Assignment 1

November 22, 2015

1 Examining the effects of ownership on software quality

1.1 The Case Of Lucene

We want to replicate the study (http://dl.acm.org/citation.cfm?doid=2025113.2025119) done by Bird et al. and published in FSE'11. The idea is to see the results of a similar investigation on an OSS system. We select Lucene (https://lucene.apache.org/core/), a search engine written in Java.

1.2 Data collection

First we need to get the data to create our table, in other words we do what is called data collection.

In our case, we are interested in checking the relation between some ownership related metrics and post-release bugs. We investigate this relation at file level, because we focus on Java and in this language the building blocks are the classes, which most of the time correspond 1-to-1 to files.

This means that our table will have one row per each source code file and as many columns as the metrics we want to compute for that file, plus one column with the number of post release bugs.

1.3 Collecting git data

For computing most of the metrics we want to investigate (e.g., how many people changed a file in its entire history) we need to know the history of files. We can do so by analyzing the versioning system. In our case, Lucene has a Subversion repository, but a git mirror (https://github.com/apache/lucene-solr.git) is also available. We use the git repository as it allows to have the entire history locally, thus making the computations faster.

We clone the repository. For this we use the python library 'sh'.

```
In [1]: import sh
        import json
        import os
        import glob2
        from collections import Counter
        import csv
```

```
In [2]: if not os.path.exists(os.getcwd() + '/lucene-solr'):
            print("Path doesn\'t exists, cloning repo:")
            sh.git.clone("https://github.com/apache/lucene-solr.git")
        else:
            pass
```

```
In [3]: git = sh.git.bake(_cwd='lucene-solr')
```

To perform the replication, we can either reason in terms of releases (see list of Lucene releases (http://archive.apache.org/dist/lucene/java/)), or we can just inspect the 'trunk' in the versioning system and start from a given date.

We decided to reason in terms of releases.

We first check which releases can be found in the git repository:

```
In [4]: tags = git.tag() # To show all the tags
        versions = []
        for t in tags:
            if t.startswith('lucene_solr_'):
                versions.append(t[12:-1].replace('_','.')) #extract the version, remove the \n
```

1.4 Bug extraction

We want to count the number of bugs for every .java file. To do that we follow these steps: 1. Select a version of the software: we will find and use the one with more bugs, so that we can have more data; 2. Find the bugs that affect that version; 3. For every bug, find the commit(s) that fixed it: it is easy to do it in the Lucene repository because if a commit fixes the bug with key=LUCENE-123 then it has, by convention, the keyword LUCENE-123 in the commit message. 4. For every commit in the resulting list, get the files that it modified. If a file was modified by a bug-fixing commit it means that it was the one with the bug, so for every one of these file the bug count is incremented by one.

We first define two methods: * extract_bug(file): extracts the bug object from the corresponding .json file. It returns None if it is not a bug (e.g. it can be an 'Improvement') or if it is not marked as 'Fixed'; * extract_version_bugs(file): extracts a list of all the bugs that affect a specific version;

```
In [5]: def extract_bug(file):
            if not file.endswith('.json'):
                    return None
            else:
                bug_json_string = open("issue_LUCENE/" + file).read()
                bug = json.loads(bug_json_string)
                bug_fields = bug.get('fields')

                if bug_fields['issuetype']['name'] != 'Bug':
                    return None

                if bug_fields['resolution'] == None:
                    return None

                if bug_fields['resolution']['name'] != 'Fixed':
                    return None
            return bug
```

```
In [6]: def extract_version_bugs(version):
            if version.endswith('.0'):
                version = version[:-2]

            version_bugs = []
            for file in os.listdir(os.getcwd() + "/issue_LUCENE"):
                bug = extract_bug(file)
                if(bug == None):
                    continue

                affected_versions = bug['fields']['versions']
                if len(affected_versions) == 0:
                    continue
                else:
                    for v in affected_versions:
                        n = v['name']
                        if (n == version) or (n == version + '.0'):
```

```
                        version_bugs.append(bug)
            return version_bugs
```

1.4.1 Most buggy release

The first thing that we need to do is select a release: we want to find the one with more bugs.

The following block counts the number of bugs that affect every release, and that are now fixed:

```
In [7]: version_bugs_count = dict()
        for file in os.listdir(os.getcwd() + "/issue_LUCENE"):

            bug = extract_bug(file)
            if(bug == None):
                continue

            affected_versions = bug['fields']['versions']
            if len(affected_versions) == 0:
                continue
            else:
                for v in affected_versions:
                    n = v['name']
                    if(version_bugs_count.get(n) == None):
                        version_bugs_count[n] = 1
                    else:
                        version_bugs_count[n] = version_bugs_count[n] + 1
```

We then use the resulting dictionary to find and checkout the release that is available as a tag in the repository and that has the highest number of bugs (we discard ALPHA or BETA versions):

```
In [8]: most_bugged_version = max(version_bugs_count, key=version_bugs_count.get)
        while ((most_bugged_version.find('ALPHA') != -1) or
               (most_bugged_version.find('BETA') != -1) or
               ((most_bugged_version not in versions) and (most_bugged_version + ".0" not in versions))

            version_bugs_count.pop(most_bugged_version)
            most_bugged_version = max(version_bugs_count, key=version_bugs_count.get)

        print("The release with the highest number of bugs is version " + most_bugged_version)
```

```
The release with the highest number of bugs is version 3.1
```

```
In [9]: version = most_bugged_version
        git.checkout("tags/lucene_solr_" + version.replace('.','_').replace('-','_0_'))

Out[9]:
```

1.4.2 Bug count

Now that we selected a version we want to apply the described bug counting procedure. We first extract the bugs that affects it:

```
In [10]: bugs = extract_version_bugs(version)
```

The following blocks extract the commits that fixed every bug in the list, and for every commit the bug count is incremented for every .java file that it affects.

We use the **git log** command, with the grep option, to find the commits, and then every commit is shown with the **git show** command: this shows only the affected files because the pretty=format option is leaved empty.

```
In [11]: commits = []
         for b in bugs:
             log = git.log("--all", "--grep=" + b['key'], "--pretty=format:%H")
             for c_hash in log:
                 if(c_hash.endswith('\n')):
                     c_hash = c_hash[:-1]
                 commits.append(c_hash)

         commits = set(commits) #To avoid considering the same commit more times (e.g. if a commit fixe
```

```
In [12]: file_bugs = dict()
         for commit in commits:
             details = git.show("--name-only","--pretty=format:",commit)
             for file in details:
                 if(file.endswith('\n')):
                     file = file[:-1]
                 if(file.endswith('.java')):
                     file = file # .split('/')[-1]
                     if(file_bugs.get(file) == None):
                         file_bugs[file] = 1
                     else:
                         file_bugs[file] = file_bugs[file] + 1
```

We now have a dictionary that contains the bug count for every file (that has bugs), and we can use it later to build the final dataset.

1.5 Ownsership metrics

1.5.1 Find all java files

To be able to compute the metrics for all the java files it is necessary to find the location of them in the repository. This is done by recursively going through the repository and storing all the file locations so that they can later be used to get the log message through git.

We decided to use the whole file path (relative to the root of the repository), because there are files with the same name.

```
In [13]: java_file_list = []
         cdir = os.getcwd()
         os.chdir(cdir + '/lucene-solr')

         ## All files
         for file_location in glob2.glob('*/**/*.java'):
             java_file_list.append(file_location)

         os.chdir(cdir)

         print("Number of Java files: " + str(len(java_file_list)))
```

```
Number of Java files: 2803
```

1.5.2 Computing the metrics for every file

Computing the metrics is done by looking at the log of every java file found by globbing througth the repository recursively. We are only interested in the contributors of every commit. We will compute the following metrics:

- **Major**: Number of minor contributors (developers whose ownership on the code in the file is less than the 5%);
- **Minor**: Number of major contributors (developers whose ownership on the code in the file is more than or equal to the 5%);
- **Total**: Total number of contributors;
- **Ownership**: Proportion of ownership for the contributor with the highest proportion of ownership.

Method:

1. Retrieve commit log through git;
2. Convert the log message into a list, so every entry contains the name of the developer that made a commit to the file;
3. Calculate the metrics (**Major**, **Minor**, **Total** and **Ownership**);
4. Do this for all the files.

```
In [14]: # File -  Minor - Major - Total - Ownership
         data_set = []

         # file_name = java_file_list[0]
         for file_name in java_file_list:
             # Name
             persons = set()

             # Metrics
             major = 0
             minor = 0
             total = 0
             ownership = 0;

             # Get Log data from git
             var = git.log("--pretty=format:%an", "--follow", file_name)
             developers_of_commits = var.split('\n')

             # Determine commits per developer
             c = Counter(developers_of_commits)

             # Get set of developers and calculate total number of commits
             for name_dev in developers_of_commits:
                 if name_dev not in persons:
                     persons.add(name_dev) # Unique set of developers of this file
                     total = total + c[name_dev] # Total number of commits of this file

             # Compute metrics
             for name in persons:
                 contribution_perc = c[name]/total
                 if contribution_perc >= 0.05:
                     major = major + 1
                 elif contribution_perc < 0.05:
                     minor = minor + 1

                 if ownership < contribution_perc:
                     ownership = contribution_perc

             # Add computed data to the data_set
             data_set.append([file_name, minor, major, len(persons), ownership])
```

1.6 Output the computed metrics and the bug count into a .csv file

```
In [15]: filename = 'dataset.csv'
```

```
In [16]: myfile = open(filename, 'w')
         wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
         wr.writerow(['File', 'Minor', 'Major', 'Total', 'Ownership', 'Bugs'])
         for row in data_set:
             if file_bugs.get(row[0]) != None:
                 row.append(file_bugs[row[0]])
             else:
                 row.append(0)
             wr.writerow(row)
         myfile.close()
```