GithubVis

David Hedin, Zackary Rutfield

May 11, 2016

1 Motivation

The purpose of this project is to try to gather and show more information from Github than previously available. The currently available statistics that can be gathered from a repository focus mainly on number of contributors and number of commits. While this is certainly useful data, some more in depth analysis can be performed. If the type of commit can be classified, mainly to either a feature (addition) or bug fix, then this could reveal much more interesting information about a project or a language.

Github's built in visualizations focus on data for individual days. For example, it can show a visualization for the number of commits that occurred on an individual day or the number of additions and deletions in commits for a single day. We wanted to present the data for a repository as it changed over time.

Our target audience is primarily computer scientists, software engineers, and project managers. Anybody who uses a code repository and needs to keep track of its progress will find our visualization useful.

2 Related Work

Data mining of Github repositories has been done in [4]. In this paper, the authors presented a broad overview of different data analysis techniques that could be applied to Github repositories. Their focus was on identifying the most popular languages and language types, classifying project domain, and categorizing bugs and bug fixes. The categorization of commits as bug fixes or features was integral to our visualization. In order to classify commits as either bugs or features, the authors used a heuristic of checking to see if the commit message contained the words 'error', 'bug', 'fix' , 'issue', 'mistake', 'incorrect', 'fault', 'defect', or 'flaw'. Machine learning techniques are used to categorize bug types, but that is beyond the scope of our project.

We also looked at the data packaging done by [2] as a possible source of our data. They presented a database called GHTorrent, which contained approximately 4TB of compressed JSON files for the MongoDB setup and over 1.5 billion rows of metadata in a MySQL relational database. While this dataset promised speed over the REST API developed by Github, it lacked the flexibility required for our project. For example, we would need to rely on the database containing the information we want whereas the REST API is guaranteed to be up to date. In the end, we decided to use the REST API.

In [3], the authors analyzed the pitfalls associated with mining Github as well as the benefits provided by a dataset as large as Github. Key pitfalls that we made sure to avoid are inactive projects, low numbers of commits, and the use of some Github repositories as a distribution platform rather than a development repository. However, Github repositories can also contain large numbers of commits with well documented commit messages.

In [1], a visualization of Github issues is presented. The authors showed the connections between different issue labels using a graph with weighted edges and nodes. While we decided not to use this paper, it provided valuable insight into important metadata stored in Github.

3 Design Evolution

Our design started off as a bare bones line graph that showed open issues over time and commits as two lines representing commits that were bug fixes and commits that were features as a stacked graph. Our

original plan was to include information about programming languages used in the commits, but upon working with the API more we discovered that we couldn't get this data.

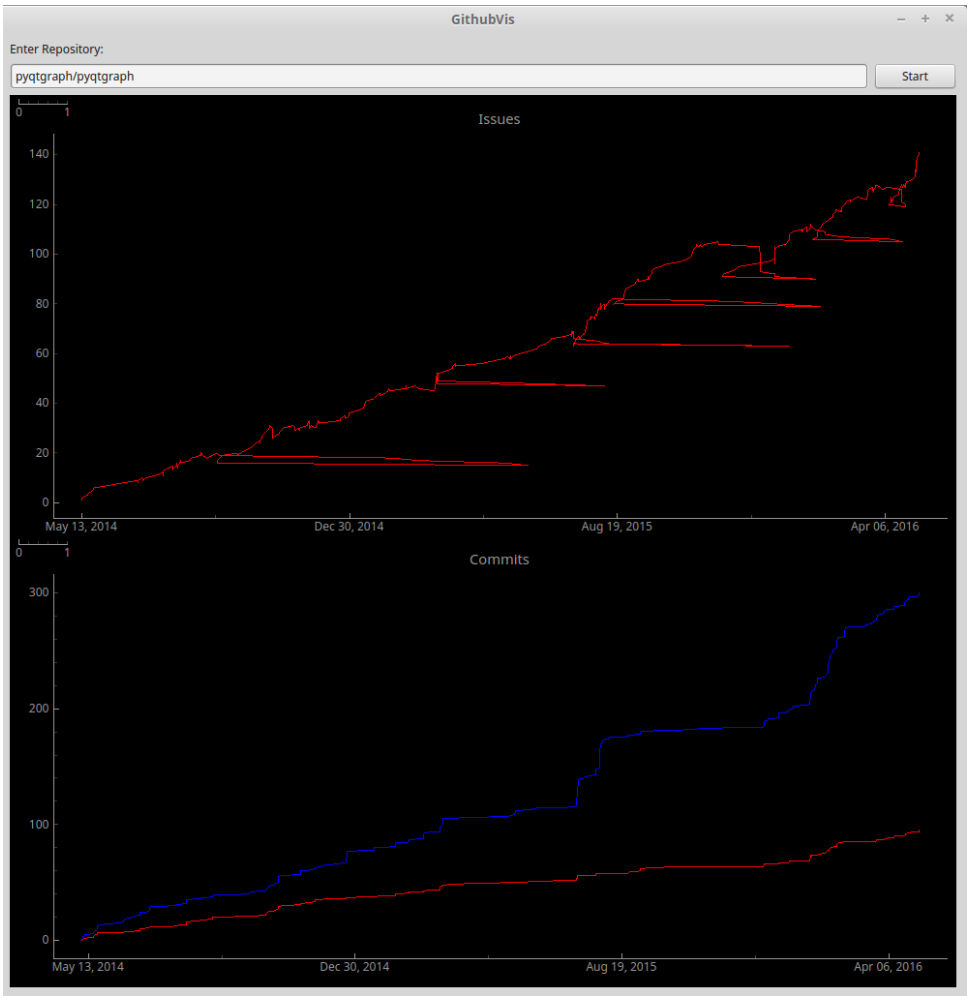

Figure 1: First working design revision of GithubVis.

One of the first immediate issues with our visualization was the fact that data was usually retrieved in reverse chronological order, but issue data contained closed issues that could be closed months or years after they were opened. This led to huge horizontal lines in the graph as they were plotted at the correct date but in the wrong order.

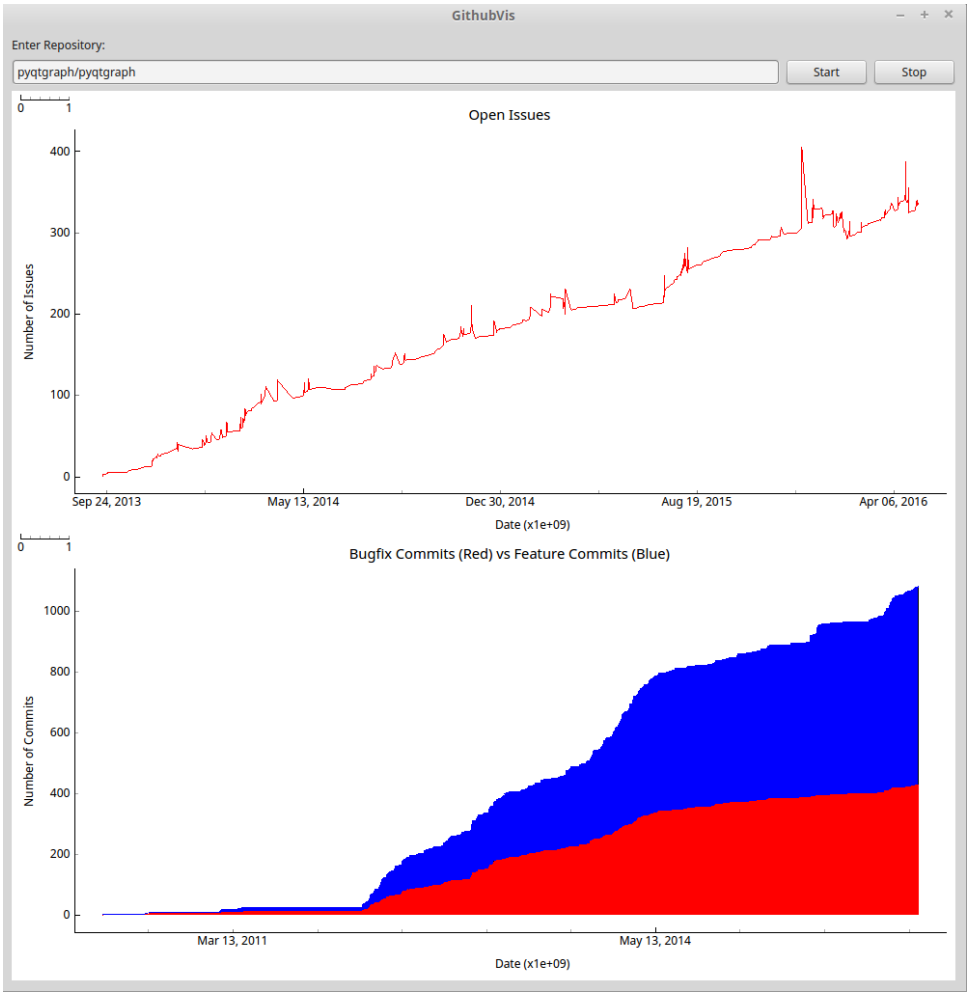

Figure 2: Second design revision of GithubVis.

The second revision of GithubVis was the version that we demoed in class. This version fixed the problem of improper handling of closed dates. Closed issues were treated as a negative one value, so that when the total number of issues was plotted, a closed issue would show a decrease in the number of open issues. For the plot of the number of commits, we decided to make it a filled and stacked graph, so that the total number of commits could be shown as the sum of the two lines plotted. This, however, confused some during the demo, so this was changed to be two independent lines instead. The scales were originally independent on either of our plots, but upon user feedback we changed them to be linked.

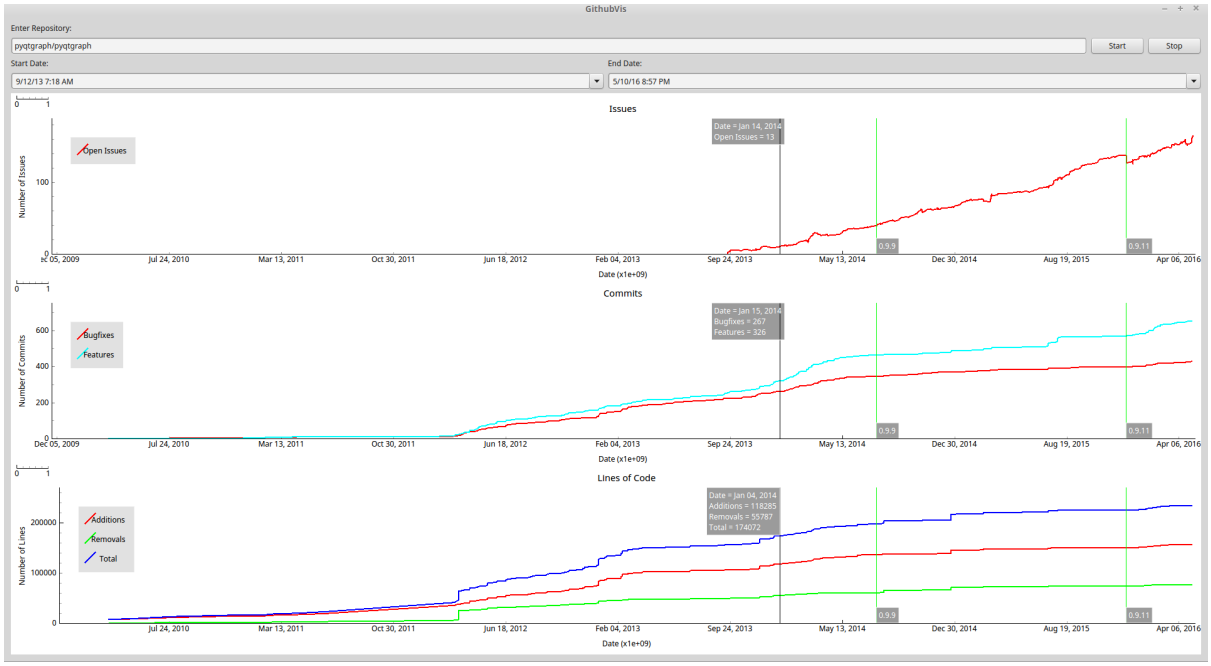

Figure 3: Current version of GithubVis.

In the final version of GithubVis, we added in legends for our plots, and also added in another plot for cumulative commit additions and deletions. We added milestone data retrieved from Github, and plotted them as vertical green lines on each graph, with a label containing the name of the milestone. We also added user interactivity besides panning and zooming by way of tracking the position of the mouse on any of the plots, and drawing a black vertical bar on each graph that displayed information at that date. Two date selectors were also added to allow the user to change the date range of the graphs to a range of interest. Colors were chosen automatically by the graphing library, in the interest of keeping the implementation flexible.

4 User Feedback

Our user feedback came directly from our target audience. Most of the students in class have some experience with git and Github in particular. Our classmates suggested a legend for our graphs, addition of a date range selection, a graph showing commit additions and deletions, showing milestones, and syncing the axes for each of our graphs for zooming and panning. We were able to implement these additions. Other feedback that we were not able to implement include a progress bar as the data gets pulled, showing the number of forks for a repository over time, and showing commits on forks as compared to the master repository. Some of these were not able to be implemented due to limitations of the API, such as implementing a progress bar. Others were not implemented due to time constraints, such as comparing forks to the main repository.

After the presentation, a suggestion to use git to clone the repository that is being looked at and use that to gather commit data was suggested. This would fix the problem of the speed at which the Github API serves commit data, but is not a perfect solution. If the repository is large, it may take a long time and large amount of space to store the repository, which will also contain data that is not relevant to the visualization, and is missing data specific to the Github API, such as issues data. It would also remove interactivity from the visualization, as the data would not be available until all of the repository is downloaded and parsed. It would be interesting to explore a combination of both methods as a way to speed up the total time to retrieve all of the data necessary for the visualization while maintaining interactivity.

If we were to continue development of our visualization moving forward, we would try to use more sophisticated methods to classify commits as bug fixes, showing more combinations of data and allowing the

user to customize the visualizations, caching our data, and implementing an auto-complete functionality to the search bar. These functionalities were explored, but ultimately cut due to time constraints.

5 Implementation

The front-end is written in PyQt, with the graphs utilizing the PyQtGraph library. The data collection was performed using PyGithub, which is a third-party wrapper to Github's REST API. Numpy and Scipy are used for some of the data processing performed during graphing. Data was stored as a list of parallel lists that contain the dates in the first list and the relative change associated with the date and the transaction type. On plotting, we took the cumulative sum of the relative changes. Currently the data structures do not contain any label, so plotting relies on knowing how the data is organized. Plots were made as a python object, however, so creating new plots is easy to do, and the visualization could be extended to have more than the initial three plots we created.

Storing the temporal data was an issue that we came across. When we processed the data in blocks, any issue that was opened in a different block than it was closed caused an issue upon plotting. At that point in development, we were calculating the cumulative sum as we pulled in new blocks, whereas in our final implementation we took the cumulative sum at plot time by leveraging the speed of NumPy's cumulative sum function.

The API had limited functionality, which caused issues. We couldn't get the total number of commits for a repository which caused us to abandon the idea of a progress bar. The API was also slow due to the HTTP requests, especially when trying to get commit additions and deletions. We also faced the limitations of 5,000 data requests per hour for authenticated users and 60 data requests per hour for non-authenticated users. While this may seem like a lot, it can easily be used when trying to pull data for larger repositories.

The code can be found at https://github.com/zrutfield/GithubVis

6 Team Contribution Breakdown

We worked evenly on both the presentation and all of our progress reports. We started out investigating the problem separately and then combining our discoveries into the final product. We utilized a pair programming methodology when debugging.

6.1 David Hedin

I mainly focused on creating the UI. This involved creating an object to plot time data using PyQtGraph, and being able to use these plots inside a PyQt window. The plot objects are able to pan and zoom, update when they receive new data, track cursor position and show the value of the data accordingly.

6.2 Zackary Rutfield

I focused on data manipulation and API integration. I worked on the objects that contain all of our data. I also worked on our initial implementation of multithreading, as well as working on implementing QThreads that allowed for the smoothness in data pulling.

References

[1] Canovas Izquierdo, Javier Luis and Cosentino, Valerio and Rolandi, Belén and Bergel, Alexandre and Cabot, Jordi. "GiLA: GitHub label analyzer" *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015.

[2] Gousios, Georgios and Spinellis, Diomidis. "GHTorrent: GitHub's data from a firehose" *Mining software repositories (msr), 2012 9th ieee working conference on*. IEEE, 2012.

[3] Kalliamvakou, Eirini and Gousios, Georgios and Blincoe, Kelly and Singer, Leif and German, Daniel M and Damian, Daniela. "The promises and perils of mining Github" *Proceedings of the 11th working conference on mining software repositories.* ACM, 2014.

[4] Ray, Baishakhi and Posnett, Daryl and Filkov, Vladimir and Devanbu, Premkumar. "A large scale study of programming languages and code quality in github" *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014.