assignment1

September 19, 2016

1 Examining the effects of ownership on software quality

This is an assignment for IN4334 - Mining Software Repositories.

Group **2hot4school**

1.1 The Case Of Lucene

We want to replicate the study done by Bird et al. and published at FSE'11. The idea is to see the results of a similar investigation on an OSS system. We select Lucene, 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 investigating 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.

1.2.1 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 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 [ ]: import sh
        import json
        import re
        from __future__ import division
        from jira import JIRA
```

We start by cloning the repository

```
In [ ]: sh.git.clone("https://github.com/apache/lucene-solr.git")
```

And we make sure that we point our 'git' command to the right directory.

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

To perform the replication, we could either reason in terms of releases (see list of Lucene releases), or we could just inspect the 'trunk' in the versioning system and start from a given date.

In this assignment, we go for the second option: We consider the 'trunk' (main branch in svn) and focus on a 6-month period in which we look at the bugs occurring to the files existing at that moment. Concerning bug data, you will consider a time window from Feb 01, 2015 to Jul 31, 2015.

Let's retrieve the list of files existing in the trunk on Feb 01, 2015.

```
In [ ]: shaFeb15 = (git("rev-list","-n 1","--before=\"2015-02-01 00:01\"","master")).stdout[:-1]
        shaFeb15
```

```
In [ ]: git.checkout(shaFeb15)
        git.status()
```

1.2.2 Collecting Java Files

After getting the snapshot right, we will get all the java files inside the repository at a given snapshot. We opted to choose **all the directory** inside the lucene-solr repository instead of only LUCENE repository alone because it is our first understanding of the problem outlined in the assignment. We will obtain the full path of the **java files** and makes them keys to our dictionary : java_files. This dictionary will have the following value indicated in the assignment : 1. Filename i.e. Foo.java 2. Package name i.e. org.apache.lucene.xxxx 3. Number of minor contributor 4. Number of major contributor 5. Number of total contributor 6. Ownership 7. Number of bugs

As the first step, we will fill value number 1 and 2 and initialized value number 3 until 7 with zero. We will use git command : ls-files to obtain all the files inside the repository and do a quick filter for .java inside the filename. For easier reference, we will save the output of the dictionary into json file : **java_files.json**.

```
In [ ]: listfiles = git("ls-files").split("\n")

        java_files = {}

        for i in listfiles:
            if i[-5:] == ".java":
                split_file = i.split("/")
                try:
                    pkg = ".".join(split_file[split_file.index("org"):-1])
                except:
                    pkg = "no package"
                filename = '/'.join(split_file[-1:])
                java_files[i] = [filename,pkg,0,0,0,0,0]

        with open('java_files.json', 'w') as fp:
            json.dump(java_files, fp)
```

1.2.3 Collecting Committer

After getting the files, we will obtain for every file, every committer that contributed to make changes in each files, and how many changes that they made for that file. We will create a dictionary with keys : java files and committer, and injecting the number of changes as the value. For our convenience, we save the output for the dictionary into json file : **ownership_new.json**

```
In [ ]: committers_data = {}

        for java_file in java_files:
            committers = git('log','--pretty=format:"%ce"',
                             '--after="2014-01-01"', java_file).splitlines()
            committers_data[java_file] = {}
```

```
            for committer in committers:
                if committer in committers_data[java_file]:
                    committers_data[java_file][committer] += 1
                else:
                    committers_data[java_file][committer] = 1

        with open('ownership_new.json', 'w') as fp:
            json.dump(committers_data, fp)
```

1.2.4 Computing Ownership

Now that we have all the contributions for each file, we are ready to compute minor, major, and ownership for each file. We do this as follows : 1. Iterate for each file in our ownership_new.json file. And for each file, iterate over each committer. 2. At first step, we compute the total contribution for each files and save them into variable total_commit. 3. We iterate again in each committer to find each contribution. This way, we can compute the ownership of that particular committer in a given file. Then we compute percentage of ownership to decide whether the committer is minor or major contributor. 4. We also compute the maximum ownership for each file

Up until this point, we are done with the ownership propertis of each files. Next, we will compute the number of bugs for each files.

```
In [ ]: with open('ownership_new.json') as data_file:
            files_contributor = json.load(data_file)

        with open('java_files.json') as data_file:
            java_files_copy = json.load(data_file)

        for filename in files_contributor:
            total_commit = 0
            max_ownership = 0
            minor = 0
            major = 0
            for committer in files_contributor[filename]:
                total_commit += files_contributor[filename][committer]
            for committer in files_contributor[filename]:
                committer_contribution = files_contributor[filename][committer] / total_commit
                if committer_contribution >= 0.05:
                    major += 1
                else:
                    minor +=1
                if committer_contribution > max_ownership:
                    max_ownership = committer_contribution
            java_files_copy[filename][2] = minor
            java_files_copy[filename][3] = major
            java_files_copy[filename][4] = minor + major
            java_files_copy[filename][5] = "{:0.2f}".format(max_ownership)
```

Now we need to get the commit before Aug 01, 2015.

```
In [ ]: shaAug15 = (git("rev-list","-n 1","--before=\"2015-08-01 00:01\"",
                        "master")).stdout[:-1]
        shaAug15

In [ ]: git.checkout(shaAug15)
        git.status()
```

1.2.5 Collecting the bugs

Now we need to get the bugs between 2015-02-01 00:02 until 2015-08-01 00:00. We will analyze the JIRA of lucene_solr project. For convenience, we will use JIRA API in Python.

JIRA provides JQL (JIRA Query Language), sql-like language that makes us easy to find particular bugs at a given time interval. As LUCENE and SOLR is a different project in JIRA package, we will use the following filters to find all bugs in the given time period. We want to focus on the bugs that is created between the given interval. We chose to inspect both LUCENE and SOLR because our focus in the beginning is to inspect all directory inside the lucene_solr repository, which contains both project LUCENE and SOLR.

1. 1st filter : (project = "LUCENE" or project = "SOLR"). This way, we make sure that we only lookup issues related to LUCENE or SOLR
2. 2nd filter : type = 'Bug'. JIRA tracking system has several type of issues e.g. Bug, Improvement, New Feature, Task, Custom Issue. We want to make sure that we only look for Bugs as indicated in the assignment.
3. 3rd filter : created >= '2015-01-01 00:02' and created <= '2015-08-01 00:00'. This is the time interval given in the assignment
4. 4th filter : status = 'Closed'. As we noticed from explanation of the assignment from the lecturer last Thursday, we noted that it is the bug that has been closed that needed to be observed. So that's why we included this filter into the query.

One of the tricks here to fetch all issues in JIRA is that we need to make a kind of loop to get all of them. This is due to the limitation of the query which has an upper limit of maximum 100 results that can be obtained at any point.

```
In [1]: jira = JIRA("https://issues.apache.org/jira/")

        checkpoint = 100
        total_issues = 0

        issues = []

        while checkpoint == 100:
            issues_page = jira.search_issues("(project=LUCENE or project=SOLR) "
                            + "and type = 'Bug' and (created >= '2015-02-01 00:02' "
                            + "and created <='2015-08-01 00:00') and status = 'Closed'"
                            , startAt=total_issues, maxResults = 100)
            for i in issues_page:
                issues.append(i.key)
            checkpoint = len(issues_page)
            total_issues += checkpoint


      File "<ipython-input-1-ed3a2bbe12be>", line 9
    issues_page = jira.search_issues("(project=LUCENE or project=SOLR) \'
                                                                        ^
SyntaxError: EOL while scanning string literal
```

```
In [ ]: print len(issues)
```

Using Git log, we fetch the commit-id and the commit subject of all commits between 2015-02-01 and 2015-08-01. Each commit is checked if its description contains ID of the bug. If it was, the commit id and the bug key will be added to a mapping, essentially linking a commit to a bug.

```
In [ ]: gitlog = (git("log","--after=\"2015-02-01 00:00\"",
                        "--before=\"2015-08-01 00:00\"",
                        "--format=%H#%s")).split("\n")
        commit_to_bug = {}
        for counter in range(0,len(gitlog)-1):
            i = gitlog[counter]
            text_split = i.split("#")
            commit_id = text_split[0]
            bug_key = re.findall(r"(\bSOLR\b-[0-9]+|\bLUCENE\b-[0-9]+)",text_split[1])
            if bug_key!= [] and commit_to_bug.has_key(bug_key[0]) == 0 and bug_key[0] in issues:
                commit_to_bug[bug_key[0]] = []
                commit_to_bug[bug_key[0]].append(commit_id)
            if bug_key!= [] and commit_to_bug.has_key(bug_key[0]) == 1:
                commit_to_bug[bug_key[0]].append(commit_id)
```

So, up until this point, we already have a mapping of commit_id and bugs inside the interval observed. Next step is to find the changed files for each commit and then compute the number of bugs for each file.

One of the caveat that we have to note here is : for one bug ID, there may be multiple commits to solve that. Hence, one file might be modified many times under one bug ID in different commits. Our logic to find number of bugs for a given file is : 1. For each bugs, iterate over each commit_id associated with that particular bug. 2. If a file is changed, add to a list of list_of_changed_files, and add the bug number by 1. 3. If a file in another commits under same bug ID already exist in list_changed_files, don't add bug by 1, so that there won't be duplicate.

```
In [ ]: # git diff-tree --name-only -r commit_id
        # for each files in all commit, find the changed files
        # commit_to_bug is a dict with value list of commit_id
        # find for each bug id, if file is already check, dont add anymore bug in it

        for key in commit_to_bug:
            list_of_changed_files = []
            for commit_id_bug in commit_to_bug[key]:
                changed_files = git("diff-tree","-r","--no-commit-id","--name-only",commit_id_bug)
                x = changed_files.split("\n")
                for y in x:
                    if y in java_files_copy and y not in list_of_changed_files:
                        java_files_copy[y][6] += 1
                        list_of_changed_files.append(y)
```

1.2.6 Wrap it up under one file

Finally, we put all the obtained data into a readable table

```
In [ ]: final_output = "Filename, Package, Minor, Major, Total, Ownership, #Bugs\n"

        for filename in java_files_copy:
            final_output +=  ",".join(str(x) for x in java_files_copy[filename]) + "\n"

        with open('result.csv', 'w') as fp:
            fp.write(final_output)
```

And we're done!