Page Rank on Github API to Sort Repositories and Contributors by Importance

Michelle Hwang† Scott Lee† Will Wang†
†EECS 126

1. INTRODUCTION

GitHub is the world's largest host of source code. As of April 2017, it has over 20 million users and 57 million repositories [2], making it an extremely valuable resource to the field of computer science. Some of the open source projects hosted on GitHub have brought about major revolutions to both industry and academia. The entire Linux kernel, for example, is hosted at `torvalds/linux`.

Users have the ability to "star" a repository if he or she finds it interesting. Stars can be used to ranked repositories in terms of popularity, and can also be used by GitHub to recommend other repositories that the user may like. Canonically, repositories are ranked by the number of stars it has. This has many benefits, as it is analogous in many ways to a democratic voting system. GitHub additionally ranks "trending" repositories by the the number of stars gained in the past day, week, or month.

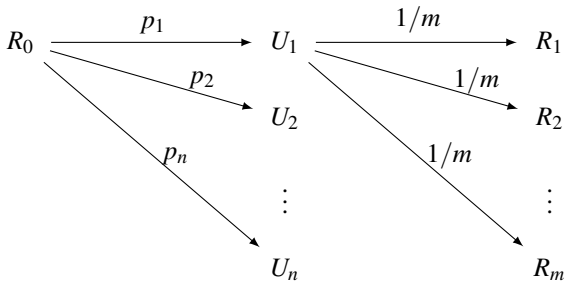
The number of stars however is sometimes uncorrelated with the actual impact that the repository has had. For example, the current repository with the most number of stars is `freeCodeCamp/freeCodeCamp` with 292k stars as of April 14, 2018 [6]. Since it has more than twice the number of stars as the second place repository, one might expect it to have been a highly influencial project. However this is not the case, as the main reason why `freeCodeCamp` as so many stars is because one of the first thing it instructors students to do is set up GitHub and star that repository. In turn, the number of stars no longer reflects the actual interest or impact of the repository, but rather the number of students it as attracted.

We propose an alternative way of ranking repositories that more accurately reflects the impact that the repository has made to computer science. Using the GitHub API [1], we were able to run a modified page rank algorithm to identify the repositories that top contributors to GitHub found interesting. In a way, we did not treat all stars equally like how a democratic election would work, but instead tried to only count the stars of people who work on or have worked on high profile projects. The end result was a ranking of the top repositories that more accurately represeted the high profile or highly influential projects that have shaped the field of modern computer science.

2. METHODS

We run a GitHub crawler that will visit users and repositories. A dictionary records many times we have visited a repository or user. We select a random repository from the top 1000 starred repositories in 2016 [5] and begin our crawler there. While on a repository, we assign each officially listed contributor i a probability $p_i = \frac{C(i)}{\sum_{i=1}^{n} C(i)}$ where $C(i)$ denotes the number of commits made by that contributor to the repository. Then, a random contributor is selected form the probabilities. We record this contributor, and increment his or her visit count. Then, from this user, we randomly select one of their starred repositories and increment that repository's visit count. We repeat the process of randomly selecting a contributor from this repository and so on. For each iteration (traversal to new repository or user), there is a small probability (denoted by `SPIDER_TRAP`) that we restart the process of visitations by selecting another repository from the top 1000 starred repositories and continue our crawling from there. An pseudo-code outline is shown in **Algorithm 1**.

2.1 Theory



First, we note that we represent GitHub as a two-layer, bipartite graph with one layer (or partition) being users U_i and the other repositories R_j. This is desired, because there is not a way to meaningfully directly link GitHub repositories to repositories or users to users (e.g. using GitHub follower/-following links is interesting but unlikely to say too much about the quality of a users' contributions). Using this, we determine that a repository is important if it has been starred by a user that we determine to be important. Important users are those to who contributes to a important repository and therefore the repositories that they star are also likely to be important. Using this cyclic model, our algorithm crawls con-

Algorithm 1 Crawler Algorithm

```
define SPIDER_TRAP, MAX_CYCLE, LAST_COUNT = .05, 3, 15
procedure CRAWL
    seen_users, seen_repos ← empty dictionaries
    while true do
        # Returns a random top 1000 repo
        curr_repo ← get_random_repo()
        while true do
            if random() < SPIDER_TRAP then
                break
            # Returns a random contributor of curr_repo, weighted by number of commits by that user
            curr_user ← get_random_contributor(curr_repo)
            if curr_user is NONE or seen curr_user MAX_CYCLE times in LAST_COUNT iterations then
                break
            seen_users[curr_user] += 1
            if random < SPIDER_TRAP then
                break
            # Returns a random repo stared by curr_user
            curr_repo ← get_random_starred_repo(curr_user)
            if curr_repo is NONE or seen curr_repo MAX_CYCLE times in LAST_COUNT iterations then
                break
            seen_repos[curr_repo] += 1
```

tinuously from an important repository to an important user and back to an important repository.

An additional note is that we also rank how important a user is to a repository by the proportion of commits they make to that repository. In search-engine PageRank, links are chosen uniformly at random, but that is because it is difficult for a crawler to determine which links are more "important" in a page. We assume that the more commits a user has for a repository, the more "important" they are to the repository, and denote this relation by transition with the probability p_i defined earlier. Lastly, we also note that it is possible to bias our results by only randomly selecting from the top 1000 repositories as starting points R_0: we rationalize this as we do not record randomly starting at a repository as a visit to that repository, and that these repositories are likely important so the users that contribute to them are also likely to be important and worth "visiting" according the the MCMC model described below.

Ideally, we would want to be able to model GitHub as a large transition matrix with each entry i, j representing the probability of traveling from node i to node j, and find the unique invariant distribution of the transition matrix: however, this is not possible because GitHub as a graph is not strongly connected (e.g. some users do not star any repositories). Therefore, we resort to manually crawling GitHub as defined above, which uses the idea of Markov chain Monte Carlo (MCMC) methods to model GitHub as a Markov Chain (in the same way as the graph we defined above) and sample from its "unique invariant distribution" by traversing the Markov Chain for a large amount of time steps. Ideally then, the proportion of times a node is sampled represents its overall proportion of the invariant distribution, and therefore importance.

We avoid pitfalls of this method using taxation to avoid spider traps - small strongly connected components that only link to themselves but not to other nodes [4]. For every iteration of the crawler, we have a small chance of jumping to a random repository node which would escape a potential cycle. In more obvious cycles (e.g. when a user only stars their own repositories that have few other contributors), we are able to detect them by noting we visited a user or repository multiple times in a short amount of time which is otherwise very unlikely. We also avoid dead ends by jumping to a random repository if we encounter one. Through all these methods, we should be able to properly rank repositories and users on overall importance and significance.

2.2 Experiment

We used one Amazon Web Service EC2 m4.large instance to run up to 5 instances of our Python GitHub crawler (defined by the pseudocode **Algorithm 1**, `crawler.py` in the repository) at a time. Concurrent crawlers were possible because visit information was stored in an Etcd [3] instance (see `store.py`) and `tmux` was used to allow them to run in the background. We ran the crawlers for 3 days and 18 hours from April 12 at 4:00 AM PST to April 15 at 10:00 PM PST with a total of over 270,000 total visits. We set the taxation probability (SPIDER_TRAP) to .05, and restarted the crawler if we had seen a repository or user 3 times (MAX_CYCLE) in the past 15 repositories or users (LAST_COUNT). The code we wrote is open source and can be found on Github.

3. RESULTS AND ANALYSIS

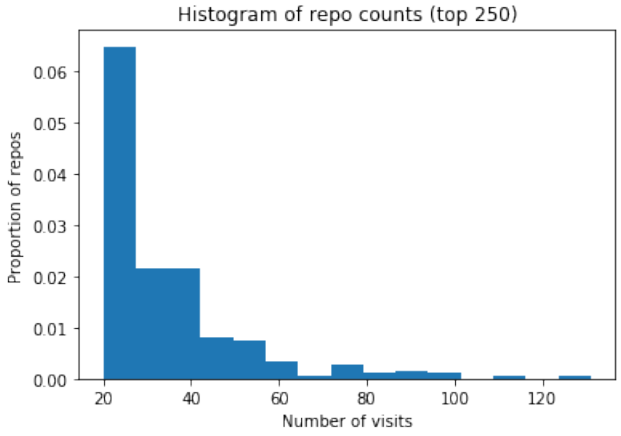
3.1 Data

To conduct our analysis, we use Python modules `numpy`, `pandas`, and `matplotlib` in a Jupyter Notebook. Our two main DataFrames (tables) of data, one for users' data and one for repo data, are as follows:

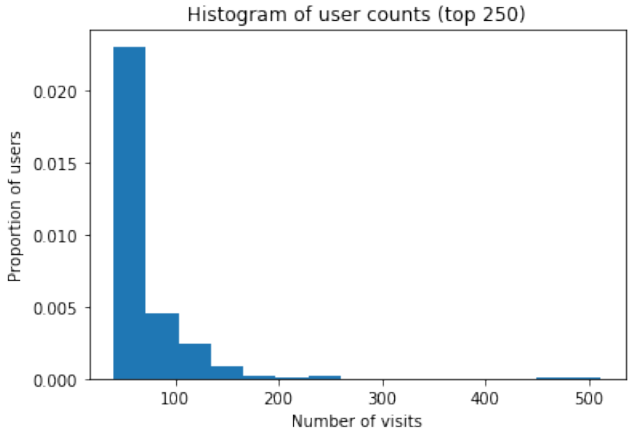
	user_id	visit_counts	login	followers	following	public_repos	contributions
0	170270	512	sindresorhus	24075.0	50.0	984.0	0.0
1	25254	454	tj	34773.0	47.0	273.0	2933.0
2	12631	248	substack	12157.0	224.0	928.0	68.0
3	66577	248	JakeWharton	45515.0	12.0	95.0	2654.0
4	810438	222	gaearon	27947.0	171.0	227.0	3155.0
5	499550	188	yyx990803	29984.0	90.0	143.0	2645.0
6	9287	167	isaacs	8129.0	2.0	373.0	1698.0
7	65632	162	antirez	9399.0	3.0	56.0	627.0
8	230541	158	mbostock	17633.0	13.0	58.0	1277.0
9	13041	155	rauchg	6223.0	703.0	102.0	3047.0

	repo	visit_counts	stargazers_count	language	forks_count
0	resume/resume.github.com	131	35703	JavaScript	940
1	twbs/bootstrap	110	123689	CSS	59001
2	facebook/react	98	93586	JavaScript	17630
3	nodejs/node	95	47642	JavaScript	10018
4	tensorflow/tensorflow	93	96424	C++	61369
5	Microsoft/vscode	92	47993	TypeScript	6460
6	nodejs/node-v0.x-archive	90	35920	None	7949
7	golang/go	84	40320	Go	5471
8	rust-lang/rust	83	27698	Rust	4854
9	facebook/react-native	78	62600	JavaScript	14229

We first generate basic statistics for users and repo data. The repo statistics for all repos are as follows: mean = 1.721; 25th, 50th, and 75th percentiles = $[1,1,2]$, max = 131. As we can see, using all of the repository data does not yield interesting results, since more than half of the visited repositories only have one visit. Therefore, we narrow down our data to the top 250 visited repositories. The statistics for the top 250 repositories are: mean = 34.69; 25th, 50th, and 75th percentiles = $[24,28,40]$; max = 131. The histogram of this data is displayed below.



For user data, the statistics for all users are as follows: mean = 2.74; 25th, 50th, 75th percentiles = $[1,1,2]$; max = 512. Similarly, using all of the users data is not meaningful, so we narrow down our data to the top 250 users. The statistics for the top 250 users are: mean = 70.16; 25th, 50th, and 75th percentiles = $[46,53,75]$; max = 512. The histogram of this data is displayed below.

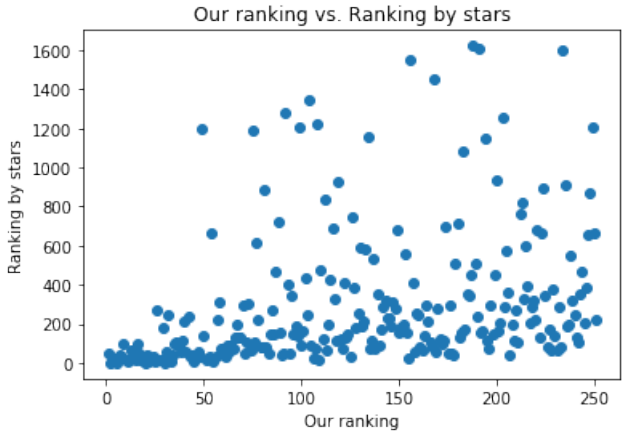


3.2 Ranking Comparison

Next, we compare our PageRank ranking of users and repositories to several different ranking methods.

3.2.1 Ranking Repositories

We compare our user ranking vs. a ranking by descending number of stars. We create a scatter plot of the top 250 repositories that compares our ranking versus the ranking by stars, as shown below. The correlation is $r \approx .344$, which indicates that there is a moderate positive correlation between our ranking and the ranking by stars.

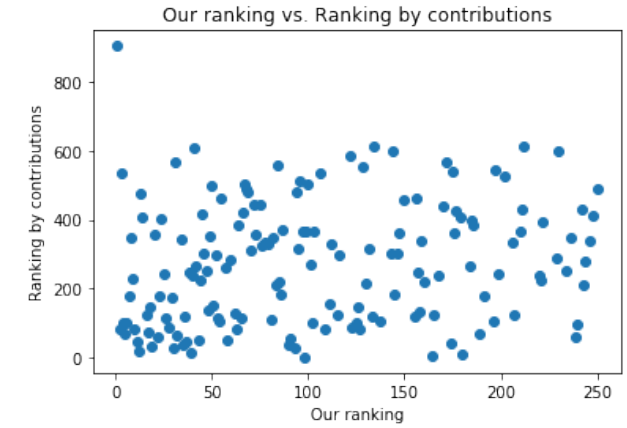


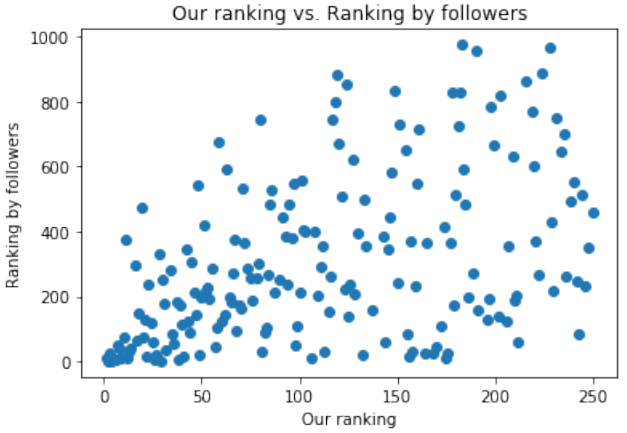
3.2.2 Ranking Users

We compare our user ranking vs. a ranking by descending number of contributions. (A *contribution* is defined as any PR, opened issue, or commit. We got the ranking by contributions from here.) We create a scatter plot of the top 250 users that compares our ranking vs. the ranking by contributions, as shown below (left). Note that we filter out users with zero

contributions. The correlation is $r \approx .175$, which indicates that there is a weak positive correlation between our ranking and the ranking by contributions.

We also compare our user ranking vs. a ranking by descending number of followers. We create a scatter plot of the top 250 users that compares our ranking vs. the ranking by followers, as shown below (right). The correlation is $r \approx .459$, which indicates that there is a moderate positive correlation between our ranking and the ranking by followers.





And just for fun, the top 10 languages in the top 250 analyzed repositories are:

language	Counts	Counts weighted by visits
JavaScript	89	3255
None	26	884
Python	19	593
C++	12	485
Go	13	472
TypeScript	7	310
C	9	289
Ruby	9	283
Shell	9	268
CSS	5	265

4. DISCUSSION

One notable outlier user from our data is user `sindresorhus`, our top-ranked user; the rankings by contributions and followers, respectively, are 904th and 8th. `sindresorhus` [7] is one of the most influential users on GitHub; he is a full-time open-sourcer, and is funded by numerous companies and users. Although he doesn't make a great deal of contributions, his work is clearly valued by many (as seen by the number of followers). The naive ranking by contributions fails to capture this.

A notable outlier from our repos is freeCodeCamp; our ranking puts it at 29th, while the ranking by star puts it at 1st. However, the reason why `freeCodeCamp/freeCodeCamp` had so many stars is because people who want to use the repo must star it; these people, who are likely learning to code for the first time, usually are not significant contributors on GitHub. Our PageRank ranking looks past this, and takes into account that users who star this repo are not as important. However, our PageRank algorithm does not avoid all the repositories that mandate starring for use: our top ranked repository `resume/resume.github.com` is a repository that generates a GitHub resume for users on the requirement that they star it. Users that are significant contributors to GitHub are more likely to star this repository because they would like to display their GitHub involvement (which increases our ranking of it) which violates our assumption that users star repositories because they think they are important.

Overall, our ranking was significantly better than ranking by contributions and stars, since these naive rankings are not able to take into account the *importance* of users (in terms of how significant their work is). The ranking by followers is more sophisticated, in that users with more followers are likely more important contributors on GitHub. There is a moderate correlation between our ranking and the ranking by contributors, which indicates that our ranking is fairly sophisticated (in that it takes into account the importance of users).

Future work in the area may revolve around limiting the crawler to only traverse to certain types of repositories, for example repositories mainly in the language Go. This could create a ranking of the most important projects of a certain language.

5. LIMITATIONS

We only ran our crawler for approximate four days coupled with having limited API tokens and rate-limiting. This led to the most visited repository only having 131 visits. Along the course of running the crawler, we observed the top repository shift between `facebook/react`, `twbs/bootstrap`, and `resume/resume.github.com` multiple times. This could be signifying that we did not run the crawler long enough to get a representative invariant distribution; while we can be decently certain the top repositories are important, we may not have a completely accurate ordering by our definition.

In addition, we did not have the time to test out differ-

ent taxation probability values, and settled at .05 because it seemed reasonable: large enough so we would break out of a cycle in an expected 20 steps, but small enough such that our visits wouldn't be heavily biased towards the top 1000 starred repositories that we pulled restart values from. We were unable to determine if pulling random repositories from the top 1000 starred repositories would significantly bias our results, and whether this bias would be appropriate.

Lastly, the probability values for transitioning from a repository to a user that we used was weighted solely based on commit numbers. It failed to take in account the quality of the commits (one-line changes vs. important features implemented) and other repository activities (opening bug reports or reviewing pull requests). It also neglected to include users who were not listed as official contributors, but may have made significant contributions to a repository through making forks and PRs.

6. REFERENCES

[1] Github developer: Rest api v3. https://developer.github.com/v3/, retrieved 04/14/2018.

[2] B. F. (bfire). Celebrating nine years of github with an anniversary sale, 2017.

[3] Etcd. https://coreos.com/etcd/ retrieved 04/12/2018.

[4] J. D. Jure Leskovec, Anand Rajaraman. *Mining of Massive Datasets. Chapter 5: Link Analysis*.

[5] minimaxir. Top 1000 github repos, by stars given in 2016. https://docs.google.com/spreadsheets/d/11bGpZq6ixlhrmQnzEUqbgbwTQwQVdtvILjp32vaOKBc/edit#gid=1735042899, retrieved 04/12/2018.

[6] Github repositories sorted by number of stars. https://github.com/search?q=stars:%3E1&s=stars&type=Repositories, retrieved 04/14/2018.

[7] sindresorhus. https://sindresorhus.com retrieved 04/17/2018.