Towards the analysis of patch and bug exploiting GitHub APIs [*]

Marco Arnaboldi
University of Illinois at Chicago
marnab2@uic.edu

Alessandro Pappalardo
University of Illinois at Chicago
apappa6@uic.edu

Andrea Tirinzoni
University of Illinois at Chicago
atirin3@uic.edu

1. INTRODUCTION

Versioning systems are one milestone in the field of software management and development especially for huge project and leading companies. In this direction several providers of this kind of service raised, providing for huge companies, but also for single or small group of developers, a simple and quick way to access those services via Internet. These providers also expose APIs in order to allow third parties to easily and automatically access information about the versioned project they own. One of those, it's GitHub and with our work we would like to prove that those APIs can be used for more higher purposes than retrieving information and downloading public accessible repositories. Maybe integrating and analysing the huge amount of retrievable information it is possible to find pattern in the developer and committers behaviour. This could be helpful in order to educate developers, but also to auto-detect common semantic mistakes they are used to make. So, the final goal of our project is to prove that aggregating information provided by GitHub about its versioned repositories it could be possible to mining from them useful information about programmers behaviour, in particular about the common they are used to do and trying to cluster them into families.

The rest of the report is organized as follows: 2 presents the proposed approach and some implementation details; preliminary results are detailed in Section 3, discussing the limitations of this work in Section 4, finally drawing some conclusions in Section 5.

2. SYSTEM DESIGN AND IMPLEMENTATION

The system was thought taking into account two main features: scalability and high patch-analysed ratio. In order to achieve these two goals the system was designed as what we define: a replicable and single stage scalable pipeline. The main idea, as shown in Figure 1, is to have a pipeline of stages that are in charge to download repositories and issues related to them, then to extract patches from those and then mix the just retrieved data in order to extract features for the machine leaning task. In order to improve the scalability, the development and future extension every stage was developed independently from the others. In this direction we adopted reactive programming paradigm using the Akka framework, defining each stage as an actor. Those actors are of three kinds, one for each stage of the pipeline: the downloader, the patch-analyser and the feature-extractor. Actually there is a fourth actor -the master- in charge to manage the communication flow between the others and to

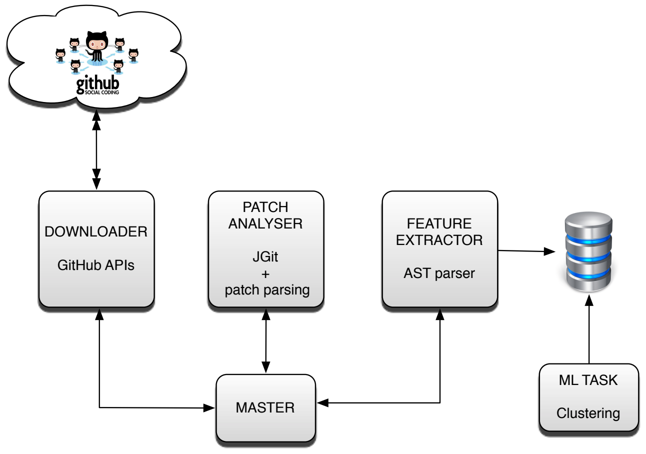

Figure 1: Overview of the proposed approach

instantiate new actors if needed. The pipeline is meant to be replicable in the sense that instantiating more than one stage at time it's possible to have several pipelines working in parallel. The mean of single stage scalable lies in the fact that it is also possible to set the number of replicas of a single stage independently by the other two. This means that in case a stage is slower than the others, just replicate its functionalities with one more actor of the same kind and in this way the bottleneck will be less strict. All of these parameters can be fixed inside a configuration class, which contains parameters for a full customization of the system. The output of the pipeline is a dataset representing a set of features describing the changes made in order to fix an issue in the repository. These data are then used to feed the machine learning task, that tries to find common pattern between changes and try to cluster them together. In this section we are going to go deep into each pipeline's stage and its duties.

2.1 Downloader

This stage of the pipeline is in charge to directly manage the GitHub APIs. These are REST APIs based on the weel known JSON format. In detail the system exploits REST call with OAuth authorization in order to increase the number of request per hour allowed (5000). The first request the downloader makes is for retrieving a list of issues that were closed. Once those are retrieved via parsing the JSON response they are filtered, all meaningless ones are discarded, where meaningless means those issues related to repositories not public or where the coommit that closed those issues is missing. In order to retrieve the information necessary to the filter step different API calls are made and the responses are mixed together in order to extract the necessary parameters for the filter. Once the repo is marked as good the downloader downloads it and send information to local repository path, relative issue and closing commit to the patch analyser stage.

2.2 Patch Analyser

This stage of the pipeline is in charge of extracting the modifications introduced by a commit, given a commit hash from the previous stage. We take advantage of the JGit library to extract the location of the modifications performed by the commit in the source code.

2.3 Feature Extractor

This stage of the pipeline receives information about diffs in a patch file and extracts features from the involved source files. The goal is to obtain a numerical dataset where each sample represents a single diff in a patch file. First, the files before and after the change are parsed to generate their AST. Then, the AST is used to extract features from the portion of code that has changed. We define a feature as the number of occurrences of some semantic attribute in the code (e.g., a for loop, a certain operation, a lambda expression, and so on). A sample is then the vector of features in the code before and after the change. Thus, we obtain a (very sparse) numerical dataset onto which we can apply clustering techniques.

2.4 Machine Learning

We apply clustering on the numerical dataset generated by our system. We decided to consider only hierarchical clustering since it provides the most general results (we get a cluster dendogram that we can cut in the desired number of clusters). We adopt the cosine similarity measure since it is the most suitable for our kind of dataset (high-dimensional and sparse feature vectors). We implement our clustering script in R, so that it can be easily integrated with Spark in case we need to scale to larger datasets.

3. EXPERIMENTAL RESULTS

Due to the restrictions imposed by the GitHub APIs, we were able to generate only a fairly small dataset. We ran our system three times, until we finished the available API calls, to collect a final dataset of 2762 samples from 107 different repositories. We applied hierarchical clustering on this dataset. The resulting dendrogram is shown in Figure 2. We can see that the cluster distribution is not completely random but we are able to distiguish some clusters. In particular, a cutting point of 30 clusters seems to be the best according to the dendrogram. We decided to evaluate our clusters by using the repository ID from which each sample is generated. In particular, we augmented each sample with its repository ID and, given the 30 clusters produced by hierarchical clustering, we computed the entropy based on such variable. We obtained an entropy of 0.7. Remembering that 0.5 implies a completely random distribution, 0.7 means that we are able to detect a slight pattern: the repository IDs are not randomly distributed in the clusters, that is, there is some relation between the changes and the

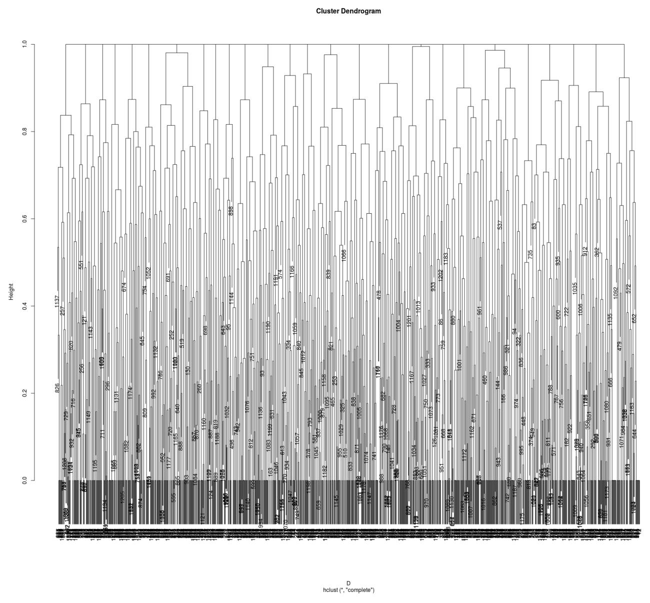

Figure 2: The dendrogram produced by hierarchical clustering

project. This may be a sign that developers of the same project tend to repeat the same semantic errors or to apply the same semantic corrections. Notice however that we would need much more samples to confirm this argument. At the moment, this was the only technique we could come up with to evaluate our clusters, since our initial idea to use bug reports was not feasible.

4. LIMITATIONS

Our current approach has several limitations. First, we are considering only Java projects. However, our code is easily extensible to support other languages. Second, we can generated only small datasets due to the restrictions imposed by GitHub on API calls. Our current solution is to run the system multiple times, waiting the necessary time between runs, to collect more samples. Our main limitation is that we are not able to exploit bug reports as we were expecting. The majority of bug reports on GitHub are poorly written and the application of some text mining algorithm on them would lead to very bad results. Thus, at the moment we lack a method to evaluate our clusters in a meaningful way. Our last limitation is that we are considering only a simple feature set. However, our code can be easily extended to include much more complicated features, even those representing dependencies between code constructs.

5. CONCLUSION AND FUTURE WORK

We presented a new approach to analyze git patches and bugs based on Machine Learning techniques. We demonstrated how we can obtain some meaningful results even by implementing a simple system with several limitations. We mentioned that our system can be easily extended to obtain much more interesting results. As a future work, we intend to improve our system's capabilities to see whether we are, indeed, able to detect more useful patterns.