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
add mail

Andrea Tirinzoni
University of Illinois at Chicago
add mail

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

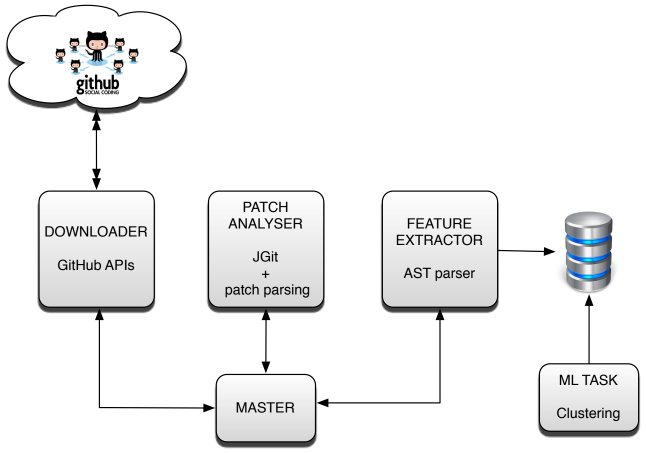

Figure 1: Overview of the proposed approach

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

2.3 Feature Extractor

2.4 Machine Learning

3. EXPERIMENTAL RESULTS

4. LIMITATIONS

5. CONCLUSION AND FUTURE WORK