Finding great software engineers with GitHub

Group #33

James Kunz (jkunz) Kenny Root (rootk)
Stanford University

December 10, 2013

Introduction

Measuring software engineering project success and developer efficiency is useful for individuals hobbyists looking to spend their time wisely and companies looking to spend their money wisely. But how do projects form? Which are the most important projects? How does a developer select projects? How productive or important is an engineer? What are good metrics for determining developer quality?

Open source repositories, where projects, engineers, and changes are readily open to inspection, provide an opportunity to study the process of software engineering in general and open source communities in particular. We will use GitHub event data to explore what might be learned.

Review of Relevant Work

The following work is of interest given the techniques employed in this paper.

PageRank Exploration of Graphs

PageRank[2] and Topic-Sensitive PageRank[5] were discussed in lecture. Concretely, they are methods to find prominent nodes in a graph. In the first case, we imagine a random walker following edges at random on a graph and measure her stationary distribution. To handle nodes with zero out-degree and other corner cases of graph topology, we periodically teleport her to a random node on the graph.

The formulation of Topic-Sensitive PageRank is very similar, except that we teleport the random walker only to a specific subset of graph nodes that are representative in some way of some property we are interested in finding elsewhere in the graph. In the classic formulation, this algorithm teleported to, for example, a manually curated set of medical webpages on the internet. The resulting stationary probability would be biased towards pages these pages linked to. In aggregate, medical pages would be given higher scores than other documents.

A more detailed explanation of our implementation follows.

Exploration of GitHub

A number of existing papers exist exploring the use of PageRank on a very limited subset of GitHub. For example [6]. In this paper, the authors directly apply PageRank to a tiny fraction of the GitHub collaboration graph. However, they simply run textbook/naïve PageRank without bringing to bear any problem-domain knowledge or intuition.

Data Collection

The GitHub Archive[1], `githubarchive.org`, maintains logs of significant actions on the Git repositories stored on GitHub. The Archive divides activity into log files, each containing an hour's worth of activity on the site. They maintain data back to February 11, 2011. The contents of these files consist of JSON objects appended one after the each. As a result, the file itself is not valid JSON. The full volume of data consists of approximately 190 GB representing over 120 million user actions.

Preprocessing

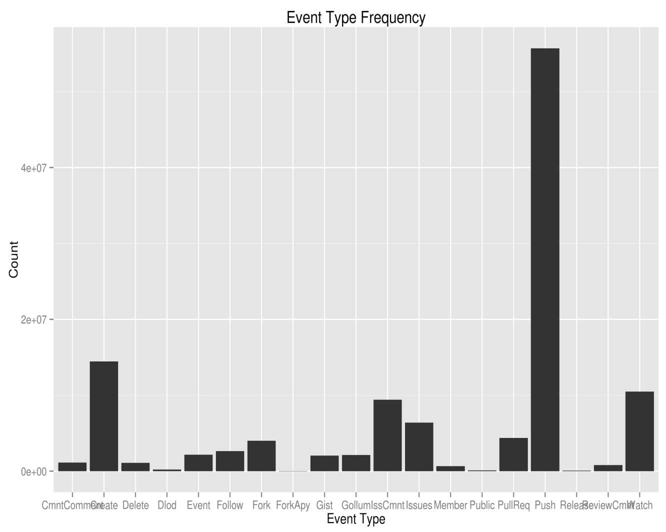
Like most large scale data, it required significant filtering and cleaning prior to use. In this dataset there were two significant problems.

First, the data format was difficult to parse reliably. In the majority of files extracted, a JSON object representing an action would appear one per line. Unfortunately for some periods of time, specifically March 10 - August 11, 2012, and again for a few hours on December 22, 2012 and September 19, 2013, the new lines were suppressed from the output. This resulted in the file consisting of a single huge line. Simply attempting to split the records on the sequence of character that usually delimited the boundary between records, `}{`, is not sufficient as this pattern often appears within quoted strings within records. These events consisted of approximately 20 GB of log volume (10.8%). In order to solve these problems, new lines were inserted at all candidate record boundaries then were removed greedily starting from the beginning of the file until the line successfully parsed as valid JSON. This process was repeated until the file was composed of valid JSON records, one per line.

Secondly, the logs format changes dramatically over the two and a half years of data. For example, information about the head of the repository on push events is not provided until October 2012. This data is key to understanding which pull requests are represented in a push event.

Graph Metrics

There were 117M events recorded in the logs across 4.2M unique repositories. They are distributed as shown in Figure 1. As can be seen, the main method through which users interact with repositories on GitHub is through Push events. Forking, issues and pull requests are also quite popular. As one might expect, the number of Push events a repository saw over the duration of the logs follow a power-law distribution. This can be seen in Figure 2.



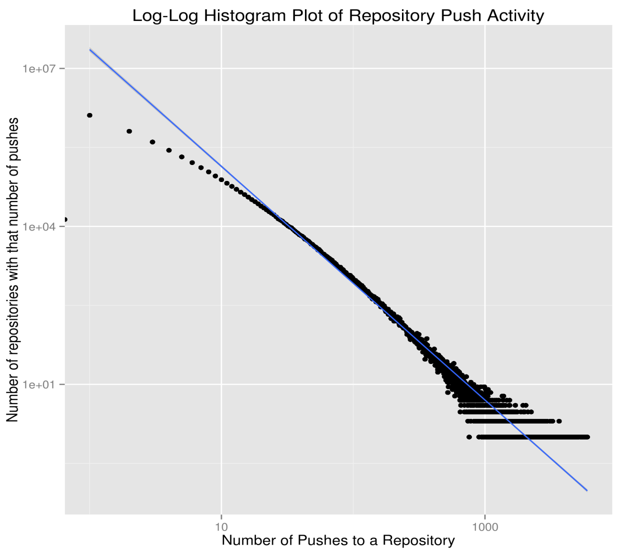


Figure 1: *Event frequency in GitHub logs*

Figure 2: *Log-Log plot of the number of repositories vs. pushes*

Algorithms Used

Converting JSON to a graph representation

We convert JSON to a graph representation through a MapReduce over the lines of the ungzipped log files. Each line is parsed as a JSON object and, if it is an action that represents an edge between a user and a repository, a key-value pair is emitted. With small modifications to this MapReduce other statistics, such as those presented in earlier sections, were generated. The key is the tuple `<user, repository>`. The value is a description of the type of connection. The reducer processes these key-value pairs, summing the number of times the user and repository were connected for each connection type. This data is useful for experimentation (eg. edge weighting) within PageRank.

Computing (Topic-Sensitive) PageRank

PageRank is simply the stationary probability of a random walker on a graph. Using a Pregel-like infrastructure, one way of computing PageRank is to:

1. Pick a node at random, dropping the random walker there.
2. With probability e select a node to teleport to at random. With probability $1-e$ select an outbound edge at random. In Topic-Sensitive PageRank, when teleporting, select a node from the topic set rather than from the graph as a whole.
3. Move to the new node.
4. Repeat to step 2 for a very large number of iterations.
5. Output the fraction of time spent on each node.

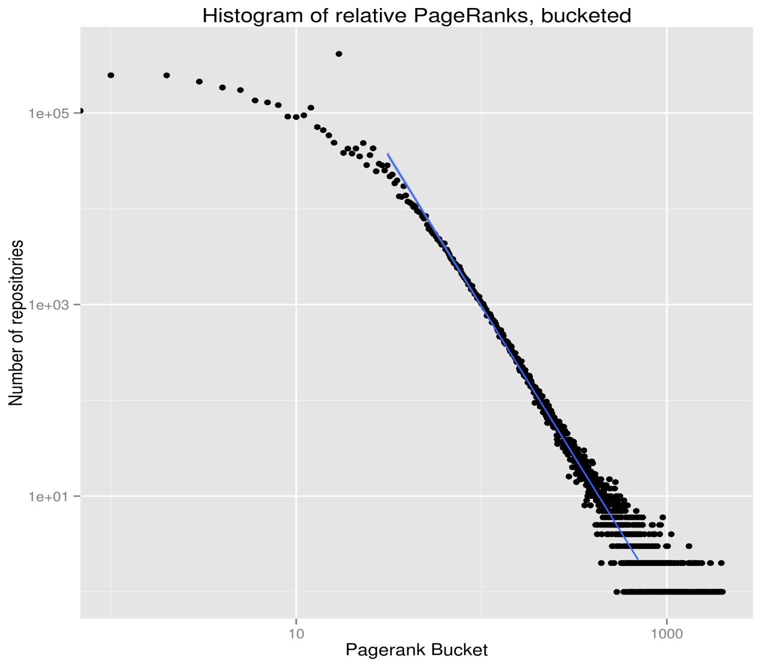
Since Pregel-like infrastructure performs a superstep for each batch of messages emitted, it is easier to randomly select all the nodes that will be teleported to ahead of time and run that many walkers concurrently on the graph.

Naïve PageRank Results

Building the graph such that there exists edges between developers and a projects weighted by the number of pull or push events and sorting by PageRank, we discover many well known projects. This formulation has a nice property that the fraction of PageRank flowing out of a repository node to an actor node is proportional to the fraction of commits to that project made by that actor. Conversely, the fraction of PageRank flowing from a repository node to an actor is proportional to the fraction of the actor's time they spend on that project (as measured by number of contributions).

Repository URL	**Relative PageRank**
https://github.com/CocoaPods/Specs	2509179.407214
https://github.com/mxcl/homebrew	1990023.300471
https://github.com/rails/rails	1237714.404296
https://github.com/mozilla-b2g/gaia	1067667.260020
https://github.com/saltstack/salt	887057.806753

The list continues with many other well known projects: Django, Mozilla Rust, Twitter Bootstrap, emberjs. Figure 3 shows a plot of relative PageRank across projects.



Naïve PageRank Analysis

One of the explicit goals of this paper was to experiment with various teleport sets. Our initial intuition was biasing the random walk by teleporting to a fixed set of hand-selected high quality actors or projects might aid in the identification of particularly important projects or people.

Before blindly applying this proposal, we decided to perform a study of where the Naive PageRank seemed to be performing really well, where it was failing, and where it is doing something interesting (though perhaps not desirable). This exercise was very instructive and the major results are discussed in the following sections. An improved formulation, informed by this analysis, is proposed below to alleviate some of the more obvious problems.

Projects

To a first order approximation, the naïve implementation of PageRank was pretty good at detecting important/impactful projects. As can be seen from the chart above, the projects are well known open source systems used in industry at major software engineering companies, startups and throughout academia. Not seeing anything particularly egregious on the project side, we turned our attention to the ranking of "actors".

Actors

Actors had a number of oddities. As we explored the data through many combinations of teleport sets, edge selection and edge weights, we discovered we could assign a broad variety of GitHub actors with high PageRank. Many classes of these actors were entirely unexpected. Some discussion of the dominant classes are below. Other potentially useful modifications to the Naive algorithm can be found in the section on Future Work.

Personal Repositories

One common occurrence among the top actors in the naïve set were users who had many personal repositories. The top such user had 1,900 repositories with a very small number (~2 on average) commits to each. For whatever reason, he decided to issue pull requests for each of these commits. To PageRank, this is mathematically the same as the link spam network discussed in 1(c) in Homework 4. As the random walker could teleport to any node in the graph, it had a relatively high probability to teleport to one of these repositories and then transition onto the user.

This reveals a fundamental property of GitHub: the nodes in the graph are not created equally. It is very cheap to create a new repo - merely a click of a button. However, it is quite expensive to create a developer: elementary school, high school and possibly even college! There is also only one sentient node: the actor. An actor makes many repos, but repos do not make actors. Repos with many actors can naturally be thought of as important whereas the inverse is not true.

Bots

Another among these unexpected users was a *bot*, or automaton, that automatically merges commits into either a small number of projects or a small number of commits across many projects. The bots are typically used as a bridge between GitHub and some other system. The two systems usually have a master-slave relationship with the bots acting to make sure they remain exact copies or *mirrors*.

An example of the bots that commit to a small number of projects “cm-gerrit.” This is a bot the CyanogenMod project uses to take reviewed and approved commits from the Gerrit code review instance and push those commits to GitHub. This allows GitHub to act as a mirror of the Gerrit instance which is considerably less performant than the GitHub infrastructure.

User “bitdeli-chef” is different type of bot that commits a single commit to several thousand projects. Bitdeli is a company that provides analytics for GitHub projects. The single commit is code that adds analytics data to a project’s *README* file. bitdeli-chef works in an interesting way that integrates well with the GitHub workflow: a project owner requests a Bitdeli Badge from the website, the badge is created as a Pull Request for the target GitHub project, and the owner of the project then merges that Pull Request.

Community Organizer

A completely unexpected type of non-bot user that appeared at the top of the PageRank list is the what we refer to as a *community organizer*. A community organizer may only do a bit of work themselves, but their major contributions are interactions with many other developers through comments, managing the merging of patches into projects they manage, and making releases for those projects. This type of person is valuable for employers to make large projects to succeed, but they are difficult to find.

An example of a community organizer is GitHub user “mmazur,” a maintainer of the PLD-Linux project. He appears to be an inactive user if one just looks at his GitHub page. The classical indications of popularity are absent: only has 2 followers, no *starred* GitHub projects, and follows no one. His *public contributions* list as GitHub calculates it is nearly devoid of any actions. However, when you look at his *public activity*, you can see he is constantly updating projects in the PLD-Linux project. In fact, his level of activity suggests that the work being done is either automated or semi-automated. Nonetheless, he appears to be the archetype of a community organizer.

The rest of the non-bot users are exactly what we expected when we first set out on this project. They are prolific contributors to large projects and they will be discussed in a subsequent section.

Failed Fix: Edge Reweighting

In response to the bot issue - where one user contributes heavily to a single repository, we experimented with variations on edge weight with the intuition: after a certain number of contributions, additional contributions aren't as informative. This approach seemed reasonably promising. Many bots fell off the top of the list; however, some really productive, senior people in the GitHub community also had their ranks significantly diminished. Generally these developers started a significant project and most of their commits are to that one project. On this metric, they and the bots are indistinguishable. Here is a glance at how the PageRank values were adjusted using this modification.

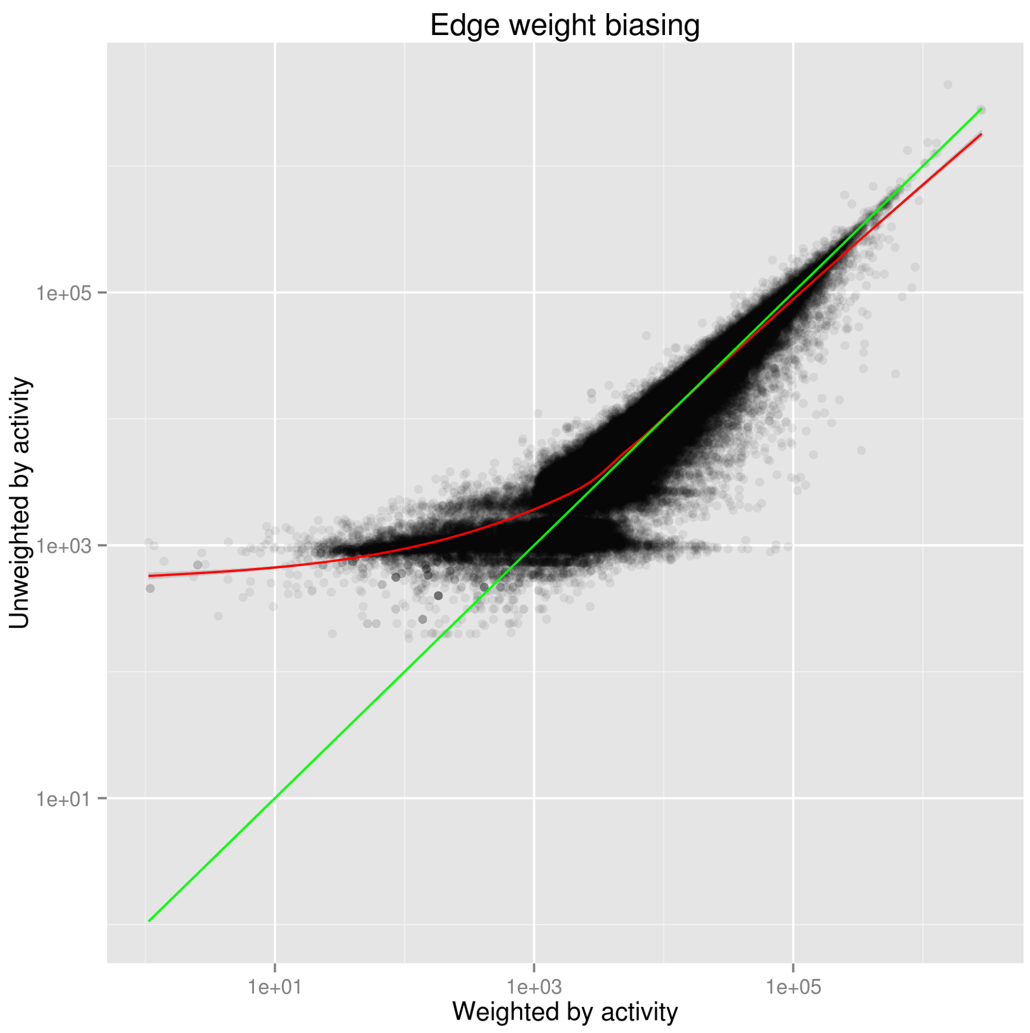


Figure 4: Distribution of PageRank scores by developer when weighted by activity and not. Among the top 100 or so actors, many bots lost significant rank, yet so did many important developers with singular focus.

Modified Formulation

Our final algorithm took the intuition developed above and reapplied it to the graph. To do so, we applied the following changes:

1. Remove actors that have never once done a pull request. This eliminates bots that simply push changes to repositories.
2. Restrict the teleport set to actors. This change prevents PageRank from identifying people with 1,000s of personal repositories as extremely productive and otherwise encapsulates the intuition discussed in the section on *Personal Repositories*, above.

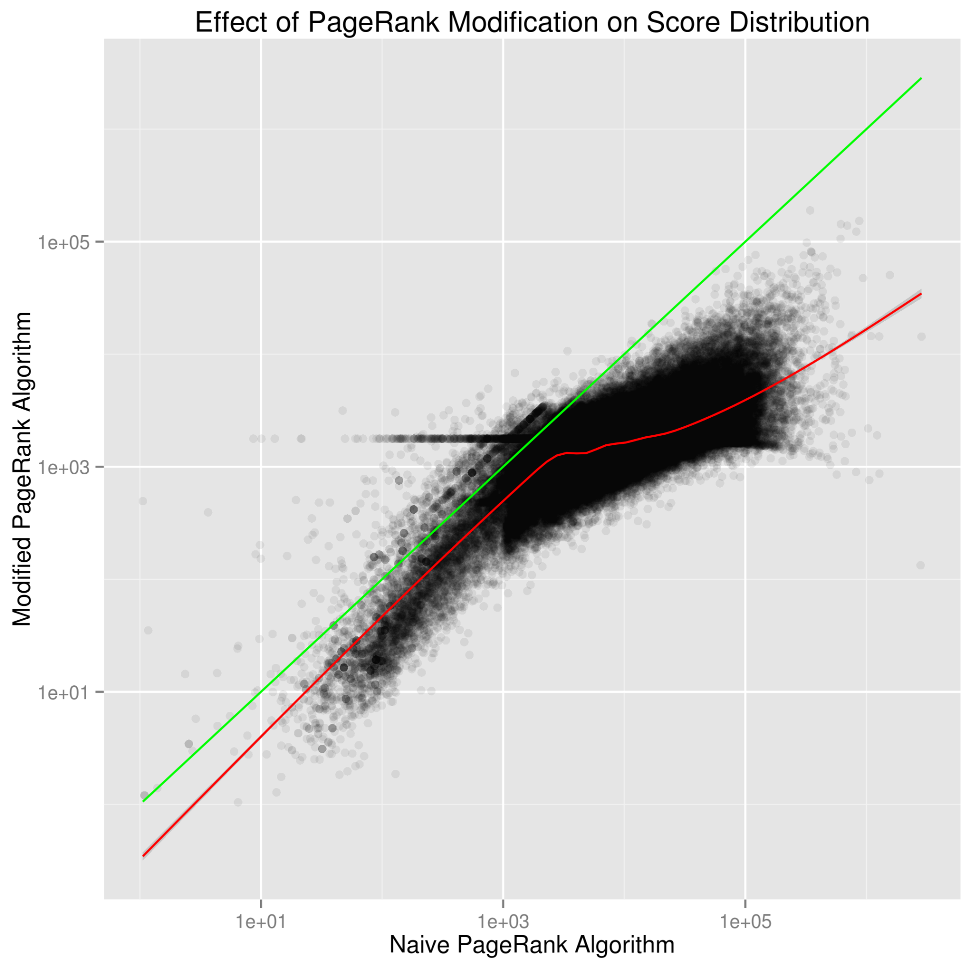


Figure 5: Plots of actor PageRank score for both Naive and Modified formulations. As can be seen in the plot, the data is pretty heavily correlated, but many actors of the types discussed above can be seen moving very far from the $y = x$ line (green). Overplotting, even with a very high point transparency, makes it difficult to see the shape of the mass of points. A spline (red) has been superimposed.

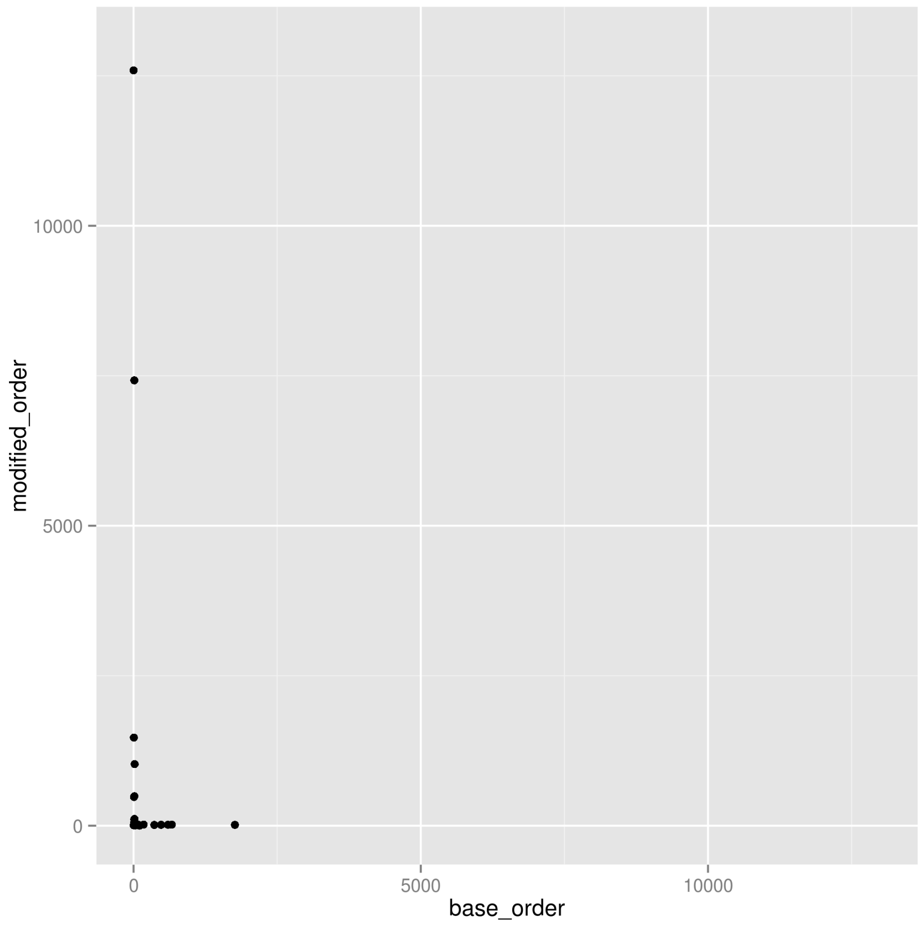


Figure 6: Rank order movement of top twenty actors. In this plot we can see those types of actors who were incorrectly ranked near the top (the four solitary dots along the positive y-axis) being significantly dropped down in the modified ordering while other, high quality actors, remain near the head of the list.

Modified Formulation Analysis: Top-Ranked Actors

The top-ranked users from our PageRank output show that this approach identifies valuable software development community members. All of them started and maintain a large role in a huge software development project.

However our PageRank is still susceptible to spam networks. The two maintainers of the Homebrew project are in the top-ranked users because they have used GitHub in an unorthodox way, essentially stealing credit from those doing the work. The two maintainers don't close *Pull Requests* and mark them as *Merged*. Instead they create entirely new commits in an apparent effort to make the Git history linear. This has the side effect of making it appear as though no one has had their Pull Request merged and is thereby considered "rejected." The maintainers, on the other hand, look incredibly productive.

There are three results that seem to not match what we're looking for: *githubteacher*, *githubtrainer*, and *githubstudent*. GitHub uses these three accounts for their namesakes in all their training classes to teach people how to use Git and GitHub features. As a result of its usage in classes, this account connects with many projects and many people whom are students in the classes thereby raising its PageRank. We decided these were legitimately high on the PageRank list since they were really used by humans and took actions that any normal developer would take.

A list of the top 15 developers according to our modified PageRank is presented here:

1. *githubteacher* – A shared account used to teach all the classes that GitHub offers.
2. fabpot – Creator of Symfony PHP framework
3. adamv – Mac OS Homebrew maintainer
4. taylorotwell – Creator of the Laravel PHP framework
5. Keithbsmiley – CocoaPods core developer
6. weierophinney – Lead developer of the Zend framework
7. mythz – Creator of servicestack.net
8. mpdehaan – Creator of Cobbler and Puppet developer
9. randx – Lead developer of GitLab
10. jacknagel – Mac OS Homebrew maintainer
11. *githubtrainer* – A shared account used with *githubteacher* above
12. *githubstudent* – A shared account used with *githubteacher* above
13. mattt – Lead developer at Heroku
14. thatch45 – CTO and co-founder of SaltStack
15. dumganhar – Primary contributor to Cocos2D-X game engine

Correlation with code durability

Another metric we computed for each developer was code durability. This metric was defined as the number of lines of code that remain in the repository divided by the number committed. To compute this number, we selected a random subset of Pull Requests and processed the Git repository data. Other reasonable formulations of this metric exist (e.g., maximum likelihood estimate of an exponential parameter determining the length of time a line tends to survive for a particular user – or at least assuming a beta prior so that a user with 3 lines of commit, 2 of which remain can be distinguished from a user with 2,000 lines of 3,000 remaining).

While sorting by this feature didn't result in quite the same immediate and dramatic successes as the PageRank approach, a null hypothesis test that durability is unrelated to $log(PR)$ has a *p*-value equal to 0.07. While not significant on the face of it, it does support that there is value in examining different forms of this metric.

Conclusions

In this project we replicated existing results in the literature that PageRank can do a reasonable job assigning scores to both projects and developers. We built upon that foundation by applying intuition gleaned from analyzing the failure modes the naive implementation. We experimented with the weights, teleport sets and edges in the graph and rigorously studied the resulting top 100 actors in each case. We were able to build intuition and understanding that we applied to resolve the most egregiously mis-scored actors.

These scores may prove useful to recruiters looking to hire the best and engineers looking to find projects with very high impact.

Future Work

Based on the observations and intuitions gleaned from this project, there are several approaches that might further yield improvement in the analysis of GitHub data.

Most of the PageRank spam networks were from bots that were not intentionally made to game the PageRank system, but their pattern had the same effect. Algorithmic detection of these bots would allow their pruning from the result set early.

Developers who initially contributed to large projects may be of more interest. Weighting earlier contributions as more important would bring those developers higher in PageRank. Care would need to be taken as applying it directly also has the negative effect of penalizing new maintainers of old projects. Time weighting contributions with the most recent contributions first would allow the system to avoid grandfathering old rankings.

More analysis into the worth of code durability, or how long contributed lines of source code are retained in the repository, would provide insight into the quality of a contribution. This could be further normalized by the amount of churn each project has in the average case. These normalized values could also be used as edge weights between a developer and a project.

A final aspect worth exploring is the construction of the graph itself. Inferring that a user that makes substantial contributions to an important project is important is reasonable. It should be difficult to claim credit for a successful project. On the other hand, it might be useful, where lacking sufficient pull/push evidence, to attach lightly-weighted edges from developers to the projects they are watching. If one of the top 15 developers listed above is watching a new project, that’s a pretty significant statement, even if those incredibly productive – yet busy – people haven’t had a chance to directly contribute.

Appendix

Background reading on tooling

Given the size of the log files involved, processing them on a single node with a single disk would be unbearably slow. Simply reading that much data from disk would take a substantial amount of time. MapReduce[3], first implemented at Google then later as an open source project Hadoop, is a tool to process large amounts of data in parallel across a number of compute nodes. It achieves concurrency by dividing a computation into three phases: map, shuffle and reduce. The map phase converts incoming records into key-value pairs. The shuffle phase collates all pairs with the same key. The reduce phase then processes sets of values belonging to a single key, emitting a final result for that key.

While MapReduce is a fantastic architecture for batch processing, many graph algorithms are cumbersome to implement using it. Specifically, many graph algorithms require propagating signals along edges. While MapReduce can perform this operation, a graph in the simplest list-of-edges form can only propagate signals over a single hop per MapReduce invocation. Pregel[4] is designed to implement these algorithms in a more elegant manner while retaining the distributed computation properties of MapReduce. It accomplishes this by considering super steps where each node has an opportunity to execute code on the present set of data before passing messages to other nodes in the graph. When all nodes vote to halt the computation, each node can emit an output value to disk. This framework, like MapReduce, was first implemented at Google and later rebuilt as an open source project Giraph.

Without these tools, the analysis below would take many days of compute time on a single machine.

Glossary

GitHub and Git itself has several terms that are used to describe actors and actions within the ecosystem. Listed below are some of the terms used to describe concepts in these two systems.

bot – software designed to perform tasks in an automated fashion; typically used to synchronize two disparate systems together.

commit – snapshot of a Git tree at a point in time.

Git – distributed revision control system for source code management. http://git-scm.org/

follow – used by GitHub users to mark another user of interest.

merge – when two trees in a Git project are joined together.

mirror – an exact copy of a software repository usually used to provide high availability.

README – the introductory text shown on the main page of a GitHub project.

star – used by GitHub users to mark a project of interest.

tree – listing of the files contained within a Git project and their associated data.

References

1. GitHub, Inc. "API v3." GitHub Developer. 16 Oct. 2013 <http://developer.github.com/v3/>.

2. Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry (1999) The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab.

3. Jeffrey Dean and Sanjay Ghemawat (2004), MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation.

4. Grzegorz Malewicz et. al (2010), Pregel: a system for large-scale graph processing. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Pages 135-146

5. Haveliwala, “Topic-Sensitive Pagerank”, Proceedings of the 11th international conference on World Wide Web (2002): 517-526

6. Thung, Ferdian; LO, David; and JIANG, Lingxiao, "Network Structure of Social Coding in GitHub" (2013). Research Collection School of Information Systems (Open Access). Paper 1687.