RepoTrack - Technical Documentation

Kanghu Shi

August 2021

1 Introduction

In this document we shall briefly present the structure of our project. The architecture is composed of two independent parts: (1) the repository analyzer and (2) the visualization tool, which we shall refer to as the **backend** and **frontend** of the application respectively.

2 Backend

The backend is written in Python, using `pydriller` as its main repository mining framework. Upon running it on a Git repository (hosted either locally or on Github's cloud service), a `.json` report encompassing the amount of work put in by each contributor into each of the repositories' files will be generated.

2.1 Implementation

We represent repositories as trees, composed of `File` (leaf) or `Package` (non-leaf) nodes. Each node has an associated list of contributors. Our code is structured under three distinct files:

1. `repostructure.py`

   Defines the data types through which we internally represent repositories. We define different types of objects to represent `File`(s), `Package`(s) and `Repository`(s) respectively. The classes are conveniently named as such.

2. `repotrack.py`

   Entry point ('main') of the backend. This is the script that is called at execution, thus dealing with commit processing (through `pydriller` functionalities)

3. `config.py`

Configuration file for defining & classifying supported extensions and regular expressions.

2.1.1 Data types

We define the following data types for representing repositories and contributors:

- `Contributor`

  A contributor is defined by its name and an associated hashmap representing the attained metrics and their respective values. Utility methods are provided to easily collect metrics from a given modification.

```
/* Contributor exemplified */
{
    "name" : "John",
    "contribution" : {
        "LOC+" : 100,
        "LOC-" : 50
    }
}
```

- `File`

  Represents a file from the repository (leaf nodes of the tree). A file is represented by its name and list of contributors which have contributed to the file. Modifications (multiple of which compose a commit) can be directly processed by a File object in order to update the metrics of its contributors.

```
/* File exemplified */
{
    "name" : "main.c",
    "contributors" : [
        /* Contributor objects */
    ]
}
```

- `Package`

  Packages are non-leaf nodes within our tree and thus may contain child nodes of their own. A number of utility methods are provided for the purpose of manipulating packages and their underlying structure (e.g. adding/removing/moving child nodes).

```
/* Package exemplified */
{
    "name" : "tests",
    "children" : [
        /* File objects */
    ],
    "contributors" : [
        /* Contributor objects */
    ]
}
```

- `Repository`

  Repository is the root package of the tree, thus englobing the entire repository structure. In essence, it is `Package` wrapped with methods to ease the process of loading & processing `git` repositories.

2.2 Execution

The script may be run with the following command (given that Python 3.4 or greater is installed):

```
python repotrack.py <source> -t <timeframe> -b <branch>
```

Where `<source>` represents the repositories' location (specified as either a local path or web URL), `<timeframe>` is the number of months for which contributions are deemed 'recent' and `<branch>` represents the name of the branch of interest. If not specified, a default timeframe of 6 months will be assigned and the branch of interest will be assumed to be the main/default one.
Example of script call:

```
python repotrack.py https://github.com/ishepard/pydriller
-t 3 -b pydriller-pygit2
```

Which will analyze the git repository of `pydriller` on branch `pydriller-pygit2` whilst considering as recent any contributions from the previous 3 months.

3 Frontend

The frontend is composed of a web page and thus makes use of HTML, CSS and JavaScript. Any report generated by the backend can be uploaded through the local file explorer & visualized as a dynamic tree chart. For visualization purposes, we leverage the powerful capabilities of the D3.js library, which allows for neat creation of graphical devices (trees, charts, graphs, etc.) and state transition & animation management.

3.1 Implementation

We shall list the files present within our frontend together with their associated functionality within our application.

1. `graph.html`

   The main HTML page which handles loading the stylesheet, (java)scripts, as well as defining the frontend's composing blocks (tree container, sidebar panels, etc.)

2. `graph.css`

   Classic CSS file which styles the different elements on the webpage.

3. `script.js`

   Essentially the "main" script of the frontend. Here, all graphical elements are actually created, using helper methods from the scripts listed below. The color palette and other standardized values are also defined within this file.

4. `graph.js`

   Handles the "heavy work" of initializing D3 and defining all the visual elements (repository tree chart and the different types of metric bar charts). Different methods are provided for manipulating the graphical elements described above (e.g. toggling a node, collapsing the tree, loading a specific contributor's metrics within the barchart, etc.).

5. `aggregation.js`

   Contains methods specific to aggregation of existing data: aggregated metrics (i.e. Forward engineering, Re-engineering, etc.) and distribution of metrics.

3.2 Execution

The frontend may be executed by simply opening the webpage (`graph.html`). The sidebar button "Choose files" will prompt up the file explorer for selecting a report to be visualized.