Domain based Classification of GitHub repositories

Abhishek Sharma (*Author*)
01FB14ECS008
Computer Science dept.
PES university
abhishek.protea.sharma@gmail.com

Andrew Robert (*Author*)
01FB14ECS026
Computer Science dept.
PES university
randrew.1996@gmail.com

Apaar Gupta (*Author*)
01FB14ECS040
Computer Science dept.
PES university
apaar.gupta.13@gmail.com

Gowri Srinivasa (*Guide*)
Professor
Computer Science dept.
PES university
gsrinivasa@pes.edu

I. Introduction - *An abstract Summary*

We attempted the clustering of the repositories by domain, by considering various parameters related to them, mainly focused around user interaction patterns. We first filtered the dataset to a much smaller time frame and retained only that part of the data which were significant and were useful for consideration to achieve the clustering. We then ran clustering on this dataset on two levels, first with a very specific parameter and then attempted to recluster and further merge the clusters by using a more generic parameter. Following which, we tagged the clusters based on the predominant domain.

To check how right we were, we manually tagged a smaller random subset of repositories and then cross verified which cluster they belonged too.

On viewing, we realized that we had achieved promising results and this approach can definitely be used as the framework for more specific modifications to achieve very good classification.

II. Introduction to context and necessity

Github is an online project hosting platform that is mainly used to store open source code repositories. Along with seamlessly integrating with Git, the popular distributed version control system, Github also provides support for various social interactions among users that resemble those offered by social networks. These include: the ability to star a repository to show interest in its content, watch a repository to receive periodic updates and forking a repository's contents to make modifications/fix bugs/add features, either to use for oneself (if the repo allows the same) or to have them accepted by the owners of the original repository.

This mechanism of forking, making additions/changes and then making a pull request to the original repository is known as the fork-and-pull model for collaboration. This model enables the building of graphs and networks resembling those of social networks and using them to make predictions and inferences.

Right now, there are various ways to filter repositories. This include refining based on language, keyword search, or number of times starred. But the domain of a repository is something very fundamental, and should be a parameter available to users. It is a difficult problem to solve though, considering how

diverse the domains can be and hence how difficult that categorization can be. However, making such a classification available would prove very useful and will have a myriad of applications, which include being used in recommender systems and for surveying / inferring trends among the community.

People generally looking at repositories will also benefit from this information since it gives them crucial information regarding what the project is about. Also, when a user wants to narrow down the search either to discover new repositories or to contribute to projects, a domain based systematization helps curtail the options.

III. Brief summary of Literature Survey

We had mentioned how a NLP based approach for tagging the repositories and thus the clusters is not feasible since looking at all the readmes will not work out for various reasons. But finally we had to use a script to extract the first few lines of the contents of all the readmes to look at them in order to tag the random subset so we could use them to check how correct our clustering was. Of course, we did this the manual way, with human intelligence and not with NLP.

Also, Our initial approach that we set out with wasn't Exactly what we ended up doing but we were definitely on the right track. We still did start out with a similarity metric that was derived from a parameter and then the clusters were further merged with a similarity metric of another parameter to end up finally with good clusters. Our approach for checking and definitively tagging the repos that need to be tagged has remained the same.

The full procedure has anyway been detailed, in the appropriate section.

IV. Problem Statement and Limitations

We want to end up with well defined clusters that can be used to definitively declare which domain repositories belong to.

But there are various limitation with the data set that we are working with.

- Firstly, the data is quite sparse. Especially the specific parameters were extremely sparse. So we had to account for that.
- We could never use plots effectively or use most of the common clustering algorithms since we didn't have the points, only a similarity metric that can be considered as the distance.
- Traditional graph clustering algorithms wouldn't work on this dataset as this comes under a special class of problems known as social network analysis, at least that's the conclusion we drew and worked with, after seeing not so promising results with traditional algorithms.
- We ended up with 5008 filtered repositories that we attempt to partition into disjoint clusters. Due to the nature of Github and the entire Web Development and JavaScript ecosystem, the repositories that we finally obtain are skewed largely towards being in that domain. This is evident after a single glance through the entire set of repositories.

V. About the Dataset

The data used in this work was collected by the GithubArchive project which continuously records the activities on the public Github Timeline and stores and makes it available analysis. In addition this data is also hosted on the Google BigQuery service which is an IaaS catered towards analysis of massive datasets with SQL like queries. This enabled the authors to perform preliminary exploratory analysis on the cloud without having to download the entire dataset.

The GithubArchive dataset is organized into various tables corresponding to different durations. These durations span a day, month or a year. Each row in these tables corresponds to the occurrence of an event in the Github Timeline. The Github Timeline provides data for more than 20 different event types but the purpose of the analysis here the following 5 were considered as useful features -

- Watch Events (Githubs nomenclature for Stars on the Timeline)
- Fork Events
- Issue Events

- Issue Comment Events
- Pull Request Events

We decided to look at the most recent month's tables and go backwards increasing the number of months as much as we could scale our methods in the short duration of time. Although the initial plan was to do almost all the data manipulation on BigQuery, this was not possible due to the usage limits imposed. BigQuery allows downloading of results up to only 16000 records which proved to be quite limiting in our case. To circumvent this difficulty and also keep to the data to a manageable amount on our own hardware the BigQuery API for Node.js was used to download to download all the records in the Github Archive dataset corresponding to any of the aforementioned events for the months of September 2016 and October 2016.

	Total	Watches	Forks	Issues	Issue Comments	Pull Requests
Records	8384758	4400039	1632399	655121	896455	800744
Repositories	1303989	698763	518621	256365	245853	417696
Users	1561560	760307	685899	384505	455669	389827

As it can be seen and reasonably understood, the watch events are the most common as they are the simplest action to perform on GitHub,followed by Forks. The other events are comparatively lesser in number and have a smaller number of unique repositories and users.

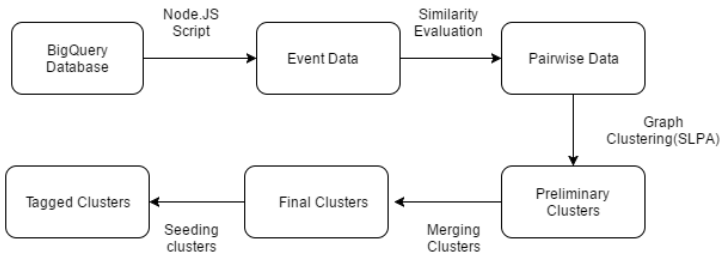
To help getting well defined and manageable clusters we need to filter this large amount of events and repositories. We need to remove the repositories that are not likely to be useful in the clustering and are too sparse or dense as these would result in clusters that are either too specific or too generic. The filtering criteria need to take care of the following situations -

1. Repositories that have very high proportions of only 1 type of event.
2. Repositories that have very large number of events by a very small number of users or a single user, possibly a bot.
3. Repositories with too few events to be helpful for clustering.

To following steps were taken to do this filtering:

1. Select only the repositories that have at least one event of each type and calculate the following for each of them: total event count, counts of each event type and proportion of each event type.
2. Remove the repositories that have a Watch Event or Fork Event proportion of greater than 0.8 and Issue and Issue Comment proportion greater than 0.7
3. To ensure that we get at least slightly recognizable repositories we also ensure that we remove the repositories with fewer than 50 Watch Event over duration in consideration.

VI. Proposed System



VII. Components of the System

Building the Graph

To cluster the selected set of repositories we model the entire system as an undirected weighted graph which the clustering algorithm then takes as its input and produces the required clusters. To build a graph we need some metric to decide if for any two repositories x,y there is an edge between these repositories and also the weight that we assign to this edge.

We build graphs as described above for each of the five Event Types that we have and in each of these graphs an edge between two repositories has weight that is calculated using the following procedure -

For a given event type E let the corresponding graph be G.E. For a pair of repositories x,y in our dataset we assign the edge between these two repositories to be the number of distinct users that performed the event E on both x and y. Let this value

be W.E(x,y).

Using the above technique, we assign weights to the edges between every pair of repositories. The edge is absent from the graph if the W(x,y) is 0 for any E.

The values for W.E(x,y) for all the event types are calculated by performing SQL queries involving JOINing the entire list of events with itself and checking for pairs of events that were done by the same user but on different repositories and counting the number of such occurrence. This list acts as the input edge list to the graph clustering algorithm.

Clustering Method

To perform the actual clustering we realised that the traditional clustering algorithms such as k-means, agglomerative clustering etc do not perform satisfactorily due to the following reasons:

1. They assume that the points being clustered are in a Euclidean Space and have well defined coordinates. This is something that does not in our case. We do not have coordinates representing each repository but only a similarity metric for pairs of repositories.

2. They do not take into account the social network like structure in the graph and make incorrect decisions when it comes to choosing nodes/ clusters to merge in a particular iteration.

To solve these problems, there exist specialized clustering algorithms that are tuned to cluster nodes arranged in a graph assuming no coordinate space for these nodes. Also they take into account a parameter known as "Betweenness" for edges and nodes that takes into account the inter community links in the social network while making the decision to merge.

For our purposes we use an algorithm known as Speaker-listener Label Propagation to cluster our graph and partition it into "communities" in social network parlance. This algorithm was chosen due to its good benchmarks, the easy availability of the implementation (in Java) and clear instructions on usage and configuration.

Preliminary Clustering

We first perform clustering on the initial data based on each of the five event types. Each of these give varying results on the number and size of clusters.

The highly dense events such as Watches and Forks give very few, large clusters. On manual scanning of the repositories in these clusters we see that these clusters are highly general and are not directly suitable tagging as assigning a single label to such a large cluster is too difficult and likely to be inaccurate.

The sparser events such as Issues, Issue Comments and Pull Requests give large number of weakly populated clusters. These clusters are opposite to those obtained using the dense events in that they are highly focussed and specific. However there are a large number of

clusters that contain between 2-10 and are far too specific.

At this stage we can either use the clusters in the first category and split them up using some metric or some other technique to come up with more reasonably sized and well distributed clusters or use those in the second category and merge them by recursively applying the clustering algorithm on the obtained clusters itself using a weight metric between clusters themselves calculated in a fashion similar to that used for individual nodes itself. We use the second approach as we felt it's simpler to merge existing clusters than to split them up.

Based on this decision to merge, we chose the preliminary 192/193 clusters obtained using the graph based on Issues. These clusters are imported into the database where we use the previously calculated pairwise information for Issues and the cluster that each repository

was assigned to to come up with pairwise metrics for all the events between the preliminary clusters.

Merging Clusters

Based on the pairwise cluster distance obtained in the previous step we create a secondary set clusters where each is a cluster of clusters. As the preliminary clusters were unevenly distributed we skew the clustering algorithm towards merging the smaller clusters together rather than merging the bigger cluster themselves. We find that such merging works best using the Pull Request events and obtain the final set of 14 clusters. In this way we obtain a final set of clusters that contain 5003 repositories.

Tagging and Seeding the Clusters

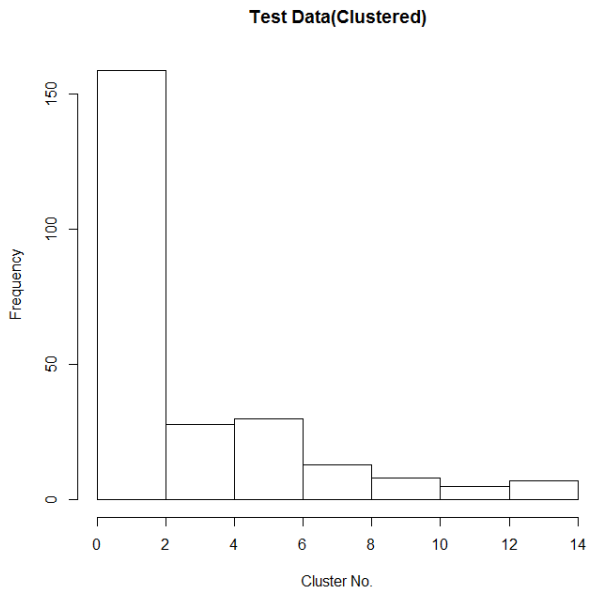
The final 5003 repositories are split up into a testing and training set. We chose 250 random repositories to be in the test set and manually tag them based on their domain without looking at the clusters that they were assigned to. We also tag each of the obtained clusters by looking at the repositories they contain after the removal of the 250 from the test set. We then compare the label

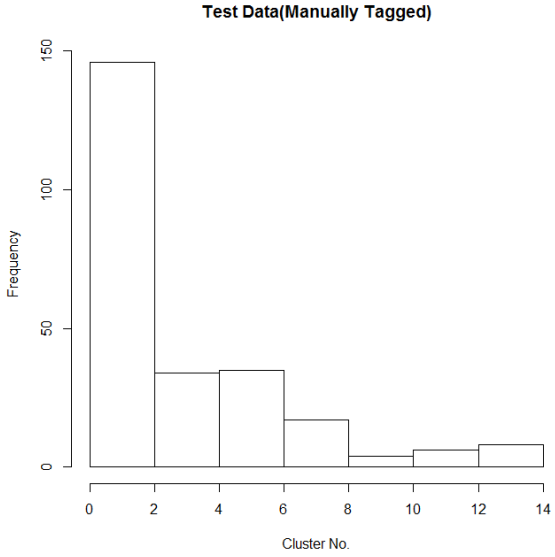
assigned to each repository in the test set to the label assigned to the cluster it is a part of and report the accuracy.

VIII. Results Achieved

We first manually tagged the 14 clusters obtained. It was found that while some clusters were very generic(web) and large, others were very specific and small. The clusters obtained were tagged as Web, Javascript, React, node.js, Android, Machine Learning, iOS, Rust, Security, Vue.js(very tiny and specific cluster), Google Go, Blogs, PHP and Vim/Word processing software. As the data used was for the last two months and we selected repositories based on density of events, this gives a very good indicator of the areas that are have been trending over that time period.

We then proceeded to manually tag 250 repositories(test data) and compared with the tags obtained from the clustering. 72 of the 250 repositories were misclassified. This was mainly because our algorithm did not account for overlapping areas/tags, sparse data and possible errors in manual tagging. This is a 71.2% accuracy and below are histograms of the tagged and clustered repositories in the test data.





It clearly shows that majority of the misclassified repositories were in the Web cluster as it is very generic and manual tagging placed it into more specific clusters.

IX. Conclusion

The GitHub ecosystem is a very vast social network filled with useful data that can be used for a wide variety of experiments. We consider this work to be only a small fraction of what is possible with the volume of data available.

Future Work:

1. The clusters obtained could be more well defined and easier to tag if we used a better selection criterion for the initial set of repositories. Also the difficulty in manually tagging the clusters can be averted by using NLP to process the READMEs of the repositories in a cluster where available and create a bag of words representing each cluster. A Naive Bayes classifier can then be used to classify repositories from the test set into clusters based on their own READMEs
2. The clustering approaches used in this work restrict the clusters to be non overlapping. But from experience we know that a single repository can easily span multiple domains.

3. The clusters and graphs obtained in this work can be used to build a recommender system where we can either recommend repositories based on the clusters or recommend similar users based on the pairwise metric between two particular users.

X. References

https://blog.insightdatascience.com/inspector-git-discover-awesome-new-repositories-on-github-b4ab0011f211#.6f9cqpk3c

ieeexplore.ieee.org/iel7/7022263/7022545/07022684.pdf

http://dl.acm.org/citation.cfm?id=2597113

https://staff.fnwi.uva.nl/m.derijke/wp-content/papercite-data/pdf/kenter-short-2015.pdf

Jierui Xie and Boleslaw K. Szymanski, SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process [https://arxiv.org/pdf/1109.5720.pdf]

XI. Contributions

The project consisted basically of downloading the data from the dataset, filtering the data to retain only the time frame we were interested in and the parameters required for our analysis. Following this, we had to convert it into the form that can be fed into the algorithm. The results from the algorithm had to be analysed and clusters had to be tagged. Then, we checked if the tagging was fine by comparing the manually tagged repositories and the clusters they had been placed in.

And owing to the fact that our project involved a lot of testing and verifying and tweaking to finally achieve results, at all points of time, we were all simultaneously doing everything every step, or simultaneously working on every aspect or providing the results required for each other. So talking about individual contributions will be a little tough, as every part was done by everyone, effectively.