Mining software repositories for code quality

Autoreferat rozprawy doktorskiej

Mikołaj Fejzer

27 kwietnia 2020

1 Wstęp

Eksploracja repozytoriów kodu źródłowego (MSR — *mining software repositories*) jest poddziedziną inżynierii oprogramowania, w ramach której bada i rozwija się metody analizy bogatych zbiorów danych, wytwarzanych podczas ewolucji oprogramowania [14]. Sama ewolucja była dogłębnie zbadana za pomocą różnych modeli stosowanych do mierzenia jakości oprogramowania oraz przeciwdziałania spadkowi jakości w czasie trwania projektu. Niedawne postępy w uczeniu maszynowym oraz wyszukiwaniu informacji pozwoliły na badanie dużych zbiorów danych i ekstrakcję wiedzy z repozytoriów [3, 11, 12, 15]. W związku z tym pojawiły się nowe trendy badań, których celem jest zapewnienie jakości oprogramowania poprzez zastosowanie eksploracji repozytoriów [14, 22]. Najważniejszymi zastosowaniami MSR w kontekście jakości jest zapobieganie powstawaniu błędów w oprogramowaniu oraz lokalizacja już istniejących [5, 16, 21, 27]. Innymi ważnymi zastosowaniami jest wsparcie inspekcji kodu źródłowego [17], detekcja wkładu programistów oraz analiza dynamiki zespołów programistycznych z uwzględnieniem interakcji międzyludzkich i socjalnej strony wytwarzania oprogramowania [24].

2 Opis wyników rozprawy

Celem tej rozprawy jest poprawa jakości oprogramowania poprzez prewencję oraz lokalizację błędów za pomocą technik wyszukiwania informacji i uczenia maszynowego. Skupiamy się na dwóch głównych zagadnieniach:

- rekomendacji recenzentów do inspekcji kodu [7],
- lokalizacji błędów w kodzie źródłowym na podstawie zgłoszeń błędów.

Dodatkowo prezentujemy badania przygotowawcze na projektach open source w celu zdobycia wglądu w role grup współpracowników oraz możliwości automatycznej detekcji błędów [8, 9].

2.1 Badania przygotowawcze

W ramach badań wstępnych przeprowadzonych na projektach dostępnych w portalu GitHub zbadaliśmy role programistów, zakres ich uczestnictwa w projektach, zarządzanie utrzymaniem oprogramowania oraz metodę detekcji błędów podczas inspekcji kodu. Podczas procesu wytwarzania oprogramowania niektórzy programiści opuszczają projekt, inni zaś dołączają, co skutkuje zmianami w wiedzy dziedzinowej w zespole programistycznym. Przeanalizowaliśmy zestawy zmian (ang. *change lists*) w projektach open source używając modelowania tematów (ang. *topic modeling*), aby określić role programistów oraz zakres ich wkładu. W tym celu wykorzystaliśmy podzbiór danych GHTorrent [10]. W konsekwencji zidentyfikowaliśmy zmiany naprawiające błędy jako jedną z najczęściej wykonywanych czynności. W związku z tym, na podstawie modelu zaproponowanego przez Kim et al. [16], przygotowaliśmy mechanizm detekcji błędów w czasie inspekcji kodu oparty o maszynę wektorów wspierających (SVM), aby pomóc opiekunom projektów open source. Przygotowany model używa wcześniejszej historii projektu jako zbioru treningowego. Używamy n ostatnich zestawów zmian do treningu, oraz oznaczamy te zestawy jako zawierające błąd błędne lub nie za pomocą algorytmu *SZZ — Śliwerski-Zimmermann-Zeller* [26]

Oryginalne wyniki opublikowane w dwóch artykułach konferencyjnych [8, 9] są następujące:

- przeprowadzamy analizę wkładu współpracowników w 42 projektach open source i definiujemy grupy programistów odpowiedzialne za większość pracy;
- przygotowujemy klasyfikator wykrywający błędy podczas inspekcji kodu w repozytoriach git i ewaluujemy go na 64 projektach, o różnych długościach historii.

Bazując na analizie komentarzy większość współpracowników nie angażuje się w komentowanie zestawów zmian. Nasza metoda jest w stanie wykryć głównych współpracowników dla analizowanych projektów. Agregacja tematów w skali dnia pozwala stwierdzić, że większość zgłaszanych problemów GitHub to błędy, a ponad połowa komentarzy dotyczących zestawów zmian jest związana z naprawianiem błędów.

Ewaluacja metody detekcji błędów pokazała, iż tego typu metodę można zastosować do projektów o różnych długościach historii oraz różnych językach oprogramowania. Rezultaty nie są wprost porównywalne z osiągniętymi przez Kim et al. [16], ze względu na inny zbiór danych, zawierający większą liczbę projektów oraz odmienne traktowanie zmian kodu w odpowiednich repozytoriach. Algorytm był w stanie osiągnąć dobre rezultaty na podstawie uczenia na różnicach zawartych w zestawach zmian z takim samym zestawem wzorców dla algorytmu *SZZ* na wszystkich rozważanych projektach.

2.2 Rekomendowanie recenzentów do inspekcji kodu źródłowego

W ramach dotychczasowych, intensywnych badań zaproponowano wiele algorytmów doboru recenzentów. Thongtanunam et al. [29] przedstawili narzędzie Revfinder, mierzące podobieństwo ścieżek plików z recenzowanego zestawu zmian do dotychczas przygotowanych recenzji. Balachandran zaproponował, aby używać informacji o autorstwie plików

zawartych w repozytoriach w celu doboru recenzentów tworząc narzędzie Review Bot [2]. Yue Yu et al. używa zarówno podobieństwa tekstu zestawu zmian jak i grafu zawierającego informacje o interakcjach deweloperów do sugerowania recenzentów [38, 39].

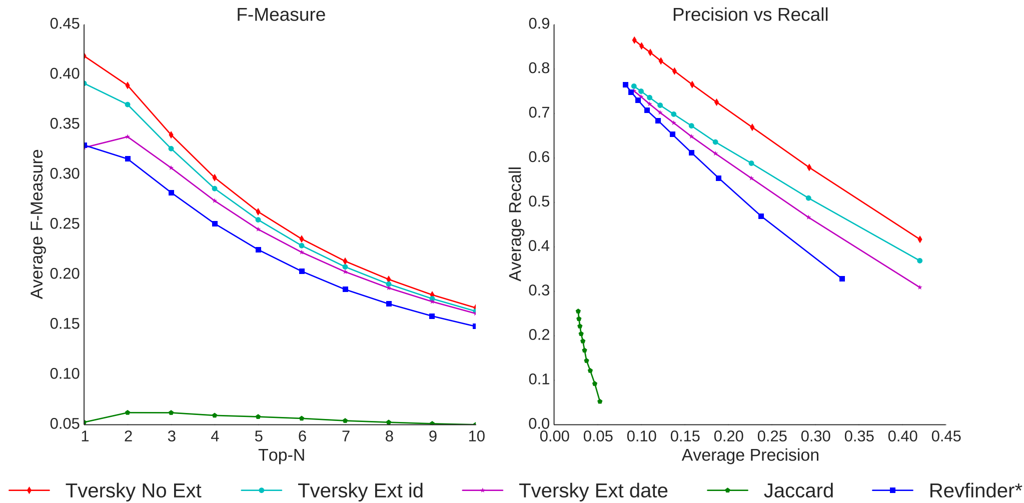
Zaproponowaliśmy nowy algorytm pozbawiony niedogodności poprzednich metod, takich jak wysoki koszt obliczeń porównywania ścieżek plików w wypadku narzędzia Revfinder lub niska celność cechująca Review Bot. Oryginalne wyniki badań zawarte w opublikowanym artykule [7] są następujące:

- przygotowaliśmy nową metodę doboru recenzentów do inspekcji kodu;
- przeprowadzamy ewaluację eksperymentalną metody;
- porównujemy tę metodę ze stosowanymi już technikami, aby dowieść jej wartości.

System inspekcji kodu źródłowego zarządza listą zestawów zmian wymagających recenzji znajdujących się na osobnych gałęziach rozwojowych repozytorium. Zazwyczaj w systemach komercyjnych oraz open source jest wielu potencjalnych kandydatów do wykonania recenzji. Można zaprezentować wszystkie zestawy zmian wszystkim dostępnym kandydatom, ale takie podejście ma kilka wad. Niektóre zestawy zmian mogą zostać pominięte [30] lub czekać relatywnie długo, inne zaś mogą zostać zaakceptowane przez osoby nieposiadające wymaganej wiedzy, co prowadzi do błędów oraz propagacji złych praktyk. System rekomendacji recenzentów zmniejsza czas oczekiwania na recenzję, dobierając recenzentów zgodnie z doświadczeniem każdego kandydata. Rekomendacja może być oparta o decyzje człowieka lub o model uczący się, który opiera się na preferencjach recenzentów oraz ich wiedzy ustalonej na podstawie historii dotychczasowych recenzji. System automatycznego doboru recenzentów oblicza współczynnik zgodności dla jeszcze niezrecenzowanych zestawów zmian oraz kandydatów. Ten współczynnik jest używany do oceny kandydatów — wyższa wartość oznacza lepszego recenzenta. Po dokonaniu recenzji, system uwzględnia ją w historii działań recenzenta celem późniejszych obliczeń.

Gdy pojawiają się nowe zestawy zmian, system musi wybrać najlepszych recenzentów. Obecnie najlepsze metody [2, 29] wczytują oraz przetwarzają całą historię projektu. Takie podejście jest niepraktyczne, ponieważ powoduje zużycie zasobów oraz czasu. Proponujemy model oparty o profile recenzentów, aby rozwiązać problem przetwarzania całej historii. Dla każdego recenzenta konstruujemy jego profil, który jest aktualizowany w momencie komentowania zestawu zmian. W konsekwencji używamy funkcji podobieństwa zdefiniowanej na zestawie zmian oraz profilu recenzenta. Służy to do określenia jak bardzo zestaw zmian jest zgodny z historią pracy danego recenzenta. Aby określić najlepszych recenzentów, obliczamy tę funkcję dla każdego profilu.

Na rozwój projektu, podczas jego trwania, ma wpływ dynamika zespołu programistów oraz różny zakres partycypacji poszczególnych osób z biegiem czasu. Należy pamiętać, że niektórzy programiści mogą opuścić projekt, inni zwiększyć swój wkład, nadając ostatnim inspekcjom kodu większe znaczenie. Jako opcjonalne rozszerzenie proponowanego algorytmu rozwiązujemy ten problem, wprowadzając w profilach recenzentów wygaszanie wykładnicze starszych recenzji. Odbywa się ono poprzez pomnożenie częstotliwości słów w profilach przez regulowany, mniejszy niż jeden czynnik.



Rysunek 1: Wyniki eksperymentów dla 4 wariantów zaproponowanej metody oraz replikacji metody Revfinder [29] na zbiorze danych Thongtanunam et al. [29] zawierającym 4 projekty open source.

Przeprowadziliśmy eksperymenty używając czterech wariantów naszej metody oraz replikacji metody Revfinder na zbiorze danych Thongtanunam et al. [29]. Różnice w naszych wariantach metod dotyczą budowy profilu i zastosowanych funkcji podobieństwa. Trzy z nich korzystają z indeksu Tverskiego. Pierwszy, oznaczony jako *Tversky No Ext*, wykorzystuje standardową konstrukcję profilu, bez wygaszania. Pozostałe dwa, oznaczone jako *Tversky Ext id* i *Tversky Ext date*, używają wygaszania odpowiednio według liczby zestawów zmian i liczby dni. Ostatni wariant, oznaczony jako *Jaccard*, oparty jest na indeksie Jaccarda jako funkcji podobieństwa oraz standardowej konstrukcji profilu, również bez wygaszania. Na Rysunku 1 zaprezentowaliśmy wyniki naszych eksperymentów. Nasza metoda używająca indeksu Tverskiego jako funkcji podobieństwa pomiędzy profilem a recenzowaną zmianą, oznaczona *Tversky No Ext* osiągnęła wyższe rezultaty dla miar precision-to-recall oraz F1-measure w porównaniu do pozostałych metod. Warianty metody z wygaszaniem nie uzyskały znaczącej poprawy w porównaniu z metodą podstawową. Dla większości eksperymentów nasza metoda jest statystycznie lepsza od metody state-of-the-art Revfinder [29]. Ponadto zużywa ona znacząco mniej zasobów obliczeniowych.

Dobór recenzentów do dużych projektów przy realistycznych ograniczeniach zasobów obliczeniowych jest trudnym zadaniem. Stworzyliśmy metodę opartą o profile recenzentów, które stanowią reprezentację ich wiedzy. Intuicyjnie, zaprezentowane wyniki pokazują, iż nasza metoda może mieć zastosowanie na dużych projektach o bogatej historii oraz wielu współpracownikach.

2.3 Adaptacyjna lokalizacja błędów programistycznych na podstawie zgłoszeń

W celu automatyzacji lokalizacji błędów stworzono wiele modeli [1, 4, 19, 20, 23, 25, 33, 36, 40]. Lokalizację błędów możemy traktować jako specyficzną formę wyszukiwania informacji, gdzie traktujemy raporty o błędach jako zapytania oraz kod źródłowy jako zbiór dokumentów. Pliki kodu źródłowego są wybierane na podstawie ich zgodności z wybranym raportem.

Informacje o błędach oraz kodzie źródłowym są dostępne w wielu ustrukturyzowanych zbiorach danych, takich jak raporty oraz zestawy zmian w repozytorium. Systemy zarządzania błędami oraz repozytoria są zazwyczaj oddzielnymi systemami, połączonymi tylko za pośrednictwem identyfikatorów błędów obecnych w metadanych zestawu zmian. Oba rodzaje systemów stanowią bogaty zbiór danych możliwy do exploracji [14].

Jednymi z najwcześniejszych modeli lokalizacji błędów było sprawdzanie wywołań API na ścieżce stosu [1] lub odchyleń w miarach kodu [4, 19]. Nowocześniejsze modele polegają na wyszukiwaniu informacji oraz uczeniu maszynowym, bazując na cechach zbudowanych z różnych źródeł danych. Przykładową cechą jest użycie danych podobieństwa tekstu [20, 40]. Innym jest użycie AST otrzymanego ze skompilowanego kodu, zawierającego bardziej szczegółowe dane, takie jak nazwy klas, metod, zmiennych lub powiązane komentarze [23]. Przetworzone ścieżki stosu stanowią inne możliwe źródło cech [33]. Niektóre algorytmy wykorzystują kompozycję innych metod, albo wykorzystując liniową kombinację wyników rankingowych [31, 32, 37] lub ucząc się rankingu trenowanego na połączonych cechach [25, 36].

W dziedzinie automatycznej lokalizacji błędów istnieją pewne wyzwania. Pierwszym z nich jest różnica między językiem naturalnym użytym w raportach błędów oraz językiem programowania zastosowanym w kodzie źródłowym. Powoduje to, iż użycie prostego dopasowania leksykalnego może powodować gorszą trafność wyboru plików. Aby tego uniknąć, zazwyczaj stosuje się specjalnie przygotowany zbiór cech, zbudowany w oparciu o dostępne dane [35, 36]. Kolejnym wyzwaniem jest fakt, iż relatywnie mała część plików zawiera błędy. Powoduje to, że zarówno model, jak i dobór danych treningowych wymagają starannego wyboru. W konsekwencji jednym z częstych problemów jest brak równowagi między pozytywnymi i negatywnymi przykładami, co prowadzi do fałszywych alarmów generowanych przez pliki ściśle związane z błędem, lecz go niezawierające, takie jak pliki wymienione w śladzie stosu.

Ye et al. [36] przedstawili znaczące wyniki, przygotowując nowy zestaw cech i nową metodę bazującą na learn-to-rank. Ponadto autorzy zaproponowali nowy, szczegółowy zbiór danych, który lepiej odzwierciedla praktyczne zastosowania niż wcześniej stosowany zbiór BugLocator [40]. Zarówno metoda jak i zbiór danych były ważnym krokiem w tej dziedzinie badań.

Dla każdego raportu błędów nasza metoda lokalizuje powiązane pliki, obliczając współczynnik prawdopodobieństwa dla każdego pliku obecnego w repozytorium w momencie zgłoszenia błędu. Pliki mające wyższe prawdopodobieństwo bycia przyczyną błędu otrzymają wyższy współczynnik niż pozostałe.

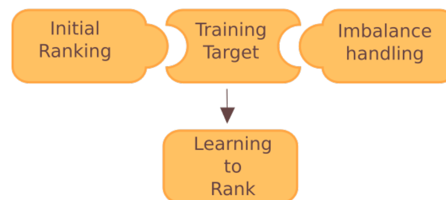
Nasze oryginalne wyniki w tej dziedzinie są następujące:

- zaproponowaliśmy nową, adaptacyjną metodę lokalizacji błędów;
- przygotowaliśmy poszerzony zbiór danych, który zawiera obliczenia pośrednie oraz gotowe cechy, poprawę brakujących opisów błędów, poprawioną cechę podobieństwa opartego na API oraz kompletny kod źródłowy naszej metody;
- zaprezentowaliśmy przegląd najnowocześniejszych podejść lokalizacji.

Uczymy nasz algorytm na zbiorze danym zawierającym raporty błędów wraz z powiązanymi zestawami zmian naprawiającymi błędy. Podobnie do innych metod traktujemy uporządkowane względem daty raporty błędów jako szereg czasowy z przesuwanym oknem [36]. To okno dzieli raporty na grupy o stałej wielkości. Ze względu na ciągły rozwój projektu przeplatany z naprawami błędów i zmianami w składzie uczestniczących programistów, każda grupa może mieć inną charakterystykę. W konsekwencji uczymy nasz algorytm na grupie n aby przewidywać błędy dla kolejnej grupy $n + 1$.

Naszym celem jest znalezienie najlepszej metody oraz jej parametrów bez użycia dodatkowych danych, niebędących w systemie zarządzania błędami lub repozytorium. W tym celu nasz algorytm dobiera parametry automatycznie, w ramach uczenia, bez użycia oddzielnego etapu dopasowania parametrów [36] albo ręcznie doboru [23, 40]. Jeśli pewne wagi muszą zostać ustalone, unikamy ustawiania ich ręcznie, lecz dopasowujemy je adaptacyjnie.

Naszym zdaniem podejście polegające na nauce rankingu jest dobrym rozwiązaniem tego problemu, chociaż można również zastosować różne algorytmy, na przykład modele klasyfikacji [27]. Na podstawie istniejących metod uogólniliśmy typową konfigurację zastosowaną do lokalizacji błędów (Rysunek 2 oraz Tabela 1).



Rysunek 2: Schemat blokowy metod lokalizacji błędów opartych o uczenie rankingu.

Uczenie modelu rankingu, poza przygotowaniem cech bazujących na jakości kodu oraz relacji między raportem a zestawem zmian, wymaga stworzenia rankingu docelowego lub algorytmicznej oceny między dobrym i złym rankingiem. W celu rozwiązania zagadnienia stworzenia rankingu docelowego, skonstruowaliśmy adaptacyjną funkcję tworzącą ten ranking, oceniającą pliki na podstawie ważonej sumy cech. Wagi są różnymi estymatorami możliwości danej cechy do rozróżnienia między błędnymi a poprawnymi plikami. Adaptacja polega na wyborze sposobu obliczania wag z kilku dostępnych funkcji. Podobna liniowa kombinacja cech jest często używana w rozwiązywaniu problemów wyszukiwania informacji. Im dana cecha lepiej jest w stanie oddzielać błędne pliki od

Tablica 1: Porównanie wybranych metod lokalizacji błędów opartych o uczenie rankingu.

	Ye et al.+ [36]	Shi et al. [25]	AmaLgam+ [31]	**Zaproponowana metoda**
Początkowy ranking ($\preceq_{IR}$)	cecha ϕ_1	BLUiR [23]	losowy	**cecha ϕ_2^* i adaptacyjna ocena**
Cel treningu	powiązane pliki: +1, niepowiązane pliki: −1	max MAP	max $e^{MAP+MRR}$	**poprawiony ranking**
Zbalansowanie treningu	top 200 niepowiązanych plików na podstawie $\preceq_{IR}$		brak opisu	**adaptacyjny wybór niepowiązanych plików**, początkowy zbiór: top 200 niepowiązane plików na podstawie cechy ϕ_2^*, zawężone do: % plików na podstawie $\preceq_{IR}$.
Metoda uczenia rankingu	SVMrank [13]	Coordinate Ascent RankLib [6]	JGAP [18]	**adaptacyjna regresja SGD**
	parami	listowo	listowo	punktowo

innych, tym większy jej wpływ na rezultat funkcji. Sam ranking docelowy jest w stanie pokonać kilka dostępnych metod.

Metoda doboru wag, która otrzymała najlepszy wynik (najwyższy MAP podczas walidacji krzyżowej na sprawdzonej grupie raportów błędów) jest używana do szkolenia punktowego modelu rankującego. Sam ranking jest poprawiony poprzez dodanie informacji o błędnych plikach. Następnie szkolimy wiele modeli regresji i wybieramy najlepiej dopasowany na podstawie wyniku walidacji krzyżowej.

Na podstawie eksperymentów na dwóch zbiorach danych pokazaliśmy iż metoda adaptacyjna osiąga wyniki poprawiające stosowane techniki [36] na dwóch wariantach zbioru danych. W tabeli 2 przedstawiono wyniki naszych eksperymentów. W szczególności poprawiamy miary *Accuracy@1*, *MAP* and *MRR* na wszystkich badanych projektach.

Tablica 2: Rezlultaty metody adaptacyjnej oraz metody state-of-the-art dla miar Mean Average Precision and Mean Reciprocal Rank na zbiorze danym Ye et al [36], zawierającym 6 projektów open source AspectJ, Birt, Eclipse, JDT, SWT oraz Tomcat. Oznaczamy wariant naszej metody „with desc", ze względu na uzupełnienie błędu w zbiorze danych, polegającym na przywróceniu brakujących opisów błędów.

	MAP						**MRR**					
Method	AspectJ	BIRT	Eclipse	JDT	SWT	Tomcat	AspectJ	BIRT	Eclipse	JDT	SWT	Tomcat
Adaptive regression	0.45	0.21	0.44	0.40	0.42	0.50	0.53	0.27	0.52	0.48	0.48	0.56
Adaptive regression with desc	0.46	0.19	0.45	0.39	0.41	0.54	0.54	0.25	0.52	0.47	0.48	0.61
Ye et al.+ [36]	0.37	0.16	0.44	0.39	0.40	0.49	0.44	0.21	0.51	0.47	0.46	0.55

3 Podsumowanie

Zbadaliśmy kilka podejść do inżynierii oprogramowania bazujących na eksploracji repozytoriów oprogramowania w celu poprawy jakości projektów. Nasze badania bazowe nad projektami open source oraz detekcją błędów zainspirowały nas do badań nad inspekcją kodu źródłowego oraz lokalizacją błędów. Zaprezentowaliśmy nowatorskie metody w tych dziedzinach, i w obu przypadkach zaproponowane algorytmy były w stanie osiągnąć lepsze rezultaty niż metody istniejące dotychczas. Wykazaliśmy wartość naszych wyników na dojrzałych, często używanych zestawach danych. Dodatkowo opublikowaliśmy kod źródłowy zarówno przygotowywania cech jak i samych algorytmów, aby uprościć możliwość odtworzenia naszych wyników.

3.1 Przyszłe kierunki badań

Istnieje kilka możliwych kierunków przyszłych badań, powiązanych z każdym z tematów poruszanych w rozprawie. Dodatkowo, ostatnie postępy w uczeniu maszynowym stanowią inspirację do szukania zastosowań wielu widoków (ang. *mutli-view learning*) na dane w dziedzinie MSR.

Dla rekomendacji recenzentów zaprezentowanej w Sekcji 2.2 użycie innych metod poprzez zespołowe łączenie wyników (ang. *ensemble*) może poprawić osiągnięte rezultaty. Odnośnie do adaptacyjnej lokalizacji błędów pokazanej w Sekcji 2.3 adaptacja bazowego algorytmu może zostać rozbudowana. Zgodnie z intuicją nasza metoda jest rozszerzalna o dodatkowe funkcje oceniania i modele regresji w odpowiednich krokach.

Jako możliwy kierunek przyszłych badań uznajemy stosowanie uczenia się z wielu widoków do celów eksploracji kodu źródłowego. Do każdego rozważanego problemu wykorzystaliśmy dane z jednego źródła, które odpowiada pojedynczemu widokowi repozytorium oprogramowania i powiązanych artefaktów. Chociaż takie podejście było wystarczające dla naszych celów, leżące u ich podstaw liczne źródła informacji można łączyć za pomocą uczenia maszynowego z wieloma widokami w sposób częściowo nadzorowany [34]. W takim przypadku każdy widok może być obsługiwany przez osobny algorytm, a wcześniej nieznane dane trafiające do jednego algorytmu mogą być przekazane do pozostałych, wzmacniając proces uczenia w kierunku optymalnego rozwiązania [28].

Bibliografia

[1] Giuliano Antoniol i Yann-Gaël Guéhéneuc. "Feature Identification: A Novel Approach and a Case Study". *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary.* IEEE Computer Society, 2005, s. 357–366. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.48.

[2] Vipin Balachandran. "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation". *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* Red. David Notkin, Betty H. C. Cheng i Klaus Pohl. IEEE Computer Society, 2013, s. 931–940. ISBN: 978-1-4673-3076-3.

[3] Christian Bird i in. "The promises and perils of mining git". *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings.* Red. Michael W. Godfrey i Jim Whitehead. IEEE Computer Society, 2009, s. 1–10. ISBN: 978-1-4244-3493-0. DOI: 10.1109/MSR.2009.5069475.

[4] Shyam R. Chidamber i Chris F. Kemerer. “A Metrics Suite for Object Oriented Design”. *IEEE Transactions on Software Engineering* 20.6 (1994), s. 476–493. DOI: 10.1109/32.295895.

[5] Marco D’Ambros, Michele Lanza i Romain Robbes. “An extensive comparison of bug prediction approaches”. *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings.* Red. Jim Whitehead i Thomas Zimmermann. IEEE Computer Society, 2010, s. 31–41. ISBN: 978-1-4244-6803-4. DOI: 10.1109/MSR.2010.5463279.

[6] Van Dang. *The Lemur Project - Wiki - RankLib.* https://sourceforge.net/p/lemur/wiki/RankLib/. The Lemur Project, [Online]. 2012. (Term. wiz. 24. 03. 2020).

[7] Mikolaj Fejzer, Piotr Przymus i Krzysztof Stencel. “Profile based recommendation of code reviewers”. *Journal of Intelligent Information Systems* 50.3 (2018), s. 597–619. DOI: 10.1007/s10844-017-0484-1.

[8] Mikolaj Fejzer, Michal Wojtyna, Marta Burzanska, Piotr Wisniewski i Krzysztof Stencel. “Open Source Is a Continual Bugfixing by a Few”. *Advances in Databases and Information Systems - 18th East European Conference, ADBIS 2014, Ohrid, Macedonia, September 7-10, 2014. Proceedings.* Red. Yannis Manolopoulos, Goce Trajcevski i Margita Kon-Popovska. T. 8716. Lecture Notes in Computer Science. Springer, 2014, s. 153–162. ISBN: 978-3-319-10932-9. DOI: 10.1007/978-3-319-10933-6_12.

[9] Mikolaj Fejzer, Michal Wojtyna, Marta Burzanska, Piotr Wisniewski i Krzysztof Stencel. “Supporting Code Review by Automatic Detection of Potentially Buggy Changes”. *Beyond Databases, Architectures and Structures - 11th International Conference, BDAS 2015, Ustroń, Poland, May 26-29, 2015, Proceedings.* Red. Stanislaw Kozielski, Dariusz Mrozek, Pawel Kasprowski, Bozena Malysiak-Mrozek i Daniel Kostrzewa. T. 521. Communications in Computer and Information Science. Springer, 2015, s. 473–482. ISBN: 978-3-319-18421-0. DOI: 10.1007/978-3-319-18422-7_42.

[10] Georgios Gousios. “The GHTorent dataset and tool suite”. *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013.* Red. Thomas Zimmermann, Massimiliano Di Penta i Sunghun Kim. IEEE Computer Society, 2013, s. 233–236. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624034.

[11] Ahmed E Hassan. “The road ahead for mining software repositories”. *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE. 2008, s. 48–57.

[12] Ahmed E. Hassan i Tao Xie. “Software intelligence: the future of mining software engineering data”. *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010.* Red. Gruia-Catalin Roman i Kevin J. Sullivan. ACM, 2010, s. 161–166. ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882397.

[13] Thorsten Joachims. “Training linear SVMs in linear time”. *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006.* Red. Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven i Dimitrios Gunopulos. ACM, 2006, s. 217–226. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150429.

[14] Huzefa H. Kagdi, Michael L. Collard i Jonathan I. Maletic. “A survey and taxonomy of approaches for mining software repositories in the context of software evolution”. *Journal of Software Maintenance* 19.2 (2007), s. 77–131. DOI: 10.1002/smr.344.

[15] Eirini Kalliamvakou i in. “The promises and perils of mining GitHub”. *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India.* Red. Premkumar T. Devanbu, Sung Kim i Martin Pinzger. ACM, 2014, s. 92–101. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597074.

[16] Sunghun Kim, E. James Whitehead Jr. i Yi Zhang. “Classifying Software Changes: Clean or Buggy?” *IEEE Transactions on Software Engineering* 34.2 (2008), s. 181–196. DOI: 10.1109/TSE.2007.70773.

[17] Shane McIntosh, Yasutaka Kamei, Bram Adams i Ahmed E. Hassan. “The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects”. *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India.* Red. Premkumar T. Devanbu, Sung Kim i Martin Pinzger. ACM, 2014, s. 192–201. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597076.

[18] Klaus Meffert. *The JGAP library.* https://sourceforge.net/projects/jgap/. [Online]. 2015. (Term. wiz. 24. 03. 2020).

[19] Raimund Moser, Witold Pedrycz i Giancarlo Succi. “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction”. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008.* Red. Wilhelm Schäfer, Matthew B. Dwyer i Volker Gruhn. ACM, 2008, s. 181–190. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368114.

[20] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen i Tien N. Nguyen. “A topic-based approach for narrowing the search space of buggy files from a bug report”. *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011.* Red. Perry Alexander, Corina S. Pasareanu i John G. Hosking. IEEE Computer Society, 2011, s. 263–272. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100062.

[21] Danijel Radjenovic, Marjan Hericko, Richard Torkar i Ales Zivkovic. “Software fault prediction metrics: A systematic literature review”. *Information & Software Technology* 55.8 (2013), s. 1397–1418. DOI: 10.1016/j.infsof.2013.02.009.

[22] Peter C. Rigby i Margaret-Anne D. Storey. “Understanding broadcast based peer review on open source software projects”. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* Red. Richard N. Taylor, Harald C. Gall i Nenad Medvidovic. ACM, 2011, s. 541–550. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985867.

[23] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid i Dewayne E. Perry. “Improving bug localization using structured information retrieval”. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013.* Red. Ewen Denney, Tevfik Bultan i Andreas Zeller. IEEE, 2013, s. 345–355. DOI: 10.1109/ASE.2013.6693093.

[24] Michael J. Scialdone, Na Li, Robert Heckman i Kevin Crowston. “Group Maintenance Behaviors of Core and Peripherial Members of Free/Libre Open Source Software Teams”. *Open Source Ecosystems: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings.* Red. Cornelia Boldyreff, Kevin Crowston, Björn Lundell i Anthony I. Wasserman. T. 299. IFIP Advances in Information and Communication Technology. Springer, 2009, s. 298–309. ISBN: 978-3-642-02031-5. DOI: 10.1007/978-3-642-02032-2_26.

[25] Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin i Xingjun Zhang. “Comparing learning to rank techniques in hybrid bug localization”. *Applied Soft Computing* 62 (2018), s. 636–648. DOI: 10.1016/j.asoc.2017.10.048.

[26] Jacek Sliwerski, Thomas Zimmermann i Andreas Zeller. “When do changes induce fixes?” *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005.* ACM, 2005. ISBN: 1-59593-123-6. DOI: 10.1145/1083142.1083147.

[27] Higor Amario de Souza, Marcos Lordello Chaim i Fabio Kon. “Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges”. *CoRR* abs/1607.04347 (2016). arXiv: 1607.04347.

[28] Shiliang Sun. “A survey of multi-view machine learning”. *Neural Computing and Applications* 23.7-8 (2013), s. 2031–2038. DOI: 10.1007/s00521-013-1362-6.

[29] Patanamon Thongtanunam i in. “Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review”. *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015.* Red. Yann-Gaël Guéhéneuc, Bram Adams i Alexander Serebrenik. IEEE, 2015, s. 141–150. ISBN: 978-1-4799-8469-5. DOI: 10.1109/SANER.2015.7081824.

[30] Erik van der Veen, Georgios Gousios i Andy Zaidman. “Automatically Prioritizing Pull Requests”. *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015.* IEEE, 2015, s. 357–361. ISBN: 978-0-7695-5594-2. DOI: 10.1109/MSR.2015.40.

[31] Shaowei Wang i David Lo. “AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization”. *Journal of Software: Evolution and Process* 28.10 (2016), s. 921–942. DOI: 10.1002/smr.1801.

[32] Shaowei Wang i David Lo. “Version history, similar report, and structure: putting them together for improved bug localization”. *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014.* Red. Chanchal K. Roy, Andrew Begel i Leon Moonen. ACM, 2014, s. 53–63. ISBN: 978-1-4503-2879-1. DOI: 10.1145/2597008.2597148.

[33] Chu-Pan Wong i in. “Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis”. *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014.* IEEE Computer Society, 2014, s. 181–190. ISBN: 978-0-7695-5303-0. DOI: 10.1109/ICSME.2014.40.

[34] Chang Xu, Dacheng Tao i Chao Xu. “A Survey on Multi-view Learning”. *CoRR* abs/1304.5634 (2013). arXiv: 1304.5634.

[35] Xin Ye, Razvan C. Bunescu i Chang Liu. “Learning to rank relevant files for bug reports using domain knowledge”. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* Red. Shing-Chi Cheung, Alessandro Orso i Margaret-Anne D. Storey. ACM, 2014, s. 689–699. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635874.

[36] Xin Ye, Razvan C. Bunescu i Chang Liu. “Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation”. *IEEE Transactions on Software Engineering* 42.4 (2016), s. 379–402. DOI: 10.1109/TSE.2015.2479232.

[37] Klaus Changsun Youm, June Ahn i Eunseok Lee. “Improved bug localization based on code change histories and bug reports”. *Information & Software Technology* 82 (2017), s. 177–192. DOI: 10.1016/j.infsof.2016.11.002.

[38] Yue Yu, Huaimin Wang, Gang Yin i Charles X. Ling. “Reviewer Recommender of Pull-Requests in GitHub”. *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, s. 609–612. ISBN: 978-0-7695-5303-0. DOI: 10.1109/ICSME.2014.107.

[39] Yue Yu, Huaimin Wang, Gang Yin i Tao Wang. “Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?” *Information & Software Technology* 74 (2016), s. 204–218. DOI: 10.1016/j.infsof.2016.01.004.

[40] Jian Zhou, Hongyu Zhang i David Lo. “Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports”. *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Red. Martin Glinz, Gail C. Murphy i Mauro Pezzè. IEEE Computer Society, 2012, s. 14–24. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227210.