Capita Selecta for Software Engineering
Pydriller Software Metrics

Gakuo Patrick

February 19, 2019

1 Introduction

In this project we are tasked to compute ten software metrics for five git repositories. We are to use the Python based Pydriller library to mine these software metrics from the git repositories. The Pydriller library offers different APIs to traverse a git repository and obtain its historical and current information (Spadini, Aniche, & Bacchelli, 2018). In this project, we use Pydriller to extract information related to commits from the beginning of the project to the end. This information will help us compute the 10 metrics which can be divided into process and developer metrics.

2 Metrics Description

We looked at the below metrics

1. **Product Metrics**
   (a) LOC - count of lines of code in a component
   (b) Cyclomatic Complexity - a measure of independent paths through a component
2. **Process Metrics**
   (a) Number of changes - number of commits for each component
   (b) Change burst - maximum number of consecutive commits editing a component
   (c) Past faults - number of faults in the previous period
   (d) Number of developers - number developers who worked on a file in a given period
   (e) Entropy of changes - a file's number of changes divided by the total number of commits in the period

3. **Developer Metrics**
   (a) Structural scattering - It is obtained for all file pairs edited by a all developers who edited a file. It is a measure of file pairs path distance.
   (b) Semantic scattering - It is obtained for all file pairs edited by a all developers who edited a file. It is a measure of file pairs content similarity (a value between zero and one).

3 Implementation architecture description

Our implementation starts with a list of paths to the five git repositories and follows the following steps and functions:

1. $computeRepoMetrics(repopath)$ -Takes a repository path and creates a $GitRepository$ object denoted as 'gr'. It then calls $analyzeCommits(repoPath, gr)$ and $calculateStructAndSemanticScattering(gr)$ whose outputs are fed to the function $parallelMetricProcessing$.

2. $analyzeCommits(repoPath, gr)$ - Takes a repository path and a $GitRepository$ object. This is the heart of all computations as it collects all information needed to compute the process and product metrics. In particular, it calls the function $calculatePeriods(gr)$ which accesses the first and last commit of the git repository through Pydriller. This function produces a list of 3 months tuples ((start_date,end_date)) starting from the fist commit date to the last commit date of a repository. Provided with this list of date interval tuples, $analyzeCommits$ traverses the git repository using the $RepositoryMining$ api and produces dictionaries, numbers, and lists needed to compute process and product metrics. These dictionaries include the following:

   (a) buggy_commits_dictionery - for each commit in a three months period, we obtain each commit's message and use the regular expression finder to see if its a bug fixing commit in the function $regularExpFinder(commitMessage)$. If the the function returns true, we can then use the SZZ algorithm to find previous commits that touched the same lines being patched using the function $getPastFaults(commit, gr)$ which uses Pydriller's $gr.get_commits_last_modified_lines(commit)$ (Spadini et al., 2018). This gives a list of hashes of all commits that touched the patched lines. We then create a dictionary of hashes as keys and (files, period_tuple) as values.
   (b) numberOfcommitsInPeriod - this is just a number increased in every commit iteration. It computes total number of commits in the period. It helps in calculating changes entropy.
   (c) modified_files - For each file we gather the unique commit hashes related to the file and store each file_name - hashes set into a dictionary. This helps us in computing changes burst for each file.

(d) authors_information - For each file we get a set of its unique developers/authors. This helps us in calculation of developer metrics.

(e) lastCommitFileComplexityAndLoc - We store each file's complexity and number of lines of code in a tuple and assign each of such values to the respective file name as a key in a dictionary.

(f) commitHashesInInterval - we collect all commit hashes in a 3 months interval into a list. This helps in computing changes burst for each file whose hashes have been collected into the modified_files above.

(g) authorEditedComponents - For each author of each commit, we collect all her edited components/files into a set and store this in a dictionary. This helps us in computing developer metrics for the period.

(h) buggy_files - in this list we check if any of the hashes related to a file in modified_files dictionary is also in the buggy_commits_dictionary. If that is the case, we add the file into the buggy_files list for this period. This helps us see if a file is buggy or not.

(i) pastFaults - in this case we assume that if a commit was a bug fixing commit (checked using the regular expression in $getPastFaults(commit, gr)$), we say its a fault reported for the file. We then sum all such occurrences. In the next period we can access this number for each file this value as the past faults.

We collect all these dictionaries into a list and this is the output of the function *analyzeCommits*.

3. $calculateStructAndSemanticScattering(gr)$ - This function takes a GitRepository object and returns scattering metrics for all files pairs contained in the repository. For structural scattering' it compares directory paths using the function $getDistanceBetweenFiles(file1, file2)$. For semantic scattering,we fetch each file pairs contents (read in file contents usin Python) and split them into strings. We then compute then count the number of words in the intersection of the two files and divide that by the sum of both files. This is done in the function $getTextSimilarity(file1, file2)$. The function *calculateStructAndSemanticScattering* returns a tuple of three lists with corresponding indexes. The first has all the pairs, the second their structural scattering and the third their semantic scattering.

4. $parallelMetricProcessing$ - as mentioned in the explanation of function number 1 above, this function takes the results of function 2 and three. It also takes a GitRepository object and a the number of threads (it is a multithreaded function). This function maps the results of function 1 and 2 above into the $compute_fileMetrics$ for parallel execution.

5. $compute_fileMetrics$ - this function takes each periods data and a dictionary of previous faults from its preceding period computed in function 2. It also takesand semantic scattering data for each file pair in the git

repository computed in function 3. It also takes a dictionary of previous faults from its preceding period. These inputs are provided as a single tuple. This function computes all the ten metrics required for this project using the stated input data. It then prints out each file's metrics into a CSV file for each time period.

(a) number of changes - for each file obtain the length of the its hashes in modified files dictionary (from function 2)

(b) change_entropy - divide number of changes above with the period's total number of changes (from function 2)

(c) lines of code - obtain lines of code from the respective dictionary obtained from function 2 for each file name

(d) complexity - obtain complexity from the respective dictionary obtained from function 2

(e) number of past faults - directly obtained from the $compute_{f}ileMetrics$ input tuple. Its value is a dictionary containing all file's commits from the previous period. It is an empty dictionary for the first period.

(f) number of developers/authors - obtain the number of developers from the respective dictionary obtained from function 2

(g) change burst - get each file name's hashes and the total hashes in the period and use the function *calculateChangeBurst* to compare the two lists for the longest matching subsection.

(h) structural_scattering and semantic_scattering = this is computed for each individual file using the function $compute_{s}ingle_{f}ileScattering$ which takes as input a file name, the authors who edited the file, the dictionary of author and the set of files they edited in the period, and the files scattering pair metrics obtained from function 3. It returns a (structural,semantic) pair.

(i) is_buggy if filemname is in the buggy_files dictionary from function 2, we give it a value of '1', else a '0'.

(j) writeCSV(fileRows,filename) - we finally use this function to write our metrics into a csv file.

4 Results and discussion

We write the results into CSV files. These processes took long to run and it is possible that we did not obtain all the repository data. Additionally, we suspect that our calculation of developer metrics could have been improved by obtaining the source code and file paths directly from the commit modifications for each commit using Pydrillers file.source_code and file.old_path/file.new_path. In this project, we calculate the scattering metrics by reading out all file paths and source code and storing their pair metrics for faster execution.

References

Spadini, D., Aniche, M., & Bacchelli, A. (2018). Pydriller: Python framework for mining software repositories. In *The 26th acm joint european software engineering conference and symposium on the foundations of software engineering (esec/fse).* doi: 10.1145/3236024.3264598