2016 IEEE International Conference on Software Maintenance and Evolution

An Empirical Study on the Characteristics of Python Fine-Grained Source Code Change Types

Wei Lin, Zhifei Chen, Wanwangying Ma, Lin Chen*, Lei Xu, Baowen Xu*
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
luckylinwei@gmail.com, {chenzhifei, wwyma}@smail.nju.edu.cn, {lchen, xlei, bwxu}@nju.edu.cn

***Abstract*—Software has been changing during its whole life cycle. Therefore, identification of source code changes becomes a key issue in software evolution analysis. However, few current change analysis research focus on dynamic language software. In this paper, we pay attention to the fine-grained source code changes of Python software. We implement an automatic tool named PyCT to extract 77 kinds of fine-grained source code change types from commit history information. We conduct an empirical study on ten popular Python projects from five domains, with 132294 commits, to investigate the characteristics of dynamic software source code changes. Analyzing the source code changes in four aspects, we distill 11 findings, which are summarized into two insights on software evolution: change prediction and fault code fix. In addition, we provide direct evidence on how developers use and change dynamic features. Our results provide useful guidance and insights for improving the understanding of source code evolution of dynamic language software.**

Keywords- fine-grained change types; Python; software evolution

I. INTRODUCTION

Recently, software has totally pervaded into people's daily life, which promotes economic growth and social progress from many aspects. However, software has been continuously evolving over time due to the changes in the requirements and environments. Change is widely accepted as an essential part of modern software's life cycle, which threatens the quality, reliability, and maintainability of software. Therefore, understanding how software evolves over time can aid programmers in developing and maintaining reliable software systems.

Over the past decades, People often mine software repository to extract rich change information so as to analyze software evolution. For instance, Ying et al. or Zimmermann et al. applied data mining to version histories in order to guide programmers along related changes: "Programmers who changed these functions also changed…" [1][2]. Hassan used the complexity of code changes to predict faults [3].

Such research works are effective and valuable but suffer from the low quality of information available for changes. Hence, many scholars hammer at gaining higher level evolutionary information about software (see Section V for a detailed discussion). Research in this direction has already produced promising results, a famous one is the work of Software Evolution and Architecture Lab at the University of Zurich. They first focused on adding structural change information to existing release history data [4], and then built a taxonomy of source code changes that defined source code changes according to tree edit operations in Abstract Syntax Tree (AST) and classified each change type with a significance level [5]. In addition, to extract fine-grained source code changes, they presented a tree differencing algorithm [6], which can get fine-grained change information. Although many approaches exist to analyze software changes, few of them focus on the characteristics of fine-grained source code change types. However, understanding the nature of change types is essential, especially for dynamic programming language. A large amount of related works are concerned with traditional mainstream programing language, such as Java, C++, and C. But interests in dynamic object-oriented languages has increased recently, both in industry and in academia [7]. What's more, dynamic features bring developers flexibility but generally unpleasant to maintain and more error-prone. Therefore, we conduct an empirical study on Python, a typical dynamic programming language, to investigate the characteristics of fine-grained change types in four aspects. And the results will benefit future research in this field:

Benefit 1. The results will provide insights on the evolution tendency of dynamic language projects, including two perspectives: across projects and across versions. For example, the results will reveal the frequencies of different change types. As another example, the results will reveal the similarity of the distributions of change type frequency across versions of the same projects, providing evidences for change prediction.

Benefit 2. The results will provide insights on the patterns of change types and their extent of association with maintenance activities in dynamic language projects, especially bug fixing. For example, the results will reveal which change types are used to fix bugs more often. Automatic program repair tool could use the findings to prune its search space.

Benefit 3. The results will provide insights on the change of Python dynamic features and assessing their implications on important software maintenance activities. For example, the results will identify which dynamic feature changes most frequently during software evolution, and then summarize some interesting patterns associated with maintenance activities.

* corresponding author

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.25

Despite the previous benefits, conducting such an empirical study perfectly is difficult, because it is challenging to implement the automatic tool. Classifying changes in thousands of commits manually is infeasible and time-consuming, so it is desirable to implement a support tool for gaining change types. To address the challenge, we implement a tool, called PyCT, which can automatically extract and classify Python source code changes. With the support of PyCT, we conduct the empirical study to investigate the characteristics of fine-grained change types.

The remainder of this paper is organized as follows. In Section II we describe our experimental methodology. Section III presents the results of our empirical study, including some key findings and implications. In Section IV we discuss the threats to validity. We review the related work in Section V and finalize with our conclusions in Section VI.

II. Methodology

In this section, we first highlight our four research questions in part A. In order to answer these research questions, we present a python source code change classification scheme in part B, and then the automatic change extraction tool PyCT is introduced in part C. In part D and part E, we describe the dataset and analysis methods used in our study respectively.

A. Research Questions

RQ1. Are the distributions of change type frequency similar across different projects?

Different projects may be developed under different requirements and intended to accomplish various tasks. Do they show similar change tendency? If the distributions of change type frequency across projects are similar, we can rank the change types from most to least common and summarize general characteristics of change types. And then if a certain project change differently from others extremely, we should pay more attention to its design and code.

RQ2. Do the distributions of change type frequency vary substantially in different versions of the same project?

Practitioners will release a new version when they fulfill some tasks, such as supporting compatibility with a certain version of programming language or fixing a few bugs. We wonder whether the distributions of change type frequency are similar across versions and do some research to investigate this issue.

RQ3. Which change types are used more often in bug fixing?

In this paper, we divide the maintenance activities into two categories: bug-fix and non-bugfix. The non-bugfix category contains new features addition, refactor and other development behaviors not involved in fixing bugs. Some literatures have examined the relationship between change and bug-fix [8]–[11]. They focused on change types only in faulty codes. However, there is no general consensus on whether a certain change type is more related to bug-fix or non-bugfix? Consequently, it is necessary to investigate the characteristics of change types in different maintenance activities.

RQ4. How dynamic features are changed in software life cycle?

Dynamic features are useful constructs that bring developers convenience and flexibility, but they are also perceived to lead to difficulties in software maintenance. Our previous work conducted an empirical study on Python, which concluded that files with dynamic features are more change-prone and there is a positive relation between the number of dynamic features and change-proneness [12]. But it is unclear that how dynamic features change and how they deal with different maintenance activities. So in this paper, we investigate the characteristic of change types about dynamic features.

B. Fine-Grained Source Code Change Types

In this part, we introduce the taxonomy of Python source code changes. Our classification scheme is based on the taxonomy proposed by Fluri and Gall [5]. They distinguished between declaration and body part changes in objected oriented language, and defined atomic change types based on AST. Since ASTs are rooted trees and these source code entities are either sub-ASTs or leafs, the basis for source code changes are elementary tree edit operations, so the detection of source code changes falls into the tree edit distance problem.

But so far, this classification of changes has been conducted on Java source code only [5], we need to adjust the change descriptions to make it suitable for Python. For example, *final* modifier can be used in declaration part of Java function while it does not exist in Python. Furthermore, the current classification method has not considered changes on exception handlings yet, which will be added in our classification scheme. Our taxonomy contains 77 kinds of fine-grained change types. For better analysis and presentation, we group the change types which are similar in syntactic structures into one change category. For example, change types about *with-structure* and *raise-structure* will be classified into the same category: *Exception Handing*. After being grouped, there are eight main change categories in total, including *Class*, *Function*, *Statement*, *Selection Structure*, *Loop Structure*, *Exception Handling*, *Import*, and *Others*.

C. PyCT

To reduce the effort for change type extraction and classification, we develop an automatic tool called PyCT based on Change Distiller for quickly comparing Python source codes.

Change Distiller is an Eclipse Plugin which has been used to extract Java source code changes in many previous works [13]-[16]. In this plugin, Subsequent AST revisions of Java files are converted into generic tree data structures and compared to compute the elementary tree edit operations that transform an original tree into a modified one. Since our change type classification scheme (introduced in part B) is also based on AST, we adopt the Change Distilling algorithm to implement our PyCT by slightly adjusting the change type definition.

The framework of PyCT is showed in Fig. 1. For each project under Git control, we retrieve all the commits from its

TABLE I. DESCRIPTIVE INFORMATION OF THE STUDIED PROJECTS

Project	Domain	Number of commits	Number of change types in a commit		
			First Quartile	*Second Quartile*	*Third Quartile*
Django	Web	21882	1	3	9
Tornado	Web	3070	1	2	6
Pandas	Data processing	13211	2	4	8
Pylearn2	Data processing	7099	2	4	12
Numpy	Science computing	14040	1	2	6
Scipy	Science computing	14459	1	3	9
Sympy	Science computing	23666	2	4	16
Nltk	NLP	11777	2	4	14
Beets	Media	6128	2	4	15
Mopidy	Media	16962	1	3	6

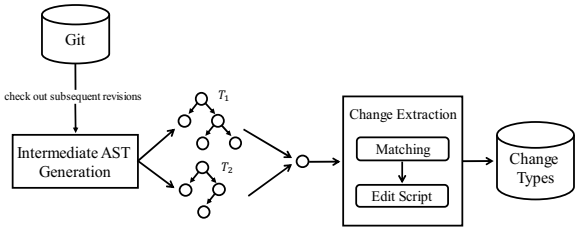

Figure 1. Fine-grained change extraction process

TABLE II. PYTHON DYNAMIC FEATURES OF FOUR CATEGORIES

Introspection		Object Changes	Code Generation	Library Loading
hasattr	isinstance	setattr	eval	__import__
getattr	issubclass	delattr	exec	reload
callable	type	del	execfile	
globals	vars			
locals	super			

source code repository and use *git-show* command to check out source code revisions. If a file is changed in a commit, we regard it in this commit is *version_current* and that in previous commit is *version_previous*. Namely, in our study, a revision is a set of source code changes done over one file in a commit. For two subsequent revisions of a Python file, ASTs are created using Python built-in *ast* module. And since the matching algorithm expects labeled and valued nodes as well as a uniquely defined parent-child relationship between hierarchically situated nodes, we create intermediate ASTs.

Then the two intermediate ASTs T_1 and T_2 are fed into our change extraction algorithm. The algorithm uses *bigram string similarity* to match source code statements and the *subtree similarity* of Chawathe et al. [17] to match source code structures and then compute the elementary tree edit operations. After the edit script generated, we classify them as change types according to the taxonomy of source code changes.

D. Dataset

To investigate the characteristics of change type, we study ten well-known open source Python projects. Table I summarizes the descriptive information for these projects. For each project, we filter the test files, which will change frequently but do not have tight relationships with program features. We use these projects to investigate the characteristics of change types across projects (RQ1). In RQ2, we studied two projects: (1) a set of Scipy versions, including v0.7.0, v0.7.1, v0.7.2, v0.9.0, v0.10.0, and v0.11.0; (2) a set of Pandas versions, including v0.15.0, v0.15.1, v0.16.0, v0.17.0, and v0.17.1. We choose these two projects to study the distributions of change type frequency across versions because of their detailed release logs. To study which change types behave differently between different maintenance activities (RQ3), we first need divide the maintenance activities into two categories: bug-fix and non-bugfix. One practical method is mining the commit message. Programmers may write a commit message to describe the fix behavior when they resolved some defects. For example, in Scipy, the message of a commit says, "Made change to fix global usage in exec_code and loop_code". So we determine a commit whose message contains the keywords "bug" or "fix" as the bug-fix commit and use the *git-grep* command to match them. With this method, we are able to find the bug-fix information of projects. For RQ4, we choose eighteen most often used and investigated dynamic features [7][12][18][19], as shown in Table II, which are classified as *Introspection*, *Object Changes*, *Code Generation* and *Library Loading*. If a changed statement contains one of the eighteen dynamic features, we record the change information.

E. Research Methods

To answer RQ1, we first calculate the frequency of each change type for ten Python projects. For the frequency of a certain change type Ct_i, the number of Ct_i is divided by the total number of change types. For example, in Django, the total number of change types in all commits is 205426, and the occurrence number of change type *Additional Class* is 19122, so the percentage of *Additional Class* in Django is 9.31% (19122/205426*100%). And then we employ the Wilcoxon sum rank test to examine whether the distributions of change type frequency are different significantly between every two projects. Wilcoxon rank sum test is a non-parametric statistical hypothesis test used to assess whether two samples' population mean ranks differ [20]. And difference is significant at 0.05 level. Furthermore, we group the ten projects into five domains according to their features. And we also employ Wilcoxon sum rank test to compare the

TABLE III. THE FREQUENCY OF EACH CHANGE TYPE ACROSS PROJECTS

Project Name	Domain	Change Types							
		Class	*Function*	*Statement*	*Selection Structure*	*Loop Structure*	*Exception Handling*	*Import*	*Others*
Django	Web	18.77%	20.29%	26.78%	5.28%	0.43%	2.95%	24.67%	0.82%
Tornado		4.17%	27.44%	31.05%	18.87%	1.18%	6.78%	8.01%	2.51%
average		11.47%	23.87%	28.92%	12.08%	0.81%	4.87%	16.34%	2.51%
Pandas	Data processing	2.87%	42.37%	22.57%	17.85%	1.09%	4.58%	5.14%	3.54%
Pylearn2		12.92%	21.67%	27.42%	6.80%	0.71%	2.30%	27.10%	1.05%
average		7.9%	32.02%	25.00%	12.33%	0.90%	3.44%	16.12%	2.30%
Numpy	Science computing	10.48%	31.11%	30.18%	9.66%	0.77%	2.70%	13.60%	1.51%
Scipy		7.54%	39.20%	27.25%	8.52%	0.39%	1.97%	13.52%	1.63%
Sympy		6.44%	46.28%	19.00%	9.20%	0.61%	1.69%	14.79%	1.98%
average		8.15%	38.86%	25.48%	9.13%	0.59%	2.12%	13.97%	1.71%
Nltk	NLP	16.63%	25.73%	25.42%	8.27%	0.79%	2.20%	19.56%	1.40%
average		16.63%	25.73%	25.42%	8.27%	0.79%	2.20%	19.56%	1.40%
Beets	Media	16.23%	22.49%	30.89%	6.84%	0.64%	1.76%	19.96%	1.21%
Mopidy		4.45%	31.46%	31.42%	12.95%	1.15%	4.26%	9.11%	5.21%
average		10.34%	26.98%	31.16%	9.90%	0.90%	3.01%	14.54%	3.21%

similarities of the distributions of change type frequency across domains.

In RQ2, we group the commits by release date to investigate whether the distributions of change type frequency are different significantly across versions. We calculate the frequency of each change type for every version and employ the Wilcoxon sum rank test in the same manner as RQ1.

In RQ3, we examine a certain change type Ct_i is more related to bug-fix behavior or non-bugfix behavior by using Fisher's exact test, which is used to examine the significance of the association between the two kinds of classification [21]. We first divide the commits into bug-fix category and non-bugfix category through keywords matching. And for each change type, the two kinds of commits are then divided into four groups, that is, (1) bug-fix commits contain this change type; (2) bug-fix commits do not contain this change type; (3) non-bugfix commits contain this change type; (4) non-bugfix commits do not contain this change type. We apply Fisher's exact test on these four groups and record the p-value and the odds ratio (OR). The p-value at 0.05 level indicates that the change type differs between bug-fix behavior and non-bugfix behavior, and then we inspect the value of OR. The OR is the ratio of the odds that bug-fix commits contain a certain change type, to the odds that non-bugfix commits contain this change type. If the OR is greater than 1, then means the commits having this change type are more related to bug-fix. While if the OR is less than 1, then indicates that this change type is more related to non-bugfix behavior. And the OR equals to 1 means that the change type behaves equally in both samples.

To investigate how dynamic features change in RQ4, we calculate the change frequency of each dynamic feature. For the change frequency of a certain dynamic feature, the number of commits involved with this feature is divided by the total number commits related to dynamic features. Furthermore, to find the most common change type about dynamic features, we group our fine-grained change types into three categories according to the elementary tree edit script, i.e., insert, delete and update, which is a different way from what illustrated in part B.

III. EXPERIMENTAL RESULTS

In this section, we present the experimental results in detail for each research question. For each of the four research questions, we decompose the results through a series of findings and implications.

A. *RQ1. Across projects*

In order to investigate this issue, we analyze the distributions of change type frequency of ten projects. Table III shows the frequency of each change type of the projects under study. The first two columns list the projects and their corresponding domains. The following eight columns present the frequencies of change categories in each project. The largest percentage and lowest percentage in each project is marked in yellow and grey respectively. Fig. 2 further depicts the distributions of change type frequency across all studied projects. We conduct research on this issue from two perspectives: (1) the frequency of change types; (2) the similarity among the distributions of change type frequency. The main results are as follows.

1) The frequency of change types

From Table III, we can see that the most common change type is *Function Change*, with an average percentage 30.80%, and *Statement Change* accounts for the second largest percentage (average percentage 27.20%). Meanwhile, *Loop Structure Change* is the most uncommon change type in all projects, with a percentage ranging from 0.39% to 1.18%. Besides *Function Change* and *Statement Change*, changes on *import* statement also account for a nice bit of percentage. After inspecting commit messages and the *diff* information between commits, we find that changes on *import* statement may be related to the stability of module dependency. For example, a commit in Scipy says that "Converted scipy to use numpy", and the corresponding changes are as follows:

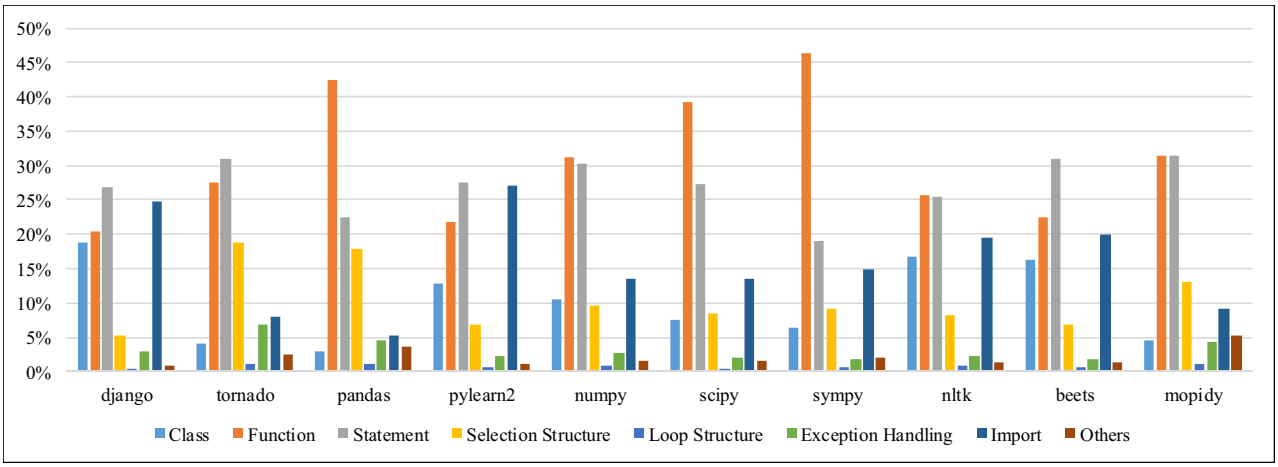

Figure 2. The distributions of change type frequency across projects

```
-   import scipy.base as Numeric
-   from scipy.distutils.core import setup
+   import numpy as Numeric
+   from numpy.distutils.core import setup
```

Finding 1: In most projects under study, *Function Change* and *Statement Change* are the most common change types, whereas *Loop Structure Change* is the most uncommon change type.

Implication 1: Practitioners should be more concerned with the stability of function. And if a programmer need modify loop structure frequently in software development, the code framework must be rethought.

2) The similarity among the distributions of change type frequency

Generally, we can observe similar distributions of change type frequency across the ten projects from Table III. That is to say, in spite of some slight differences, there are numerous characteristics of change types in common among the studied projects. For example, *Function Change* and *Statement Change* account for the majority in most studied projects. To further confirm this, we employ the Wilcoxon sum rank test to provide statistical evidence. And the p-values are all larger than 0.05, which shows there is no significant difference among the distributions of change type frequency across projects.

What's more, we calculate the average percentage of change types for each domain and the results can also be seen in Table III (the row called *average*). Likewise, we employ Wilcoxon sum rank test to examine the similarity across domains and the p-values are all larger than 0.05, which also shows no significant difference of the distributions of change type frequency across domains.

During software lifetime, changes are mainly performed for three purposes, i.e. adding new features, fixing bugs, and refactoring [22]. These change purposes will be the same in all kinds of projects, even if they belong to different domains. Besides, these projects are developed by the same programming language, the usage of syntactic structures may be similar. Therefore, it is likely that there are high similarities in the distributions of change type frequency across projects and across domains. But this conjecture need to be further confirmed in the future works.

Finding 2: The distributions of change type frequency share similar trends across studied projects.

Finding 3: There is no significant differences among the distributions of change type frequency across studied domains.

B. *RQ2. Across versions*

To investigate whether the distributions of change type frequency are similar in different versions, we study two groups of versions, one from Scipy, and the other from Pandas. The key observations are as follows.

For Scipy, the frequencies of eight main change categories across six versions are shown in Fig. 3. Fig. 3 reveals that the distributions of change type frequency vary a little across different versions. For example, the percentage of *Class Change* in v0.9.0 accounts for less than in others, while the percentage of *Exception Handling Change* accounts for much more. To further examine the similarity of distributions of change type frequency statistically, we adopt the hypothetical test method, Wilcoxon rank sum test, to compare the distributions of change type frequency of different versions in pairs. The statistical results show that the distribution of change type frequency in v0.7.0 is different from that in v0.7.1 and v0.7.2. The distribution of change type frequency in v0.9.0 also shows statistical differences from that in v0.7.0, v0.10.0, and v0.11.0. And the results of rest versions show no significant differences between each other. After inspecting the release notes of Scipy, we discover some rules: the versions whose number are 0.x.0 are main branches and all contain many new features, numerous bug-fixes, improved test coverage, and better documentation. The distributions of change type frequency are similar between v0.10.0 and v0.11.0 may because of the same release purpose. Whereas v0.7.1 and v0.7.2 are purely bug-fix releases with no new features compared to v0.7.0, which may result in their distributions of change type frequency are different from v0.7.0. As for v0.9.0, it is the first Scipy release to support Python 3. There are a certain degree of differences between

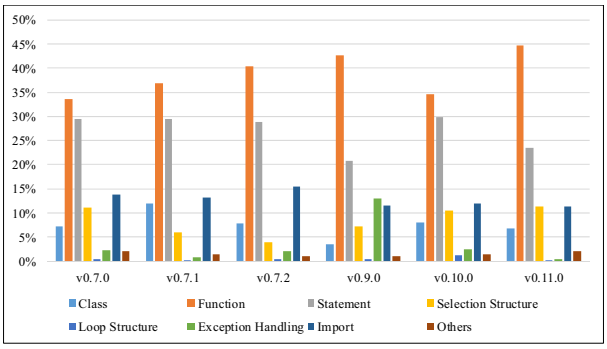

Figure 3. The frequency comparison between different versions of Scipy

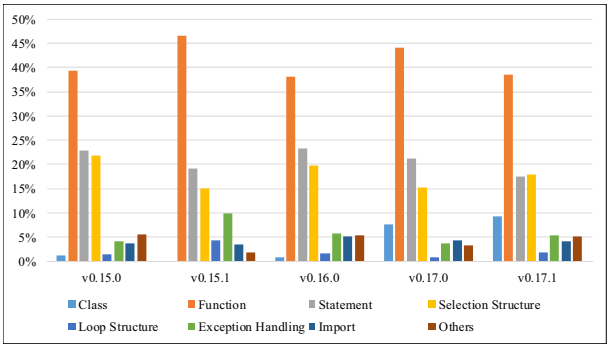

Figure 4. The frequency comparison between different versions of Pandas

Python 2 and Python 3, which may lead to the huge changes among v0.9.0 and others. For example, Python2 differs from Python3 in the *raise* syntax and this may lead to the *Exception Handling Change* accounts for more percentage in v0.9.0 than in others. The modified lines related to *raise* syntax in v0.9.0 maybe as follows:

```
- raise ValueError, "Number must be integer <= 1200."
+ raise ValueError("Number must be integer <= 1200.")
```

Finding 4: Considering project Scipy, the distributions of change type frequency are different significantly between bug-fix version and non-bugfix version.

Finding 5: Language evolution may have a huge impact on software evolution.

For Pandas, the frequency comparison results are presented in Fig. 4. For each version, changes on *function* and *statement* account for the largest and the second largest percentage respectively. Similar trends can be observed in other change types in spite of some subtle variance. The results of Wilcoxon sum rank test consistently shows that all the studied versions of Pandas have no significant difference on the distributions of change type frequency, as the p-values are all greater than 0.05. We also review the release notes of Pandas and find that all the releases have similar purposes, including some API changes, several new features, enhancements, and performance improvements along with a few bug fixes. That is to say, each version has similar release purpose and the distributions of change type frequency among these versions have no significant difference.

TABLE IV.
BUG-FIX RELATED CHANGE TYPES IN SCIPY AND MOPIDY

Scipy			Mopidy		
Change type	***P-value***	***Odds ratio***	***Change type***	***P-value***	***Odds ratio***
If Insert	0.0492	1.7811	Statement Update	0.0062	1.2590
Import Update	0.0165	1.9018	If Insert	0.0003	1.4706
			Import Insert	0.0001	1.4633
			Raise Update	0.0001	2.7377
			CE Update*	0.0006	1.4334

*Conditional Expression Update

TABLE V.
CHANGE TYPES BEHAVE DIFFERENTLY BETWEEN TWO KINDS OF MAINTENANCE ACTIVITIES

Bug-fix Related		Non-bugfix Related	
Change type	***Percentage of projects***	***Change type***	***Percentage of projects***
CE Update	6/10	Import Update	5/10
If Insert	6/10	Additional Function	5/10
		Function Renaming	5/10
		Parameter Insert	5/10

Finding 6: Considering project Pandas, the distributions of change type frequency across versions show no significant difference.

From the experimental results above, we can see that the way software evolves and changes has a lot to do with release purpose.

Implication 2: The distributions of change type frequency are closely related to the release purposes. Hence, we'd better use the previous versions which have similar release purposes as train data when predicting changes.

C. *RQ3. Maintenance activities*

As observed in RQ2, the distributions of change type frequency in pure bug-fix versions are different from others. Furthermore, will each specific change type be more related to bug-fix activity or non-bugfix activity? That is to say, we want to know how strongly a bug-fix commit is associated with the presence or absence of a change type. If quite a few bug-fix commits are associated with a certain change type, we have reasons to think this change type is used more in fixing bugs.

In order to investigate this issue, we apply Fisher's exact test on ten Python projects. A comprehensive overview of the test results of some common change types is given in

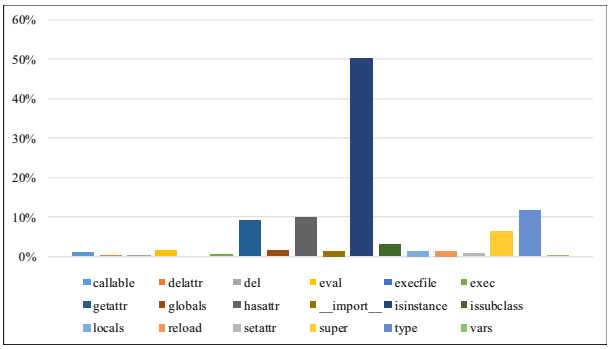

Figure 5. Change frequency of each dynamic feature

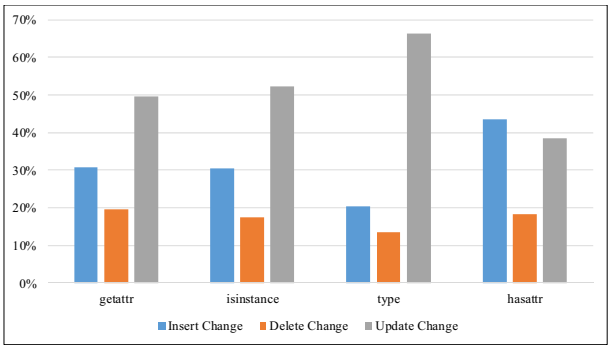

Figure 6. Frequencies of change types about dynamic features

Appendix A, the values in the table are odds ratios which are marked by * if their corresponding p-values are at 0.05 level. For illustration, we take Scipy and Mopidy as specific examples. Table IV presents the change types which are bug-fix related in Scipy and Mopidy, i.e., the change types with p-value less than 0.05 and odds ratio greater than 1. As can be seen, Scipy witnesses two change types that appear more frequently in bug-fix activity. But for Mopidy, there are five change types that are bug-fix related.

In order to weaken project dependency and make generalized conclusion, we calculate the number of projects involved in each change type. We consider a change type Ct_i bug-fix related if it appears as bug-fix related change type in over a half projects. Table V summarizes the bug-fix related and non-bugfix related change types among all studied projects.

For the bug-fix related change type, we can see from Table V that in six of the ten projects, *Conditional Expression Update* and *If Insert* are more related to bug fixing. This observation is consistent with Martinez's work [23]. Martinez et al. presented the abundance of eighteen bug-fix patterns from the analysis of six open source projects. Their experimental results show that the most frequent changes to fix bugs are changes in *if condition* statements.

For non-bugfix change type, Table V presents that *Import Update*, *Additional Function*, *Function Renaming*, and *Parameter Insert* are the most common non-bugfix change types. As stated above, the non-bugfix activity mainly involves adding new features and refactoring. When adding new functional module, we may gain access to the code in another modules by the process of importing them. For example, we need import *csv* module if we want our Python program to read or write a csv file. And it is intuitive that refactoring will be likely to involve in renaming the functions.

Finding 7: In our studied projects, *if* structure related change types are more related to bug-fix, especially *Conditional Expression Update* and *If Insert*.

Finding 8: Our experimental results show that *import* statement and *function* structure change types are mainly related to non-bugfix activity, especially *Import Update*, *Additional Function*, *Function Renaming*, and *Parameter Insert*.

Implication 3: When fixing bugs, practitioners should pay more attention to *if* structure, especially check the conditional expression.

D. ***RQ4.*** *Dynamic feature*

In order to investigate the issue that how dynamic features are changed, we study eighteen famous dynamic features in Python from the following three perspectives: (1) dynamic features change most frequently; (2) the most common change type about dynamic features; (3) dynamic features and maintenance activities. The main results are presented as follows.

1) Dynamic features change most frequently

As shown in Fig. 5, in all commits involved with dynamic features, commits contain *isinstance* account for over a half percentage, and commits contain *type* account for the second largest, with a percentage of 11.77%. The two dynamic features account for the third and fourth largest percentage are *hasattr* and *getattr* respectively. Other features change rarely in our studied projects. These top four dynamic features all belong to *Introspection* category, a mechanism used to examine the state of an object or the local scope at runtime [18]. As we all know, dynamic language lack a static type system and is popular for their flexibility, expressivity, and succinctness. Not having to support a static type system frees dynamic languages up for including powerful dynamic feature. But type is important in a way, non-static type system will result in weaker performance and safety [18]. So that we need to check the type and take different actions according to object's type sometimes in Python. For example, in Django commit *bdca5ea*, the original code and modified code are as follows:

```
@@ -288, 9 +289, 9 @@
-    if type(s) == str:
-       s = s.decode('utf-8')
-    elif type(s) != unicode:
-       raise TypeError(s)
+    if type(s) == bytes:
+       s = s.decode('utf-8')
+    elif type(s) != six.text_type:
+       raise TypeError(s)
```

Finding 9: In our studied projects, *isinstance*, *type*, *hasattr*, and *getattr* are the top four dynamic features that change frequently.

2) The most common change type about dynamic features

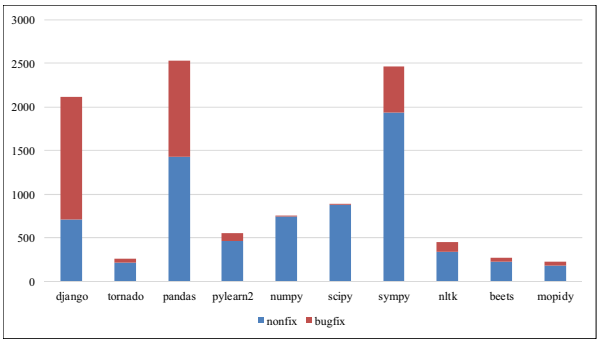

Figure 7. Changes containing dynamic features

Since other dynamic features were somewhat less likely change in projects, we choose the top four dynamic features that change frequently in *Finding 9* to investigate this issue, i.e. *isinstance*, *type*, *hasattr*, and *getattr*.

We group our 77 fine-grained change types into three categories according to the elementary tree edit script. Fig. 6 presents the average frequency of change types about four dynamic features. As shown in Fig. 6, *Update* is the most common change type in most studied dynamic features, with a percentage ranging from 37.60% to 65.26%. Only in terms of *hasattr*, the most frequent change type is *Insert*, accounting for 43.38% on average, which a little more than the percentage of *Update* (38.40%). What's more, insertions about dynamic features on a code element are more than deletions. When using dynamic features, careless programmers may use them improperly, may forget some statements, or may add unwanted statements. Our results show that the first case is more common than the other two.

Finding 10: The change actions on dynamic features follow the pattern that *Update* is the most common type, followed by *Insert* and *Delete* respectively.

3) Some discussion about dynamic feature changes

Continuing the study of RQ3, we also want to know how dynamic features are changed to deal with different maintenance activities. How many changes containing dynamic feature usage are related to bug fixing, and how many are related to other activities?

Finding 11: Among the changes containing dynamic features in the studied projects, about 23% are related to bug-fix, while 77% are related to non-bugfix.

From Fig.7 we can see that in total, about 23% of changes containing dynamic features are bug-fix changes, while about 77% are non-bugfix changes. We manually reviewed the changes and found some interesting patterns.

Pattern 1: *Type widening.* In this pattern, developers use dynamic features to ensure that the program can accept some data with widen types. For example, the commit *3ac0961* of Django shows, developers use *isinstance* to ensure that the type of the variable is *basestring* instead of *str*, thus fixing a bug.

```
-   if setting in tuple_settings and type(setting_
    value) == str:
-       setting_value = (setting_value)
+   if setting in tuple_settings and isinstance
    (setting_value, basestring):
+       setting_value = (setting_value)
```

Pattern 2: *Type addition.* In this pattern, developers use dynamic features to ensure that the variable could be types other than the original ones. For example, developers use *isinstance* to accept both strings and promises from gettext_lazy in the commit *e4e28d9* of Django.

```
-   assert isinstance(message, basestring), ("%s
    should be a string" % repr(message))
-   assert isinstance(message, basestring), ("'%s'
    should be a string" % message)
+   assert isinstance(message, (basestring,
    Promise)), ("%s should be a string" %
    repr(message))
+   assert isinstance(message, (basestring,
    Promise)), ("'%s' should be a string" %
    message)
```

Pattern 3: *Attribute guarantee*. In this pattern, developers want to guarantee that some object has particular attributes in the context so that they can do something special with the object. For example, Scipy developers used *hasattr* to guarantee attributes in the commit *40a4a36*.

```
@@ -1174,12 +1175,10 @@
-   try:
-       del self.sampleF
-       del self.samplelogprobs
-       del self.sample
-   except AttributeError:
-       pass
+   for var in ['sampleF, samplelogprobs,
     sample']:
+       if hasattr(self, var):
+           exec('del self.' + var)
```

In these type-aware context, developers want to keep variables flowing into the context belonging to particular types, so they use dynamic features to aid the task. However, the flexibility of dynamic features may come with problems. Researchers and developers may pay more attentions on how to use dynamic features correctly.

Implication 4: Developers had better to be particularly concerned about the type and attributes of objects while coding by dynamic language. And when using dynamic features, they should pay more attention on how to use them correctly than adding or deleting them.

IV. Threats to Validity

In this section, we discuss the main threats to the construct, internal, and external validity of our study.

A. Threats to construct validity

The most important threat to construct validity in our experiments is the accuracy of the identification of bug-fix commits. Kim et al. point out that it needs high quality bug-fix information to reduce superficial conclusions, but many bug-fixes are polluted [24]. We distinguish bug-fix commits through key words matching, such heuristic makes a strong assumption on the development process and the developer's behavior: it assumes that developers generally put syntactic features in commit texts enabling to recognize repair actions, which is not really true in practice [25]–[27]. That is to say,

not all commits contain words such as “bug” or “fix” in their messages fixing bugs. For instance, in Mopidy, the message of a commit says “docs: Add bug fix to changelog”. This message can be matched by key words “fix” and “bug”, but cannot be classified as a bug-fix commit in fact. Although a number of previous studies (e.g. [10], [24],[28]) used the same technique to extract bug-fix information, it may still not be absolutely perfect. However, this inaccuracy may not have a great effect on our conclusion. We have manually inspected the commit message of some projects and found that the number of polluted commits can be accepted.

B. Threats to internal validity

Internal validity threats in our study mainly concern the limitations inherited from Change Distilling. Although Change Distilling has a great improvement compared to previous technique [17], there are still some deficiencies. For example, the best match approach may match reoccurring statements that are not at the same positon in the method body. Such mismatches can have, as in some particular cases, tremendous impact on the extraction of other changes.

C. Threats to external validity

Threats to external validity mainly concern the possibility to generalize our findings. In this study, we studied ten widely used Python projects and exhausted our ability to cover various domains. Although we believe our results can reveal the characteristics of change types well in many projects, we do not intend to draw general conclusions on all Python software, because of all the projects we studied are open source. Consequently, our findings may not be generalized to commercial software. More comprehensive and general results still require further case studies and a more wide variety of projects in the future works.

V. Related Work

A. Source code changes

Change is broadly accepted as a crucial part of software evolution. Consequently, an increasing number of works have studied software changes. Omori and Maruyama proposed a mechanism for recording all editing operations a developer has applied to source code on an integrated development environment, which has a practical use from the viewpoint of its performance [29]. Canfora et al. empirically investigated the relationships between the complexity of a software system and factors that could influence it, measured using source code change entropy [22]. Rastkar and Murphy proposed the use of multi-document summarization techniques to generate a concise natural language description of why code changed so that a developer can choose the right cause of action [30]. As for fine-grained software modifications, Daniel proposed several metrics to quantify modification requests (MRs) and use these metrics to create visualization graphs that can be used to understand the inter-relationships[31].

B. Taxonomy of change type

Many existing literatures introduced change type classification schemes or list change types as a base of their analysis. Li and Offut presented a change type taxonomy which contains add, delete, and accessibility changes of classes, methods, and attributes, as well as value-change operations of attributes [32]. Xing and Stroulia developed an approach to enable the model-based differencing of UML class models, with the overall goal of detecting class co-evolution patterns [33]–[35]. Another taxonomy of change types introduced in the work of Feng and Maletic distinguished between atomic changes and composite changes [36]. In this paper, we choose the taxonomy proposed by Fluri and Gall [5], they distinguished between declaration and body part changes in object oriented software and built the change types on an analysis of elementary tree edit operations. Furthermore, they also introduced the concept of change significance.

C. Change type study

Besides taxonomy, Fluri et al. also discovered the patterns of change types [37], they explored whether change types appear frequently together in time and whether they describe specific development activities. Sun et al. proposed a static CIA technique to calculate the impact of each change type [38]. Giger et al. explored prediction models for whether a source file will be affected by a certain type of source code change [39]. Another research about change type is done by Kidwell et al. [40], they extended the change taxonomy developed by Fluri and Gall [5] to analyze software faults. Romano developed a tool called WSDLDiff to perform a study aimed at analyzing the evolution of web services using the fine-grained changes extracted from the subsequent versions of four real world WSDL interfaces [41], which is similar to our study in a way. As for the relationship between change type and bug-fix, Thung et al. proposed a combination of machine learning and code analysis techniques to identify root cause from the changes made to fix bugs [42]. Martinez presented a new mechanism to formalize bug-fix patterns based on change types and focus on the abundance of bug-fix patterns [23].

VI. Conclusion

Identification of fine-grained source code changes is a key issue in software evolution analysis. In this paper, we choose Python, a dynamic language which is widely used but rarely studied, to do some empirical research so as to investigate the characteristics of fine-grained change types. We first introduce a taxonomy of Python source code changes and then implement an automatic tool PyCT based on Change Distiller to reduce the effort for change extraction and classification.

In order to explore the characteristics of change types, we study ten widely used open source Python projects with thousands of commits and observe the following key findings: (1) across projects: The distributions of change type frequency show no significant difference across most studied projects and the most common change type are *Function Change* and *Statement Change* while the most uncommon change type is *Loop Structure Change*; (2) across versions: The similarities of the distributions of change type frequency across versions may rely on the release purposes; (3) between maintenance activities: In our studied projects, *if* structure related change types are more likely to be used for fixing bugs, while *import*

statement and *function* structure related change types are more associated with non-bugfix; (4) dynamic features: The dynamic features that change frequently in our studied projects are *isinstance*, *type*, *hasattr*, and *getattr*. What's more, the most common change action on these four features is *Update*.

We believe our observations will improve the understanding of Python software evolution and hence aid practitioners in developing high quality software systems.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments and constructive suggestions. This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (91418202, 61472178, 61472175, 61432001, 61272080) All support is gratefully acknowledged.

REFERENCES

[1] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history,"IEEE Trans. Softw. Eng., vol. 30, no. 9, pp. 574–586, 2004.

[2] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," IEEE Trans. Softw. Eng., vol. 31, no. 6, pp. 429–445, 2005.

[3] A. E. Hassan, "Predicting faults using the complexity of code changes," Proc. - Int. Conf. Softw. Eng., pp. 78–88, 2009.

[4] B. Fluri, H. C. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," Proc. - Fifth IEEE Int. Work. Source Code Anal. Manip. SCAM 2005, pp. 66–74, 2005.

[5] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," IEEE Int. Conf. Progr. Compr., vol. 2006, pp. 35–45, 2006.

[6] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," IEEE Trans. Softw. Eng., vol. 33, no. 11, pp. 725–743, 2007.

[7] Laurence Tratt, "Dynamically Typed Languages," Adv. Comput., vol. 77, pp. 149–184, 2009.

[8] S. Kim, T. Zimmermann, K. Pan, and E. James Whitehead, "Automatic identification of bug-introducing changes," Proc. - 21st IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2006, pp. 81–90, 2006.

[9] H. Osman, M. Lungu, and O. Nierstrasz, "Mining Frequent Bug-Fix Code Changes," Csmr-Wcre, pp. 343–347, 2014.

[10] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," Empir. Softw. Eng., vol. 14, no. 3, pp. 286–315, 2009.

[11] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," IEEE Int. Conf. Progr. Compr., pp. 70–79, 2009.

[12] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, "An empirical study on the impact of Python dynamic features on change-proneness," 27th Int. Conf. Softw. Eng. Knowl. Eng., pp. 134–139, 2015.

[13] Z. Xing, Y. Xue, and S. Jarzabek, "Distilling useful clones by contextual differencing," Proc. - Work. Conf. Reverse Eng. WCRE, pp. 102–111, 2013.

[14] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," 2011 33rd Int. Conf. Softw. Eng., pp. 351–360, 2011.

[15] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," ACM SIGPLAN Not., vol. 45, no. 10, p. 302, 2010.

[16] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactoring and software defects," Proc. - Int. Conf. Softw. Eng., pp. 35–38, 2008.

[17] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," ACM SIGMOD Rec., vol. 25, no. 2, pp. 493–504, 1996.

[18] B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad, "Tracing dynamic features in python programs," Proc. 11th Work. Conf. Min. Softw. Repos. - MSR 2014, pp. 292–295, 2014.

[19] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of python applications," Conf. Res. Pract. Inf. Technol. Ser., vol. 91, no. Acsc, pp. 17–25, 2009.

[20] F. Wilcoxon, "Individual Comparisons by Ranking Methods," vol. 1, no. 6, pp. 80–83, 1945.

[21] R. Fisher, "Statistical methods for research workers", no. V. 1925.

[22] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How changes affect software entropy: An empirical study", vol. 19, no. 1. 2014.

[23] M. Martinez, L. Duchien, and M. Monperrus, "Accurate Extraction of Bug Fix Pattern Occurrences using Abstract Syntax Tree Analysis," 2014.

[24] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," 2011 33rd Int. Conf. Softw. Eng., pp. 481–490, 2011.

[25] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced?: Bias in Bug-Fix Datasets," Proc. ESEC/FSE, pp. 121–130, 2009.

[26] R. Wu, H. Zhang, S. Kim, and S. C. Cheung, "ReLink: Recovering links between bugs and changes," Proc. ACM SIGSOFT Symp. Found. Softw. Eng., pp. 15–25, 2011.

[27] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, "A Machine Learning Approach for Text Categorization of Fixing-issue Commits on CVS," Proc. 2010 ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas., pp. 6:1–6:10, 2010.

[28] S. Kim, T. Zimmermann, E. J. Whitehead, and A. Zeller, "Predicting faults from cached history," Proc. - Int. Conf. Softw. Eng., pp. 489–498, 2007.

[29] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," Proc. MSR, no. January, pp. 31–34, 2008.

[30] S. Rastkar and G. C. Murphy, "Why did this code change?," Proc. - Int. Conf. Softw. Eng., pp. 1193–1196, 2013.

[31] D. M. German, "An empirical study of fine-grained software modifications," Empir. Softw. Eng., vol. 11, no. 3, pp. 369–393, 2006.

[32] L. Li and a J. Offutt, “Algorithmic analysis of the impact of changes to object-oriented software,” Proc. Int. Conf. Softw. Maint. ICSM-96, pp. 171–184, 1996.

[33] Z. Xing and E. Stroulia, “Data-mining in Support of Detecting Class Co-evolution,” SEKE ’04 Proc. 16th Int. Conf. Softw. Eng. Knowl. Eng., pp. 123–128, 2004.

[34] E. Stroulia, “Understanding class evolution in object-oriented software,” Proceedings. 12th IEEE Int. Work. Progr. Comprehension, 2004., pp. 34–43, 2004.

[35] Z. Xing and E. Stroulia, “UMLDiff: An Algorithm for Object-oriented Design Differencing,” Proc. 20th Int. Conf. Autom. Softw. Eng., pp. 54–65, 2005.

[36] T. Feng and J. I. Maletic, “Applying Dynamic Change Impact Analysis in Component-based Architecture Design,” Proc. Seventh ACIS Int. Conf. Softw. Eng. Artif. Intell. Networking, Parallel/Distributed Comput., pp. 43–48, 2006.

[37] B. Fluri, E. Giger, and H. C. Gall, “Discovering patterns of change types,” ASE 2008 - 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng. Proc., pp. 463–466, 2008.

[38] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, “Change impact analysis based on a taxonomy of change types,” Proc. - Int. Comput. Softw. Appl. Conf., pp. 373–382, 2010.

[39] E. Giger, M. Pinzger, and H. C. Gall, “Can we predict types of code changes? An empirical analysis,” IEEE Int. Work. Conf. Min. Softw. Repos., pp. 217–226, 2012.

[40] B. Kidwell, J. H. Hayes, and A. P. Nikora, “Toward extended change types for analyzing software faults,” Proc. - Int. Conf. Qual. Softw., pp. 202–211, 2014.

[41] D. Romano and M. Pinzger, “Analyzing the evolution of web services using fine-grained changes,” Proc. - 2012 IEEE 19th Int. Conf. Web Serv. ICWS 2012, pp. 392–399, 2012.

[42] F. Thung, D. Lo, and L. Jiang, “Automatic recovery of root causes from bug-fixing changes,” Proc. - Work. Conf. Reverse Eng. WCRE, pp. 92–101, 2013.

Fisher's exact test results (*OR*) of Some Common Change Types (RQ3) [a]

A.

Change Types \ Projects	Django	Tornado	Pandas	Pylearn2	Numpy	Scipy	Sympy	Nltk	Beets	Mopidy
Additional Class	1.1642*	0.2803*	0.5298*	0.9421	0.0000	0.0000	0.9101	1.0084	0.9613	0.6543*
Removed Class	1.0606	0.4133*	0.5266*	1.0350	0.0000	0.0000	0.9499	1.0584	1.0856	0.4126*
Class Renaming	0.8109	0.5725	0.3455*	0.5547	0.0000	0.0000	0.7926	0.9813	0.4270	0.3262
Additional Function	1.2195*	0.5429*	0.6564*	0.8924	2.4965	0.6510	0.8990*	1.0310	0.8273*	0.5887*
Removed Function	1.0530	0.4899*	0.6972*	1.1051	2.9344*	0.5622	0.9422	1.0693	0.8743	0.6432*
Function Renaming	0.6343*	0.5931	0.5849*	0.3813*	0.0000	0.0000	0.7716*	0.7315	0.4859	0.9369
Statement Insert	1.4137*	0.6665*	0.9156	0.8464*	0.8102	0.6577	1.0883*	1.1322	0.9232	0.7062*
Statement Delete	0.9650	0.7895	0.8595*	0.7910*	0.2766	0.6210	1.0133	1.1240	0.9972	0.8142
Statement Update	1.1413*	0.7284*	0.9706	1.3847*	0.8377	0.5617*	1.1758*	1.1238	0.9890	1.2590*
If Insert	1.8878*	1.0875	1.2348*	1.0095	0.9503	1.7811*	1.1980*	1.3461*	1.2104	1.4706*
If Delete	1.1770*	0.7517	1.0784	0.8378	1.6607	1.2623	1.2100*	1.0516	0.8085	0.8968
Conditional Expression Update	1.1245*	1.4421*	1.4644*	1.0600	3.4061*	0.6274	1.1700*	1.1257	0.9329	1.4334*
Else_Part Insert	1.5523*	0.6141	1.2450*	1.1592	3.8501	1.2192	1.2192*	1.5117*	0.8068	0.6770
Else_Part Delete	1.4138*	0.5923	1.1504	0.7145	2.2670	1.5448	1.1455*	1.4233	0.6152	0.7236
Parameter Insert	1.2433*	0.5655*	0.7731*	0.6946*	0.0000	0.7578	0.8351*	0.9472	0.5139*	0.8470
Parameter Delete	0.5143*	0.6150	0.6166*	0.5628*	0.0000	0.0000	0.9812	0.8673	0.4930*	0.7398
Parameter Renaming	2.0188*	1.6350	0.4665*	0.9214	0.0000	0.7679	1.1737	1.0027	0.6058	1.5209
Import Insert	1.0701	0.6476*	0.7792*	0.8057	0.2085	0.4933	0.8603*	0.9005	0.8300*	1.4633*
Import Delete	0.8537	0.5636*	0.5466*	0.9994	0.3883	0.1370*	0.9607	1.1409	1.0055	0.7723
Import Update	0.7082	1.3022	0.5977*	1.0089	0.6657	1.9018*	0.8819*	0.9364	0.8631	0.7680*
Try Insert	1.3810*	1.7748*	0.9982	0.9923	0.0000	0.0000	1.0384	1.1856	0.8236	1.0999
Try Delete	1.1325	0.7488	1.0605	1.0077	0.0000	0.0000	1.0500	1.2555	1.1051	1.3886
Try Update	1.1871*	0.9001	1.3017*	1.3355	0.0000	1.5090	1.1277	1.5188*	0.9395	0.8447
Raise Insert	3.6763*	0.4056	0.8161	1.2532	0.0000	0.0000	0.7807	3.7442*	0.0000	2.0360
Raise Delete	2.0037*	9.7944	1.5189	0.4685	0.0000	0.0000	0.8003	8.0136*	3.8657	0.0000
Raise Update	1.0424	3.8205*	0.8842	0.7197	0.0000	1.4105	1.1053	1.1266	0.8802	2.7377*

a. The p-value at 0.05 level is marked by *, the odds ratio less than 1 is marked with an underline and greater than 1 is marked by gray.