Section: Experimental Setup. Data collection and processing

1. **Data Collection:**

   Step 1: Select the latest rails issues. In this assignment, I focused on the latest issues that are related to the rails project. Furthermore, I selected the most recent issues because they imply fresh information. With recent issues, I can accurately diagnose the reported problems situation and analysis the activity of the developers involved in the studied issues.

   Step 2: Crawl issues data. I performed a crawler that is customized based on the official GitHub API to crawl data of the chosen 500 issues. For each investigated issue, I specifically gather the following information:

   a. **Issue metadata:** Basic information that sheds light on each issue's features (e.g., issue number, issue title, the number of comments, the number of labels, issue state, state reason, date of creating the issue, and date of closing the issue).
   b. **Issue labels data:** All labels were collected for each selected issue.
   c. **Issue user data:** For each issue, I gathered its user data which reported the issue.

   I collected recent 500 issues which roughly stretch over four months from Jul 30 until Sep 10. In particular, I collected 500 issues with 269 distinct users and 29 different labels (i.e., categories).

2. **Data Processing**

   I selected only the features that I will need in order to investigate the RQs findings and drop others. So, I dropped all features except just the following:

   i. **For issue data:** issue number, issue title, the number of comments, the number of labels, issue state, state reason, created at, closing at.
   ii. **For label data:** the name of the label
   iii. **For user data:** the login (i.e., user name).

   Then, I convert the type of columns (created at and closed at) into date data type.

Section: Experimental Results.

1. ***RQ1*. How do the number of issues evolve across time?**

   a. Motivation:

   Developers of software repo are usually interested to track how issues are distributed over time to detect correctly what factors cause the increase of issues. Analysis of the issues that evolve over time also helps developers to specify the times at which they are almost expected to report a lot of issues. This assists them easily to investigate the reasons behind these increases and the same for decreasing the issues.

b. **Approach:**

First, I calculated how many issues per day were reported using group by query. **Then**, I convert the number of issues column into a time series data type in order to adapt to drawing a time series plot.

c. **Findings:**

Ruby-on-Rails developers are precisely tracking the issues and addressing them as soon as possible. Looking at the top plot in Figure 1, I notice that there are a lot of peaks which means an increase in the number of issues, and a lot of bottoms which is a decrease in the number of the issues. However, the peaks don't take a long time and are soon going to decrease. This may be interpreted as the developers of Ruby-on-Rails being so active and fixing the bugs as soon as possible. When looking at the third part of Figure 1 which is the trend plot, The trend changes its direction and doesn’t mostly go up or down. This notice support that Ruby-on-Rails developers control effectively the increase of issues and fix them in good time manners.

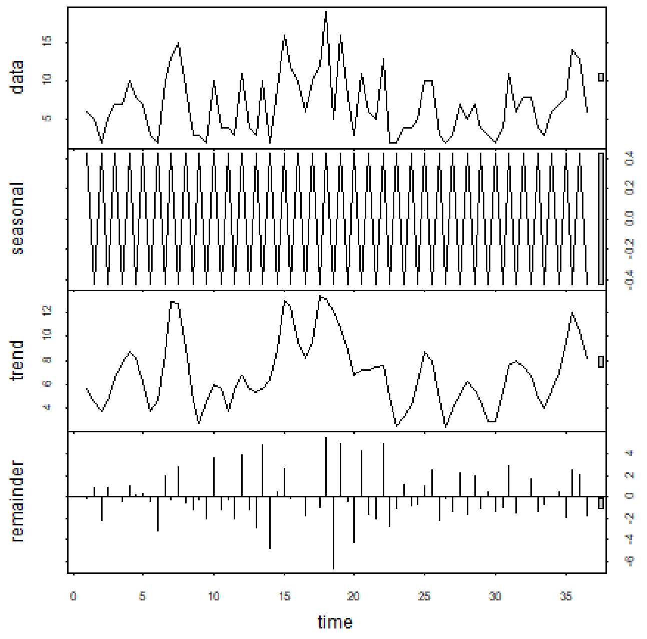

Figure 1

2. ***RQ2*. Are there any periods in which we get more issues?**

a. **Motivation:**

Software repo owners and developers usually have a tight time combining developing new features and fixing bugs and even tracking developers’ issues. They need to schedule adequately these tasks to utilize accurately the time. Studying the time periods that record more issues is useful to employ the appropriate time to track the issues and interact with them.

b. **Approach:**

At the beginning, I grouped the issues by creating date and count the number of issues for each day. **Then**, new column holds day number (i.e., the number of the day in the week ordered from one for Monday to seven for Sunday) were added and calculated from creating date column. **After that**, I used a correlation test between the day number and the number of issues to analyze the correlation. **Finally**, I summed the number of issues for each day number to show which days have most issues reported.

c. **Findings:**

There is a correlation between the weekday and the number of issues reported. The results obtained in this investigation revealed that there is a correlation between the day name and the number of issues reported on that day (p-value=0.01). This result indicates that developers may don't report issues directly the moment they face them. They probably accumulate the issues and then report them all in their free time. Looking at Figure 2, I observed that there is a density of issues that are backlogged near the weekends. The days in the Figure are written as numeric values ordered from number (1) which is Monday to number (7) which is Sunday. In the last 500 issues of rails, developers reported approximately 40% (200 out of 500) of issues at the ends of the weeks (i.e., Thursdays 91 issues and Fridays 109 issues). In addition, developers almost take the weekends into account and Lesly worked on the weekends. As noticed in Figure 2, Saturday and Sunday have the fewest number of issues reported.

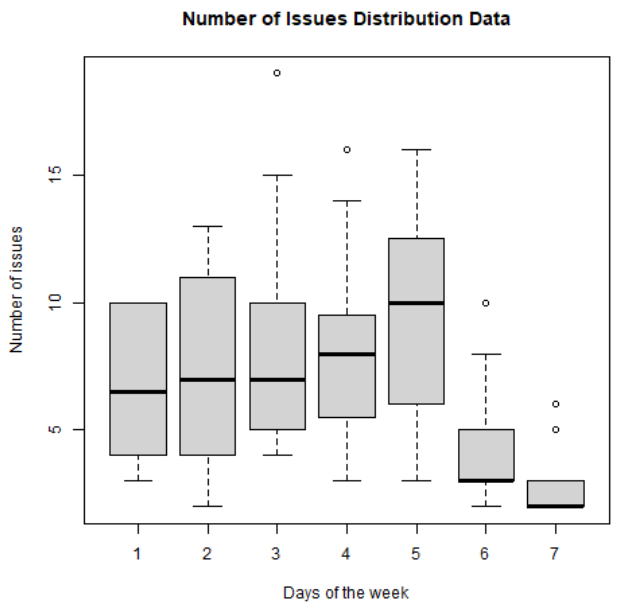

Figure 2

3. ***RQ3*. Is there anyone who reports more issues than others?**

   a. **Motivation:**

   Developers are often interested to report issues to the project repo that they used a lot of times in order to improve the experience of the project. Some of these developers are so active and report most of the issues he finds. Those developers sometimes hold inspired ideas or dangerous bug issues. Their excitement activity may be related to their strong interesting and good experience. Repositories owners should concern and take these developers' interests into account and track their reported issues and what mind to hire them especially if they have real and important ideas.

   b. **Approach:**

   Each issue often is reported by just one user. As a result, I grouped the issues by the user who reported and calculate the number of his issues. In order to calculate the number of distinct users, the minimum, and the maximum of number of issues. **Then**, I count how many users contributed how many accumulated totals of issues. In addition, I filtered only the issues reported by the user who has the maximum contributions to study and track his issues as individual. **After that**, I grouped his issues by issue state to analysis the situation of his issues. In order to calculate the age of his issues, I add new column that calculated from the time difference in days between creating the issue and its closing date. **Finally**, I counted how many issues that have zero comments and the average of the number of labels in each issue.

   c. **Findings:**

   The issues are distributed over users with an average equal to 12.07 issue per user and a median of 9 issue per user. About 269 users contributed to the last 500 rails' issues with a different accumulation of issues ranging from 1 to 28 issues/user. About half of users (**42.2%**) have just one contribution (i.e., reported one issue). The user which holds the username "**fatkodima**" achieved the maximum number of issues which is **5.6%** (28 out of 500) issues. **96%** (27 out of 28) of his issues have been closed. His issues often live in an average of 3.9 days and then going to close. In addition, **44%** (12 out of 27) of his closed issues have been closed on the same day they were created. **37%** (10 out of 27) of issues have been closed without any comment. So, this may mean he has a kind of confusing or blender and report issues that either the bugs are fixed but the state is not "fixed" and has no state reason, feedback is acted on however there are a few comments, or to show that work is not planned. For more details, I manually investigated his GitHub profile, I found that all of his repositories are forked and don't have any owned repository. Moreover, which supports this hypothesis is the average label count is very high which is 6.75 labels/issue he reported. For instance, this issue "*Update* *`rubocop-performance` and enable more performance-related cops*" reported by "**fatkodima**" and has 12 different labels. It seems that he reported unclear and unmanageable issue.

4. ***RQ4*. What is the most popular category (label)?**

   a. **Motivation:**

   It is common that each issue is involved in a specific topic or topics called labels. Those labels classify issues into groups in order to facilitate tracking the issues. Studying these categories contribute to knowing which topics are trend and developers complain about it a lot. This may be helpful to the owners of the repo to get more attention to these labels and in order to try to fix their bugs.

   b. **Approach:**

   For each issue there are many labels related to it. So, I collected all labels that related to studied issues. **First**, I counted how many issues in each label to show which label has the maximum number of issues and to know the number of distinct labels. **Second**, I selected only the issues that hold the label which achieve the maximum number of issues. **Finally**, I grouped those maximum label issues by the state of them to study their current situations.

   c. **Findings:**

   Almost the issues distributed in the categories with an average of 20 issues per category or label. There are several categories in the last 500 rails' issues which reach 29 categories. **31%** (9 out of 29) of categories have only one issue. This may belong to a few bugs reported related to these categories. Recently, the category "**activerecord**" is the trend, its related issues attained **35%** (i.e., 175 issues out of 500) of the studied issues. Active Record in Ruby-on-Rails is employed as a layer responsible for representing business data and logic. Active Record plays a role the same as the M in MVC - the model. Hot discussions about ***activerecord*** indicate that this layer may suffer from a lot of bugs. **33%** (58 out of 175) of ***activerecord*** issues are still open. However, the reasons which cause closing the closed issues are still unknown. As an illustration, **83%** (97 out of 117) of closed issues don't contain any information about why they have been closed. However, this category is the trend GitHub users didn't interact with it effectively. Only two comments/issue is the average of comments in this category. This also is another evidence of the bugs plenty in the repository and developers may find a difficulty to deal with them.