Prioritizing code documentation effort: Can we do it simpler but better?

Appendix A. Role of code documentation in software quality assurance

Code documentation is a key component of software quality assurance. Good code documentation can greatly promote the understanding of software system, accelerate the process of learning and reusing code, increase developer productivity, simplify maintenance, and therefore improve the reliability of software [1-6]. In contrast, poor code documentation is one of the main reasons for the rapid deterioration of software system quality [4]. Therefore, code documentation is an irreplaceable necessity to enhance software reliability. To sum up, code documentation plays a fundamental role at least in the following areas of software quality assurance.

(1) Program Comprehension. Code documentation is an important aid for program comprehension during software development and maintenance [7-9]. It is a common strategy for programmers to understand project code starting with code documentation [10]. Combined code related design documentation can help participants achieve significantly better understanding than using only source code [11]. For example, many programmers wrote comments (a form of code documentation) to actively record the technical debt [12] in the code itself (that is, mark the test, improvement, and fix to be completed in the code with comments, also known as self-admitted technical debt [13]) to assist the subsequent software understanding and maintenance.

(2) Test case generation. Test case generation is among the most labor-intensive tasks in software testing. Because code documentation written by tabular expressions is precise, readable, and can clearly express the intended behavior of the code, such documentation documents are widely used in the test case generation [14-18], which makes evaluation of test results inexpensive and reliable. In general, they can be used to generate oracle [19] used to determine whether any test results (input and output pairs) meet the specification.

(3) API Recommendation. Application Programming Interfaces (APIs) are a means of code reuse. The goal of API recommendation techniques is to help developers perform programming tasks efficiently by selecting the required API from a large number of libraries with minimal learning costs. Clearly, API documentation (a typical type of code documentation, such as Javadoc[1]) is an important source of information for programmers to learn how to use API correctly [20]. In practice, API documentation is widely used in API recommendation [21-26]. By analyzing the similarity between words in API documentation and code context or natural language words in programming tasks, the accuracy of API recommendation can be enhanced.

(4) Bug detection. Bug detection techniques have been shown to improve software reliability by finding previously unknown bugs in mature software projects [27, 28]. Bug detection based on code comments is one of the most extensively studied bug detection techniques [29-31]. For a function or API, developers often write comments (natural language type or Javadoc type) to indicate the usage. An inconsistency between comments and body of a function indicates either a defect in the function or a fault in the comment that can mislead the function callers to introduce defects in their code. Bug detection based on code comments is to search for such inconsistencies to find bugs in software.

(5) Program repair. Automated program repair (APR) is a technique for automatically fixing bugs by generating patches that can make all failure test cases pass for a buggy program. Although APR has great potential to reduce bug fixing effort, the precision of most previous repair techniques is not high [32-35]. For a defect, often hundreds of plausible patches are generated, but only one or two are correct. In order to improve the precision of APR, code documentation have been applied in this field recently. For example, the literature [36] used Javadoc comments embedded in the source code to guide the selecting of patches. As a result, a relatively high precision (78.3%) is achieved, significantly higher than previous approaches [35, 37-39].

The above-mentioned works have a direct contribution on enhancing software reliability, and it can be seen that these works heavily depend on the code documentation. Undoubtedly, if there are high-quality code documentations, the effectiveness and efficiency of many quality assurance activities could be greatly improved.

1 https://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

Appendix B. Other performance comparison results

Table B1 Performance comparison

(a) Precision under the top 5%

Project	Precision				
	SCM	SCM_FS	VSM	VSM_FS	PageRank
NanoXML	0.02(N)	0.01(N)	0.03(N)	0.03(N)	0
JExcelAPI	0.3(S)	0.18(S)	0.35(M)	0.46(M)	0
JGraphT	0.27(M)	0.31(M)	0.38(S)	0.26(L)	0.5
Ant	0.02(L)	0.03(L)	0.03(L)	0.12(M)	0.21
JHotDraw	0.08(L)	0.08(L)	0.1(L)	0.15(S)	0.2
ArgoUML	0.05(L)	0.08(L)	0.04(L)	0.11(L)	0.22
jEdit	0.04(L)	0.06(L)	0.03(L)	0.07(L)	0.23
JMeter	0.12(M)	0.1(L)	0.18(S)	0.18(S)	0.27
Wro4j	0.1(L)	0.06(L)	0.13(L)	0.28(N)	0.27

(b) Recall under the top 5%

Project	Recall				
	SCM	SCM_FS	VSM	VSM_FS	PageRank
NanoXML	0.01(N)	0(N)	0.01(N)	0.01(N)	0
JExcelAPI	0.07(S)	0.04(S)	0.09(M)	0.13(M)	0
JGraphT	0.05(M)	0.06(M)	0.07(S)	0.05(L)	0.09
Ant	0.05(L)	0.1(L)	0.13(L)	0.48(M)	0.76
JHotDraw	0.27(L)	0.27(L)	0.35(L)	0.51(S)	0.62
ArgoUML	0.16(L)	0.26(L)	0.18(L)	0.37(L)	0.75
jEdit	0.15(L)	0.24(L)	0.12(L)	0.26(L)	0.84
JMeter	0.11(M)	0.09(L)	0.16(S)	0.16(S)	0.25
Wro4j	0.16(L)	0.09(L)	0.23(L)	0.48(N)	0.46

(c) F_1 under the top 5%

Project	F1				
	SCM	SCM_FS	VSM	VSM_FS	PageRank
NanoXML	0.01(N)	0(N)	0.01(N)	0.01(N)	0
JExcelAPI	0.11(S)	0.06(S)	0.14(M)	0.19(M)	0
JGraphT	0.08(M)	0.09(M)	0.11(S)	0.08(L)	0.15
Ant	0.03(L)	0.05(L)	0.05(L)	0.19(L)	0.32
JHotDraw	0.11(L)	0.12(L)	0.15(L)	0.22(S)	0.29
ArgoUML	0.07(L)	0.12(L)	0.07(L)	0.16(L)	0.33
jEdit	0.06(L)	0.09(L)	0.04(L)	0.11(L)	0.34
JMeter	0.11(M)	0.09(L)	0.16(S)	0.17(S)	0.25
Wro4j	0.11(L)	0.07(L)	0.16(L)	0.33(N)	0.32

(d) ER under the top 5%

Project	ER				
	SCM	SCM_FS	VSM	VSM_FS	PageRank
NanoXML	0.01(N)	0(N)	0.02(N)	0.02(N)	0
JExcelAPI	0.22(S)	0.13(S)	0.25(M)	0.33(M)	0
JGraphT	0.19(M)	0.21(M)	0.3(S)	0.17(L)	0.4
Ant	0.15(L)	0.25(L)	0.29(L)	0.67(M)	0.87
JHotDraw	0.4(L)	0.41(L)	0.6(M)	0.75(S)	0.85
ArgoUML	0.4(L)	0.48(L)	0.45(L)	0.67(L)	0.92
jEdit	0.27(L)	0.4(L)	0.21(L)	0.43(L)	0.89
JMeter	0.32(M)	0.27(L)	0.46(S)	0.46(S)	0.62
Wro4j	0.39(L)	0.27(L)	0.52(L)	0.75(N)	0.81

Appendix C. Discussions

In this section, we analyze the influence of various factors on the effectiveness of the PageRank approach. First, for the old data sets, we analyze the utility of classes in the whole project. Then, we analyze the influence of the weights of dependence relationships among classes. Third, we observe whether the PageRank approach is superior to a simple unsupervised approach. Fourth, we observe whether the PageRank approach is superior to supervised approaches after rebalancing. The third and fourth observations help us further analyze the effectiveness of the PageRank approach.

C.1. The utility of classes in the whole project

In the result of Fig. 4 in Section 5.1 in our paper[2], our analysis reveals that the effectives of the PageRank approach on libraries in the old data sets is affected by the number of classes used for experiments. In the old data sets, the proportions of classes scored by graduate students in the three libraries are as follows: NanoXML (75%), JExcelAPI (11%), and JGraphT (91%). As can be seen, many classes in these three libraries were not scored. Because the PageRank approach uses the dependence relationships among classes to evaluate the importance of classes, it is natural that many dependence relationships are missing when only a part of classes in a library are used. Therefore, the true performance of the PageRank approach is suppressed, especially for the JExcelAPI library.

To observe the influence of the number of classes, we design a new PageRank approach named “PageRank_allClasses”, which uses all classes in a library to construct the dependence graph and recalculate the importance score of each class. Taking the JExcelAPI library as an example, the difference between the original PageRank approach in RQ1 and the PageRank_allClasses approach is that: the original PageRank approach uses 50 classes scored by graduate students to calculate the importance scores of each classes and then obtains their rankings. In contrast, the PageRank_allClasses approach first uses all the classes (i.e., 458) to calculate the importance scores of each classes and then extracts the classes that have actual category labels (i.e. 50 classes scored by users) to observe their rankings. The experimental settings and steps are the same as those in Section 6.

Fig. C1 shows a comparison of precision, recall, and F_1 between the original PageRank approach and the PageRank_allClasses approach on the old data sets. It can be observed that these two PageRank approaches have little difference on the NanoXML and JGraphT libraries. This is expected as the number of classes scored by graduate students in McBurney et al.’s study [40] is close to the total number of classes in each library. However, on the JExcelAPI library (which has low proportion of classes scored by graduate students in McBurney et al.’s study [40]), the effectiveness of the PageRank_allClasses approach has been greatly improved compared with the original PageRank approach.

The above result indicates that the number of classes used for the experiment in a library would affect the performance of the PageRank approach. The more classes are missing, the greater the influence on the PageRank approach will be. This is the reason why the original PageRank approach does not perform well on the JExcelAPI library (as shown in RQ1 and RQ2 in Section 5). As a result, we suggest that if a future code documentation effort prioritization study employs users (such as graduate students) to score the importance of a library, all the

2 Prioritizing code documentation effort: Can we do it simpler but better? Status: under review.

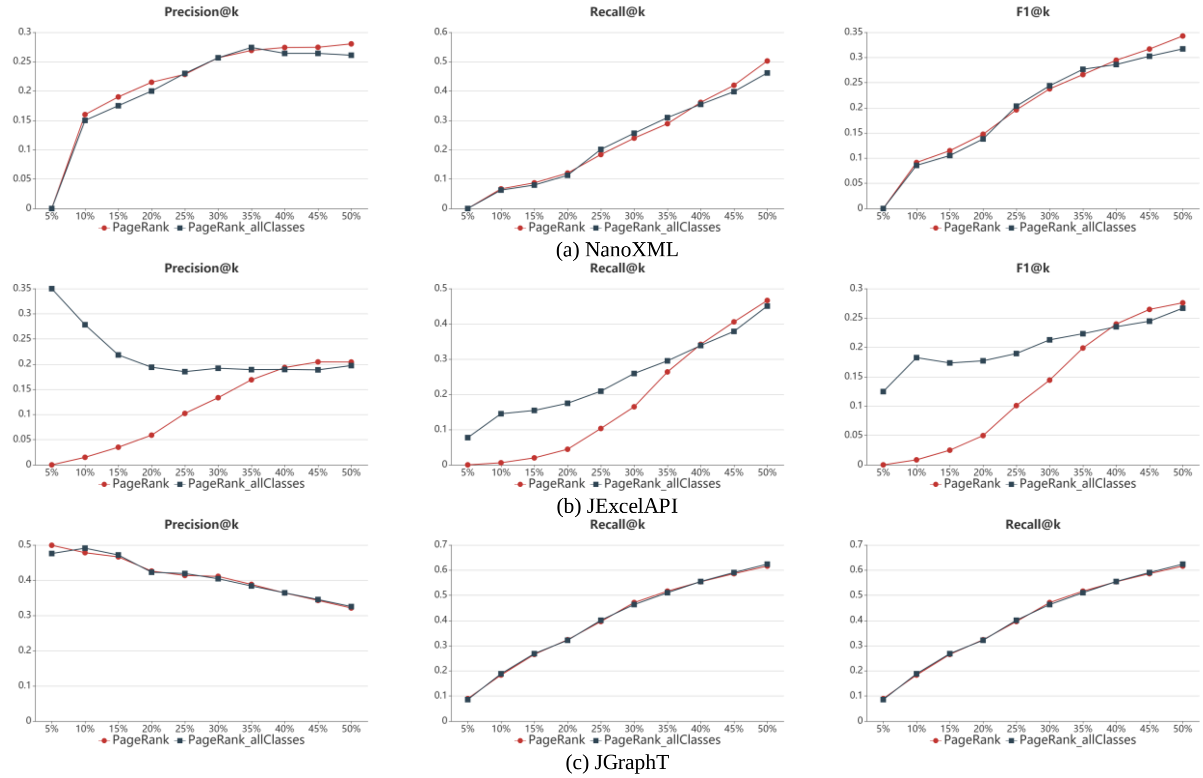


Fig. C1. Comparison of precision, recall, and F_1 between two types of PageRank approaches on the old data sets

classes in the project (library or application) should be scored (if it is cannot rate all modules, at least all the modules that participants have used or seen). Otherwise, due to the insufficient representativeness of the scored classes, the resulting conclusion based on them may be biased.

C.2. The influence of weights of dependence relationships

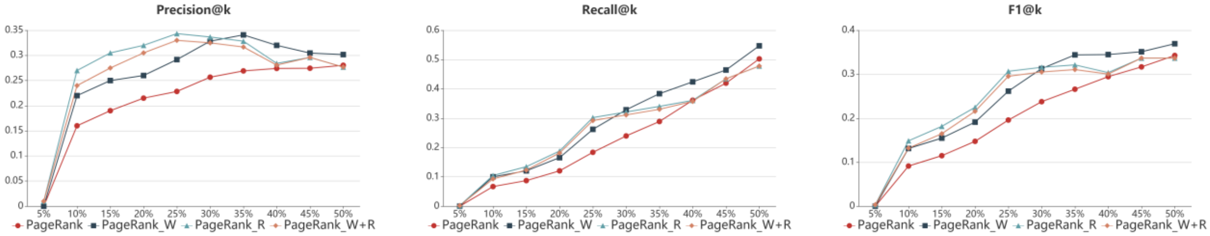
As shown in equation (2) in our paper, for each class on a given project (library or application), the resulting *PR* (i.e. class importance) depends on L_u and $TL(v)$. Since L_u and $TL(v)$ are based on the edge weight in the inter-module class dependence graph, we know that, in nature, they depend on the weight of the four types of dependences: CI, CA, CM, and MM. As shown in Section 3 of our paper, in the PageRank approach, we assign the same weight to these four types of dependences. In other words, for the simplicity of computation, we do not distinguish their contributions when computing class importance.

In the following, we investigate the influence of weights of dependence relationships. Following the work in [41], we use the following two methods to assign the weights to dependence relationships:

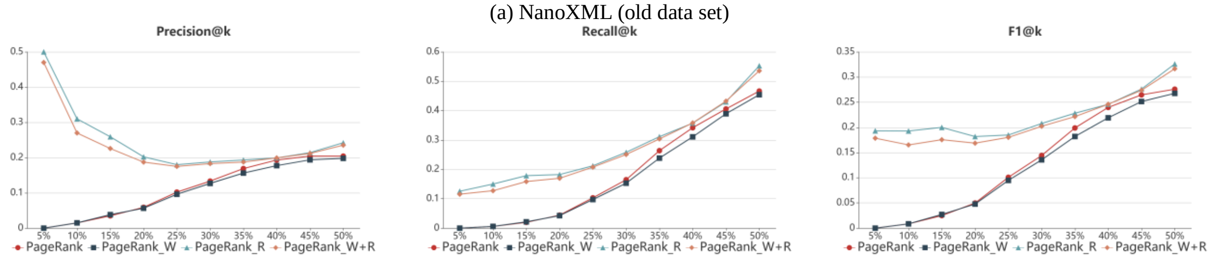
- Empirical weight. Literature [41] believed that different dependences had different contributions when computing class importance and hence assigned different multiplication coefficients to different dependences when expressing the weight of edges. We call this improved method of assigning weights empirical weighting. Referring to the settings in literature [41], we assign the multiplication coefficient (3, 3, 2, and 4) to the four types of dependence relationships (CI, CA, CM and MM) used in this paper. In this way, the equation of weight:
  W(u, v) = CI(u, v)+CA(u, v)+CM(u, v)+MM(u, v)
  is changed to:
  W(u, v) = 3CI(u, v)+3CA(u, v)+2CM(u, v)+4MM(u, v).
- Back recommendation. In literature [41], they call the edge from A to B a forward recommendation and the edge from B to A a back recommendation. In particular, “the weight of the forward recommendation from A to B is given by the dependency strength of the cumulated dependencies from A to B. The weight of the back recommendation from B to A is a fraction F of the weight of the forward recommendation from A to B” [41]. Let the weight matrix of the class dependence graph only using forward recommendation be *R*, then the weight matrix of the class dependence graph adding back recommendation be $R + \frac{1}{F} \times R^T$. Here, *T* represents the matrix transpose. The class dependence graph corresponding to equation (2) has only “forward recommendation” edge, i.e. the weight matrix is *R*. (Note: the definition of forward recommendation edge in [41] is the same as the Out-Edge in Section 3.2). Therefore, when we combine “backward recommendation” to improve the weight matrix *R*, *R* is changed to $R + \frac{1}{2} \times R^T$, where $\frac{1}{2}$ is the best value of $\frac{1}{F}$ reported in literature [41].

For the simplicity of presentation, we use “PageRank_W” to denote the PageRank approach using empirical weights, use “PageRank_R” to denote the PageRank approach using back recommendation, and use “PageRank_W+R” to denote the PageRank approach using both.

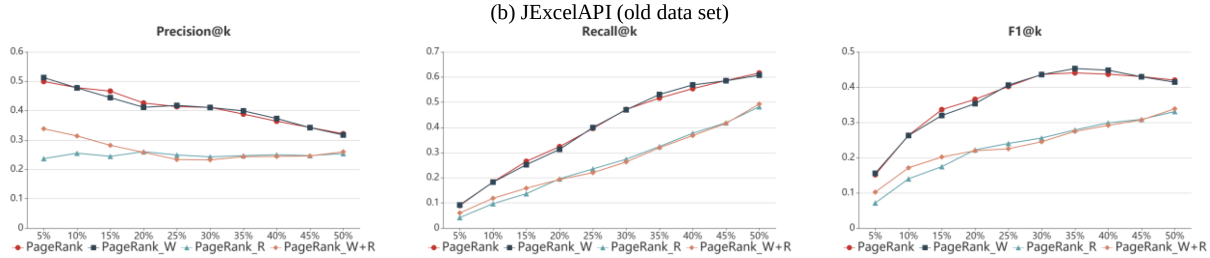
We repeat the experimental steps in Section 5 to obtain the



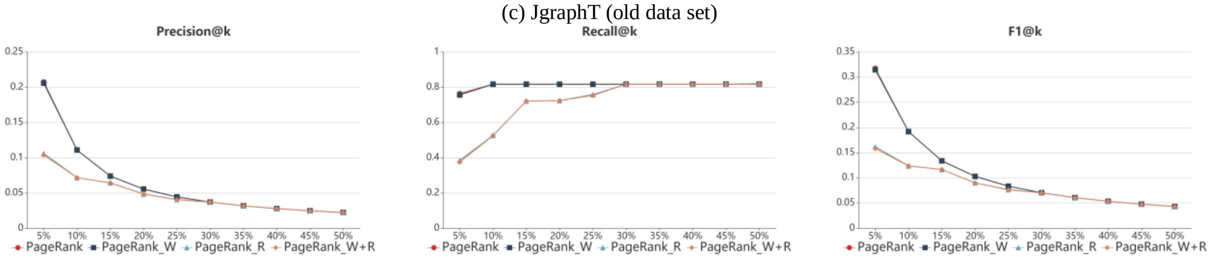
(a) NanoXML (old data set)



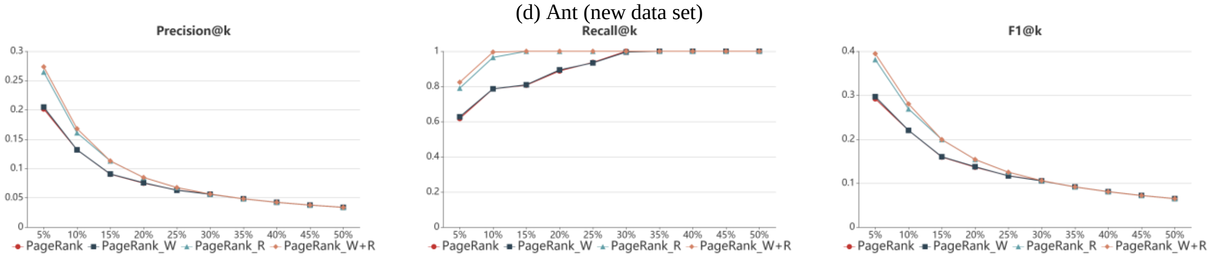
(b) JExcelAPI (old data set)



(c) JgraphT (old data set)



(d) Ant (new data set)

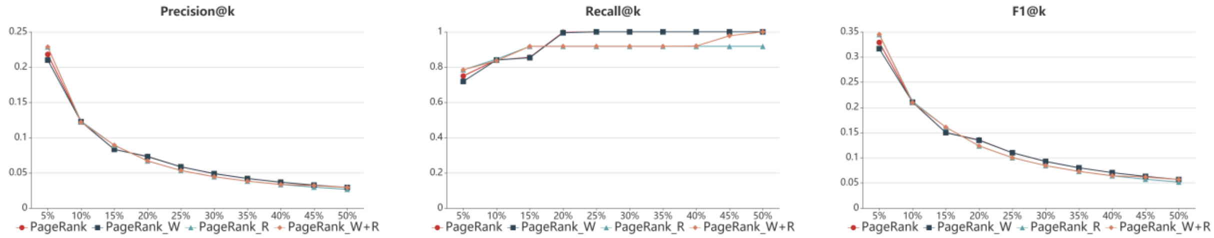


(e) JHotDraw (new data set)

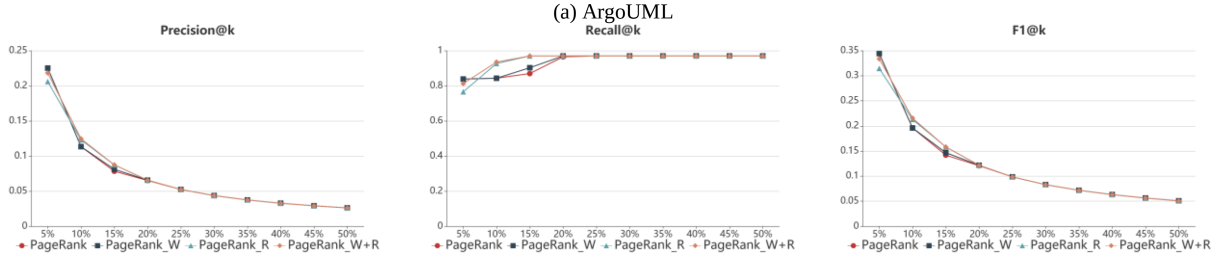
Fig. C2. Comparison of precision, recall and F_1 between the original PageRank model and three improved PageRank models on API libraries

results of “PageRank_W”, “PageRank_R”, and “PageRank_W+R”. Fig. C2 and C3 show the comparisons of precision, recall, and F_1 among these four PageRank approaches. From these figures, we make the following observations. First, PageRank_W is close to PageRank and PageRank_W+R is close to PageRank_R on almost all projects (i.e., all libraries or applications, except PageRank_W is evidently not close to PageRank on the NanoXML library). This shows that empirical weights have less impact on the PageRank model than back recommendation. Second, compared with the original PageRank approach, PageRank_R and PageRank_W+R, which use the back recommendation, show completely different effectiveness on different projects (libraries or applications): some are obviously improved (e.g. NanoXML, JExcelAPI, JHotDraw, and JMeter), some are obviously decreased (e.g. JGraphT and Ant), and some are little changed (e.g. ArgoUML, jEdit, and Wro4j).

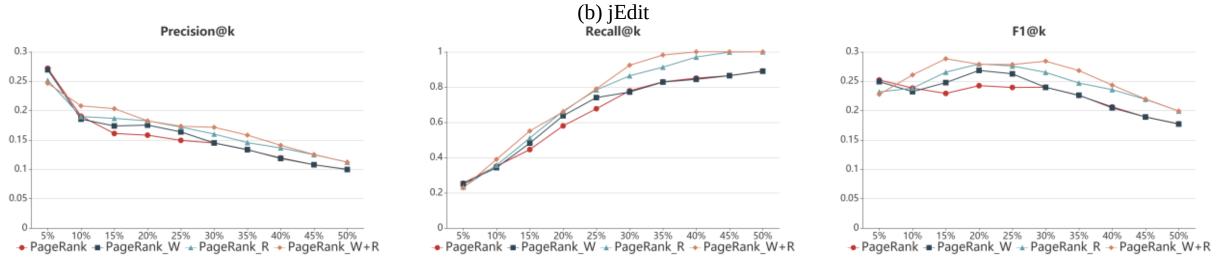
Therefore, for the PageRank approach used in this paper, the influence of empirical weights is relatively small, and the influence of back recommendation is relatively large. Since empirical weights and back recommendation do not always improve or decrease PageRank performance, we suggest that these two improvement methods for PageRank should be used cautiously in practice.



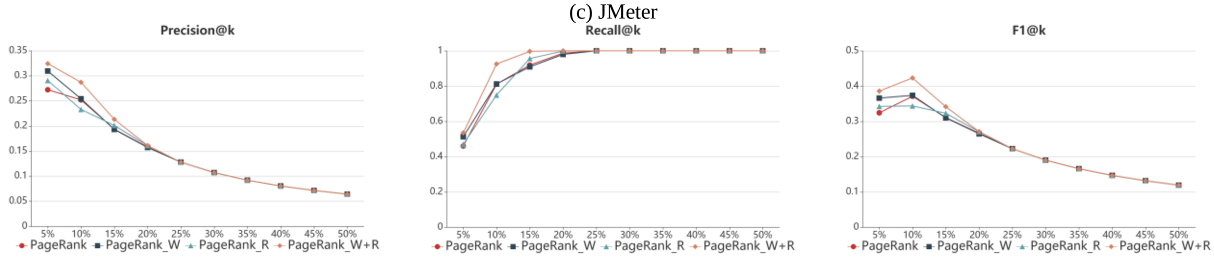
(a) ArgoUML



(b) jEdit



(c) JMeter



(d) Wro4j

Fig. C3. Comparison of precision, recall and F_1 between the original PageRank model and three improved PageRank models on applications in the new data sets

C.3. The comparison of PageRank and supervised approaches after rebalancing

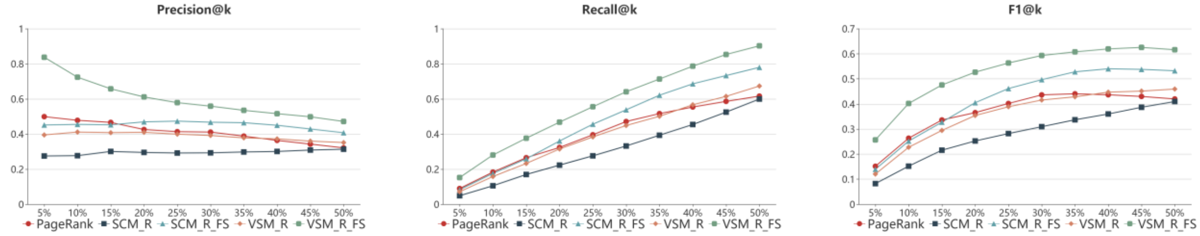
As stated in Analysis 2 of RQ1 (Section 5.1), we do not use the rebalancing technique for supervised ANN approaches for three reasons (the original authors of supervised ANN approaches we compared did not use the rebalancing technique [40], the rebalancing technique may have negative effects [42], and the comparison results also show that ANN approaches are not suitable for imbalanced data sets (if no specific processing is done)).

However, considering that data sets are highly imbalanced (especially new data sets), the effectiveness of supervised approaches is likely to be hidden by this imbalance. Naturally, the following questions arise: is the PageRank approach superior to supervised approaches after rebalancing? Exploring this question is helpful to further understand the effectiveness of the PageRank approach.

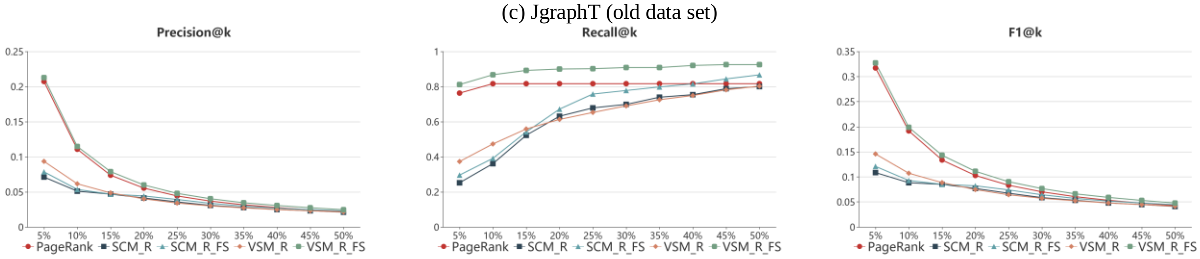
We apply SMOTE (synthetic minority over-sampling technique) [43] to rebalance the imbalanced training data, as it is considered “de facto” standard in the framework of learning from imbalanced data [44] and the most influential data preprocessing/sampling algorithm [45]. For the simplicity of presentation, we use (SCM_R, VSM_R, SCM_R_FS, and VSM_R_FS) to represent the four supervised approaches (SCM, VSM, SCM_FS, and VSM_FS) after rebalancing.

We repeat the experimental steps in Section 5 to obtain the results of four supervised approaches (SCM_R, VSM_R, SCM_R_FS, and VSM_R_FS). It should be noted that, as described in Sections 5, C.2 and C.3, the effectiveness of the PageRank approach has suffered from a number of factors on the NanoXML and JExcelAPI projects. Therefore, in this section, we exclude these two projects to avoid unnecessary interference with the experimental analysis. We only compared the PageRank approach to supervised approaches for the remaining large projects.

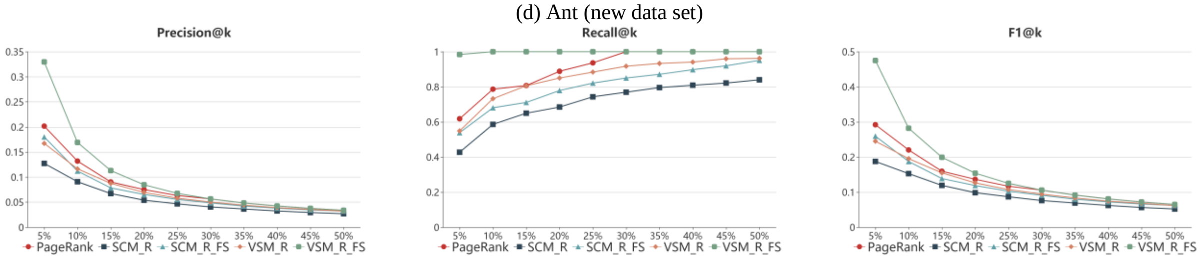
Fig. C4 and C5 show the comparisons of precision, recall, and F_1 among these four supervised approaches after rebalancing and the PageRank approach. From these figures and Fig. 4 and 5 in RQ1 (Section 5.1), we make the following observations. First, almost all four supervised approaches show improved performance after rebalancing compared to without rebalancing (RQ1). Only on the JgraphT project, the performance of SCM_R show decreased performance after rebalancing. This shows that the rebalancing technique is



(c) JgraphT (old data set)



(d) Ant (new data set)



(e) JHotDraw (new data set)

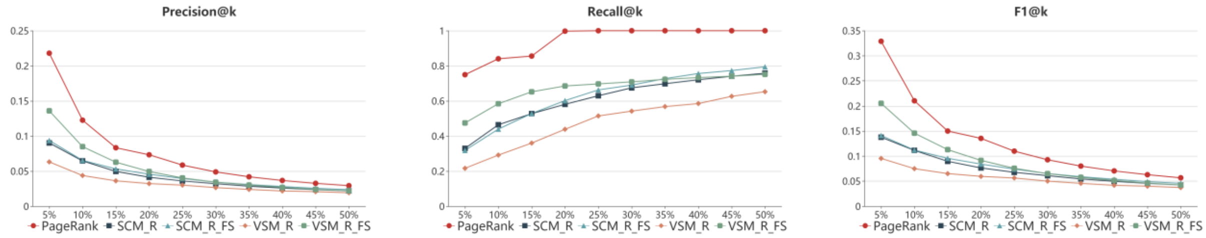
Fig. C4. Comparison of precision, recall and F_1 between the PageRank model and supervised models after rebalancing on the API libraries ("@k" denotes the corresponding results of the model when k takes different thresholds)

beneficial to supervised approaches for documentation effort prioritization. Second, except VSM_R_FS, the effectiveness of the other three supervised approaches (SCM_R, VSM_R, and SCM_R_FS) after rebalancing is still significantly lower than that of the PageRank approach, especially at small thresholds (5% and 10%). Third, after rebalancing, the effectiveness of VSM_R_FS is significantly higher than that of PageRank on four projects (JgraphT, JHotDraw, JMeter, and Wro4j), and is significantly lower than that of PageRank on one project (ArgoUML), and is comparable to that of PageRank on two projects (Ant and jEdit). Combining the second and third observations, the PageRank approach perform best or second best on most projects, so the PageRank approach still does not lose its qualification as a good baseline.

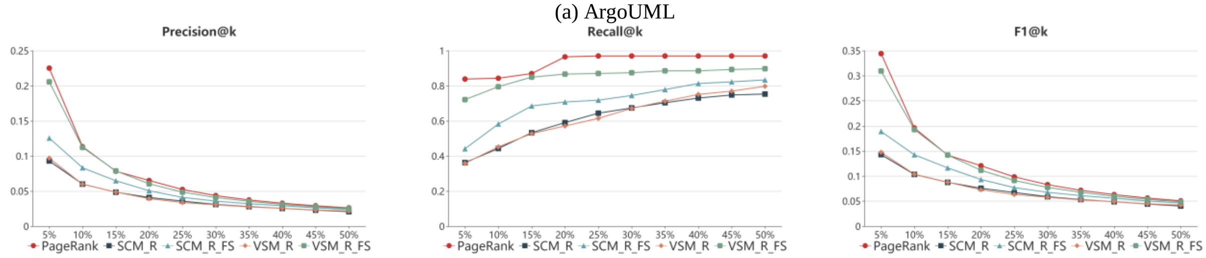
In addition, we have an interesting finding that VSM_R_FS is significantly more effective than PageRank at small thresholds (5% and 10%) when the recall of PageRank is not good enough (the recall is less than or close to 0.6). This seems to indicate that there is some kind of complementary relationship between the PageRank approach and the VSM_R_FS approach. This means that when the PageRank score fails, we can use the tf/idf (term frequency–inverse document frequency) in the module as a substitute for identifying the important module. We analyze the causes for this complementarity because two factors play a dominant role in the program understanding. One factor is the direct and indirect dependences between modules, as we analyzed in Section 6. Another factor is the meaning of words or relationships between words in the module. Intuitively, programmers need to look to the meanings or relationships of words in a module to understand the functionality provided by the module itself. When the former factor plays a dominant role, the PageRank approach is more effective. When the latter factor plays a dominant role, the VSM_R_FS approach is more effective.

Although the performance of the PageRank approach is worse than that of the VSM_R_FS approach on some projects, the PageRank approach still has the following two significant advantages compared with the VSM_R_FS approach. (1) The PageRank approach is unsupervised and does not need to collect training sets. In practice, label collection of training sets may be time-consuming or difficult. If there is no labeled training set, there is no way to apply supervised approaches, such as for a new project. (2) The time cost of PageRank approach is much lower than that of supervised approaches (e.g., VSM_R_FS). Supervised approaches need to build models on training data before they can be used for prediction (calculation). The PageRank approach is unsupervised, no modeling is required, and the computation time is negligible. Note: for the metric collection time, the PageRank approach also takes less time than supervised approaches, because the number of features (only 4) extracted by PageRank is less than that of supervised approaches (for VSM_R_FS, thousands of features (words) are required).

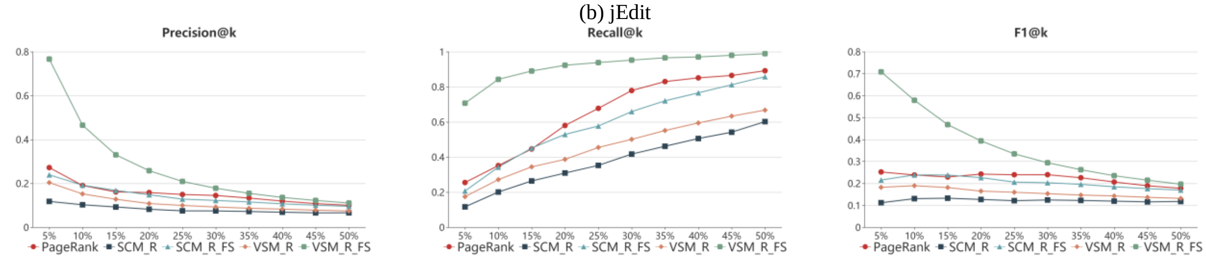
Table C2 shows the modeling (calculation) cost of the PageRank approach and four rebalanced supervised approaches under 100 bootstrap samples. As can be seen from Table C2, the time cost of supervised approaches are hundreds to hundreds of thousands of times that of the PageRank approach.



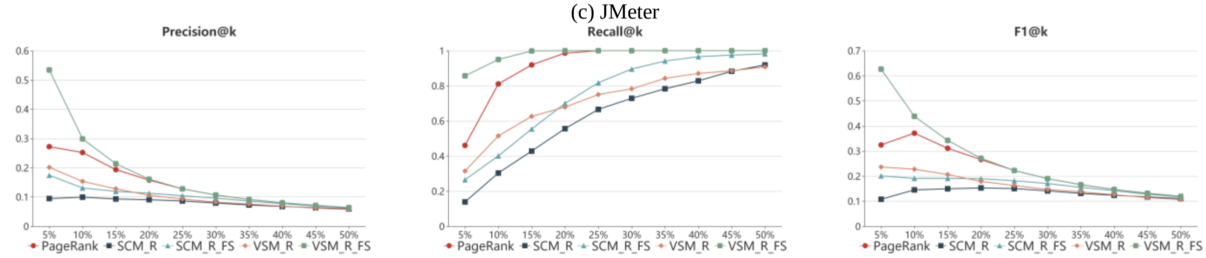
(a) ArgoUML



(b) jEdit



(c) JMeter



(d) Wro4j

Fig. C5. Comparison of precision, recall and F_1 between the PageRank model and supervised models after rebalancing on applications in the new data sets ("@k" denotes the corresponding results of the model when k takes different thresholds).

Table C2 Comparison of modeling (calculation) time (unit: second)

Project	SCM_R	SCM_R_FS	VSM_R	VSM_R_FS	PageRank
nanoxml-2.2.1	6.5	7.9	6.4	18.4	0.002
jexcelapi-2.6.12	29.5	20.5	27.3	96.3	0.002
jgrapht-0.9.1	116.4	77.3	242.8	715.2	0.004
ant-1.6.1	3.5	3.4	904.2	1555.0	0.018
argouml-0.9.5	2.2	3.1	1391.6	3213.8	0.014
jedit-5.1.0	4.1	4.6	979.8	2692.1	0.009
jhotdraw-6.0b.1	3.0	2.9	399.8	934.4	0.009
jmeter-2.0.1	37.4	124.7	214.1	365.0	0.005
wro4j-1.6.3	54.5	218.5	501.1	504.8	0.007

Taken together, the PageRank approach is still a good baseline, especially considering that PageRank approach does not need the collection of training sets and the time cost is minimal.

Appendix D. Metric table

Table D1 List of **S**tatic source **C**ode **M**etrics (SCM) and **T**extual **C**omparison **M**etrics (TCM) in McBurney et al.'s study [40]

Type	Metric	Description	Tools for measuring metrics
SCM: Size	LOC	Number of lines of code including comments but not empty lines	SourceMonitor[1]
	Statements	Number of executable code statements	
SCM: Complexity	%Branch	Branch statements account for percentage of statements	
	Calls	Number of statements for method calls	
	Calls Per Statement	Number of statements for method calls / Statements	
	Methods Per Class	Average number of methods for each class	
	Statements Per Method	Average number of statements contained in each method	
	Avg. Depth	Average number of branch layers nested in a function	
	Max Depth	Maximum number of branch layers nested in a function	
	Avg. Complexity	Average McCabe Cyclomatic Complexity of methods	
	Max Complexity	Maximum McCabe Cyclomatic Complexity of methods	
	WMC	The sum of McCabe Cyclomatic Complexity of methods per class	Metrics[2]
	NOF	Number of fields of a class	
SCM: Object Oriented	DIT	Number of ancestor classes a given class has	Metrics[2]
	NSC	Number of children classes a given class has	
	LCOM	Lack of cohesion of methods	
	NORM	Number of methods in a class overridden by its child classes	
	Abstract	A class is or is not an abstract class	
SCM: Others	%Comments	Annotated line account for percentage of all lines	SourceMonitor[1]
TCM	Class Appearance	The class name appears or doesn't appear in the two bodies of text	Scripts[4]
	Package Appearance	The package name appears or doesn't appear in the two bodies of text	
	Combination Appearance	The class and package name appears or doesn't appear in the two text	
	First Overlap[3] metric	Overlap similarity that words with splitting on camel case.	
	Second Overlap[3] metric	Overlap similarity that words without splitting on camel case.	
	First STASIS[3] metric	STASIS similarity that words with splitting on camel case.	
	Second STASIS[3] metric	STASIS similarity that words without splitting on camel case.	

1. SourceMonitor is a tool for collecting static source code metrics, which can be downloaded from: http://www.campwoodsw.com
2. Metrics is an Eclipse plugin to extract object-oriented metrics or complexity metrics.
3. Overlap and STASIS are textual and semantic similarity metrics. For specific definitions, please refer to the paper [40].
4. TCM are collected by McBurney et al.'s script [40], in which the script that collect the STASIS metric can be accessed from http://www.cis.upenn.edu/~paulmcb/research/doceffort/.

References

[1] B.J. Bauer, D.L. Parnas. Applying mathematical software documentation: an experience report. 10th Annual Conference on Computer Assurance, COMPASS (1995), pp. 273-284.

[2] J. Kotula. Source code documentation: an engineering deliverable. 34th Technology of Object-Oriented Languages and Systems, TOOLS (2000), pp. 505.

[3] M. Visconti, C.R. Cook. An overview of industrial software documentation practice. 22nd International Conference of the Chilean Computer Science Society, SCCC (2002), pp. 179-186.

[4] L.C. Briand. Software documentation: How much is enough? 17th European Conference on Software Maintenance and Reengineering, CSMR (2003), pp. 13.

[5] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. Empirical Software Engineering, 10(1) (2005), pp. 31-55.

[6] M.A. Oumaziz, A. Charpentier, J. Falleri, X. Blanc. Documentation reuse: Hot or not? An empirical study. International Conference on Software Reuse, ICSR (2017), pp. 12-27.

[7] F. Lehner. Quality control in software documentation based on measurement of text comprehension and text comprehensibility. Information processing & management, 29(5) (1993), pp. 551-568.

[8] U. Dekel, J.D. Herbsleb. Reading the documentation of invoked API functions in program comprehension. International Conference on Program Comprehension, ICPC (2009), pp. 168-177.

[9] H.H. Schoonewille, W. Heijstek, M.R.V. Chaudron, T. Kühne. A cognitive perspective on developer comprehension of software design documentation. 29th ACM international conference on Design of communication, SIGDOC (2011), pp. 211-218.

[10] T. Roehm, R. Tiarks, R. Koschke, W. Maalej. How do professional developers comprehend software? IEEE/ACM International Conference on Software Engineering, ICSE (2012), pp. 255-265.

[11] C. Gravino, M. Risi, G. Scanniello, G. Tortora. Do professional developers benefit from design pattern documentation? A replication in the context of source code comprehension. International Conference on Model Driven Engineering Languages and Systems, MoDELS (2012), pp. 185-201.

[12] W. Cunningham. The WyCash portfolio management system. ACM Sigplan Oops Messenger, 4(2) (1992), pp. 29-30.

[13] A. Potdar, E. Shihab. An exploratory study on self-admitted technical debt. International Conference on Software Maintenance and Evolution, ICSME (2014), pp. 91-100.

[14] D.K. Peters, D.L. Parnas. Generating a test oracle from program documentation (Work in Progress). International Symposium on Software Testing and Analysis, ISSTA (1994), pp. 58-65.

[15] D.K. Peters, D. L. Parnas. Using test oracles generated from program documentation. IEEE Transactions on Software Engineering, 24(3) (1998), pp. 161-173.

[16] M. Clermont, D.L. Parnas. Using information about functions in selecting test cases. ACM SIGSOFT Software Engineering Notes, 30(4) (2005), pp. 1-7.

[17] X. Feng, D.L. Parnas, T.H. Tse. Tabular expression-based testing strategies: A comparison. Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION, TAICPART-MUTATION (2007), pp. 134-134.

[18] X. Feng, D.L. Parnas, T.H. Tse, T. O'Callaghan. A comparison of tabular expression-based testing strategies. IEEE Transactions on Software Engineering, 37(5) (2011), pp. 616-634.

[19] E.J. Weyuker. On testing non-testable programs. The Computer Journal, 25(4) (1982), pp. 465-470.

[20] M.P. Robillard, R. DeLine. A field study of API learning obstacles. Empirical Software Engineering, 16(6) (2011), pp. 703-732.

[21] H. Zhong, L. Zhang, T. Xie, H. Mei. Inferring resource specifications from natural language API documentation. 24th IEEE/ACM International Conference on Automated Software Engineering, ASE (2009), pp. 307-318.

[22] C. McMillan, D. Poshyvanyk, M. Grechanik. Recommending source code examples via API call usages and documentation. 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE@ICSE (2010), pp. 21-25.

[23] L.W. Mar, Y. Wu, H.C. Jiau. Recommending proper API code examples for documentation purpose. 18th Asia Pacific Software Engineering Conference, APSEC (2011), pp. 331-338.

[24] M.P. Robillard, Y.B. Chhetri. Recommending reference API documentation. Empirical Software Engineering, 20(6) (2015), pp. 1558-1586.

[25] C. Treude, M.P. Robillard. Augmenting API documentation with insights from stack overflow. IEEE/ACM International Conference on Software Engineering, ICSE (2016), pp. 392-403.

[26] Q. Huang, X. Xia, Z. Xing, D. Lo, X. Wang. API method recommendation without worrying about the task-API knowledge gap. 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE (2018), pp. 293-304.

[27] S. Hangal, M.S. Lam. Tracking down software bugs using automatic anomaly detection. IEEE/ACM International Conference on Software Engineering, ICSE (2002), pp. 291-301.

[28] D. Hovemeyer, W. Pugh. Finding bugs is easy. SIGPLAN Notices, 39(12) (2004), pp. 92-106.

[29] L. Tan, D. Yuan, G. Krishna, Y. Zhou. /*icomment: bugs or bad comments?*/. ACM Symposium on Operating Systems Principles, SOSP (2007), pp. 145-158.

[30] S.H. Tan, D. Marinov, L. Tan, G.T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. 5th IEEE International Conference on Software Testing, Verification and Validation, ICST (2012), pp. 260-269.

[31] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, H.C. Gall. Analyzing APIs documentation and code to detect directive defects. IEEE/ACM International Conference on Software Engineering, ICSE (2017), pp. 27-37.

[32] Z. Qi, F. Long, S. Achour, M. Rinard. Efficient automatic patch generation and defect identification in Kali. International Symposium on Software Testing and Analysis, ISSTA (2015), pp. 257–269.

[33] E.K. Smith, E.T. Barr, C.L. Goues, Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. 10th ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference, ESEC/SIGSOFT FSE (2015), pp. 532-543.

[34] F. Long, M.C. Rinard. An analysis of the search spaces for generate and validate patch generation systems. IEEE/ACM International Conference on Software Engineering, ICSE (2016), pp. 702-713.

[35] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, M. Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. Empirical Software Engineering, 22(4) (2017), pp. 1936-1964.

[36] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang. Precise condition synthesis for program repair. IEEE/ACM International Conference on Software Engineering, ICSE (2017), pp. 416-426.

[37] X.D. Le, D. Lo, C.L. Goues. History driven program repair. 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER (2016), pp. 213-224.

[38] F. Long, M. Rinard. Automatic patch generation by learning correct code. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL (2016), pp. 298-312.

[39] S. Mechtaev, J. Yi, A. Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. IEEE/ACM International Conference on Software Engineering, ICSE (2016), pp. 691-701.

[40] P.W. McBurney, S. Jiang, M. Kessentini, N.A. Kraft, A. Armaly, M.W. Mkaouer, C. McMillan. Towards prioritizing documentation effort. IEEE Transactions on Software Engineering, 44(9) (2018), pp. 897-913.

[41] I. Sora. Helping program comprehension of large software systems by identifying their most important classes. 10th international conference on evaluation of novel approaches to software engineering, ENASE (2015) pp. 122-140. Communications in Computer and Information Science, vol 599. Springer, Cham.

[42] Q. Song, Y. Guo, M.J. Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Transactions on Software Engineering, 45(12) (2019), pp. 1253-1269.

[43] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer. SMOTE: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, 16 (2002), pp. 321-357.

[44] A. Fernández, S. Garcia, F. Herrera, N.V. Chawla. Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. Journal of Artificial Intelligence Research, 61 (2018), pp. 863-905.

[45] S. García, J. Luengo, F. Herrera. Tutorial on practical tips of the most influential data preprocessing algorithms in data mining. Knowledge-Based Systems, 98 (2016), pp. 1-29.