commit timeline:

   This feature is a time series of all the points at which there was a commit made to the repository. The way it was implemented is by taking all the time points at which commits were made from the commit date and sorting them. This gives us a time series of all the commits made to the project. Then they are grouped based on hours or days letting us understand how many commits were made on a day or in an hour. This feature helps us understand the number of commits generally made on days and if the project received continuous commits or if the users spent only a fixed time committing to the repository.

   ‘

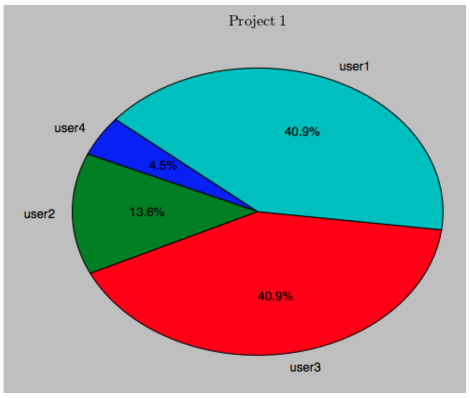


Figure 19: Number of comments by each user

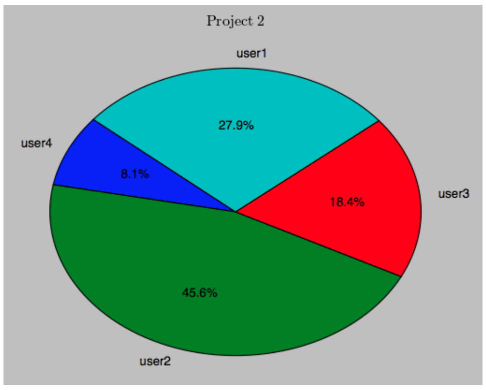


Figure 20: Number of comments by each user

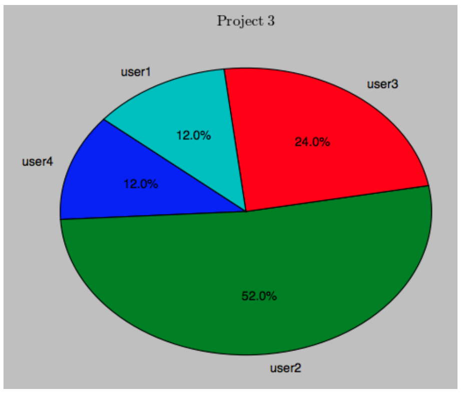


Figure 21: Number of comments by each user

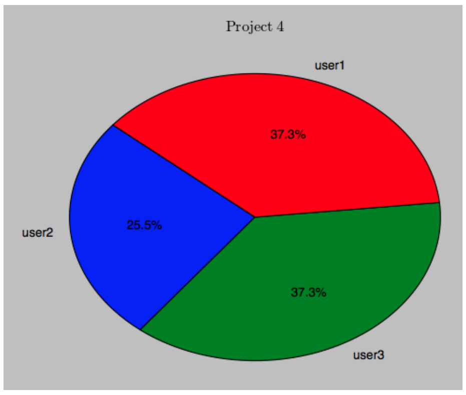


Figure 22: Number of comments by each user

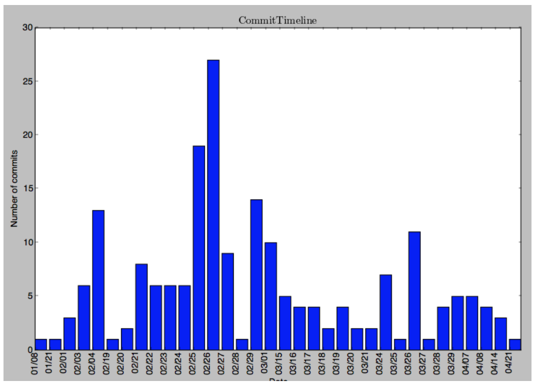


Figure 25: Commit timeline

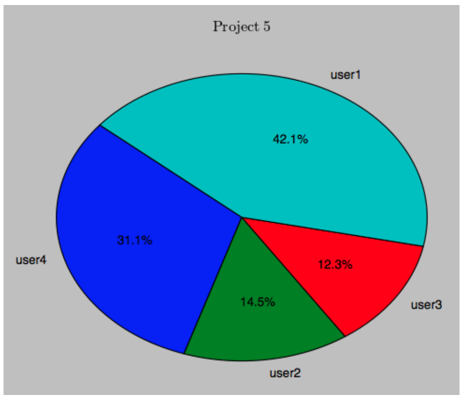


Figure 23: Number of comments by each user

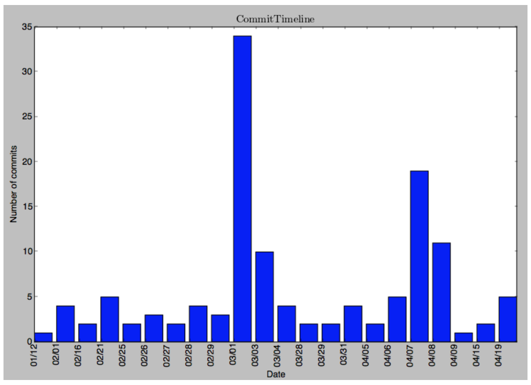


Figure 26: Commit timeline

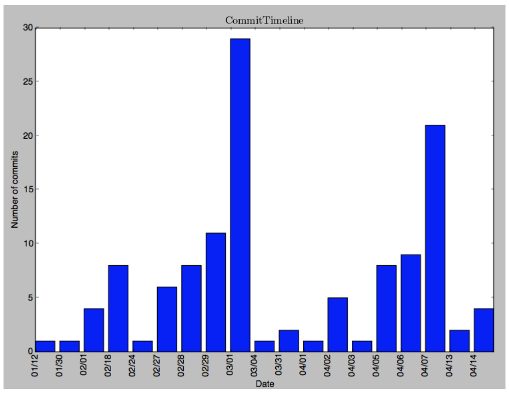


Figure 24: Commit timeline

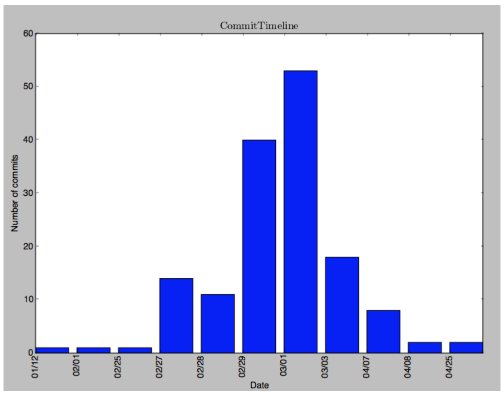


Figure 27: Commit timeline

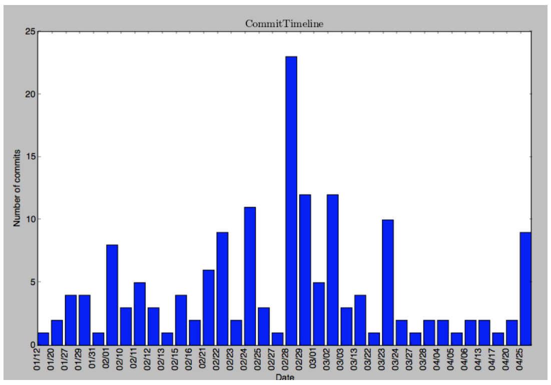


Figure 28: Commit timeline

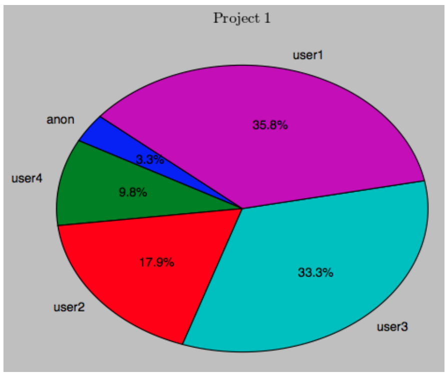


Figure 29: Number of commits per user

8. Number of commits per user:

   This feature indicates the total number of commits made by each user in their repository and the percentage of commits they contributed to the project. This is implemented by taking a count on the total number of commits each user made. This feature helps us identify how much each user contributed to the project. It helps us see whom among the users actually contributed to the project and if there were any user acting as an outlier.

9. Milestone's lifetime:

   This feature shows us the total lifetime of a milestone. This just represents how long the milestone was alive in the project. It is the difference between each milestone's created date and the due date. This gives us an idea of how long the milestones were planned to be and if they were short lived. It also gives us an idea of the proper usage of milestones feature.

10. Quickly closed issues:

    This feature shows us the metrics of issues that were closed too soon. It shows the total number of issues that were closed within an hour of creation and within a day of creation in each project. This is done by finding out the difference between the created time and the closed time first, then removing all the issues where the difference is more than a day and then bucketing these into the first hour and the first day. This shows if

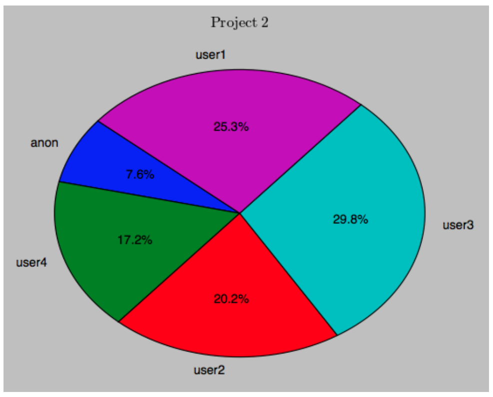


Figure 30: Number of commits per user

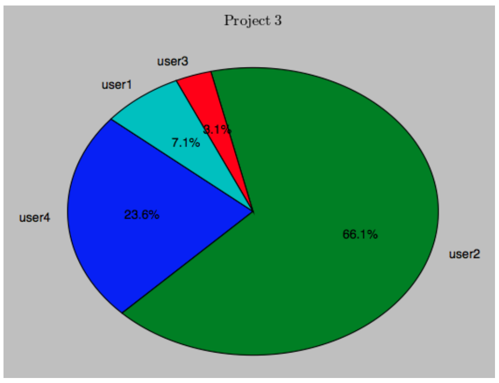


Figure 31: Number of commits per user

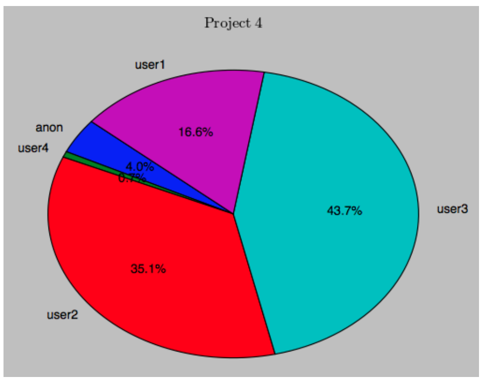


Figure 32: Number of commits per user

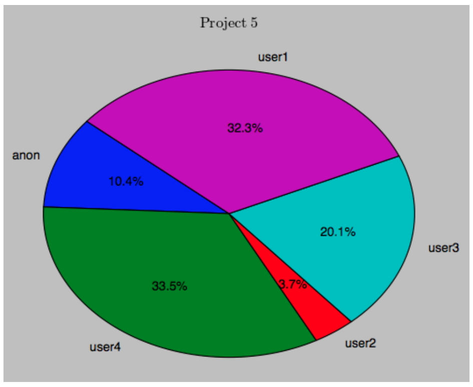


Figure 33: Number of commits per user

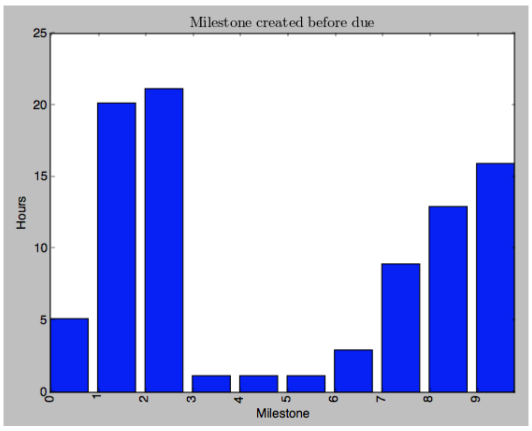


Figure 36: Milestone's lifetime

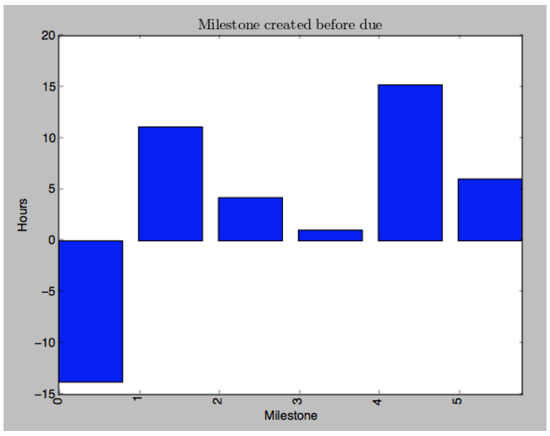


Figure 34: Milestone's lifetime

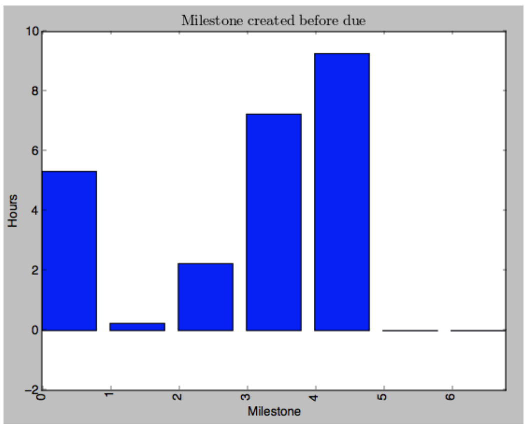


Figure 37: Milestone's lifetime

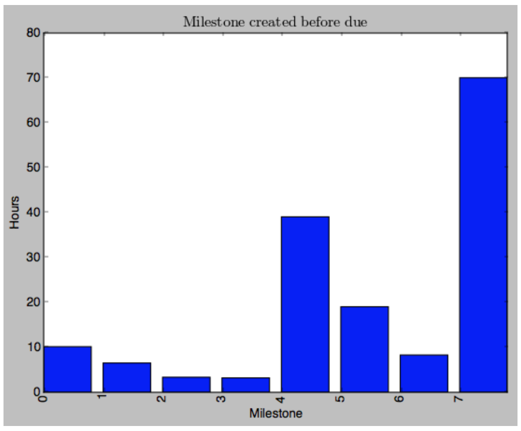


Figure 35: Milestone's lifetime

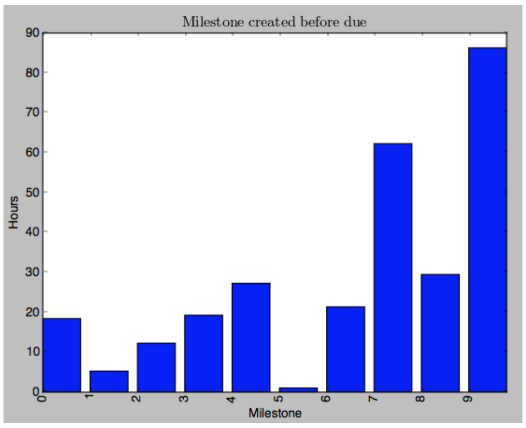


Figure 38: Milestone's lifetime

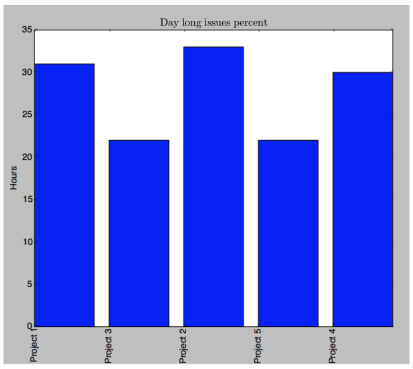


Figure 39: Quickly closed issues

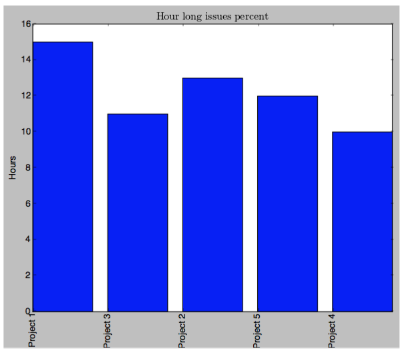


Figure 40: Quickly closed issues

the issues were actually us ed properly or if they were just issues to show work that was done earlier.

11. Number of issues reopened:

    This shows the number of issues reopened in each project. This feature is implemented by just taking a count of all the issues were reopening was one of the actions performed. This shows us if the issues were looked after properly and if the users used git properly by re-opening issues rather than creating new ones for fixes in existing issues.

12. Time gap between successive issue close times:

    This feature takes all the close times of issues in a project, sorts them in increasing order and calculates the time gap between successive issues. This indicates the time interval between the closing times of two continuous issues. This is implemented by finding the difference between closing times of issues. This feature is useful is seeing is the distribution was normal or if too many issues were closed next to each other showing a very bad practice or improper usage of git.

13. Issues with no milestones:

    This features shows us issues that do not have any milestones attached to them. This is implemented by counting all the issues that do not have any milestone assigned to them. This shows us the improper usage of milestones and if the issues were properly organised into milestones or not.

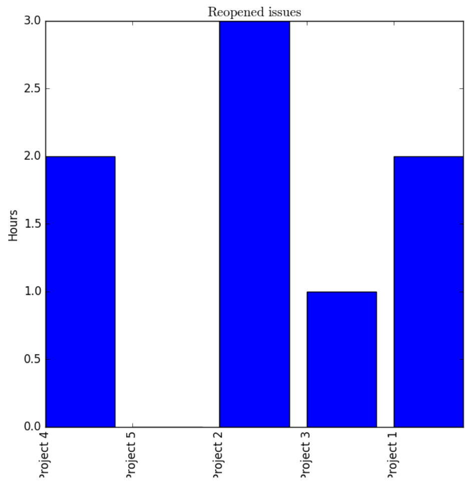


Figure 41: Number of issues reopened

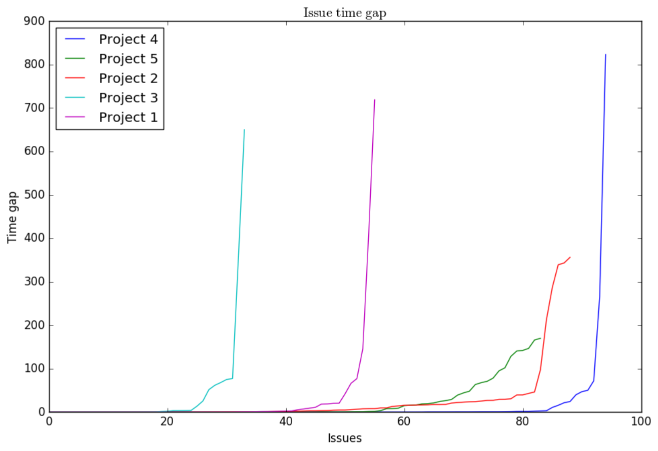


Figure 42: Time gap between successive issue close times

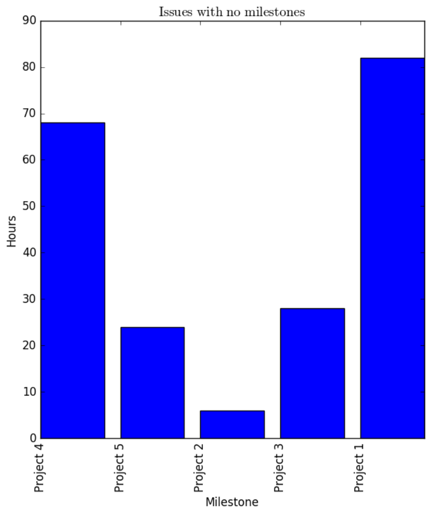


Figure 43: Time gap between successive issue close times

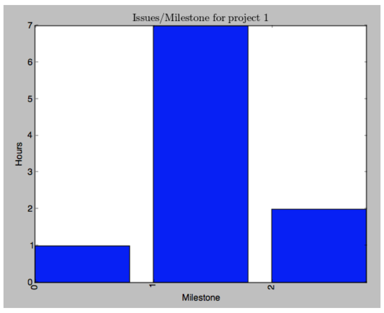


Figure 44: Number of issues per milestone

14. Number of issues per milestone:

    This feature is just the number of issues assigned to each milestone in a project. This is done by counting the number of issues assigned to each issue in the project. This shows how hard each milestone relatively was and if the milestones were actually useful.

4. BAD SMELL DETECTORS

With the features extracted, we came up with a set of bad smell detectors that show different bad smells from each of the corresponding features. The procedure to do this was to first rank each of the feature from 1 to 5, 1 being the best and 5 being the worst. This is done with different metrics for different features and always ranks the projects from best to bad smelling ones. Then, we assign a set of features to each of the given bad smell detectors and find the average of their scores to see how each project is in each of the bad smells. Finally, we take an average of all the bad smells and see the overall stink rate of each of the projects. The features contributing to each of these bad smells is given below.

1. Git not properly used

   This bad smell indicates that the features in git have not been used properly for the project development. The features include:

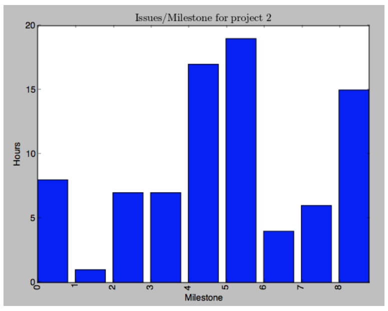


Figure 45: Number of issues per milestone

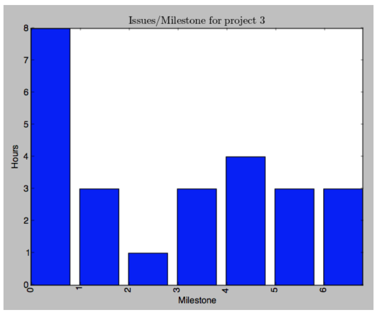


Figure 46: Number of issues per milestone

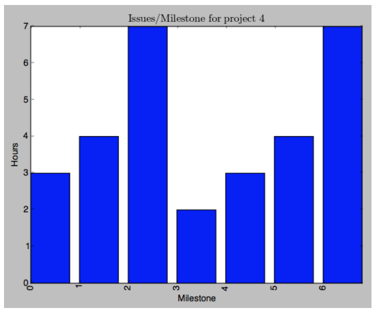


Figure 47: Number of issues per milestone

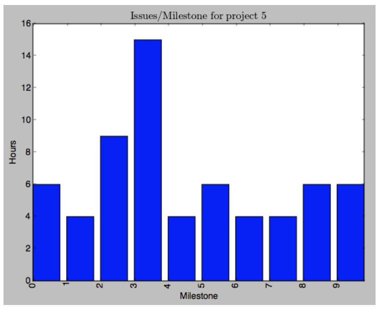
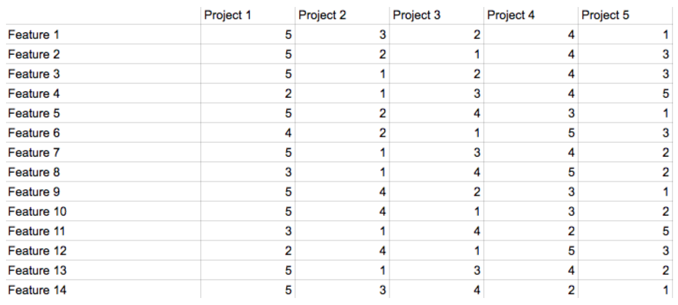


Figure 48: Number of issues per milestone



	Project 1	Project 2	Project 3	Project 4	Project 5
Feature 1	5	3	2	4	1
Feature 2	5	2	1	4	3
Feature 3	5	1	2	4	3
Feature 4	2	1	3	4	5
Feature 5	5	2	4	3	1
Feature 6	4	2	1	5	3
Feature 7	5	1	3	4	2
Feature 8	3	1	4	5	2
Feature 9	5	4	2	3	1
Feature 10	5	4	1	3	2
Feature 11	3	1	4	2	5
Feature 12	2	4	1	5	3
Feature 13	5	1	3	4	2
Feature 14	5	3	4	2	1

Figure 49: Bad Smell-Data

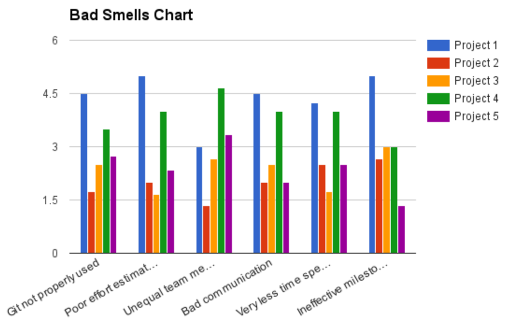


Figure 50: Bad smell

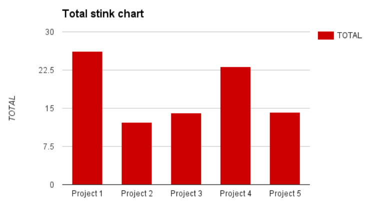


Figure 51: Stink rate chart

- Issue open time period
- Issues closed before milestone due
- Commit timeline
- Number of issues reopened

2. Poor effort estimation

   This bad smell shows that when the effort taken for a particular piece of work is decided, the estimate was not near the real time it took. The features for this bad smell are:

   - Issue open time period
   - Issues closed before milestone due
   - Issues closed after milestone due

3. Unequal team member contribution

   This bad smell shows that the contribution levels of the team members were not equal. Either a few team members contributed way too much to the project or a few team members contributed very less to the project. The features for this bad smell include:

   - Number of issues assigned to each user
   - Number of comments by each user
   - Number of commits per user

4. Bad communication

   This bad smell shows the communication between the team members regarding the project were not happening much or were not effective enough. The features for this bad smell are:

   - Number of comments on an issue
   - Number of comments by each user

5. Very less time spent on project

   This bad smell shows the amount of time spent on the project was very less and most of the work done were accomplished in a small time period. This also shows that the members did not focus much on things like meeting deadlines and closing issues properly, etc. The features that indicate this bad smell are:

   - Issues closed after milestone due
   - Commit timeline
   - Quickly closed issues
   - Time gap between successive issue close times

6. Ineffective milestone usage This shows that the milestone feature was not used properly. This includes characters like not enough milestones creates, not many issues assigned to a milestone, etc. The features that come under this bad smell are as follows:

   - Milestone's lifetime
   - Issues with no milestones
   - Number of issues per milestone

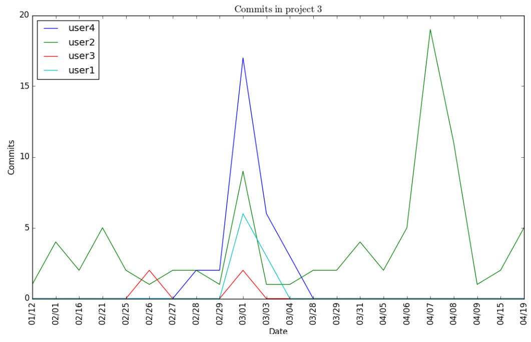


Figure 52: Early Warning Detection

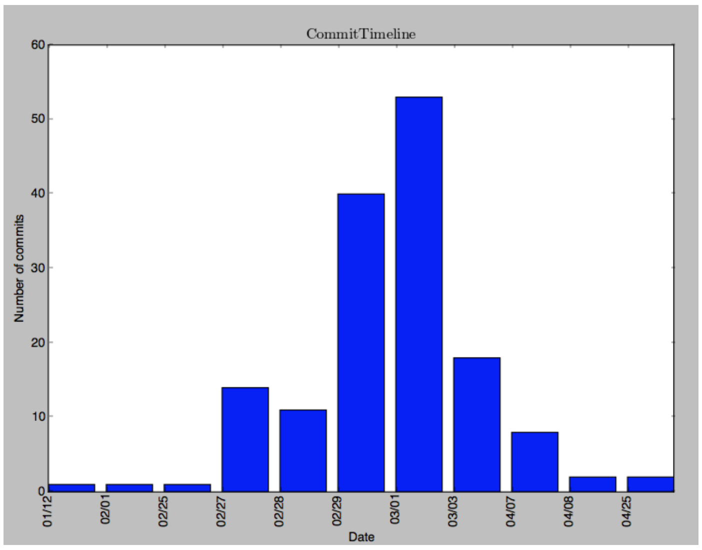


Figure 53: Early Warning Detection

5. EARLY WARNING DETECTION

After analysing our data and the time at which each of these bad smells occur, these are some of the early warnings that we detected from the data.

- If you look at the figure, its very evident that user2 from project3 had started and worked alone on the project for the first one and a half months. As he/she is the only person working, it is a early warning that he/she might contribute way too much or end up doing most of the work. This turns out to be true in the end because this user ended up having 60% issues assigned to them, 52% of the total comments, 66% of the total commits.
- In this figure, this team has made their first commit on 01/12, the next commit on 02/01 and the third on 02/25. These huge gaps show that the users were not using git properly and were just working without without committing their changes. This, in fact, is seen in the way they close their issues. In the issue closing time gap graph, it is seen that this project has closed around 86 issues with the time gap of an hour, meaning they have continuously closed all the issues. They also have a very good number of day long and hour long issues. All these show that they were not using git properly, as shown in the early warning.

6. CONCLUSION

Github is a very useful tool for product management and having it used in a project makes the software development cycle very productive. Having a whole class of students do their projects on github and using this data to analyse the software engineering practices has helped us out in understanding the problems and advantages of various techniques used in software product development. Processing github data and getting the features out of it to detect bad smells has given us better intuitions on what mistakes the project developers had made and how they can be fixed. Thus, the above work has helped us understand the mistakes that software developers generally do when using a project management tool and how to overcome them to be more effective and useful.