Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka
specjalność: Inżynieria oprogramowania

Praca dyplomowa - magisterska

Predykcja defektów na poziomie metod w celu zredukowania wysiłku związanego z zapewnieniem jakości oprogramowania

Mateusz Kutyba

słowa kluczowe:
pierwsze
drugie
trzecie

krótkie streszczenie:
Bardzo krótkie streszczenie w którym powinno się znaleźć omówienie tematu pracy i poruszanych terminów. Tekst ten nie może być zbyt długi.

opiekun pracy	dr hab. inż. Lech Madeyski	.....	.....
dyplomowej	*Tytul/stopień naukowy/imię i nazwisko*	*ocena*	*podpis*

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**
a) kategorii A (akta wieczyste)
b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)
** niepotrzebne skreślić*

pieczątka Instytutu, w którym student wykonywał pracę

Wrocław 2015

Spis treści

Streszczenie

Streszczenie po polsku

Abstract

Abctract in english

Rozdział 1

Wstęp

Prawdopodobnie nie istnieją programy wolne od błędów. Z całą pewnością istnieją programy, które zawierają zbyt dużą liczbę błędów. Każdy kto tworzy oprogramowanie chciałby aby było ono wolne od wad. Podstawowym narzędziem pozwalającym na sprawdzenie czy program działa poprawnie są testy. Powstają coraz bardziej wyszukane metody i metodyki testowania oprogramowania, a wszystko po to oprogramowanie działało zgodnie z oczekiwaniami, czyli aby cechowało się wysoką jakością. Testowanie i inspekcje kodu pozwalają zapewnić odpowiednią jakość oprogramowania, ale są kosztowne. Bazując na zasadzie Pareto [11] wiemy, że około 80% defektów pochodzi z 20% modułów [7]. Wiedząc, które moduły należy poddać inspekcji, można znacznie obniżyć ilość pracy potrzebną do znalezienia większości błędów, a co za tym idzie znacząco zmniejszyć koszt takich inspekcji.

Kod źródłowy oprogramowania zazwyczaj składa się z wielu plików, które są organizowane w pakiety (ang. *packages*). W języku Java pliki zawierają klasy (ang. *class*), a każda klasa może zawierać metody (ang. *method*). Powstało wiele metod predykcji defektów, na różnych poziowach granulacji: od pakietów, przez pliki, klasy, metody, na pojedynczych zmianach (ang. *hunk* [12,13]) skończywszy [5,8,9,20]. Jak wykazano m.in. w [15] i [25], predykcja błędów na poziomie metod dostarcza dokładniejszych danych na temat lokalizacji błędów, dzięki czemu ich odnajdywanie jest efektywniejsze.

Predykcja defektów oprogramowania wykorzystuje techniki eksploracji danych, głównie są to metody statystyczne i metody uczenia maszynowego. Kluczowym elementem w tych procesach są właśnie dane. To na ich podstawie algorytmy uczenia maszynowego są w stanie formułować reguły decyzyjne. W inżynierii oprogramowania tymi danymi są róznego rodzaju metryki oprogramowania. Podział metryk oraz ich zastosowanie zostały szerzej opisane w rozdziale 2.1.

Gromadzenie danych (metryk) z projektów jest czasochłonne, wymaga dużo pracy — pobierania lub kopiowania projektów, mocy obliczeniowej do wyliczenia metryk. Potrzebne jest stworzenie uniwersalnych rozwiązań służących do tego celu oraz nastawienie na możliwość rozszerzania zestawu narzędzi, które mogą być ze sobą dowolnie zestawiane. Te wymagania spełnia platforma DePress [32], która jest rozwijana przy udziale studentów i pracowników Politechniki Wrocławskiej oraz pracowników Capgemini Polska. Więcej informacji o DePress zawarto w rozdziale 3.3.

Wstępne przeszukiwanie literatury wykazało niewielką ilość źródeł ściśle odpowiadających zagadnieniu predykcji defektów na niskim poziomie granulacji. Jest to główny

kierunek tych badań a ich celem jest przede wszystkim opracowanie nowego modelu, który miałby służyć do efektywnego wskazywania miejsc w oprogramowaniu, w których znajdują się błędy. Pozwololiłoby to na ograniczenie ilości pracy potrzebnej do przejrzenia krytycznych miejsc i naprawienia błędów.

1.1. Cele pracy

Cele pracy dyplomowej:

- Przegląd literatury pod kątem predykcji defektów oprogramowania, szczególnie na niskim poziomie granulacji.
- Budowa nowych lub rozbudowa istniejących narzędzi służących do wyliczenia metryk oprogramowania, współpracujących z wersjonowanymi repozytoriami kodu (SVN oraz Git).
- Zebranie danych z projektów o otwartych źródłach na potrzeby predykcji defektów.
- Budowa modeli predykcji z wykorzystaniem zebranych danych.
- Ocena stworzonych narzędzi oraz zebranych danych.
- Ewaluacja modeli predykcji i ocena ich skuteczności.

1.2. Związek z innymi pracami

Na początku prac dokonano przeglądu literatury aby określić aktualny stan wiedzy (ang. *state of the art*) w badanej dziedzinie. Przegląd literatury pozwolił udzielić odpowiedzi na następujące pytania:

- **Jakie istnieją metody predykcji defektów na poziomie metod i jaka jest ich skuteczność?**
  Istnieje wiele modeli predykcji defektów, jednak większość z nich opiera się na danych dotyczących klas, pakietów lub modułów. Odpowiedzią na powyższe pytanie jest zbiór modeli predykcji defektów na niskim poziomie granulacji, na przykład metod lub bloków kodu.
- **Jakie są możliwości usprawnienia lub rozwinięcia istniejących metod?**
  Zostały zebrane wszelkie możliwości ulepszenia lub rozszerzenia badań wskazanych przez autorów, określone np. jako "Dalszy rozwój".
- **Jakie są sposoby ekstrakcji zmian kodu źródłowego na poziomie metod?**
  Jakie są sposoby porównywania wersji kodu źródłowego, jakiego rodzaju dane (metryki) są uzyskiwane.

Podczas wstępnego rozpoznania dziedziny zauważono, że liczba publikacji jest niewielka. W związku z tym postanowiono przeprowadzić wyszukiwanie w dwóch etapach. W pierwszym etapie przeszukano elektroniczne zbiory, natomiast w drugim etapie przejrzano bibliografie pozyskanych publikacji a także wszystkie publikacje ich autorów w celu odnalezienia dodatkowych tekstów.

Przeszukiwalne zbiory cyfrowe. Przeszukano poniższe zbiory z użyciem ustalonych wyrażeń, za pomocą wyszukiwarek udostępnianych w postaci aplikacji internetowej:

- IEEE Xplore,
- Science Direct,
- ACM Digital Library,
- Springer Link,
- ISI Web of Science.

Wybrano te zbiory ponieważ pokrywają one większość publikacji inżynierii oprogramowania oraz są używane jako źródła w innych przeglądach z tej dziedziny [20, 39].

Szara literatura. Wstępne przeszukiwanie wykazało niewielką ilość źródeł ściśle odpowiadających zagadnieniu predykcji defektów na niskim poziomie granulacji. Aby pokryć znaczną część szarej literatury zdecydowano, aby przeszukać następujące źródła:

- Google Scholar.
- Lista odnośników w znalezionych źródłach pierwotnych.
  Zgodnie z metodą śnieżnej kuli [18] przejrzano listy referencji źródeł pierwotnych w celu odnalezienia dodatkowych istotnych (relewantnych) publikacji.
- Inne publikacje autorów znalezionych źródeł pierwotnych.
  Przeszukano bazę DBLP [30] szukając według nazwisk autorów dotychczas zgromadzonych źródeł pierwotnych.

Po dokonaniu przeglądu literatury oraz oceny znalezionych źródeł, wybrano te najbardziej istotne z punktu widzenia niniejszej pracy:

- *Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes.* [10]
- *Method-level bug prediction.* [15]
- *Comparing fine-grained source code changes and code churn for bug prediction.* [16]
- *Fault-prone Module Prediction Using Version Histories.* [22]
- *Reconstructing fine-grained versioning repositories with git for method-level bug prediction.* [23]
- *Historage: fine-grained version control system for Java.* [24]
- *Bug prediction based on fine-grained module histories.* [25]

1.3. Struktura pracy

TODO Omówienie zawartości rozdziałów

Rozdział 2

Wprowadzenie

2.1. Rola metryk w inżynierii oprogramowania

Norma IEEE 1061-1998 [2] definiuje metrykę jako "funkcję odwzorowującą jednostkę oprogramowania w wartość liczbową. Ta wyliczona wartość jest interpretowalna jako stopień spełnienia pewnej własności jakości jednostki oprogramowania."

W inżynierii oprogramowania metryki są wykorzystywane we wszystkich fazach procesu wytwarzania oprogramowania. Pozwalają na porównywanie ze sobą różnych elementów lub różnych projektów ponieważ są danymi liczbowymi. W fazie projektowania mogą służyć m.in. do szacowania nakładu pracy potrzebnego do realizacji projektu. W fazie produkcji i testów do mierzenia jakości aplikacji, wydajności pracy czy złożoności programu.

Metryki można podzielić według różnych kryeriów. Ze względu na typ artefaktu jaki opisują dzieli się je na:

- Metryki produktu (inaczej metryki kodu źródłowego). Są bezpośrednio wyliczane z kodu źródłowego programu. Przykładem takich metryk są:
  - Zestaw metryk CK [6], do którego należą:
    - uśrednione metody na klasę (ang. *Weighted Methods per Class*, WMC),
    - głębokość drzewa dziedziczenia (ang. *Depth of Inheritance Tree*, DIT),
    - liczba dzieci (ang. *Number of Children*, NOC),
    - zależność między obiektami (ang. *Coupling Between Objects*, CBO),
    - odpowiedzialność danej klasy (ang. *Response For a Class*, RFC),
    - brak spójności metod (ang. *Lack of Cohesion of Methods*, LCOM).
  - OO — metryki obiektowe, np.:
    - liczba atrybutów (ang. *Number of attributes*, NOA),
    - liczba metod (ang. *Number of methods*, NOM),
    - liczba dziedziczonych metod (ang. *Number of methods inherited*, NOMI).
  - LOC — liczba linii kodu.
- Metryki procesu (inaczej metryki zmian). Określają zmienność atrybutu w czasie. Oblicza się je dla zadanych przedziałów czasowych. Niezbędna do ich obliczenia jest historia projektu, którą można uzyskać dzięki systemom kontroli wersji (jak SVN czy Git). Przykłady metryk procesu:
  - liczba modyfikacji (rewizji) pliku (ang. *Number of Revisions*, NR),
  - liczba autorów zmieniających plik (ang. *Number of Distinct Commiters*, NDC),

- liczba zmienionych linii kodu (ang. *Number of Modi?ed Lines*, NML),
- wiek pliku (ang. *Age*, AGE),
- liczba refaktoryzacji pliku (ang. *(Number of Refactorings*, NREF),
- liczba dodanych (usuniętych, zmienionych) metod,
- liczba dodanych (usuniętych, zmienionych) atrybutów.

Dodatkowo można podzielić metryki z uwagi na cel pomiaru [19]:

- metryki złożoności,
- metryki szacowania nakładu,
- metryki funkcjonalności.

Model predykcji defektów to narzędzie, które na podstawie wartości metryk danego projektu dokonuje wskazania defektów znajdujących się w tym projekcie. Aby poprawnie zinterpretować wskazania dostarczane przez model predykcji defektów zdefiniować defekt. Norma 982.2 IEEE/ANSI [1] definiuje defekt jako anomalię w produkcie, która może być:

- zaniechaniami i niedoskonałościami znalezionymi podczas wczesnych faz cyklu życia oraz
- błędami zawartymi w oprogramowaniu wystarczająco dojrzałym do testowania lub działania.

Istniejące badania wykazały, że metryki procesu przewyższyły metryki produktu w kontekście budowania modeli predykcji defektów [12, 15, 26, 33]. Z tego powodu w dalszej części pracy zrezygnowano z wykorzystania metryk produktu, biorąc pod uwagę jedynie metryki procesu.

2.2. Koszty zapewnienia jakości

W inżynierii oprogramowania występuje kilka różnych definicji jakości. Na potrzeby niniejszej pracy przyjęto definicję Kana [27] "brak defektów w produkcie". Zarządzanie jakością oprogramowania polega na podejmowaniu działań mających na celu zapewnienie jakości tworzonego oprogramowania poprzez szereg testów, które wspierają cały proces rozwoju oprogramowania.

- Na etapie zbierania wymagań — weryfikacja czy określone wymagania będą możliwe do zweryfikowania (przetestowania).
- Na etapie projektowania — zaplanowanie procesu testowego, wybór środowisk testowych.
- Na etapie kodowania — definiowanie i realizacja scenariuszy i przypadków testowych oraz rejestracja defektów.
- Na etapie zamknięcia projektu — testy integracyjne, testy akceptacyjne, testy operacyjne.

Jak wykazano w [3] koszty zapewnienia jakości są prawie proporcjonalne do wielkości modułu. Dlatego badacze biorą pod uwagę wysiłek związany z działaniami mającymi na celu zapewnienie jakości [28, 35, 38]. Zmniejszenie wysiłku i kosztu związanego z zapewnieniem jakości to obecnie jeden z głównych kierunków badań [25].

Podstawowym celem pracy dyplomowej jest stworzenie narzędzi w postaci wtyczek do środowiska KNIME, służących do gromadzenia metryk oprogramowania z systemów kontroli wersji oraz zgromadzenie jak największej ilości metryk w publicznym repozytorium. Następnym krokiem jest stworzenie modelu (modeli) predykcji defektów oraz ich ewaluacja, biorąc pod uwagę wysiłek związany z zapewnieniem jakości oprogramowania. Stworzenie narzędzi pozwalających na zautomatyzowane gromadzenie metryk z dostępnych projektów (na przykład Open Source) pozwoli rozszerzyć publiczne zbiory

danych. Dzięki temu będzie możliwe wykorzystanie tych danych do tworzenia modeli predykcji defektów oprogramowania dzięki: większym zbiorom uczącym; ewaluacji modeli na większych zbiorach danych.

2.3. Systemy kontroli wersji jako źródło danych o projektach

Jak wspomniano wcześniej w rozdziale 2.1 aby obliczyć metryki procesu, konieczne jest uzyskanie historii projektu. Przegląd literatury pozwolił na wyodrębnienie sposobów i narzędzi, które pozwalają na porównywanie różnych wersji kodu źródłowego. Jak wykazano w [26, 36, 37] predykcja na poziomie plików jest bardziej efektywna niż na poziomie pakietów. Idąc dalej w kierunku uszczegółowienia wyników predykcji, można przypuszczać, że predykcja na poziomie metod byłaby skuteczniejsza niż na poziomie plików. Badanie [25] wykazało, że pliki zawierające błędy zawierają prawie lub ponad 10 metod, natomiast tylko kilka metod zawiera błędy (mediana 1–2). Jest to nie tylko odpowiedź na pytanie czy predykcja na poziomie metod jest skuteczniejsza, ale również wskazanie przyczyny takiego stanu rzeczy. Jednakże aby w pełni wykorzystać możliwości ograniczenia kosztów jakości poprzez predykcję na poziomie metod, potrzebne są skuteczne modele, dostarczające wiarygodnych wyników.

Ze względu na powyższe zależności, podjęto decyzję o prowadzeniu dalszych prac w kierunku budowy modeli predykcji na poziomie metod. Poniżej wypisano techniki porównywania kodu źródłowego na poziomie metod.

- ChangeDistiller [14] — polega na odwzorowaniu kodu źródłowego Java w strukturze drzewiastej, jaką jest AST (ang. *Abstract Syntax Tree*) a następnie wyodrębnieniu zmian pomiędzy dwiema wersjami przy użyciu algorytmów porównywania drzew.
- Historage [24] — wykorzystuje system kontroli wersji Git do przechowywania zidentyfikowanych zmian w kodzie na niskim poziomie.
- APFEL [41] — jest wtyczką do środowiska Eclipse, która zbiera w bazie danych niskopoziomowe zmiany w kodzie. Działa z systemem kontroli wersji SVS i źródłami Java.
- C-REX [21] — wyodrębnia fakty z historii kodu źródłowego języka C, a następnie porównuje ze sobą kolejne wersje.
- Kenyon [4].
- Beagle [17].

2.3.1. Historia błędów w projektach

Metryki, które stanowią dane wejściowe w modelach predykcji są zmiennymi niezależnymi (ang. *independent variables*). Pełny zestaw danych potrzebny do wytrenowania modelu obejmuje także zmienne zależne (ang. *dependent variables*). Zmienna niezależna reprezentuje wyjście (wynik), oraz może być używana do testowania modelu, żeby ocenić jego skuteczność. W predykcji defektów oprogramowania zmienną zależną jest liczba błędów lub zmienna określająca czy występuje błąd.

Aby uzyskać informacje o błędach w projekcie stosuje się metody linkowania błędów. Linkowanie polega na odszukaniu powiązań pomiędzy zmianą zapisaną w repozytorium kodu, a błędem zgłoszonym w systemie śledzenia zmian (ang. *Issue Tracking System*, ITS), takim jak JIRA, Bugzilla, IBM Rational ClearQuest czy innym.

Metoda używa w tej pracy opiera się na metodzie SZZ [40]. Jej zaletą jest porównywanie czasu naprawienia błędu zapisanego w ITS z czasem wysłania poprawki do systemu kontroli wersji.

2.4. Przeznaczenie narzędzi

Narzędzia stworzone w ramach pracy dyplomowej wchodzą w skład platformy (ang. *framework*) DePress[1]. DePress (*Defect Prediction in Software Systems*) jest rozszerzalną platformą pozwalającą na budowanie przepływu pracy (ang. *workflow*) w sposób graficzny, dzięki temu, że jest oparty na projekcie KNIME. Głównym celem DePress jest wspieranie analizy empirycznej oprogramowania. Pozwala na zbieranie, łączenie i analizę danych z różnych źródeł, jak repozytoria oprogramowania czy metryki.

2.4.1. Ograniczenia dotyczące realizacji

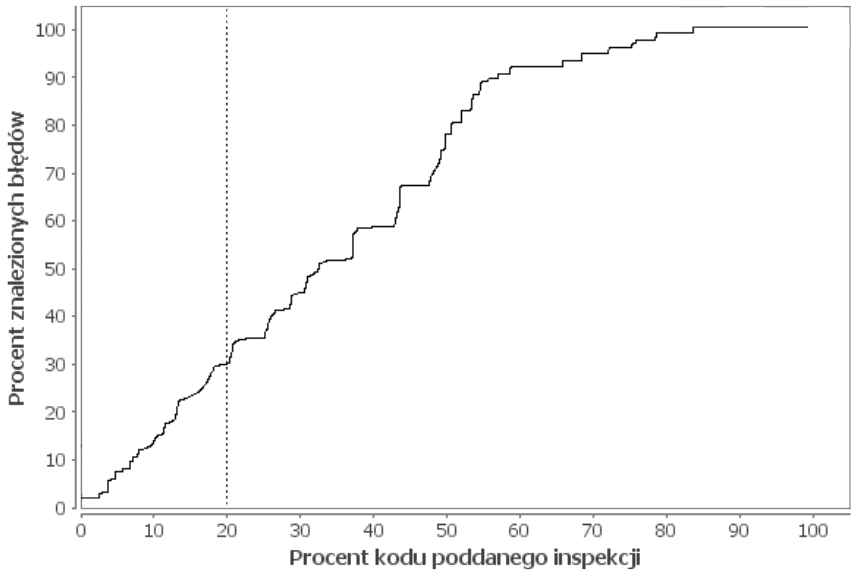
Poniżej wypisano ograniczenia dotyczące realizacji badań:

- badanie tylko projektów Open Source,
- badanie tylko projektów napisanych w języku Java,
- wykorzystanie narzędzi platformy DePress, lub stworzenie nowych narzędzi w ramach platformy,
- badania prowadzone w środowisku KNIME.

2.4.2. Metoda oceny

Podstawową oceną efektywności tworzonych modeli była ocena oparta na wysiłku (ang. *effort-based evaluation*). Dla każdego modelu została stworzona krzywa efektywności, przykład takiej krzywej przedstawia rysunek 2.1. Porównanie efektywności modeli polega przede wszystkim na porównaniu procentowej ilości błędów znalezionych w określonej ilości kodu. Przyjęto, że wartością graniczną kodu poddanego inspekcji będzie 20%. Taka sama wartość jest stosowana w innych badaniach. W przedstawionym przykładzie dokonując przeglądu 20% kodu, znajdzie się w nim 30% encji z defektami (w przypadku tych badań są to metody).

Rys. 2.1. Wykres krzywej efektywności



Oprócz określenia wysiłku, do oceny modeli przyjęto również inne miary. TODO dopisać jakie i jak się je liczy, na pewno będą: Accuracy, Cohen's kappa coefficient, AUC

1. http://depress.io

Rozdział 3

Omówienie infrastruktury pomiarowej

...

3.1. Knime

Knime
o knime

3.2. Weka

Weka
o Weka weka w knime

3.3. DePress

ogólny podział na grupy wtyczek
czego można było użyć, wykorzystywane wtyczki (JIRA, Bugzilla)

JIRA Bugzilla eclipse apache
czego brakuje, trzeba było napisać

3.4. AstCompare (roboczo)

AstCompare plugin co było co dodano (nowe metryki, nowa tabela wyjściowa)

3.4.1. Wymagania

3.4.2. Implementacja

3.4.3. Testowanie i gromadzenie danych

Metryki zapoczątkowane przez autorów i zaimplementowane przez poprzednika (Piotr Mitka [**?**]):

- allMethodHistories
- methodHistories
- authors
- stmtAdded
- maxStmtAdded
- avgStmtAdded
- stmtUpdated
- maxsSmtUpdated
- avgStmtUpdated

- stmtDeleted
- maxStmtDeleted
- avgStmtDeleted
- stmtParentChanged
- churn
- maxChurn
- avgChurn
- decl
- cond

Metryki dodane przeze mnie:

- elseAdded
- elseDeleted
- loopsAdded
- loopsUpdated
- loopsDeleted
- variablesAdded
- variablesUpdated
- variablesDeleted
- assigmentsAdded
- assigmentsUpdated
- assigmentsDeleted
- returnsAdded
- returnsUpdated
- returnsDeleted
- nullsAdded
- nullsUpdated
- nullsDeleted
- casesAdded
- casesUpdated
- casesDeleted
- breaksAdded
- breaksUpdated
- breaksDeleted
- objectsAdded
- objectsUpdated
- objectsDeleted
- catchesAdded
- catchesUpdated
- catchesDeleted
- throwsAdded
- throwsUpdated
- throwsDeleted

Dodatkowo zaimplementowano drugie wyjście wtyczki z wszystkimi historiami metod (do linkowania błędów): commit_id, nazwa metody, autor, komentarz, data.

3.4.4. Omówienie zebranych danych pomiarowych

Wymagania dla repozytorium metryk

Struktura danych i zawartość zbioru

Rozdział 4

Modele predykcji i ich ewaluacja

Wykorzystanie RandomForest [25, 26, 29, 31, 34]

Udział % metod z błędami jest niewielki — modele predykcji mają tendencję do wskazywania wszystkich jako 0, bo mało FP — logistic regression tak miało w [25].

Walidacja modelu przez 10-fold cross validation oraz przez testowanie na zbiorze danych z następnego wydania (ang. *release*).

Wykorzystywane algorytmy:

- RandomForest
- J48
- NaiveBayes
- Bagging
- AdaBoostM1
- RacedIncrementalLogitBoost
- LMT
- BFTree
- RandomTree

Przebadane projekty:

Rys. 4.1. Projekty

robocza tabelka							
			NCSS	# of methods	Buggy methods	% of buggy	daty
log4j	v_1_2final	v1_2_11	9904	1746	70	4,01%	...
ecf	Root_Release_3_0	Root_Release_3_3	63375	12260	174	1,42%	...
jdt.debug	R2_0	R3_0	103466	9652	172	1,78%	...
ant	ANT_170	ANT_180	56487	9697	296	3,05%	
cassandra	cassandra-1.2.0-rc2	cassandra-2.0.0	93654	15545	557	3,58%	

- NCSS = Non Commenting Source Statements
- pierwszy wiersz w wyniku (Sort by NCSS) polega na posortowaniu metod od najmniejszych i sprawdzaniu ich w tej kolejności
- kolejne wyniki są dla modeli opartych na wymienionych wyżej algorytmach klasyfikacji
- wyniki porównuję głównie na podstawie efektywności znajdowania błędów — sortuję wyniki wg prawdopodobieństwa wystąpienia błędu w metodzie (algorytmy

Rys. 4.2. Wyniki

		% of bugs found in 20% of NCSS	Accuracy	Kappa	AUC
log4j	Sort by NCSS	9,68	?	?	?
	RandomForest	92,86	0,99	0,81	0,94
	J48	82,86	0,98	0,64	0,76
	NaiveBayes	71,43	0,96	0,44	0,75
	Bagging	84,29	0,98	0,57	0,89
	AdaBoostM1	75,71	0,97	0,51	0,77
	RacedIncrementalLogitBoost	77,14	0,96	0,00	0,51
	LMT	78,57	0,97	0,47	0,80
	BFTree	80,00	0,97	0,58	0,78
	RandomTree	90,00	0,99	0,83	0,92
ecf	Sort by NCSS	25,00	?	?	?
	RandomForest	81,03	0,99	0,56	0,84
	J48	55,17	0,99	0,12	0,54
	NaiveBayes	50,57	0,97	0,20	0,72
	Bagging	59,20	0,99	0,10	0,77
	AdaBoostM1	62,07	0,99	0,00	0,73
	RacedIncrementalLogitBoost	58,62	0,98	0,10	0,67
	LMT	51,72	0,99	0,00	0,50
	BFTree	51,72	0,99	0,00	0,50
	RandomTree	77,59	0,99	0,59	0,76
jdt.debug	Sort by NCSS	64,53	?	?	?
	RandomForest	100,00	0,98	0,52	0,84
	J48	100,00	0,98	0,47	0,75
	NaiveBayes	100,00	0,98	0,36	0,75
	Bagging	100,00	0,98	0,43	0,80
	AdaBoostM1	100,00	0,97	0,42	0,78
	RacedIncrementalLogitBoost	100,00	0,98	0,00	0,50
	LMT	100,00	0,98	0,43	0,76
	BFTree	100,00	0,98	0,49	0,76
	RandomTree	100,00	0,98	0,48	0,73
ant	Sort by NCSS	15,15	?	?	?
	RandomForest	57,43	0,97	0,44	0,84
	J48	47,30	0,97	0,33	0,70
	NaiveBayes	31,76	0,94	0,24	0,76
	Bagging	40,54	0,97	0,29	0,81
	AdaBoostM1	34,12	0,97	0,00	0,76
	RacedIncrementalLogitBoost	33,11	0,96	0,25	0,74
	LMT	33,45	0,97	0,21	0,78
	BFTree	36,82	0,97	0,22	0,67
	RandomTree	57,77	0,97	0,45	0,71
cassandra	Sort by NCSS	0,00	?	?	?
	RandomForest	83,84	0,96	0,41	0,74
	J48	82,94	0,95	0,32	0,61
	NaiveBayes	68,04	0,96	0,23	0,64
	Bagging	74,33	0,96	0,35	0,70
	AdaBoostM1	79,53	0,95	0,31	0,65
	RacedIncrementalLogitBoost	79,53	0,95	0,29	0,62
	LMT	75,04	0,95	0,33	0,66
	BFTree	85,28	0,95	0,34	0,64
	RandomTree	91,74	0,96	0,41	0,67

podają zwycięzcę oraz prawdopodobieństwa dla 0 i 1), następnie zliczam ile % błędów zostanie odnalezione przy przeglądaniu 20% kodu w tej kolejności
- wyniki w tabeli poniżej
- przykładowe wykresy efektywności dla 2 projektów poniżej
- na wykresach wybrane 2 projekty które wyszły najgorzej, w pozostałych projektach te krzywe są jeszcze bardziej strome

4.1. Ocena rozwiązania

np. Goal-Question-Metric

- Stopień realizacji wymagań funkcjonalnych
- Poprawność rozwiązania (funkcjonowania systemu): weryfikacja (symulacja), testowanie
  utworzenie nowego modelu predykcji defektów, bazującego na zgromadzonych metrykach w celu weryfikacji przydatności narzędzi i zgromadzonych metryk
- Właściwości (parametry) rozwiązania
- Porównanie z innymi rozwiązaniami

Rozdział 5

Podsumowanie i propozycja dalszych badań

- Wady, zalety, ograniczenia
- Wnioski
- Zakres zastosowań, rozwój i dalsze badania

Bibliografia

[1] Ieee guide for the use of ieee standard dictionary of measures to produce reliable software. *IEEE Std 982.2-1988*, 1989.

[2] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1998*, 1998.

[3] Arisholm E., Briand L. C., Johannessen E. B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[4] Bevan J., Whitehead Jr E. J., Kim S., Godfrey M. Facilitating software evolution research with kenyon. W: *ACM SIGSOFT Software Engineering Notes*, 30(5):177–186. ACM, 2005.

[5] Catal C., Diri B. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[6] Chidamber S. R., Kemerer C. F. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[7] Concas G., Marchesi M., Murgia A., Tonelli R., Turnu I. On the distribution of bugs in the eclipse system. *Software Engineering, IEEE Transactions on*, 37(6):872–877, 2011.

[8] D'Ambros M., Lanza M., Robbes R. An extensive comparison of bug prediction approaches. W: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 31–41. IEEE, 2010.

[9] D'Ambros M., Lanza M., Robbes R. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[10] Dyer R., Rajan H., Nguyen T. N. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. W: *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, 23–32. ACM, 2013.

[11] Endres A., Rombach H. D. *A handbook of software and systems engineering: empirical observations, laws and theories*. Pearson Education, 2003.

[12] Ferzund J., Ahsan S. N., Wotawa F. Empirical evaluation of hunk metrics as bug predictors. W: *Software Process and Product Measurement*, 242–254. Springer, 2009.

[13] Ferzund J., Ahsan S. N., Wotawa F. Software change classification using hunk me-

trics. W: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 471–474. IEEE, 2009.

[14] Fluri B., Wursch M., PInzger M., Gall H. C. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[15] Giger E., D'Ambros M., Pinzger M., Gall H. C. Method-level bug prediction. W: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 171–180. ACM, 2012.

[16] Giger E., Pinzger M., Gall H. C. Comparing fine-grained source code changes and code churn for bug prediction. W: *Proceedings of the 8th Working Conference on Mining Software Repositories*, 83–92. ACM, 2011.

[17] Godfrey M. W., Zou L. Using origin analysis to detect merging and splitting of source code entities. *Software Engineering, IEEE Transactions on*, 31(2):166–181, 2005.

[18] Goodman L. A. Snowball sampling. *The annals of mathematical statistics*, 148–170, 1961.

[19] Górski J. *Inżynieria oprogramowania: w projekcie informatycznym*. Wydaw. MIKOM, 2000.

[20] Hall T., Beecham S., Bowes D., Gray D., Counsell S. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

[21] Hassan A. E., Holt R. C. C-rex: an evolutionary code extractor for c. W: *CSER Meeting*. Citeseer, 2004.

[22] Hata H. Fault-prone module prediction using version histories. 2012.

[23] Hata H., Mizuno O., Kikuno T. Reconstructing fine-grained versioning repositories with git for method-level bug prediction. *IWESEP '10*, 27–32, 2010.

[24] Hata H., Mizuno O., Kikuno T. Historage: fine-grained version control system for java. W: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, 96–100. ACM, 2011.

[25] Hata H., Mizuno O., Kikuno T. Bug prediction based on fine-grained module histories. W: *Proceedings of the 2012 International Conference on Software Engineering*, 200–210. IEEE Press, 2012.

[26] Kamei Y., Matsumoto S., Monden A., Matsumoto K.-i., Adams B., Hassan A. E. Revisiting common bug prediction findings using effort-aware models. W: *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 1–10. IEEE, 2010.

[27] Kan S. H. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] Koru A. G., El Emam K., Zhang D., Liu H., Mathew D. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473–498, 2008.

[29] Lessmann S., Baesens B., Mues C., Pietsch S. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[30] Ley M., Herbstritt M., Ackermann M. R., Hoffmann O., Wagner M., von Keutz S., Hostert K. Dblp bibliography, 2014.

[31] Liaw A., Wiener M. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[32] Madeyski L., Majchrzak M. Software measurement and defect prediction with depress extensible framework. *Foundations of Computing and Decision Sciences*, 39(4):249–270, 2014.

[33] Mende T., Koschke R. Revisiting the evaluation of defect prediction models. W: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, 7. ACM, 2009.

[34] Mende T., Koschke R. Effort-aware defect prediction models. W: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 107–116. IEEE, 2010.

[35] Menzies T., Milton Z., Turhan B., Cukic B., Jiang Y., Bener A. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[36] Nguyen T. H., Adams B., Hassan A. E. Studying the impact of dependency network measures on software quality. W: *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 1–10. IEEE, 2010.

[37] Posnett D., Filkov V., Devanbu P. Ecological inference in empirical software engineering. W: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 362–371. IEEE Computer Society, 2011.

[38] Rahman F., Posnett D., Hindle A., Barr E., Devanbu P. Bugcache for inspections: hit or miss? W: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 322–331. ACM, 2011.

[39] Riaz M., Mendes E., Tempero E. A systematic review of software maintainability prediction and metrics. W: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 367–377. IEEE Computer Society, 2009.

[40] Śliwerski J., Zimmermann T., Zeller A. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[41] Zimmermann T. Fine-grained processing of cvs archives with apfel. W: *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, 16–20. ACM, 2006.