Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki

ROZPRAWA DOKTORSKA

Modelowanie jakości kodu źródłowego na podstawie danych gromadzonych w systemach kontroli wersji

mgr inż. Wojciech Frącz

PROMOTOR: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH

PROMOTOR POMOCNICZY: dr inż. Jacek Dajda

Kraków 2019

Dziękuję promotorowi pomocniczemu dr inż. Jackowi Dajdzie za ciągłą współpracę w wielu moich naukowych przedsięwzięciach, a w szczególności przy badaniach prowadzonych w ramach niniejszej rozprawy.

Dziękuję promotorowi dr hab. inż. Markowi Kisiel-Dorohinickiemu, prof. AGH oraz dr hab. inż. Aleksandrowi Byrskiemu, prof. AGH za cenne merytoryczne uwagi do przeprowadzanych prac.

Dziękuję dr. inż. Marcinowi Kurdzielowi za naukowe wsparcie przy opanowywaniu metod uczenia maszynowego.

Dziękuję mojej Żonie za ciągłe motywowanie mnie, bym sięgał wyżej.

Dziękuję mojej Mamie za poświęcony mi czas, bo dzięki niej jestem tu gdzie jestem.

Spis treści

Niniejsza rozprawa jest dostępna także w wersji elektronicznej pod adresem
`https://fracz.com/phd`

Streszczenie

Czytelność kodu źródłowego, rozumiana także jako jego jakość, jest jedną z głównych cech wpływających na jego niezawodność, możliwość ponownego użycia, a przede wszystkim – koszty utrzymania oprogramowania. Kod, pomimo swojej nazwy kojarzącej się z ukrywaniem informacji, powinien być napisany w sposób maksymalnie prosty, pozwalający innym programistom na szybkie jego zrozumienie. Dzięki temu dowolny programista z zespołu pracującego nad projektem w razie pojawienia się nowych wymagań będzie mógł szybko wprowadzić pożądane funkcjonalności, nie naruszając przy tym dotychczasowego działania systemu.

Problem utrzymywania kodu odpowiedniej jakości w tworzonym oprogramowaniu jest znany zarówno programistom, jak i osobom nietechnicznym odpowiedzialnym za projekt. Dobrze rozumiana jest konieczność częstej refaktoryzacji, czyli poprawy jakości kodu bez zmiany jego funkcjonalności. Większość osób związanych z programowaniem jest także świadoma istnienia *zapachów kodu* czy metryk, które wskazują symptomy sugerujące konieczność przeprowadzenia takich przekształceń. W dodatku, mnogość istniejących narzędzi potrafiących je wykrywać powoduje, że w trakcie tworzenia oprogramowania problem jego jakości jest jawny przez cały czas, co umożliwia minimalizację ryzyk z nim związanych.

Pomimo świadomości istnienia kodu o różnej jakości, w dalszym ciągu nie udało się jednoznacznie określić zestawu cech, które świadczą o odpowiedniej czytelności kodu źródłowego. Jest to pojęcie subiektywne, nieco inaczej rozumiane przez każdą osobę związaną z programowaniem. Analiza istniejących metryk kodu źródłowego pokazuje, że żadna z nich nie mówi jednoznacznie o jakości kodu źródłowego.

Autor niniejszej rozprawy pochyla się nad problemem zbudowania jakościowego modelu kodu źródłowego (Source Code Quality Model – SCQM), którego zadaniem jest automatyczna klasyfikacja jakości kodu. Wykorzystano w tym celu metody uczenia maszynowego znane z przetwarzania języka naturalnego i analizy jego wydźwięku. Skoro skutecznie rozwijane są rozwiązania poprawnie klasyfikujące pozytywny lub negatywny wydźwięk analizowanego tekstu, w podobny sposób można potraktować kod

źródłowy – jako mechanizm komunikacji programisty z komputerem – i przeanalizować, czy komunikacja ta przebiega na odpowiednim poziomie jakości.

Rozprawa wykazuje, że jakościowy model kodu źródłowego zrealizowany na podstawie dwukierunkowej rekurencyjnej sieci neuronowej cechuje większa trafność w rozpoznawaniu jakości kodu źródłowego niż jakiekolwiek z dostępnych obecnie rozwiązań. W przeprowadzonej ewaluacji zbudowany model poprawnie sklasyfikował jakość kodu dla 79% przykładów, podczas gdy druga pod względem skuteczności istniejąca metryka osiągnęła jedynie 57%.

Aby osiągnąć ten cel, autor rozprawy wybrał ponad 15000 zmian wprowadzających refaktoryzację do 350 najpopularniejszych projektów z portalu GitHub tworzonych w języku Java. Posłużyły one do zbudowania danych treningowych dla zaprojektowanego modelu, który został nauczony subiektywnego pojęcia kodu źródłowego wysokiej jakości.

Ponadto, w celu zwiększenia skuteczności stworzonego modelu przeprowadzono badanie wśród programistów mające na celu zebrać opinie o jakości zaprezentowanych przykładów kodu źródłowego. W badaniu wzięło udział ponad 500 programistów, oddając ponad 5000 głosów i klasyfikując w ten sposób 645 przykładów refaktoryzacji. Zbiór tych refaktoryzacji został wykorzystany zarówno do trenowania jak i walidacji modelu. Zostały one także opublikowane jako jakościowy *benchmark* kodu źródłowego.

Wykorzystanie jakościowego modelu kodu źródłowego (SCQM) do automatycznej klasyfikacji nowych zmian w projekcie pozwoli ograniczyć czas poświęcany na jego utrzymanie, a co za tym idzie – obniży koszty tworzenia oprogramowania. Ponadto, może on być wykorzystany do analizy jakości istniejącego kodu w celu zidentyfikowania miejsc wymagających refaktoryzacji lub nawet błędów niezauważonych podczas przeglądów kodu. Zbudowany jakościowy *benchmark* kodu źródłowego może być wykorzystany przy ewaluacji innych prac mających na celu analizę lub klasyfikację jakości kodu źródłowego.

Abstract

A source code readability, also known as its quality, is one of the most important factors that influence software reliability, reusability, and its maintenance costs. Although the word "code" may be associated with information hiding, the source code of any program should be written in the easiest possible way. The straightforward implementation enables other programmers to understand it rapidly. In consequence, when new software requirements arise, any member of the development team is able to introduce the desired changes without spending to much time on finding an appropriate solution or breaking existing features.

The demand for maintaining appropriate quality of the source code is often known for both programmers and non-technical people involved in the project. They understand the need for continuous refactorings of the source code, that is improving its quality without behavioral changes. Most of the programmers are also aware of *code smells* or source code metrics that indicate possible problems. Moreover, they tend to remember the source code quality requirements during development due to the numerous tools that support detecting potential source code defects in real time.

Although there is a common awareness of the benefits from maintaining an appropriate source code quality level, the set of attributes that clearly indicates its readability has yet to be discovered. Nowadays, the source code quality is still a strongly opinion based concept. Moreover, the analysis of the existing source code metrics reveals the fact that none of them can be used to classify it apparently.

The author of the thesis contributes to this problem by building a Source Code Quality Model (SCQM) that classifies the quality of the source code automatically. It leverages the machine learning methods commonly used in natural language processing. The model design assumes treating the source code as a mean of communication between the programmer and the computer. Such attitude enabled the author to use the sentiment classification methodology to predict whether such communication is conducted at the appropriate quality level.

The thesis concludes that the Source Code Quality Model implemented on top of a bidirectional recurrent neural network reaches better accuracy in classifying the

source code quality than currently available tools or metrics. An evaluation of the model results in detecting appropriate source code quality for 79% of examples. The second best compared solution succeeded in 57% only.

This achievement has been preceded with a gathering of over 15000 source code changes that introduce refactorings to 350 top Java open source projects from the GitHub. Collected samples have been used to build a training dataset for the designed model which has been taught an opinion based classification of the source code readability.

Moreover, a subset of gathered refactoring samples has been further categorized in a source code quality survey in order to further improve the total accuracy of the model. More than 500 programmers took part in the experiment, leaving over 5000 opinions about the presented source code examples. In consequence, besides the training dataset collected automatically, the author managed to build a source code quality *benchmark* out of the 645 samples that have been precisely classified by humans. The *benchmark* has been used to both training and validation of the model.

Adopting the SCQM automatic source code quality classifications will increase the speed of development and decrease maintenance costs. The model can also be used on existing software for detecting lower quality source code fragments or even defects unnoticed during code reviews. Last but not least, the source code quality *benchmark* can be employed to evaluate any future work that builds a solution for code quality classification.

Rozdział 1

Wstęp

Oprogramowanie jest zestawem instrukcji i danych, które przetworzone przez komputer realizują założone cele. Instrukcje te wyrażone są w postaci kodu źródłowego implementowanego przez programistów. Kod ten, poza poprawnością weryfikowaną przez dostarczanie użytkownikom odpowiednich funkcjonalności, cechuje też wiele innych aspektów.

Jedną z takich cech jest efektywność kodu, czyli ilość zasobów (czasu, pamięci) koniecznych do zrealizowania przez dany kod określonego zadania. Można łatwo porównać dwie implementacje realizujące ten sam cel pod kątem efektywności. Wystarczy zmierzyć czas, w którym obydwie postaci generują odpowiedź lub wyznaczyć minimalną ilość pamięci operacyjnej koniecznej do jej ukończenia. Mówimy wtedy, że dany kod źródłowy lub dany algorytm jest bardziej efektywny od innego.

Inną cechą, według której można porównać kody źródłowe jest ich czytelność. Jeśli przez efektywność kodu rozumiemy *możliwość szybkiego wykonania go przez komputer*, to czytelność kodu można przedstawić jako *jego zdolność do bycia zrozumianym przez czytającego go programistę*. Można określić, że dany kod jest bardziej czytelny od innego, jeśli średni czas poświęcony na jego zrozumienie przez programistę o podobnym doświadczeniu jest krótszy.

Po krótkim zastanowieniu się nad problemem czytelności kodu źródłowego okazuje się, że o ile efektywność kodu źródłowego możemy w łatwy i automatyczny sposób zmierzyć, o tyle jego czytelność nie jest już problemem dużo bardziej złożonym. Mamy tu do czynienia bowiem z subiektywną opinią programisty o danym przykładzie. Opinie te mogą się różnić, a różnice te mogą wynikać z przyzwyczajeń analizujących je osób, różnego poziomu doświadczenia lub znajomości dziedziny, w której osadzony jest rozwiązywany oprogramowaniem problem [1].

Mimo tak luźnej definicji czytelności kodu źródłowego, jest to aspekt często analizowany w literaturze. Przekłada się ona bowiem bezpośrednio na wiele innych aspektów, dotyczących nie tylko samego kodu ale także całego tworzonego oprogramowania. Są to np. niezawodność, przenośność, a także możliwość ponownego użycia i stopień skomplikowania implementowanych funkcjonalności, czy w końcu koszty utrzymania całego systemu [1, 2].

Czytelność kodu bywa także utożsamiana z jego jakością. Precyzyjniej, czytelność kodu źródłowego jest ściśle związana z jakością tworzonego oprogramowania [3], dlatego możemy mówić także o jakości kodu źródłowego. Kod jest wyższej jakości, jeśli łatwiej jest w nim odnaleźć i szybko naprawić ewentualny błąd. Kod jest wyższej jakości, jeśli łatwiej jest do niego dodać nową, wymaganą przez użytkowników funkcjonalność. Kod jest wyższej jakości, jeśli dowolny programista z zespołu jest go w stanie szybko zrozumieć i wykonać wspomniane wcześniej czynności. Pomimo swojej nazwy, „kod" źródłowy nie powinien ukrywać informacji o sposobie swojego działania. Nie powinien być *szyfrem* trudnym do zrozumienia. Czytelnie napisany kod źródłowy powinna zrozumieć nawet nietechniczna osoba, która nie potrafi programować, co bardzo dobrze pokazuje prelekcja *Nie koduj, pisz proszę* [4]. Robert C. Martin w swojej popularnej książce w całości poświęconej technikom pisania czytelnego kodu [5] pisze, że *kod źródłowy powinno czytać się tak jak książkę*.

Jest wiele technik, które przypominają programistom lub nawet wymuszają na nich utrzymywanie kodu na odpowiednim poziomie jakości. Zostały one opisane dokładnie w rozdziale 2. Niemniej jednak, najpopularniejszą z nich od dłuższego czasu są przeglądy kodu źródłowego [6]. Technika ta polega na ocenie kodu źródłowego napisanego przez innego programistę. Ocenia on jego czytelność oraz poprawność, co może skutkować brakiem akceptacji i oddaniem przesłanej implementacji wraz z uwagami do autora kodu w celu refaktoryzacji, czyli poprawy jego jakości [7].

Podstawowym wsparciem programistów podczas lub nawet przed wykonaniem przeglądu kodu źródłowego są narzędzia poddające kod statycznej analizie [8]. Narzędzia takie nazywane są także *linterami* (z ang. *lint* – niepożądane strzępki materiału znajdywane na ubraniu) [9, 10]. Ich zadaniem jest analiza kodu źródłowego w celu odnalezienia w nich podejrzanych miejsc, zwanych *zapachami kodu* (ang. *code smells*) [5]. Zapachy kodu wskazują potencjalny problem z kodem źródłowym. Przykładowo może to być zbędny komentarz lub powtórzenie kilka razy tych samych instrukcji. Jest to więc prawdopodobny problem z kodem źródłowym, który może być automatycznie wykryty i zasugerowany autorowi tak, że programista poświęcający swój czas na przegląd kodu nie musi już zwracać uwagi na tego typu problemy.

Oczywiście zapachy kodu mogą być dużo bardziej zawiłe, identyfikując np. nieużywane zmienne czy zbyt długie lub zagnieżdżone fragmenty i konstrukcje, które mogą prowadzić do błędnego wykonania programu. Nie można więc powiedzieć, że lintery są prostymi programami i jak wskazano w [11], zdecydowanie przyspieszają sesje przeglądów kodu. Niemniej jednak narzędzia te nie potrafią jednoznacznie sklasyfikować jakości kodu źródłowego. Bez wątpienia, implementacja nacechowana zapachami kodu może być uznana za kod niskiej jakości. Czy można natomiast odwrócić to spostrzeżenie? Fakt, że *linter* nie odnalazł żadnego zapachu w kodzie źródłowym nie świadczy od razu o jego odpowiedniej jakości. Nadal wymagany jest przegląd kodu, by to rozstrzygnąć.

Ze względu na brak dostępnych narzędzi potrafiących poprawnie sklasyfikować jakość kodu źródłowego, autor rozprawy postanowił połączyć znane metody uczenia głębokiego i wiedzę od programistów wykonujących przeglądy by zbudować jakościowy model kodu źródłowego. Jego zadaniem będzie automatyczna ocena jakości kodu źródłowego. Taka ocena będzie niezwykle przydatna podczas przeglądu kodu, ponieważ kod odpowiedniej jakości będzie mógł być pominięty przez programistę, a co za tym idzie – przeglądy kodu zostaną skrócone, a programiści będą mogli więcej czasu poświęcić na rozwój oprogramowania. Ponadto, automatyczna klasyfikacja jakości kodu podczas jego tworzenia pomoże wypracować wśród programistów przyzwyczajenie ciągłego dbania nie tylko o poprawne działanie kodu, ale także o jego czytelność i utrzymywalność. To z kolei powinno przyczynić się do dalszego rozwoju inżynierii oprogramowania oraz poprawy komfortu pracy programistów.

1.1. Teza i cele rozprawy

Programista tworząc oprogramowanie poświęca 10 razy więcej czasu na czytanie kodu niż na jego pisanie [5]. Wynika to z faktu, że aby poprawnie zaimplementować wymaganą funkcjonalność, konieczne jest zrozumienie kodu, który już istnieje. Należy podjąć na tym etapie szereg decyzji: w której części oprogramowania należy kod dodać, jak go zaimplementować by nie wprowadzić regresji, czyli naruszenia działających dotąd funkcjonalności oraz jak nowy kod źródłowy zorganizować by można było go w przyszłości rozwijać [12].

Ponadto, kod jest także analizowany w trakcie pracy programistów przy wykonywaniu wspomnianej już we wstępie praktyki przeglądów kodu źródłowego. Pomimo swojej skuteczności w odnajdywaniu defektów kodu i identyfikowaniu zbyt niskiej jego jakości podczas przeglądów [6, 13], czynność ta nie jest chętnie wykonywana przez programistów [14]. Zdecydowana większość woli pisać kod niż analizować czytelność

przykładu nienapisanego przez siebie, uważając że jest to mniej kreatywne zajęcie. W książce „Best kept secrets of peer code reviews" [14] spotykamy się nawet z nazwaniem tego zjawiska jako *opór przed wykonaniem przeglądu kodu źródłowego* (z ang. *Resistance to Code Review*). Autorzy twierdzą, że główną winę za ten stan rzeczy ponosi spadek produktywności, gdy potrzeba podjęcia kodu do przeglądu pojawia się w trakcie dnia zbyt często. Z kolei publikacja [15] bada powody, dla których praktyka przeglądów jest bądź nie jest wdrażana w różnego rodzaju przedsiębiorstwach. Z badań wynika, że poważnym problemem w tej praktyce są także względy społeczne i emocje towarzyszące przeglądom kodu w zespole programistów. Nikt nie lubi być poprawiany, niechętnie też wskazujemy błędy w kodzie dobrego kolegi z pracy lub nawet przełożonego.

Częstość wykonywania przeglądów kodu mogłaby być zmniejszona gdyby programista miał możliwość w trakcie swojej pracy dokonania automatycznej oceny jakości tworzonego przez siebie kodu źródłowego. Niestety, w literaturze nie odnaleziono istniejącego modelu umożliwiającego taką klasyfikację. Wspomniane na początku niniejszego rozdziału narzędzia skupiają się na statycznej analizie kodu źródłowego i wykrywają fragmenty kodu uznawane za potencjalnie problematyczne. Nie potrafią one natomiast stwierdzić, czy dostarczona implementacja jest czytelna i będzie mogła być łatwo zrozumiana i zmodyfikowana, gdy pojawią się nowe wymagania.

Zdaniem autora rozprawy możliwe jest stworzenie jakościowego modelu kodu źródłowego, który będzie wyznaczał jakość istniejącej implementacji lub wprowadzanej przez programistę zmiany. Modelu, który nie będzie – jak *lintery* – skupiał się wyłącznie na skończonej liście zapachów kodu, ale będzie analizował kod całościowo i będzie potrafił wskazać, że dana zmiana rzeczywiście podnosi jego czytelność lub ją obniża i w konsekwencji wymaga wykonania refaktoryzacji. Dzięki temu tworzony kod źródłowy nie tylko nie będzie posiadał zapachów kodu wyeliminowanych przez *lintery*, ale będzie też kodem czytelnym, który może być dużo szybciej przeanalizowany przez programistę pod kątem defektów funkcjonalnych. To z kolei podniesie jakość całego oprogramowania, przyspieszy jego rozwój i obniży koszty jego utrzymania. Konieczność utrzymywania kodu wysokiej jakości oraz analiza dostępnych narzędzi próbujących ją klasyfikować skłoniły autora rozprawy do postawienia następującej tezy:

> *Opracowanie jakościowego modelu kodu źródłowego z użyciem technik uczenia maszynowego na podstawie danych gromadzonych w systemach kontroli wersji pozwoli na wykazanie większej trafności w rozpoznawaniu kodu niskiej jakości niż dostępne obecnie narzędzia wykorzystujące jego statyczną analizę.*

Mając na uwadze dążenie do potwierdzenia postawionej powyżej tezy, zostały przyjęte następujące cele rozprawy.

1. **Analiza istniejących rozwiązań pozwalających na utrzymanie kodu źródłowego wysokiej jakości.** Dzięki dokładnemu poznaniu literatury o przedstawionym problemie oraz rozwiązań, które motywują do tworzenia kodu wysokiej jakości zostanie uzasadniona ważność tworzonego rozwiązania, a czytelność kodu zostanie przedstawiona jako nietrywialna w klasyfikacji, ale niezwykle istotna cecha kodu źródłowego.

2. **Opracowanie jakościowego modelu kodu źródłowego**, który będzie w sposób automatyczny klasyfikował jakość kodu. Skuteczność tego modelu zostanie porównana do istniejących narzędzi i technik klasyfikujących jakość kodu źródłowego przy użyciu jego statycznej analizy. Jest to główny cel rozprawy.

3. **Zgromadzenie danych treningowych dla jakościowego modelu.** Odpowiednio duża liczba przykładów kodu źródłowego o różnej jakości będzie stanowić podstawę wiedzy dla tworzonego modelu. Dane te zostaną pozyskane z systemów kontroli wersji otwartych projektów.

4. **Zaprojektowanie i implementacja modelu.** Po dokonaniu rozpoznania istniejących metod uczenia maszynowego zostanie zaprojektowany i zaimplementowany odpowiedni model potrafiący wydobyć wiedzę z zebranych danych.

5. **Zebranie opinii o jakości kodu źródłowego od programistów.** W ramach rozprawy zostanie przeprowadzony eksperyment, w którym programiści wykonają przegląd kodu źródłowego zaprezentowanych przykładów fragmentów oprogramowania o różnej jakości. To pozwoli na wprowadzenie do modelu rzeczywistych opinii programistów o jakości kodu.

6. **Zbudowanie zbioru danych walidacyjnych pozwalających na ewaluację jakościowych modeli kodu źródłowego.** Na podstawie opinii pozyskanych od programistów w przeprowadzonym badaniu zostanie stworzony i udostępniony zestaw danych walidacyjnych, który umożliwi ewaluację tworzonego modelu, a także sprawdzenie efektywności dowolnych innych metod starających się poprawnie klasyfikować jakość kodu źródłowego.

7. **Przygotowanie narzędzia oferującego funkcjonalność klasyfikacji kodu za pomocą stworzonego modelu.** Dzięki temu rezultat rozprawy będzie mógł być w prosty sposób zintegrowany z istniejącymi narzędziami, które wspierają tworzenie oprogramowania lub wykonywanie przeglądów kodu źródłowego.

1.2. Zakres prac i wyzwania stawianie rozprawie

Jakościowy model kodu źródłowego będzie ograniczony do jednego języka programowania – Java. Język ten został wybrany ze względu na jego popularność. Według ankiety przeprowadzanej w 2018 roku przez jeden z największych portali dla programistów – StackOverflow[1] – Java jest najpopularniejszym i najchętniej stosowanym obiektowym językiem programowania [16] (uplasowała się na piątym miejscu, zaraz za niezorientowanymi obiektowo językami: JavaScript, HTML, CSS, SQL). Dzięki popularności Javy, łatwiej będzie pozyskać dane treningowe dla modelu oraz znaleźć programistów, którzy wezmą udział w badaniu opinii o jakości kodu. Ponadto, wynikowy jakościowy model kodu źródłowego znajdzie szerokie zastosowanie dzięki dużej liczbie projektów implementowanych w tym języku programowania.

Ograniczenie się do Javy implikuje także ograniczenie się do narzędzi i metryk, z którymi będzie porównany tworzony model. Naturalnym będzie więc wybór jednego z popularnych *linterów* dla Javy i porównanie się do efektów jego klasyfikacji.

Pomimo ograniczenia się do wybranego języka programowania, przygotowanie odpowiednich danych treningowych jest nie lada wyzwaniem. Pierwszym problemem, który się pojawia jest heterogeniczność prowadzonych dzisiaj projektów. Ze względu na mnogość języków programowania oraz technologii w nich wykorzystywanych praktycznie nie istnieją już projekty tworzone z użyciem tylko jednego z nich. To utrudnia jednoznaczne stwierdzenie, że dany projekt jest tworzony w danym języku programowania, co z kolei może utrudnić automatyczną analizę dostępnych, otwartych projektów pod kątem wystąpienia w nich poszukiwanych zmian refaktoryzacyjnych.

Kolejnym problemem może okazać się sama identyfikacja tych zmian. W trakcie pracy nie wymaga się od programistów, by w jakiś specjalny sposób oznaczali przeprowadzane refaktoryzacje. Problem ten wymaga więc dokładnej analizy tak, aby możliwe było pozyskanie dużej liczby takich zmian w sposób możliwie zautomatyzowany.

Po zebraniu przykładów kodu o różnej jakości dużym wyzwaniem będzie przeprowadzenie badania mającego na celu zebranie opinii o jakości kodu od programistów. Jak każdy eksperyment wymagający udziału respondentów, musi być on starannie przygotowany ponieważ będzie go bardzo trudno powtórzyć. Ponadto, takie badania wymagają poświęcenia sporej ilości czasu oraz odpowiedniej liczby ludzi, których w kontekście zaprojektowanego badania musi jeszcze łączyć jedna profesja – programisty.

Na koniec, gdy już wszystkie konieczne dane zostaną zebrane, jakościowy model

[1] https://stackoverflow.com

musi zostać zaimplementowany oraz wytrenowany posiadanymi przykładami kodu o różnej jakości. To z kolei wymaga dostępu do dużych zasobów obliczeniowych.

Pomysł na stworzenie jakościowego modelu kodu źródłowego jest więc pomysłem ambitnym, stawiających przed doktorantem wiele wyzwań. Jak jednak pokaże lektura niniejszej rozprawy, oczekiwany efekt udało się osiągnąć, a wszystkie niewiadome, które budziły niepewność na etapie początkowym udało się rozwiązać w prostszy lub bardziej skomplikowany – ale zawsze skuteczny – sposób.

W trakcie gromadzenia danych i budowy modelu wdrożono jeszcze kilka mniejszych ograniczeń, które umożliwiły implementację modelu (zob. sekcję 3.2). Oczywiste jest więc, że jakościowy model kodu źródłowego przedstawiony w tej rozprawie nie zastąpi całkowicie programisty wykonującego przegląd kodu źródłowego.

1.3. Struktura rozprawy

W rozdziale 2 niniejszej rozprawy przedstawiono istniejące techniki ułatwiające pisanie kodu wysokiej jakości oraz praktyki i narzędzia, które umożliwiają kontrolę tego procesu. Opis ten stanowi zarys aktualnego stanu wiedzy na temat czytelności kodu źródłowego oraz przedstawia aktualne trendy w rozwoju i badaniu tej tematyki.

Rozdział 3 przedstawia koncepcję jakościowego modelu kodu źródłowego. Opisano założenia, które powinien on spełniać oraz zaprojektowano mechanizm jego działania.

W rozdziale 4 opisano sposób zebrania danych treningowych, które umożliwiły przekazanie do tworzonego modelu odpowiedniej wiedzy na temat jakości kodu źródłowego. Pojawia się tutaj sporo szczegółów implementacyjnych, które pozwolą lepiej poznać i zrozumieć problemy, z którymi zmierzył się doktorant, a także przedstawią tok rozumowania towarzyszący autorowi w trakcie analizy problemu. W rozdziale 5 opisano sposób przeprowadzenia i wyniki badania zaprojektowanego w celu pozyskania opinii programistów na temat jakości kodu źródłowego.

Rozdział 6 zawiera opis implementacji sieci neuronowej oraz przebiegu jej trenowania. Przedstawiono różne kombinacje parametrów uczenia i podjęte próby, które w konsekwencji doprowadziły do oczekiwanego rezultatu i powstania jakościowego modelu kodu. Uzyskany efekt był następnie poddany analizie porównawczej stworzonego model z istniejącymi rozwiązaniami, co zostało przedstawione w rozdziale 7.

Rozdział 8 zamyka niniejszą rozprawę, przedstawiając sposób realizacji postawionych jej celów i potwierdzając poprawność postawionej tezy. Znalazły się tu także propozycje dalszego rozwoju zaproponowanego rozwiązania.

Rozdział 2

Problem utrzymania kodu źródłowego odpowiedniej jakości

Jak opisano we wstępie, jakość kodu źródłowego przekłada się bezpośrednio na koszty utrzymania oprogramowania [1]. Konieczność utrzymywania kodu źródłowego wysokiej jakości został zidentyfikowany niedługo po pojawieniu się pierwszych języków programowania. Już w latach 70. XX wieku proponowano przeprowadzanie formalnych inspekcji kodu źródłowego [17] mających na celu kontrolę jego jakości. Dziś ta praktyka znana jest jako przeglądy kodu źródłowego [6].

W tym rozdziale autor rozprawy przygląda się problemowi utrzymania odpowiedniej jakości kodu źródłowego w tworzonym oprogramowaniu oraz przedstawia istniejące techniki pozwalające programistom na pozostawianie po sobie kodu zrozumiałego dla innych osób.

2.1. Definicja jakości kodu źródłowego

Jakość kodu źródłowego należy rozumieć jako jego cechę mówiącą o tym *jak dobrze jest on napisany* [18]. Im wyższa jakość kodu, tym łatwiej będzie rozwijać oparte o niego oprogramowanie.

Praca programisty wprowadzającego modyfikację do istniejącego oprogramowania składa się na ogół z czterech, następujących po sobie etapów [18]:

1. Programista określa *gdzie* i *jak* zmienić lub dodać kod źródłowy.
2. Implementuje zmianę.

3. Upewnia się, że nie wprowadził regresji (tj. naruszenia działającej dotąd funkcjonalności).

4. Upewnia się, że wprowadzana zmiana spełnia założone wymagania.

Istniejący kod jest odpowiedniej jakości, gdy czas poświęcony przez programistów na wprowadzenie nowej funkcjonalności lub naprawienie defektu powodującego problem jest porównywalny z subiektywnym rozmiarem tego zadania. To założenie będzie spełnione wyłącznie, gdy programista poświęci największą ilość czasu swojej pracy na etap drugi, tj. dostarczenie rozwiązania dla aktualnego zadania. Wysiłek włożony w pozostałe etapy powinien być minimalizowany. Innymi słowy, programiści powinni szybko odnaleźć się w istniejącej implementacji (1), a po wprowadzeniu zmian nie powinni mieć wątpliwości co do spełnienia nowych wymagań (4) czy nienaruszenia już istniejących (3).

Szybkie przejście od zadania do implementacji jest możliwe wyłącznie gdy programista nie napotka trudności przy próbie zrozumienia istniejącego kodu źródłowego. Z tego powodu jakość kodu źródłowego jest często utożsamiana z jego czytelnością [3]. Czytelność powinna tu być rozumiana jako łatwość zrozumienia użytych konstrukcji językowych, nazw elementów czy zaprojektowanego algorytmu.

Kod odpowiedniej jakości nie powinien pozwalać na przypadkowe wprowadzanie regresji. Struktura dobrze napisanego kodu będzie zapewniać programiście świadomość wszystkich konsekwencji swoich modyfikacji. Dlatego wprowadzając zmianę do czytelnego i dobrze zrozumianego kodu programista już w trakcie projektowania i implementacji swojego rozwiązania zadba o weryfikację istniejących funkcjonalności.

W czwartym etapie programista najczęściej posługuje się automatycznymi testami oprogramowania. Testy są kodem źródłowym, którego zadaniem jest wykorzystanie danego przypadku użycia systemu i zweryfikowanie, czy testowane oprogramowanie zachowuje się zgodnie z wymaganiami. Istnieją nawet praktyki sugerujące tworzenie kodu testów przed implementacją [19], co w założeniu ma powodować jeszcze większe zwrócenie uwagi programisty na postawione przed nim wyzwania. Nie każda struktura kodu źródłowego pozwala na jej dokładne przetestowanie. Z tego powodu, na jakość kodu źródłowego składa się również jego podatność na bycie testowanym. Tylko przy zachowaniu *testowalnego* kodu źródłowego programista może szybko zakończyć etap pracy poświęcony weryfikowaniu nowych zmian. Co więcej, pozostawienie po każdej wprowadzonej zmianie zestawu testów weryfikujących nowe wymagania znacząco ułatwia uniknięcie regresji. Każde bowiem wymaganie, które pojawiło w trakcie rozwoju oprogramowania posiada przypisany mu test.

Skoro wysoka jakość kodu źródłowego pozwala na łatwiejsze utrzymywanie oprogramowania, przekłada się ona bezpośrednio na koszty tworzenia oprogramowania. Ponadto, system oparty o kod wysokiej jakości jest systemem niezawodnym.

Nie bez powodu więc problemowi jakości kodu źródłowego poświęcono już dotąd wiele uwagi. Należy jednak zauważyć, że w dalszym ciągu jakość kodu źródłowego jest cechą subiektywną, postrzeganą różnie przez różnych programistów.

2.2. Refaktoryzacja sposobem na utrzymanie jakości kodu

Refaktoryzacja jest techniką polegającą na modyfikacji kodu źródłowego tak, by poprawić jego jakość ale nie zmienić funkcjonalności [7]. Jest to podstawowa metoda utrzymania kodu źródłowego na odpowiednim poziomie jakości.

2.2.1. Techniki refaktoryzacji

Istnieje wiele sposobów na wykonywanie refaktoryzacji [20]. Najprostszą refaktoryzacją kodu jest choćby poprawienie nazw zmiennych, metod, klas występujących w kodzie źródłowym tak, aby nie było konieczne zrozumienie implementacji w celu poznania realizowanej przez danych fragment kodu funkcjonalności. Taka refaktoryzacja nie zmienia struktury kodu ani sposobu jego wykonania. Wpływa jednak na czytelność kodu źródłowego, a dzięki temu też bezpośrednio na jego jakość.

Podczas refaktoryzacji programista może jednak wykonywać znacznie większe przekształcenia kodu źródłowego, przy zachowaniu wspomnianego wyżej warunku: nie może zmienić się rezultat działania kodu. Najczęściej stosowanymi przekształceniami refaktoryzacyjnymi są zidentyfikowanie i ujednolicenie powtarzającego się kodu oraz podzielenie długiego fragmentu kodu na kilka mniejszych, nazwanych odpowiednio operacji [21]. W literaturze można trafić na badania proponujące sposoby wyboru odpowiedniej techniki refaktoryzacji po analizie zastanego kodu [22] lub nawet sugerujące z jakich etapów powinna składać się czynność refaktoryzacji [23].

Możliwe do wykonania przekształcenia refaktoryzacyjne są ściśle zależne od używanego języka programowania. Refaktoryzacje dotyczące klas spotykamy tylko w językach zorientowanych obiektowo, a zmiany doprecyzowujące typ parametru funkcji – tylko w językach statycznie typowanych. Badanie [24] sprawdza nawet wpływ używanego języka programowania na sposób pojmowania jakości kodu źródłowego. Niektóre elementy języków programowania pomagają w jej utrzymaniu bez stosowania

żadnych dodatkowych technik. Mogą także skrócić czas konieczny do implementacji danego rozwiązania [25] lub pomóc w jego zrozumieniu i wykorzystaniu [26].

Istnieje wiele narzędzi wspomagających proces refaktoryzacji kodu źródłowego [27]. Często oprogramowanie, w którym programista pisze kod – Integrated Development Environment (IDE) – posiada wbudowane funkcje proponujące i przeprowadzające refaktoryzację automatycznie lub jedynie z małą pomocą programisty. Przykładowo: wystarczy zmienić nazwę zmiennej w jednym miejscu, a IDE rozpropaguje ją w pozostałych miejscach gdzie została użyta. Takie wsparcie zdecydowanie przyspiesza wykonywane przekształcenia i zachęca do częstszego myślenia o jakości kodu w trakcie pracy.

2.2.2. Dlaczego kod wymaga refaktoryzacji?

Problem utrzymania kodu wysokiej jakości jest na tyle złożony, że nie można bezpośrednio powiązać go z intencją programisty, który go napisał w swojej pracy. Głównymi powodami, dla którego programista zostawia po sobie nieczytelny kod są limity czasowe na wykonanie powierzonych mu zadań oraz często zmieniające się wymagania stawiane wytwarzanemu oprogramowaniu. W książce „Clean Code" [5] przeczytamy, że w miarę rozwoju oprogramowania kod „gnije". Na początku projektu, gdy jego funkcjonalności są skromne i doskonale określone, refaktoryzacja może w ogóle okazać się zbędną praktyką. Z biegiem czasu, pomysł na implementację, który wcześniej wydawał się idealny do dostarczenia danej funkcjonalności może już nie sprawdzać się tak dobrze po otrzymaniu nowych wymagań. Programista jest wtedy postawiony przed dylematem: dodać wymaganą funkcjonalność w krótszym czasie, godząc się na wybór nieoptymalnego pod względem jakości rozwiązania, czy może poświęcić więcej czasu i najpierw dostosować strukturę kodu do nowych wymagań.

W ten sposób w kodzie źródłowym pojawia się zjawisko zwane „długiem technologicznym" (z ang. *techincal debt*) [28]. Jest ono rezultatem niepoświęcania wystarczającego czasu na dbałość o jakość kodu źródłowego, co skutkuje pogorszeniem jakości wytwarzanego oprogramowania. Według autorów wspomnianej publikacji, pojęcie to zostało stworzone z myślą o nietechnicznych osobach odpowiedzialnych za projekt, by uświadomić im konieczność przeprowadzania refaktoryzacji kodu. Łatwiej jest wytłumaczyć, że czas zapożyczony od technologii na jej nieoptymalnym wykorzystaniu trzeba spłacić, wykonując refaktoryzację.

Autorzy publikacji [21] zbadali powody, dla których programiści wykonują refaktoryzację. Ku ich zaskoczeniu, głównym powodem wykonywanych refaktoryzacji nie było zidentyfikowanie defektów kodu, ale właśnie zmieniające się wymagania pochodzące

od użytkowników. Artykuł [29] proponuje nawet refaktoryzację jako sposób zrozumienia kodu, przy jednoczesnej poprawie jego jakości. Dzięki temu kolejny programista, który trafi w to miejsce nie będzie ponownie poświęcał czasu na zrozumienie przesadnie skomplikowanego kodu źródłowego. Takie zachowanie Robert C. Martin określa jako *Zasadę dobrego harcerza: zostaw kod lepszym niż go zastałeś* [5].

2.3. Zapachy kodu, czysty kod

Z biegiem czasu zauważono, że konieczność przeprowadzenia refaktoryzacji kodu źródłowego może być zasugerowana przez wystąpienie w nim typowych konstrukcji, które sugerują pojawienie się długu technologicznego. Nazwano je zapachami kodu, z ang. *code smells* [30, 5].

W pracy [30] przedstawiony został podział zapachów kodu na kategorie problemów, które reprezentują. Książka [5] prezentuje listę zapachów kodu wraz z proponowanymi nazwami i sugerowanymi technikami refaktoryzacji pozwalającymi na ich uniknięcie. Poniżej przedstawiono niektóre z kategorii zapachów kodu i podano ich przykłady.

- **Bloaters**, czyli fragmenty kodu źródłowego, które przesadnie się rozrosły i wymagają podzielenia na mniejsze bloki kodu, które będzie można łatwiej zrozumieć. Przykładowe zapachy: *God Class* – klasa realizująca zbyt wiele funkcjonalności, *Too Long Method* – zbyt długi kod w metodzie, wymagający wyekstrahowania go do kilku mniejszych, poprawnie nazwanych metod.

- **Object-Orientation Abusers**, czyli konstrukcje w kodzie naruszające ogólnie przyjęte zasady programowania obiektowego. Jest to na przykład *Refused Bequest*, czyli dostarczanie implementacji dla wybranych klas rozszerzających w klasie nadrzędnej.

- **Change Preventers**, czyli taka organizacja kodu źródłowego, która utrudnia lub wręcz uniemożliwia jego rozwój. Przykładem będzie tu *Shotgun surgery*, czyli rozrzucenie danej funkcjonalności po wielu rożnych klasach w systemie.

- **Dispensables**, czyli niepotrzebny już kod, który z jakiegoś powodu jest pozostawiony w oprogramowaniu. Zaliczymy tutaj *Dead code*, czyli instrukcje, które ze względu na umieszczenie ich w kodzie nigdy nie zostaną osiągnięte lub nawet *Code Duplication*, przy czym w tym przypadku wyrzucenie kodu będzie polegać na jego ujednoliceniu.

- **Copulers**, jako grupa reprezentująca fragmenty kodu niepotrzebnie związane ze sobą. Przykład: *Feature Envy* jest nadużywaniem istniejącego interfejsu klasy zamiast wzbogacenia jej o nowe zachowanie a *Misplaced Responsibility* – po prostu umieszczeniem funkcjonalności w złym miejscu.

Jakość kodu jest często kojarzona z liczbą występowania w nim zapachów kodu [31]. W literaturze można nawet znaleźć przykłady prac wiążące występowanie zapachów kodu z łatwością utrzymania oprogramowania i częstością wprowadzania w nim zmian [32, 33]. Książka [5] nie pozostawia już żadnych wątpliwości, że zapachy kodu są związane z jego jakością i czytelny kod źródłowy nienacechowany zapachami nazywa po prostu „czystym kodem".

Zapachy kodu zyskały swoją popularność dzięki możliwości ich automatycznego wykrywania. Istnieje szereg narzędzi, które potrafią przeanalizować kod źródłowy pod kątem występowania w nich zapachów kodu [34]. Wynikiem takiej analizy jest raport wskazujący miejsca w kodzie, którym należy się przyjrzeć, gdyż zidentyfikowany w nich zapach kodu może sugerować konieczność refaktoryzacji kodu. Narzędzia analizujące kod źródłowy pod kątem jego jakości wyrażonej w liczbie odnalezionych w nim potencjalnych problemów nazywane są statycznymi analizatorami kodu lub *linterami* [35, 8]. Słowo *statyczne* reprezentuje tutaj fakt, że programy te nie wykonują kodu źródłowego, a jedynie budują z niego odpowiednią postać (np. drzewo rozbioru syntaktycznego), w której z kolei wyszukiwane są zapachy kodu.

Z przeprowadzone badania [36] analizującego znajomość zapachów kodu wśród programistów wynika, że u zdecydowanej większości z nich to pojęcie jest im znane. Co więcej, większość respondentów wskazała, że korzysta z narzędzi wykrywających zapachy kodu ponieważ utożsamiają oni kod zawierający zapachy z kodem niedopracowanym lub nieprzemyślanym. Powyższe badanie potwierdza również, że najczęściej występującymi zapachami kodu są *bloaters* wymagające podziału kodu na mniejsze bloki oraz *dispensables*, w szczególności – duplikacja kodu. Te statystyki pokrywają się z najczęściej przeprowadzanymi typami refaktoryzacji omówionymi w [21].

Pomimo skutecznego powiązania zapachów kodu z jakością i łatwością utrzymania oprogramowania, [33] wskazuje wiele innych postaci kodu źródłowego, które nie zawierają zapachów kodu, a mimo to zostały wskazane przez programistów jako niepożądane. To uświadamia, że zapachy kodu nie są jedynym symptomem nieodpowiedniej jakości kodu źródłowego.

2.4. Metryki kodu źródłowego

Metryki kodu są miarami, które za pomocą wartości liczbowej reprezentują jego wybraną cechę [37]. W pracy dyplomowej „Ocena przekształceń refaktoryzacyjnych z wykorzystaniem metryk kodu źródłowego" [20] autor przeprowadził klasyfikację metryk z punktu widzenia ich skuteczności w rozpoznawaniu kodu niskiej jakości. Wynika z niej, że najlepszą metryką sugerującą nieodpowiednią jakość kodu jest złożoność cyklomatyczna (ang. *cyclomatic complexity*). Mierzy ona stopień skomplikowania programu na podstawie liczby punktów decyzyjnych w zadanym kodzie źródłowym [38]. Im wartość tej metryki jest wyższa, tym kod jest uznawany za bardziej skomplikowany, a co za tym idzie – mniej czytelny czyli niższej jakości.

Innym przykładem interesującej metryki kodu źródłowego w kontekście jego czytelności jest złożoność poznawcza (ang. *cognitive complexity*) [39]. Ustala ona koszt poszczególnych instrukcji rozumiany jako „trudność zrozumienia danej konstrukcji przez programistę". Im więcej takich konstrukcji się pojawi, tym kod uznawany jest za trudniejszy w interpretacji.

Metryki kodu, choć analizują kod w inny sposób w stosunku do zapachów kodu, są często z nimi utożsamiane [40]. Zbyt duże odchylenie wartości danej metryki jest utożsamiane z zapachem przeanalizowanego kodu wskazującym na konieczność jego refaktoryzacji. To połączenie zapachów i metryk kodu źródłowego może też mieć swoją przyczynę w narzędziach, które obliczają metryki kodu [34]. W celu ich wyznaczenia, konieczne jest przygotowanie podobnej struktury w oparciu o kod źródłowy, więc to zadanie jest często realizowane także przez narzędzia do statycznej analizy kodu. Często jest to to samo narzędzie, którego programista używa do wykrywania zapachów kodu źródłowego. Przykładem może być tu choćby Checkstyle użyty do ewaluacji jakościowego modelu zbudowanego w ramach tej rozprawy (zob. rozdział 7). Został on użyty zarówno do wykrycia zapachów kodu jak i do wyznaczenia wartości poszczególnych metryk.

2.5. Przeglądy kodu źródłowego

Przegląd kodu jest praktyką polegającą na sprawdzaniu kodu źródłowego przez innego programistę w celu wykrycia problemów z jego jakością, a także defektów funkcjonalnych. Technika ta jest niezwykle skuteczna w utrzymaniu kodu źródłowego o odpowiedniej jakości [6, 11, 41, 42]. Ponadto, ogromną zaletą wykonywania przeglądów kodu jest rozprzestrzenianie się wiedzy o systemie w zespole programistów i przenikanie wiedzy pomiędzy nimi [43]. Dzięki temu poziom umiejętności w zespole

wyrównuje się. Co więcej, przeglądy w znaczącym stopniu minimalizują *bus factor* projektu, czyli ryzyko przestoju gdy jeden z programistów postanowi opuścić zespół [44].

Początkowo przeglądy kodu były nazywane „inspekcjami" i zostały zdefiniowane przez M. Fagana [17, 45]. Inspekcję zdefiniowano jako etapowy, bardzo formalny proces, któremu może podlegać kod źródłowy, ale też wszystkie inne artefakty będące wynikiem procesu wytwarzania oprogramowania, np. dokumentacja, projekty interfejsu użytkownika czy testy. Grono programistów zbierało się by wspólnie prześledzić i przedyskutować wprowadzane zmiany. Spotkania te trwały bardzo długo, przez co były niechętnie stosowane. Niemniej jednak, już wtedy (lata 70. XX wieku) odkryto, że korzyści płynące z analizy kodu źródłowego przez programistę, który nie jest jego autorem są niezwykle duże. Fagan twierdził, że dzięki inspekcjom w trakcie trwania całego projektu można zaoszczędzić 54 godziny czasu pracy programisty na każde 1000 linii kodu. Te szacunki zostały potwierdzone przez późniejsze prace [46, 14].

Szybko jednak zidentyfikowano problem przeciągających się spotkań inspekcyjnych i zaczęto szukać sposobu ich przyspieszenia [47]. Z biegiem czasu zauważono, że przeglądy kodu mogą być dużo mniej formalne [6, 48]. Nie trzeba wyznaczać specjalnych dni lub pory w ciągu dnia, kiedy programiści powinni wykonywać przeglądy. Aktywność tą można wpleść pomiędzy pisanie kodu źródłowego wtedy, kiedy jest wygodnie oderwać się od bieżących obowiązków. Inspekcje kodu zaczęto nazywać przeglądami kodu źródłowego. Rewolucja, która przekształciła formalne inspekcje w nieformalne, lekkie przeglądy kodu źródłowego w znacznym stopniu przyczyniła się do ich popularyzacji. Dzięki znajomości tej praktyki programiści są świadomi istnienia problemu kodu źródłowego niskiej jakości.

Pomimo przedstawionych korzyści płynących ze stosowania przeglądów kodu i ciągłego rozwoju tej praktyki, jest ona niechętnie wdrażana przez programistów. W literaturze można odnaleźć wiele badań poświęconych przyczynom takiej sytuacji.

Autorzy [49] próbują zidentyfikować, dlaczego przeglądy niektórych zmian są wykonywane szybciej niż innych. Bez zaskoczenia, z przeprowadzonego badania wynika, że ilość czasu koniecznego do wykonania przeglądu wynika przede wszystkim z wielkości danej zmiany. Co więcej, wyniki przeprowadzonego wśród programistów badania opinii o przeglądach kodu pokazują, że im więcej kodu zawiera dana zmiana tym mniej skupiają się oni na szczegółach, a w konsekwencji – pozostawiają po sprawdzeniu takiej zmiany mniej uwag [50]. Na czas wykonania przeglądu mają także wpływ inne czynniki, takie jak precyzyjny opis zadania lub spójność analizowanej zmiany z innymi funkcjonalnościami dodawanymi do tej pory w projekcie [49].

Poza czasem poświęconym na przegląd kodu źródłowego, który według programistów jest czasem wykorzystanym mniej efektywnie niż gdyby pisali oni kod [14], częstym zarzutem przeciwko wprowadzaniu tej praktyki są aspekty miękkie. Jak wspomniano już we wstępie, istnieje prawdopodobieństwo pogorszenia się relacji koleżeńskich w zespole po zbyt często odrzucanym kodzie w przeglądzie kodu [15]. To może spowodować obawy przed oddaniem kodu do sprawdzenia, a także – dla sprawdzającego – obawy przed zgłoszeniem autorowi problemów z przesłanym rozwiązaniem. Co więcej, autorzy publikacji [51] twierdzą, że nie każdy programista posiada cechy osobowości pozwalające mu na bezstresowe wykonanie przeglądu kodu źródłowego innej osoby.

Co ciekawe, w literaturze możemy spotkać się także z sugestiami, że nadużywanie praktyki przeglądów kodu może powodować obniżenie jakości kodu źródłowego. Jeśli programista otrzymuje zbyt wiele implementacji do sprawdzenia, może zacząć mniej przykładać się do wykonywanej czynności i akceptować zmiany, które przy jego większej czujności zostałyby odrzucone [52]. W ten sposób cały zespół jest przekonany, że ze względu na ustalone zasady tworzenia oprogramowania nie rośnie w nim dług technologiczny, podczas gdy przez niedbałe wykonanie przeglądów nie tylko się on pojawia, ale jest ukrywany i po zaakceptowaniu danej zmiany nie jest już dalej analizowany.

Sytuacja wygląda podobnie wśród programistów, którzy są pewni że wyniki ich pracy będą poddane weryfikacji przed wdrożeniem. Programiści przestają zastanawiać się nad najlepszym sposobem rozwiązania postawionego przed nimi problemu, a zaczynają myśleć o tym jak napisać kod, by był on zaakceptowany podczas przeglądu [53]. To z kolei może przerodzić się w tworzenie rozwiązań, które „tylko działają”, a ich struktura pozostawia wiele do życzenia. Pominięto w nich bowiem etap refaktoryzacji będąc pewnym, że w trakcie przeglądu i tak ich rozwiązanie zostanie skrytykowane i pokierowane w inną stronę. Jeśli połączymy taką sytuację z niedbałym wykonaniem przeglądu, do oprogramowania dostanie się w efekcie kod gorszej jakości niż gdyby przeglądy kodu w ogóle nie były wdrożone.

Oczywiście opisane wyżej sytuacje nie zdarzają się często, a liczba korzyści wynikających ze stosowania przeglądów kodu zdecydowanie przewyższa liczbę niebezpieczeństw z nimi związanych. Z tego powodu rozwój tej praktyki jest obserwowany w literaturze od lat. W kolejnych podsekcjach przedstawiono istniejące propozycje wsparcia i usprawnienia praktyki przeglądu kodu źródłowego.

2.5.1. Narzędzia wspierające przeglądy kodu

Odpowiednie wsparcie narzędziowe zdecydowanie pomaga we wkomponowaniu przeglądów kodu do procesu wytwarzania oprogramowania. Praktyka ta stała się na tyle popularna, że zaczęły powstawać narzędzia w pełni jej poświęcone. Jednym z takich narzędzi jest aplikacja Gerrit Code Review [54], która wprowadza do procesu wytwarzania oprogramowanie wymaganie zaakceptowania kodu przez innego programistę przed dołączeniem kodu do produktu. Nawet bez korzystania ze specjalnych narzędzi, jedynie publikując swój kod źródłowy np. na platformie GitHub, programista ma możliwość stworzenia ze swojej zmiany *pull requestu*, czyli prośby o dołączenie napisanego przez siebie kodu do projektu, po uprzednim jego sprawdzeniu.

Przy wprowadzeniu praktyki przeglądów kodu źródłowego do procesu wytwarzania oprogramowania niezwykle ważnym jest, by przed sprawdzeniem kodu źródłowego przez programistę każdą zmianę poddać analizie za pomocą narzędzia statycznie analizujących kod źródłowy – *lintery* (zob. poprzednie sekcje 2.3 i 2.4). Ich zadaniem jest wykrycie potencjalnych defektów i naruszeń jego jakości jeszcze przed poświęceniem na to czasu innego programisty. Dzięki temu osoba sprawdzająca kod nie powinna już zwracać uwagi na te aspekty, które zostały wykryte przez stosowane narzędzia. To pozwala na skrócenie przeglądów, a właśnie czas poświęcany na ich wykonanie jest najczęściej wskazaną przyczyną niechętnego stosowania tej praktyki [14, 42].

W pracy [55] zaproponowano nawet integrację o nazwie *ReviewBot*, która w odpowiedzi na kod przesłany do przeglądu automatycznie publikuje informacje uzyskane z różnych narzędzi wykrywających zapachy kodu i publikuje je formie komentarzy. Symuluje w ten sposób automatyczne wykonanie przeglądu kodu pod kątem występowania w nim zapachów kodu.

2.5.2. Wybór odpowiedniej osoby do przeglądu kodu

Spora liczba publikacji i badań mających na celu usprawnienie przeglądów kodu źródłowego skupia się wokół wyboru odpowiedniej osoby, która powinna go wykonać [56, 57, 58, 59, 60]. Metody te w głównej mierze analizują historię tworzonego kodu źródłowego. Zakłada się, że jeśli dana osoba zna lepiej dany kod źródłowy, będzie w stanie szybciej i dokładniej sprawdzić nową zmianę, która jest do niego wprowadzana.

Każda z przytoczonych prac realizuje algorytm wyboru odpowiedniej osoby w nieco inny sposób, jednakże można mieć do tego pomysłu wiele zastrzeżeń.

Po pierwsze, algorytm wyboru osoby, która powinna sprawdzić dany fragment kodu ma sens tylko przy dużych zespołach, gdzie rzeczywiście taki dylemat może się

pojawić. Metody te nie będą mieć zastosowania w małych zespołach, gdzie często przegląd kodu wykonuje po prostu osoba, która w danej chwili może się oderwać od zadań bieżących [14].

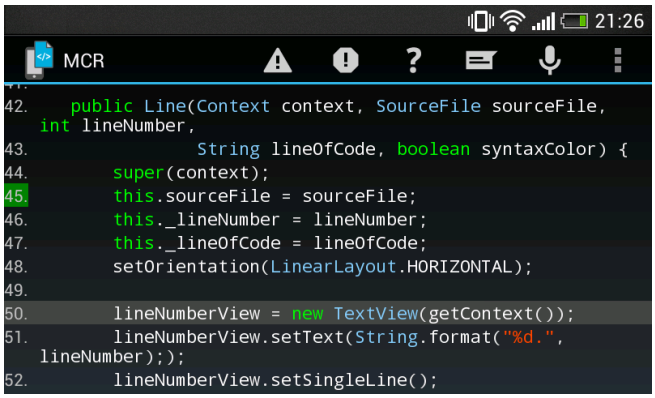
Po drugie, sugerowanie osoby, która powinna sprawdzić daną zmianę w kodzie źródłowym zmniejsza rozprzestrzenianie się wiedzy o tworzonym systemie wśród programistów [61], co zostało wymienione wcześniej jako jedna z zalet płynących ze stosowania omawianej praktyki. Jeśli narzędzie sugerujące osobę do sprawdzenia kodu będzie opierać się na znajomości danego fragmentu przez programistę, szybko powstaną miejsca w systemie, w których jedne osoby się specjalizują, a pozostałe – nie wiedzą o nim prawie nic. To zwiększa niepożądany, omówiony wcześniej wskaźnik *bus factor* całego projektu [44].

W końcu, niedawno przeprowadzane badanie potwierdza, że stosowane rozwiązania w bardzo nielicznych przypadkach przyczyniły się do zasugerowania innej osoby niż ta, która już do tego zadania była wcześniej przydzielona [62].

Pomimo wielkich nadziei związanych z przyspieszeniem wykonywania przeglądów kodu dzięki przydzielaniu odpowiednich osób i pomimo wielu prac zrealizowanych w ostatnim czasie w kierunku rozwoju tego pomysłu, wydaje się że dalsze inwestowanie czasu w ulepszanie przeglądów kodu na tym obszarze mija się z celem. Inaczej sytuacja wygląda z próbami wprowadzania elementów grywalizacji do przeglądów kodu źródłowego.

2.5.3. Przeglądy kodu na urządzeniach mobilnych

Skuteczność przeglądów kodu źródłowego w zapewnianiu odpowiedniej jakości oprogramowania spowodowała podjęcie prób wykonywania ich również przy użyciu dużo mniejszych ekranów. Autor rozprawy w swoich wcześniejszych pracach [63, 64] proponuje nawet przeniesienie przeglądów kodu na urządzenia mobilne (zob. przykładowy zrzut ekranu podczas wykonywania przeglądu na rysunku 2.1). W publikacjach



Rys. 2.1: Przegląd kodu przy użyciu urządzenia mobilnego.

tych wykazano, że nie tylko przegląd kodu przy użyciu telefonu komórkowego jest możliwy ale sama praktyka realizowana w ten jakże wygodny sposób zachowuje wszystkie korzyści znane ze stosowania jej w sposób klasyczny. Ponadto, osoba wykonująca przegląd kodu na małym ekranie jeszcze bardziej skupia się na niektórych elementach jakości kodu źródłowego, takich jak np. odpowiednie nazewnictwo elementów. Okazuje się bowiem, że siedząc w wygodniej sofie w pokoju socjalnym lub w samolocie podczas wyjazdu służbowego możemy traktować przegląd kodu źródłowego jako bardzo przyjemną i pożyteczną czynność.

2.5.4. Gamifikacja przeglądów kodu

Gamifikacja jest wykorzystaniem elementów znanych z gier w środowisku, które dotąd nie było z nimi kojarzone [65]. Technika ta sprawdziła się na wielu różnych płaszczyznach [66]. Dziś nikogo już nie dziwią programy lojalnościowe w różnego rodzaju placówkach handlowych, gdzie za zgromadzone punkty możemy otrzymać małe upominki. Pojawił się więc pomysł, by spróbować wprowadzić tę technikę do przeglądów kodu źródłowego. W literaturze można odnaleźć wiele badań sprawdzających efekty takiej integracji [67, 68, 69].

Jedną z pierwszych prac, której celem było nagradzanie programistów nie tylko za pisanie nowego kodu, ale także za zabieranie głosu w dyskusjach i pozostawianie komentarzy w trakcie przeglądów kodu źródłowego jest publikacja [70]. Autorzy proponują wprowadzenie rankingu programistów opartego o *współczynnik udzielania się w projekcie* (z ang. *contribution factor*), do którego obliczenia używana jest między innymi aktywność podczas przeglądów kodu.

Autor rozprawy również przyczynił się do popularyzacji przeglądów kodu za pomocą grywalizacji. Narzędzie Code Review Analyzer [67] wprowadza do platformy Gerrit Code Review [54] mechanizm nagradzania programistów punktami i odznakami na podstawie ich zaangażowania w projekt w ostatnim czasie. W ewaluacji tego rozwiązania zostało wskazane, że tak proste nagradzanie czy nawet „zauważanie" małych osiągnięć programistów motywuje ich do wykonywania przeglądów kodu, a w efekcie – do podniesienia jakości tworzonego oprogramowania. Podobne narzędzie zostało opracowane także w ramach innego badania [68]. Autorzy niektórych publikacji proponują nawet zupełnie nową aplikację wspierającą przeglądy kody źródłowego, w której od razu osadzone zostały elementy gier i nagród [69].

Dzięki gamifikacji można zmotywować programistów do częstszych przeglądów i w ten prosty sposób utwierdzić ich w przekonaniu, że jest to czynność tak samo

istotna jak tworzenie nowego kodu. Nie wpływa to jednak na szybkość wykonywanego przeglądu, a jedynie czyni ten proces bardziej komfortowym.

2.5.5. Analiza komentarzy po przeglądzie kodu

Opisane w poprzednich podsekcjach techniki usprawniające przegląd kodu skupione są przede wszystkim wokół osoby go wykonującej – co zrobić, by mogła ona kod ocenić dokładniej i w krótszym czasie? Zdecydowanie mniej badań skupia się na autorze kodu, którego zadaniem jest z kolei zrozumienie uwag otrzymanych po przeglądzie i wprowadzenie zmian do kodu, które są przez nie wymagane.

JavadocMiner jest narzędziem powstałym w ramach jednego z pierwszych odnotowanych w literaturze badań, które skupiły się nie na jakości kodu źródłowego, ale na jakości komentarzy w nim pozostawionych [71]. Stworzony model analizuje nie tylko poprawność językową komentarza, ale także czy rzeczywiście odnosi się do kodu, w którego kontekście występuje.

W literaturze można odnaleźć także publikacje, w których podjęto próby analizy sentymentu tekstu w komentarzach pozostawianych po przeglądach kodu [72]. Autorzy narzędzia *SentiCR* [73] pokazują, że takie klasyfikacje mogą automatycznie wskazać ważniejsze uwagi, na których autor kodu powinien skupić się w pierwszej kolejności.

2.5.6. Analiza kodu przesłanego do przeglądu

Ciekawym badaniem próbującym usprawnić przeglądy kodu źródłowego jest analiza istnienia klasy wyróżniającej się (ang. *salient class*) we wprowadzanych do kodu źródłowego zmianach [74]. Okazuje się, że spory odsetek zmian dotyczy – lub jest spowodowany – wprowadzaniem modyfikacji tylko w jednej klasie w kodzie źródłowym. Pozostałe zmiany wynikają z konieczności dostosowania kodu jej używającej do nowego interfejsu lub wymagań.

Zidentyfikowanie, od której klasy rozpoczyna się dana zmiana może zasugerować rozpoczęcie sprawdzenia kodu właśnie od niej. W ten sposób pozostałe modyfikacje będą miały jasną przyczynę dla osoby przeglądającej kod i w efekcie – szybciej zostaną one zrozumiane a przegląd kodu zakończony.

2.6. Uczenie maszynowe a jakość kodu

Zbliżając się powoli do sedna niniejszej rozprawy, autor dokonał szczegółowego przeglądu literatury traktującej o wsparciu metod uczenia maszynowego w kontekście klasyfikacji jakości kodu źródłowego. Zadaniem tworzonego w ramach rozprawy jakościowego modelu kodu źródłowego będzie automatyczna klasyfikacja jakości kodu źródłowego. Jednym z jego zastosowań może być automatyczne wsparcie programistów piszących kod lub wykonujących przegląd kodu. W tej sekcji przedstawiono aktualne badania nad jakością kodu w kontekście znanych metod uczenia maszynwego.

Większość z tych badań opiera się na założeniu, że kod źródłowy jest formą komunikacji człowieka z komputerem. Reguły pisania kodu źródłowego są podobne do reguł budowy języka naturalnego. W związku z tym, metody uczenia maszynowego wykorzystywane przy problemach językowych (NLP) powinny dać się wykorzystać przy pracy z kodem źródłowym [75].

2.6.1. Jakościowe modele kodu źródłowego

PR-Miner był jednym z pierwszych narzędzi, które starało się odkryć zasady pisania kodu ustalone w zespole programistycznym na podstawie istniejącego kodu źródłowego [76]. Według autorów nie tylko oszczędzało to czas na ciągłym dostosowywaniu konfiguracji *linterów* używanych w projekcie, ale także pozwalało na odkrycie wielu nieudokumentowanych reguł panujących w zespole, które były przekazywane wyłącznie słownie a istniejące narzędzia nie potrafiły ich nawet wziąć pod uwagę. Co więcej, wspomniane narzędzie po odkryciu tych reguł potrafiło także wyszukać przykłady ich naruszeń w kodzie źródłowym.

Z kolei *Bugram* [77] wykorzystuje sieć neuronową nauczoną konstrukcji językowych występujących w projekcie w celu wykrywania sytuacji nieoczekiwanych w nowym kodzie źródłowym. Autorzy przyjmują założenie, że gdy tak zbudowany model wykryje ciąg konstrukcji o relatywnie niskim prawdopodobieństwie wystąpienia, może to oznaczać błąd w kodzie lub przynajmniej – potrzebę jego refaktoryzacji. Po przeanalizowaniu 16 projektów z kodem źródłowym napisanym w Javie, *Bugram* znajduje 59 podejrzanych konstrukcji, z których 42 zostają potwierdzone przez programistów jako miejsca, którym rzeczywiście zwrócono by uwagę podczas przeglądu kodu.

W innej pracy zaprezentowano, jak zbudować narzędzie wykrywające zapachy kodu bez ustalania z góry reguł, które mają być wykrywane [78]. Rozwiązanie to wspiera język JavaScript. Autorzy wykazali, że model starający się nauczyć potencjalnych zapachów kodu tylko analizując dane zidentyfikował większość problematycznych

konstrukcji, które są zdefiniowane w istniejących narzędziach. Ponadto, był w stanie wskazać niektóre przypadki zapachów kodu, które były na tyle zawiłe, że statyczne *lintery* je ignorowały.

W rozprawie doktorskiej „Leveraging Machine Learning to Improve Software Reliability" [79] autor proponuje model *QTEP*, który służy do wykrywania miejsc w kodzie potencjalnie najbardziej narażonych na pojawienie w nich defektów. Może to wynikać z ich stopnia skomplikowania, pokrycia testami czy nawet zbyt małej liczby przeglądów kodu, którym były w trakcie rozwoju projektu poddane. QTEP automatycznie klasyfikuje zmiany o podwyższonym ryzyku wprowadzane do kodu źródłowego i ostrzega o swoich predykcjach osoby wykonujące przegląd kodu. we wnioskach rozprawy jasno stwierdzono, że metody uczenia maszynowego analizujące kod źródłowy mogą pomóc w utrzymaniu wysokiej jakości oprogramowania dzięki przewidywaniu błędów w analizowanych zmianach oraz wykrywaniu regresji.

2.6.2. Generowanie kodu źródłowego

Bardzo ciekawym zastosowaniem uczenia maszynowego w kontekście kodu źródłowego jest możliwość generowania krótkich fragmentów kodu na podstawie słownego opisu pożądanej funkcjonalności. Okazuje się, że odpowiednio wytrenowany model jest w stanie przekształcić wyrażenie *each element parse double separated by a tab and get max* w prostą funkcję, rozdzielającą dany ciąg znaków z użyciem wskazanego separatora, zrzutować każdy element do typu zmiennoprzecinkowego, wybrać z nich maksymalną wartość i zapisać wynik tego działania w zmiennej [80].

Oczywiście jeszcze daleko nam do stworzenia modelu, gdzie zamiast pisać kod będziemy opowiadać o tym, co chcielibyśmy od niego uzyskać. Jeśli natomiast weźmiemy pod uwagę funkcjonalność automatycznego podpowiadania kodu w środowisku programistycznym (IDE), wykorzystanie w tym celu technik uczenia maszynowego jest już dużo bardziej realne. Co więcej, podjęto i opisano już pierwsze próby przewidywania, co programista ma zamiar napisać za pomocą wytrenowanej przykładami kodu źródłowego sieci neuronowej [81].

Jeśli wytrenowany model potrafi generować kod źródłowy, może on być także użyty w celu sprawdzenia, czy dany kod źródłowy jest poprawny syntaktycznie. Wątpliwe jest, że taki model będzie szybszy w wykrywaniu błędnego kodu niż kompilator. Model rozumiejący język programowania jest jednak w stanie nie tylko wykryć błąd syntaktyczny, ale także zasugerować odpowiedni opis błędu programiście [82] lub nawet automatycznie opisywaną poprawkę wprowadzić [83]. Takie rozwiązania już brzmią zdecydowanie sensowniej i wykraczają poza możliwości kompilatorów.

2.6.3. Automatyczne wykonywanie przeglądów kodu

Zostało już wspomniane, że przegląd kodu może – i powinien – być wspierany przez wstępne sprawdzenie kodu za pomocą dostępnych *linterów* (zob. sekcję 2.3). Niemniej jednak programista analizujący przekazaną mu zmianę jest w dalszym ciągu niezbędny, by finalnie potwierdzić odpowiednią jakość kodu źródłowego. Wszak fakt, że narzędzie nie wykryło żadnych zapachów kodu, a wszystkie wyznaczone metryki nie wzbudziły podejrzeń nie oznacza, że dany kod jest czytelny. Czy można wobec tego całkowicie zdjąć z programisty obowiązek wykonywania przeglądów kodu?

Jedną z najnowszych prac analizujących możliwość automatycznego wykonywania przeglądów kodu źródłowego jest publikacja *Intelligent code reviews using deep learning* [84]. Autorzy wykorzystują w niej rekurencyjną sieć neuronową, która jest wytrenowana danymi historycznymi z repozytorium danego projektu. Model ten posiada zarówno wiedzę o kodzie, który był wysyłany do projektu jak i o wynikach przeglądu wykonanego przez programistę. Mając dostęp do wystarczająco dużej liczby historycznych przeglądów kodu sieć uczy się zasad przeglądu panujących w danym projekcie tak, że w efekcie jest w stanie sugerować w którym miejscu kodu osoba sprawdzająca kod dodałaby uwagę. To z kolei sugeruje osobie, która rzeczywiście sprawdza kod na których miejscach się skupić, co przyspiesza wykonywanie przeglądów kodu.

Autorzy tego rozwiązania, nazwanego *DeepCodeReviewer*, skupili się na języku C#. W przeprowadzonej ewaluacji DCR osiągnął akceptowalność proponowanych przez niego komentarzy w trakcie przeglądu kodu na poziomie 54%. Jest to wynik zdecydowanie lepszy od istniejących do tej pory narzędzi dodających automatyczne komentarze, pochodzące od narzędzi poddających kod statycznej analizie [55].

Wykorzystywany w tym badaniu model sieci neuronowej jest w wielu aspektach podobny do modelu zaprojektowanego w niniejszej rozprawie. To, co je różni to sposób pozyskania danych i cel budowy modelu. Publikacja [84] udowadnia, że możliwe było za pomocą uczenia maszynowego dopasowanie odpowiednich komentarzy z historycznych przeglądów kodu dla nowo pojawiających się zmian.

Podobna praca sugeruje, że narzędzie proponujące miejsca, które mogą być bezpiecznie pomięte w trakcie wykonywania przeglądu na podstawie historycznych danych może zmniejszyć ilość kodu wymagającego sprawdzenia nawet o 25% [85].

Pojawienie się takich publikacji na krótko przed zakończeniem prac nad jakościowym modelem kodu źródłowego pokazuje, że obrana metodyka była zgodna z panującymi trendami, a temat jakości kodu jest w obecnym czasie bardzo popularny.

2.7. Podsumowanie

Szczegółowo omawiając pojęcie jakości kodu źródłowego autor rozprawy przedstawia, jak ważna i trudna jest dbałość o ten aspekt oprogramowania. Utrzymywanie czytelnego kodu wpływa nie tylko na komfort pracy programistów, ale także na stabilność, niezawodność i łatwość utrzymania oprogramowania. Ponadto, przedstawione rozwiązania analizujące jakość kodu źródłowego i wspierające utrzymanie jej na odpowiednim poziomie pokazują, że większość programistów jest tego świadoma i stara się minimalizować ryzyko powstania zbyt dużego długu technologicznego, stosując różne narzędzia czy praktyki.

Pomimo istnienia wielu metod, które potrafią w sposób automatyczny wykryć błędy w kodzie lub nawet zasugerować uwagę, która powinna być dodana w trakcie przeglądu – wciąż w dostępnej literaturze brak jest propozycji jakościowego modelu kodu źródłowego oraz zbioru danych walidacyjnych, który pozwoliłby na ewaluację nowych rozwiązań.

Rozdział 3

Koncepcja jakościowego modelu kodu źródłowego – SCQM

Jakościowy modelu kodu źródłowego – Source Code Quality Model (SCQM) – powinien klasyfikować jakość kodu podobnie do programisty. Z tego powodu nie można przygotować listy reguł czy symptomów, które pozwalają go ocenić. W ten sposób działają narzędzia, które opierają się na statycznej analizie kodu opisane w poprzedniej sekcji 2.3. Przy opracowywaniu modelu konieczne jest więc wyjście ponad statyczną analizę i nie skupianie się wyłącznie na wartościach metryk lub detekcji zapachów kodu. W tym rozdziale przedstawiono koncepcję rozwiązania postawionego problemu.

3.1. Wymagania stawiane modelowi

Jakościowy model kodu źródłowego powinien naśladować analizę kodu przeprowadzaną przez programistę wykonującego przegląd. Dlatego wynikiem takiej analizy powinna być również decyzja podobna do tej podejmowanej przez programistę – czy dany kod powinien być zaakceptowany i dołączony do projektu? Jeśli wprowadzana funkcjonalność jest poprawna oraz kod jest odpowiedniej jakości – odpowiedź powinna być twierdząca. Jeśli nie, osoba starająca się go zrozumieć – a także SCQM – powinny zgłosić potrzebę wprowadzenia do niego usprawnień lub przeprowadzenia refaktoryzacji. Opisywane rozwiązanie nie będzie sprawdzać pierwszego wspomnianego czynnika – czyli poprawności implementacji. Weryfikacja tego wymagania nadal pozostanie obowiązkiem programisty, który powinien dostarczyć wystarczająco szczegółowy zestaw testów pokrywających daną funkcjonalność. SCQM natomiast sprawdzi, czy jakość

zaprezentowanego kodu jest odpowiednia i nie wprowadza do projektu długu technologicznego.

Doprecyzowując: wynikiem analizy kodu źródłowego za pomocą jakościowego modelu powinna być klasyfikacja mówiąca o tym na ile dany kod byłby postrzegany przez programistę jako „czysty".

Warto tutaj podkreślić, że model będzie miał zastosowanie w dwóch sytuacjach.

1. Analiza jakości istniejącego kodu źródłowego. W sekcji 3.5 wprowadzono w tym celu opis wariantu bezwzględnego jakościowego modelu kodu źródłowego – Absolute Source Code Quality Model (aSCQM). Klasyfikacja jakości w tym przypadku oznacza czytelność zaprezentowanego kodu źródłowego.

2. Analiza zmiany jakości kodu źródłowego po wdrożeniu proponowanej modyfikacji. W sekcji 3.6 wprowadzono w tym celu opis wariantu względnego jakościowego modelu kodu źródłowego – Relative Source Code Quality Model (rSCQM). Klasyfikacja jakości w tym przypadku wskazuje na poprawę lub pogorszenie jakość kodu po wdrożeniu analizowanej zmiany.

Ważnym wymaganiem niefunkcjonalnym modelu jest jego przenośność i łatwość zastosowania w istniejących narzędziach wykorzystywanych przez programistów: platformach wspierających przegląd kodu, środowiskach programistycznych IDE lub systemach kontroli wersji. Wdrożenie klasyfikacji pochodzących z modelu SCQM do jednej z tych platform wykracza poza zakres niniejszej rozprawy. Niemniej jednak, opis przeprowadzenia analizy porównawczej w celu wykazania prawdziwości tezy rozprawy nie powinien pozostawić wątpliwości jak wykorzystać stworzony model w dowolnym narzędziu.

Podsumowując, zaimplementowany jakościowy model kodu źródłowego powinien spełniać następujące dwa wymagania.

1. Wynikiem analizy kodu źródłowego przez SCQM jest klasyfikacja przeanalizowanego kodu jako kodu wysokiej bądź niskiej jakości.

2. Implementacja modelu jest łatwa do uruchomienia i zaaplikowania do dowolnie wybranego kodu źródłowego we wspieranym języku programowania.

3.2. Zakres działania modelu

Jakościowy model SCQM będzie budowany dla języka Java. Wybór ten był podyktowany głównie popularnością tego języka i dobrą jego znajomością wśród większości programistów (zob. szczegóły w sekcji 1.2). Model będzie wspierać tę wersję Javy, którą będzie wspierać wybrany parser języka użyty do przygotowania danych wejściowych (zob. sekcja 4.3). W momencie pisania rozprawy jest to Java 10.

W trakcie gromadzenia oraz przygotowywania danych na podstawie których przeprowadzono uczenie modelu SCQM (rozdział 4) oraz podczas samego procesu uczenia (rozdział 6) wprowadzono szereg kolejnych ograniczeń w stosunku do kodu źródłowego, który może być sklasyfikowany za pomocą stworzonego, jakościowego modelu. Dzięki nim możliwe było zawężenie problemu do stopnia pozwalającego metodom uczenia maszynowego przyswojenie wiedzy o jakości kodu źródłowego z przygotowanych próbek, przy jednoczesnym zachowaniu ogólności dla przypadków, które mają być analizowane w przyszłości. Powody wprowadzanych ograniczeń wraz z konsekwencjami, które ze sobą niosą, zostały dokładnie opisane w kolejnych rozdziałach. W tym miejscu zostają one tylko pokrótce wymienione.

1. Model SCQM analizuje metody klas – zarówno ich sygnaturę jak i ciało. Bierze także też pod uwagę fakt istnienia komentarza nad nią – jeśli wystąpił (zob. sekcja 4.2.1). Z tego powodu model nie analizuje jakości kodu w blokach statycznych kodu. Nie bierze też pod uwagę sposobu deklaracji i liczby pól klasy. Model nie wykryje także poprawy jakości kodu związanej z refaktoryzacją, której zakres jest szerszy niż pojedyncza metoda i wynika na przykład z poprawienia kolejności występowania metod lub przeniesienia ich do innej klasy.

2. Przygotowanie danych dla modelu wiąże się ze zbudowaniem drzewa rozbioru syntaktycznego kodu źródłowego AST (zob. sekcja 4.3). Takie podejście powoduje, że model SCQM

   (a) nie bierze pod uwagę formatowania kodu źródłowego; sprawdzaniem poprawnego formatowania kodu zgodnego z przyjętymi w zespole regułami powinny zajmować się wyspecjalizowane narzędzia potrafiące przyjąć odpowiednią konfigurację zasad i ją wyegzekwować przed przeglądem kodu (zob. sekcję 2.3);

   (b) nie analizuje nazewnictwa elementów kodu źródłowego (metod, parametrów, zmiennych), gdyż takie informacje są tracone na etapie budowania

drzewa AST; autor rozprawy zgadza się z faktem, że nazewnictwo elementów ma niejednokrotnie duży wpływ na jakość i czytelność kodu źródłowego, ale próby automatycznego proponowania trafniejszych nazw dla elementów w kodzie źródłowym znacząco wykraczają poza zakres prowadzonej rozprawy i wymagałyby stosowania technik przetwarzania i analizy języka naturalnego.

3. Model SCQM został ograniczony do analizy metod nie dłuższych niż 200 tokenów w rozbiorze syntaktycznym AST (zob. sekcja 4.4). W zależności od skomplikowania kodu metody, przekłada się to na około 30-40 linii kodu; nie jest to więc duże ograniczenie, gdyż według [5] oraz autora rozprawy metody o tej długości zazwyczaj nie zawierają już czystego kodu i sam fakt zgromadzenia w obrębie jednej z nich większej liczby linii kodu już sugeruje konieczność jej refaktoryzacji.

4. Ze względu na sposób porównywania zmian refaktoryzacyjnych, model SCQM nie potrafi sklasyfikować zmiany refaktoryzacyjnej przeprowadzonej na przeciążonych metodach (zob. sekcja 4.2.2).

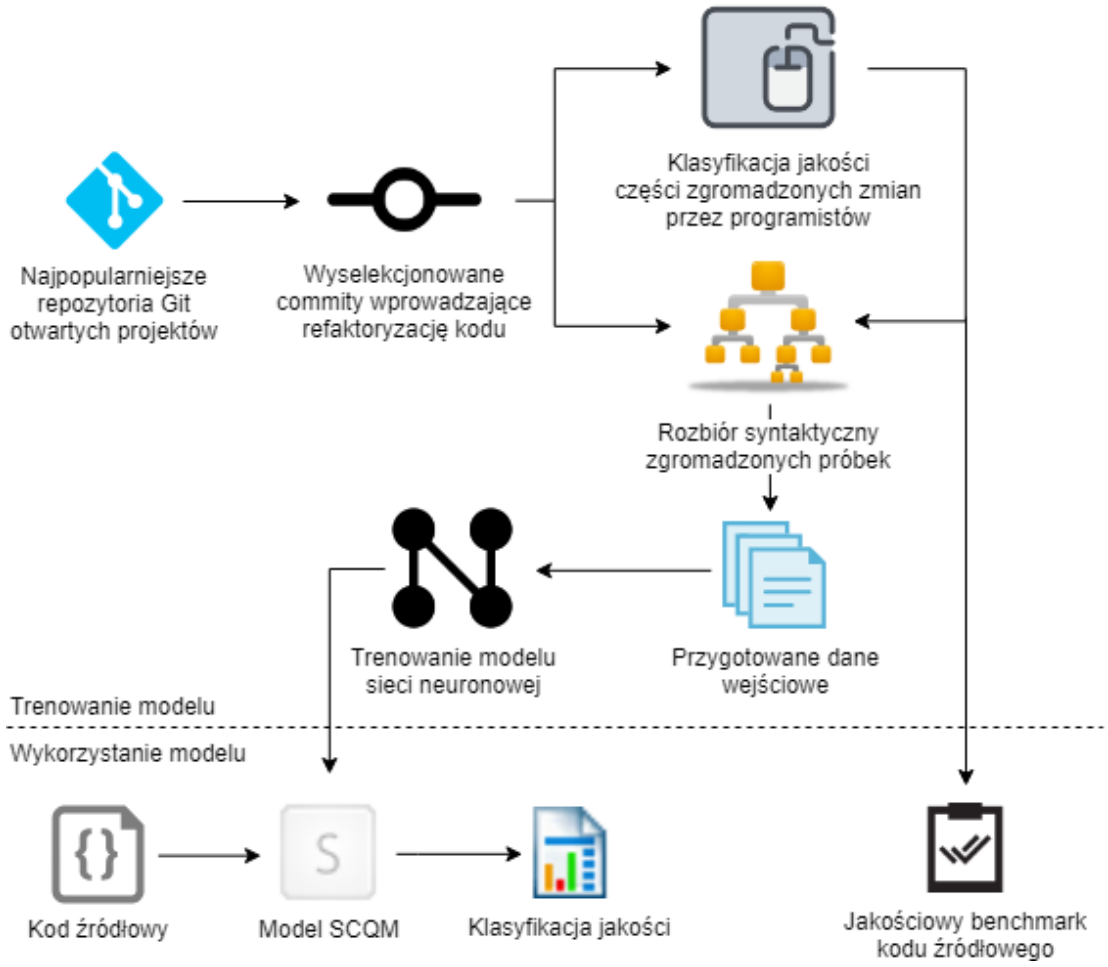
3.3. Przyjęta metodyka badań

Do implementacji jakościowego modelu kodu źródłowego zostaną użyte metody uczenia głębokiego. Zostaną podjęte próby przekazania do modelu wiedzy programistów na temat źródła ich decyzji o sklasyfikowaniu danego kodu jako kodu odpowiedniej bądź nieodpowiedniej jakości. Rysunek 3.1 przedstawia poglądowy diagram obranej metodyki pozyskania odpowiednich danych wejściowych, sposobu ich przygotowania, a także trenowania i ewaluacji jakościowego modelu. W kolejnych sekcjach opisano poszczególne etapy przedstawione na diagramie.

3.3.1. Zgromadzenie odpowiednich danych wejściowych

W każdym problemie, w którym pojawiają się metody uczenia głębokiego, etap przygotowania danych jest jednym z najważniejszych. To od nich zależy skuteczność uczenia a następnie trafność klasyfikacji.

Budowa modelu SCQM musi rozpocząć się od zgromadzenia wystarczająco dużej liczby przykładów kodu ocenianego jako kod wysokiej i niskiej jakości. Tylko wtedy możliwe będzie odkrycie wzorców, które programista podczas przeglądu rozpoznaje jako symptomy „czystego” kodu.



Rys. 3.1: Metoda trenowania i ewaluacji jakościowego modelu

Inspirując się pracami [24, 86], podjęto próbę zgromadzenia odpowiednich próbek kodu o wysokiej i niskiej jakości z otwartych projektów z platformy GitHub. Użyta metodyka, opisana dokładnie w sekcjach 4.1 oraz 4.4, pozwoliła na wyodrębnienie z popularnych projektów na GitHubie zmian refaktoryzacyjnych. Następnie założono, że kod przed refaktoryzacją jest kodem niższej jakości niż kod po refaktoryzacji. W ten sposób zgromadzono przykłady kodu źródłowego o pożądanych cechach.

3.3.2. Przygotowanie danych wejściowych

Analizowany kod źródłowy musi być przekształcony do formatu, który może być podany jako wejście modelu. W tym celu każdą próbkę poddano rozbiorowi syntaktycznemu. Każdemu tokenowi z tego rozbioru przypisano kolejne liczby całkowite, otrzymując w ten sposób ciąg reprezentujący zadany kod źródłowy. Metoda przygotowania wejścia modelu wraz z przykładami została opisana w sekcjach 4.3 oraz 4.5.

3.3.3. Klasyfikacja jakości części próbek przez programistów

W ramach pracy nad rozprawą zostanie przeprowadzone badanie umożliwiające zgromadzenie opinii programistów na temat jakości kodu lub wprowadzanych do niego zmian.

Będzie ono polegać na przedstawieniu programistom przykładów kodu realizującego podobne funkcjonalności, ale różniącego się jakością lub sposobem implementacji. Zadaniem programisty będzie wykonanie przeglądu obydwu przykładów i ocena, które z zaprezentowanych rozwiązań cechuje jego zdaniem wyższa jakość kodu. Koncepcja doświadczenia została dokładnie opisana w kolejnej sekcji 3.4, a sposób jego przeprowadzenia w rozdziale 5.

Na podstawie badania autor rozprawy wprowadza do modelu SCQM rzeczywiste opinie programistów, a co za tym idzie – podnosi końcową trafność i użyteczność opracowywanych klasyfikacji.

3.3.4. Implementacja modelu

Posiadając odpowiednie dane wejściowe należy zaimplementować model, który będzie w stanie przyswoić wiedzę zawartą w zebranych przykładach kodu wysokiej i niskiej jakości.

Istniejące prace poświęcone modelowi struktury kodu źródłowego przyjmują założenie, że kod źródłowy jest formą komunikacji programisty z komputerem (zob. sekcję 2.6). Z tego powodu techniki używane do budowania modeli wiedzy o języku naturalnym powinny sprawdzić się – i sprawdzają się [75] – w zetknięciu z kodem źródłowym.

Tworzony model powinien potrafić podjąć decyzję o jakości zadanego kodu źródłowego na podstawie ciągu tokenów uzyskanych z jego rozbioru syntaktycznego (zob. sekcja 4.3). Sekwencyjna reprezentacja drzewa rozbioru syntaktycznego pozwala traktować problem rozpoznawania kodu źródłowego w podobny sposób, w jaki rozwiązuje się inne problemy modelowania złożonych ciągów – na przykład problemu analizy wydźwięku tekstu w języku naturalnym [87]. Z tego powodu podjęto decyzję o wykorzystaniu rekurencyjnej sieci neuronowej (RNN), posiadającej zdolność rozpoznawania wzorców w danych sekwencyjnych. W analizowanym problemie ciągiem wejściowym będzie właśnie rozbiór syntaktyczny kodu źródłowego, a jego możliwe tokeny stworzą alfabet obsługiwany przez sieć neuronową. Ponadto, rekurencyjne sieci neuronowe pozwalają na podanie sekwencji wejściowych o zmiennej długości, co jest warunkiem koniecznym by przeanalizować rozbiór syntaktyczny dowolnego kodu źródłowego.

Ze względu na fakt, że ocena kodu źródłowego powinna obejmować całą strukturę metody, podjęto również decyzję o wykorzystaniu dwukierunkowej rekurencyjnej sieci neuronowej [88]. Na jakość danego fragmentu kodu składa się bowiem nie tylko to, co wystąpiło po danym węźle drzewa AST, ale również tokeny, które są przed nim. Dwukierunkowa RNN powinna być w stanie odkryć te zależności.

Co więcej, jakość kodu nie wynika tylko z kolejnych operacji następujących po sobie. Mogą występować zależności pomiędzy składowymi kodu, które są od siebie w znacznym stopniu oddalone, a mimo wszystko powodują, że programista klasyfikuje dany kod jako odpowiedniej bądź nieodpowiedniej jakości. Z tego powodu do zbudowania sieci neuronowej użyto komórek Long short-term memory (LSTM) [89], które wykazują zdolność powiązania ze sobą odległych elementów ciągu wejściowego.

3.3.5. Uczenie modelu

Po zebraniu danych i zaimplementowaniu sieci neuronowej należy zbudować model wiedzy, który będzie potrafił sklasyfikować przykłady ze zbioru testowego oraz nowe przykłady metod, które nie znalazły się w zgromadzonym zbiorze danych.

Jak pokaże lektura rozdziału 6, do uczenia modelu wykorzystano zarówno próbki kodu zebrane automatycznie i te sklasyfikowane przez programistów w przeprowadzonym badaniu. Z ich części wydzielono zbiór testowy, który pozwolił na monitorowanie osiąganej przez model trafności w zależności od różnych konfiguracji parametrów sieci neuronowej i sposobu usuwania szumu z danych treningowych.

3.3.6. Ewaluacja modelu

Po skutecznym przekazaniu wiedzy do modelu SCQM, ostatnim krokiem prowadzącym do weryfikacji tezy rozprawy będzie porównanie klasyfikacji dowolnego kodu źródłowego przez stworzony jakościowy model oraz dowolne inne narzędzie potrafiące dokonać statycznej analizy kodu.

Oczekuje się, że klasyfikator oparty o SCQM będzie trafniej rozpoznawać kod niskiej jakości. Model SCQM został porównany z jednym z najpopularniejszych narzędzi dostarczającym możliwość statycznej analizy kodu źródłowego napisanego w Javie: Checkstyle. Ponadto, porównano klasyfikacje uzyskiwane z modelu do wartości jakościowych metryk: złożoności cyklomatycznej, NPath oraz NCSS. Wyniki ewaluacji przedstawiono w rozdziale 7.

3.4. Jakościowy *benchmark* kodu źródłowego

W literaturze nie odnaleziono zbiorów walidacyjnych (z ang. *benchmark*[1]) dotyczących problemu jakości kodu źródłowego, które pozwoliłyby rozstrzygnąć o prawidłowości klasyfikacji pochodzących z jakościowego modelu kodu źródłowego.

W celu zbudowania zbioru walidacyjnego oraz poprawienia jakości danych wejściowych zaplanowano przeprowadzenie badania pozwalającego na pozyskanie od programistów opinii na temat jakości kodu źródłowego.

Uczestnikom badania, będącym studentami lub absolwentami studiów informatyki, zostały zaprezentowane pary podobnych implementacji tej samej funkcjonalności. Ich zadaniem było wskazanie, którą z nich cechuje wyższa jakość wykonania.

Podczas projektowania badania podjęto szereg decyzji mających na celu zminimalizowanie prawdopodobieństwa przypadkowych klasyfikacji, takich jak:

- pytanie wstępne o liczbę lat doświadczenia w programowaniu,
- wymaganie zebrania przynajmniej trzech takich samych głosów od różnych respondentów na temat danego przykładu, by uznać go jako sklasyfikowany,
- mechanizm sprawdzania czujności respondenta polegający na prezentowaniu mu w losowych momentach już sklasyfikowanego wcześniej przez innych przykładu i porównywaniu oddanego głosu z głosami już znanymi.

Szczegółowy opis przeprowadzenia badania oraz stworzonej w tym celu platformy został zamieszczony w dalszej części rozprawy w rozdziale 5.

Sklasyfikowane w ten sposób próbki kodu źródłowego zostały użyte do zbudowania zbioru walidacyjnego, za pomocą którego została przeprowadzona ewaluacja jakościowego modelu kodu źródłowego (zob. rozdział 7). Ponadto, część ocenionych przez programistów próbek kodu została wykorzystana w trakcie uczenia modelu (zob. sekcję 6.5). Cały zbiór zgromadzonych w wyniku doświadczenia danych został opublikowany jako jakościowy *benchmark* kodu źródłowego (zob. sekcję 5.6).

Przeprowadzenie tego badania wraz z analizą i udostępnieniem jego wyników jest ważnym osiągnięciem niniejszej rozprawy doktorskiej. Zbudowany zbiór przykładów kodu o różnej, sklasyfikowanej przez programistów jakości jest pierwszym takim osiągnięciem odnotowanym w literaturze. Może on zostać wykorzystany do ewaluacji dowolnego innego rozwiązania mającego na celu analizę lub klasyfikację jakości kodu źródłowego.

[1]od tego momentu wystąpienia słowa *benchmark* w niniejszej rozprawie będą odnosić się do zestawu zgromadzonych danych walidacyjnych

3.5. Model bezwzględny aSCQM

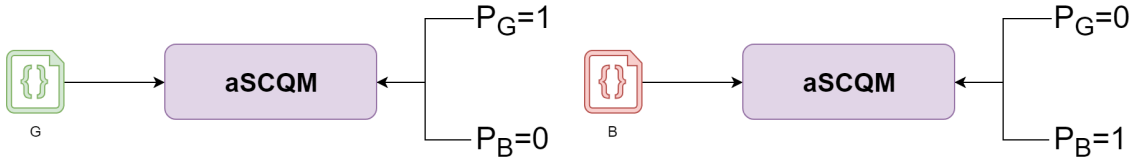
Zadaniem modelu bezwzględnego jest analiza istniejącego kodu źródłowego i klasyfikacja jego jakości. Model aSCQM będzie użytecznym wsparciem programisty piszącego kod. Przed próbą zrozumienia wybranego fragmentu kodu będzie możliwa wstępna klasyfikacja jego jakości. Negatywny wynik będzie sugerować konieczność przeprowadzenia refaktoryzacji.

Model bezwzględny na wejściu otrzymuje jedną próbkę kodu źródłowego. Zadaniem modelu jest sklasyfikowanie jej jakości, czyli wyznaczenie prawdopodobieństwa określającego szanse na to, że zadany kod jest kodem „czystym", czyli wysokiej jakości. Rysunek 3.2 przedstawia schemat działania modelu aSCQM.

Rys. 3.2: Schemat działania modelu aSCQM.

Na wejściu zadany jest kod źródłowy S (z ang. *sample*). Wynikiem klasyfikacji są dwie liczby P_G oraz P_B przyjmujące wartości z zakresu $[0, 1]$ oraz spełniające warunek $P_G + P_B = 1$. Oznaczają one prawdopodobieństwa, że zadany kod jest odpowiednio wysokiej, bądź niskiej jakości. Oznaczenia P_G oraz P_B pochodzą z języka angielskiego – G od ang. *good* – dobry, oraz B – *bad* – zły.

W trakcie uczenia modelu aSCQM na wejściu zostaną podane kolejno przykłady kodu wysokiej jakości (G) oraz kodu niskiej jakości (B). Do uczenia ze wzmocnieniem zostaną użyte odpowiednie wektory zawierające stuprocentowe prawdopodobieństwo odpowiedniego parametru – P_G dla próbek pozytywnych oraz P_B dla próbek negatywnych. Rysunek 3.3 zawiera schematy przedstawiające sposób uczenia modelu aSCQM.



(a) przypadek z kodem wysokiej jakości na wejściu

(b) przypadek z kodem niskiej jakości na wejściu

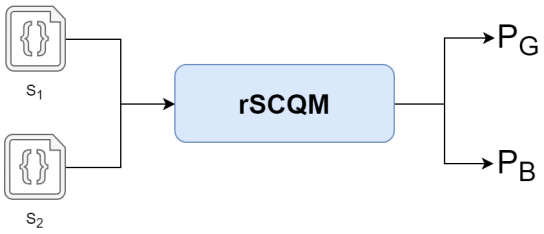
Rys. 3.3: Schemat uczenia modelu aSCQM

Model bezwzględny umożliwi klasyfikację jakości kodu źródłowego w istniejących systemach. Stara się odpowiedzieć na pytanie: **Czy przedstawiony kod jest kodem wysokiej jakości?**

3.6. Model względny rSCQM

Model względny ma za zadanie ocenić zmianę w kodzie źródłowym. Zdecydowana większość przeglądów kodu skupia się właśnie na ocenie wprowadzanej zmiany, a nie na klasyfikacji całego kodu źródłowego. Model rSCQM powinien więc być bardziej przydatny przy potencjalnej integracji jakościowego modelu z platformami wspierającymi przegląd kodu źródłowego.

W odróżnieniu od modelu bezwzględnego, na wejściu powinien on otrzymać dwie próbki kodu. Pierwsza z nich to postać kodu źródłowego przed zmianą, a druga – po zmianie. Zadaniem modelu jest zwrócenie klasyfikacji oznaczającej prawdopodobieństwo, że dana zmiana podnosi jakość kodu źródłowego. Rysunek 3.4 przedstawia schemat działania modelu rSCQM.

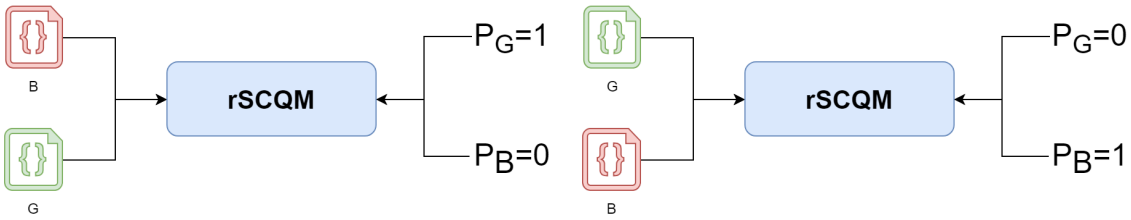


Rys. 3.4: Schemat działania modelu rSCQM.

Na wejściu zadana jest zmiana kodu źródłowego w postaci dwóch próbek S_1 i S_2 (z ang. *sample*), będące odpowiednio kodem źródłowym przed i po wprowadzonej zmianie. Podobnie jak w modelu bezwzględnym, wynikiem klasyfikacji są dwie liczby P_G oraz P_B spełniające te same założenia. W przypadku modelu względnego oznaczają one prawdopodobieństwa, że zadana zmiana odpowiednio podnosi lub obniża jakość kodu.

W trakcie uczenia modelu rSCQM na wejściu zostaną podane przykłady zmian kodu źródłowego. Z każdej zmiany zostanie wybrany kod wyższej jakości (G) oraz niższej jakości (B). Do uczenia ze wzmocnieniem zostaną użyte odpowiednie wektory zawierające stuprocentowe prawdopodobieństwo odpowiedniego parametru – P_G dla zmian podnoszących jakość kodu (charakteryzująca się niższą jakością próbka B podana jako pierwsza – zmiana refaktoryzacyjna) oraz P_B dla zmian, które ją pogarszają (próbka B podana jako druga – zmiana wycofująca refaktoryzację, obniżająca jakość kodu). Rysunek 3.5 zawiera schematy przedstawiające sposób uczenia modelu rSCQM.

Model względny naśladuje programistę wykonującego przegląd kodu zmiany wprowadzanej do projektu. Stara się odpowiedzieć na pytanie: **Czy wprowadzana zmiana podnosi jakość kodu źródłowego?**



(a) przypadek ze zmianą poprawiająca jakość kodu

(b) przypadek ze zmianą pogarszającą jakość kodu

Rys. 3.5: Schemat uczenia modelu rSCQM

3.7. Możliwe zastosowania modelu

Model aSCQM będzie mógł być wykorzystywany do oceny jakości istniejącego kodu. To pozwoli np. na automatyczne porównanie wyników pracy różnych zespołów programistycznych. Dzięki takiemu rozwiązaniu można będzie nie tylko skupić się na efektach pracy programistów w postaci liczby napisanych linii kodu czy zamkniętych zadań, ale także na jakości dostarczanych przez nich rozwiązań. Takie narzędzie byłoby z pewnością docenione na szczeblu menedżerskim oraz wśród osób odpowiedzialnych za zatrudnienie czy ewaluację pracy programistów.

Model rSCQM może mieć bardzo pozytywny wpływ na popularyzację praktyki przeglądów kodu źródłowego. Jak opisano w sekcji 2.5, przeglądy kodu są niechętnie wykonywane przez programistów ze względu na ich czasochłonność i niski – w stosunku do tworzenia kodu – poziom kreatywności wykonywanej pracy. Sugestie o jakości proponowanych zmian od modelu względnego powinny przyspieszyć przegląd kodu dzięki sugerowaniu miejsc, na które należy zwrócić uwagę oraz tych, które można pominąć, gdyż automatyczna klasyfikacja nie wykazała w nich żadnych podejrzeń.

Wykorzystanie analiz aSCQM w środowiskach programistycznych IDE pozwoli na wskazanie miejsc w istniejącym systemie, które wymagają refaktoryzacji. Takie fragmenty kodu mogłyby być specjalnie oznaczane jak ostrzeżenia o istnieniu długu technologicznego. Stanowiłoby to bardzo użyteczny dodatek do już wyświetlanych ostrzeżeń pochodzących z kompilatorów lub *linterów*.

Model rSCQM sprawdzi się w integracjach z platformami wspierającymi przeglądy kodu. Wdrożenie powinno polegać na dostarczeniu mechanizmu, który po każdej nadesłanej zmianie, przed powiadomieniem osoby odpowiedzialnej za wykonanie przeglądu, podda jej zawartość automatycznej klasyfikacji. rSCQM identyfikuje te zmiany, które pogarszają jakość kodu i zwraca informacje o nich do integrowanej platformy. Ta z kolei zaznaczy w interfejsie użytkownika wskazane miejsca z informacją, że jakościowy model kodu źródłowego podejrzewa w nich istnienie problemów z jakością.

Integracja SCQM leży poza zakresem rozprawy, jednakże powyższy opis oraz dalsze szczegóły implementacji powinny jasno zarysować sposób jej ewentualnego przeprowadzenia. Ponadto, w dodatku G zaprezentowano i omówiono przykład integracji oparty o skrypt dla systemu kontroli wersji Git. Jest to prosta integracja ostrzegająca programistę o wykonywaniu zmiany, którą SCQM sklasyfikował jako nieodpowiedniej jakości.

3.8. Podsumowanie

Projekt jakościowego modelu kodu źródłowego SCQM oraz analiza jego dwóch wariantów – względnego i bezwzględnego – jasno zakreśla zakres i cel prowadzonej rozprawy. Opisana metodyka powinna doprowadzić do uzyskania satysfakcjonujących wyników i otrzymania klasyfikatora jakości kodu źródłowego, który patrzy na kod „oczami programisty". Kolejne rozdziały rozprawy przedstawiają szczegóły implementacji SCQM oraz uzyskane rezultaty.

Rozdział 4

Przygotowanie danych

W niniejszym rozdziale opisano sposób zgromadzenia danych, za pomocą których model SCQM powinien posiąść pożądaną wiedzę oraz sposób ich przygotowania. Opisano także szczegółowo budowę sieci neuronowej odpowiedzialnej za klasyfikację oraz badanie, którego celem było pozyskanie jak największej i najdokładniejszej wiedzy ekspertowej na temat jakości kodu źródłowego.

4.1. Źródło danych

Naturalnym źródłem, z którego można pozyskać dane przy analizowaniu kodu źródłowego są otwarte projekty (ang. *open source*). Ich licencje pozwalają na bezpłatne zapoznanie się z kodem źródłowym, a zazwyczaj także na jego wykorzystanie lub nawet modyfikację. Nad kodem źródłowym otwartych projektów często pracuje też spora liczba programistów – szczególnie gdy bierzemy pod uwagę projekty popularne, używane przez znaczną liczbę użytkowników.

Te cechy powinny czynić z popularnych otwartych projektów doskonałe przykłady kodu, który jest kodem czystym i kodem wysokiej jakości. Skoro jego większość była przeglądnięta przez kilku programistów, jest duża szansa że nie tylko jest on działający, ale także zrozumiały i czytelny. Niestety, sam wybór najpopularniejszych otwartych projektów nie pozwoliłby na zbudowanie jakościowego modelu kodu. Nie można założyć, że sama otwartość i popularność projektu niesie ze sobą odpowiednią jakość w całym kodzie źródłowym.

Kod źródłowy przechowywany jest w repozytorium kodu źródłowego, zwanego też systemem kontroli wersji (ang. *Version Control System*, VCS). W stosunku do zwykłego przechowywania plików na dysku, najważniejszą zaletą korzystania z repozytoriów kodu jest fakt, że oprócz przechowywania aktualnej wersji kodu, repozytorium

przechowuje całą historię tworzenia danego projektu. Historia ta podzielona jest na kolejne zmiany (ang. *commit*[1]), które jedna po drugiej tworzą pełną historię powstawania projektu od pierwszego pliku aż po stan na dzień dzisiejszy.

Jednym z najpopularniejszych systemów kontroli wersji w momencie pisania rozprawy jest Git [90]. Jest to system darmowy i bardzo funkcjonalny, pozwalający na łatwe zarządzanie kodem projektu. Z tego powodu jest on bardzo często wybierany jako system kontroli wersji dla otwartych projektów.

Przykładem popularnej platformy, która udostępnia darmowe repozytoria Git do przechowywania repozytoriów otwartych projektów jest GitHub[2]. W celu pozyskania danych, które pozwolą na zbudowanie SCQM w pracy nad rozprawą wykorzystano popularne repozytoria kodu źródłowego Git przechowywane na platformie GitHub.

4.1.1. Wybór repozytoriów kodu

GitHub, poza udostępnieniem możliwości darmowego przechowywania repozytoriów kodu otwartych projektów, oferuje także funkcjonalności, które umożliwiają zbudowanie społeczności wokół nich. Jednym z takich mechanizmów jest mechanizm oznaczania przez użytkowników wybranych projektów gwiazdkami. W dokumentacji GitHuba [91][3] możemy przeczytać, że celem tej funkcjonalności jest możliwość oznaczenia interesujących projektów tak, by można je było potem szybciej odnaleźć. Przekazanie gwiazdki przez użytkownika oznacza także docenienie pracy włożonej w projekt przez jego autorów, a sama liczba gwiazdek projektu rozumiana jest jako jego popularność czy użyteczność.

Liczba gwiazdek projektu była głównym kryterium wyboru repozytoriów, z których pobrano próbki kodu różnej jakości pozwalające na zbudowanie SCQM.

Poza popularnością projektu, ważne było też skupienie się tylko na projektach pisanych w analizowanym w rozprawie języku programowania – Java. Ze względu na heterogeniczność projektów (bardzo ciężko wskazać choćby jeden przykład projektu pisanego wyłącznie w jednym jezyku programowania), mogłoby się wydawać że odnalezienie odpowiednich projektów będzie trudnym zadaniem. Na szczęście, Github na bieżąco analizuje typy plików w repozytorium kodu każdego projektu i automatycznie klasyfikuje projekty jako pisane w danym języku programowania, jeśli większość z kodu w repozytorium jest napisana właśnie w nim.

---

[1] od tego momentu wystąpienia słowa *commit* w niniejszej rozprawie będą odnosić się do zmiany kodu źródłowego wprowadzanej do repozytorium

[2] `https://github.com`

[3] `https://help.github.com/articles/about-stars/`

Mając dostęp do wszystkich opisanych wyżej narzędzi oraz do API, za pomocą którego można przeszukiwać istniejące projekty na GitHubie, wybór potencjalnie przydatnych w rozprawie repozytoriów sprowadził się do wykonania następującego zapytania przedstawionego na Listingu 4.1.

```
https://api.github.com/search/repositories?q=language:java&sort=stars&
    ↪ order=desc
```

Listing 4.1: Zapytanie do GitHub Search API wybierające repozytoria kodu źródłowego do dalszej analizy

Zapytanie `language:java` wybiera tylko projekty, których głównym językiem programowania jest Java. Kolejne parametry zapytania ustawiają malejące sortowanie po liczbie przyznanych danemu repozytorium gwiazdek. Do dalszej analizy wybrano 350 pierwszych projektów zwróconych w wyniku powyższego zapytania. Dziesięć pierwszych z nich przedstawiono w Tabeli 4.1 (obecny wynik zapytania może różnić się od tego, który uzyskano w trakcie pracy nad rozprawą). Pełną listę repozytoriów kodu, które wybrano na tym etapie, zamieszczono w dodatku A.

Tabela 4.1: Najpopularniejsze repozytoria z GitHuba wybrane do dalszej analizy (pierwsze 10)

L.p.	**Nazwa projektu**	**Liczba gwiazdek**
1.	ReactiveX/RxJava	32732
2.	elastic/elasticsearch	31391
3.	iluwatar/java-design-patterns	30268
4.	square/retrofit	29079
5.	square/okhttp	28072
6.	google/guava	26045
7.	PhilJay/MPAndroidChart	23589
8.	JakeWharton/butterknife	21761
9.	JetBrains/kotlin	20664
10.	JakeWharton/butterknife	20506

Wyniki zwracane przez API są podzielone na strony, dlatego w celu pobrania nazw wszystkich 350 repozytoriów kodu napisano prosty skrypt, który odpowiednio operując parametrami `page` i `per_page` wykonał to zadanie automatycznie. Kod źródłowy tego skryptu znajduje się w pliku `0_github-query.php` w repozytorium [92].

4.1.2. Identyfikacja zmian wprowadzających refaktoryzację

Każdy *commit* oprócz zmian wprowadzanych w kodzie źródłowym zawiera także informację o autorze zmiany, datę jej wprowadzenia do kodu oraz krótki komentarz podany przez programistę w chwili tworzenia zmiany, nazywany z ang. *commit message*. Powinien on zawierać informacje, które pozwolą na zrozumienie, co dana zmiana wprowadza do kodu źródłowego w trakcie przeglądania historii projektu. Dzięki temu nie trzeba analizować zmian w kodzie źródłowym by poznać cel wprowadzonego do repozytorium przyrostu.

Prace [24, 86] pokazują, że komentarze do zmian w repozytorium mogą być wykorzystane do wybrania potencjalnie cennych dla danego badania zmian w kodzie. Autorzy [24] analizują przyczyny błędów w otwartym oprogramowaniu, przez co wyszukują *commity*, które takie defekty naprawiają. W tym celu przeglądnięto repozytoria kodu pod kątem *commitów* zawierającyech w swoim *commit message* odpowiednie frazy, np. *exception handling*, *memory leak* czy *optimization problem*. Autorzy drugiego wspomnianego badania [86] z kolei szukają przyczyn wycofywania danych zmian z projektu, przez co wyszukują w historii projektu *commitów* z komentarzem zawierającym automatycznie dodawany w takiej sytuacji przez narzędzie Git komentarz *This reverts commit ....*

Powyższe badania pokazują, że analizowanie informacji podanych w *commit message* pozwala na automatyczne odnalezienie zmian w historii projektu, które posiadają poszukiwane cechy. W przypadku rozprawy, poszukiwanymi zmianami są *commity*, które poprawiają jakość kodu. Jak wspomniano na początku tej sekcji, sama otwartość i popularność projektu nie gwarantuje od razu wysokiej jakości całego kodu źródłowego. Wyszukanie zmian, które były refaktoryzacją kodu projektu powinno pozwolić na zidentyfikowanie tych fragmentów kodu, które były podczas przeglądu lub przy próbie jego modyfikacji zidentyfikowane jako wymagające naprawy. Dzięki temu, można z dużą dozą prawdopodobieństwa wskazać, że kod przed zmianą refaktoryzacyjną w opinii programisty jest kodem gorszym od tego, który został wprowadzony do repozytorium po tej zmianie.

Dokładnie taka wiedza jest konieczna do zbudowania jakościowego modelu kodu źródłowego. Głównym założeniem SCQM jest wyjście poza sztywne ramy statycznej analizy czy identyfikacji możliwych zapachów kodu. SCQM powinien analizować kod i sugerować konieczność jego refaktoryzacji w sposób zbliżony do programisty. Pomysł z wykryciem zmian refaktoryzacyjnych w projekcie jest więc pierwszym przybliżeniem do materializacji modelu, który pozwoli na weryfikację tezy rozprawy.

Analogicznie do wspomnianych już prac [24, 86], w celu odnalezienia zmian dokonujących refaktoryzację kodu w wybranych projektach, wybrano słowa jednoznacznie wskazujące na poprawę jakości kodu we wprowadzanej zmianie. Zdecydowana większość *commit message* jest pisana w języku angielskim, dlatego też słowa były poszukiwane w tym właśnie języku. Tabela 4.2 zawiera poszukiwane słowa oraz frazy, w które planowano „trafić" za ich pomocą. Oczywiście, lista docelowych fraz nie zamyka komentarzy, które pasowały do zadanych słów kluczowych, ale pozwala odgadnąć intencję autora w ich wyborze.

Tabela 4.2: Słowa kluczowe użyte do identyfikacji *commitów* zawierających refaktoryzację

Słowo	**Docelowe frazy**
refactor	refactor code, refactor method, some refactorings
readability	work on readability, improve readability

W początkowej fazie pracy przy identyfikacji zmian refaktoryzacyjnych brano pod uwagę także słowa `improve` (frazy docelowe: improve the code structure, improve the quality, small improvements) oraz `reorganize` (frazy docelowe: reorganize methods, reorganize classes). Pozyskane dane jednak pokazały, że te słowa często pojawiały się w *commit messages* zmian, które nie wykonywały refaktoryzacji, ale np. poprawiały interfejs użytkownika.

Ograniczenie się do tematu zmiany

Szybko okazało się, że kryterium wyszukiwania zmian po wystąpieniu jednego z założonych słów w dowolnym miejscu *commit message* jest zbyt luźne. Często bowiem refaktoryzacje były wykonywane przy okazji innej zmiany – np. dodania nowej funkcjonalności do aplikacji bądź naprawienia innego błędu. Było to jasne po analizie kilkunastu losowo wybranych komentarzy zaklasyfikowanych przez skrypt jako refaktoryzacje.

Przykład takiego *commit message* zaprezentowano na Listingu 4.2. Taki *commit*, pomimo niewątpliwej wartości dla projektu, nie mógł być użyty przy budowie danych wejściowych dla SCQM, gdyż oprócz zmian refaktoryzacyjnych zawierał też inne – niekoniecznie związane z jakością kodu. Nie sposób też z takiej zmiany automatycznie oddzielić kod odpowiedzialny za wykonanie przez programistę danego zadania od kodu, który zmienia się w związku z refaktoryzacją. To skłoniło autora rozprawy

do pierwszego zawężenia algorytmu gromadzenia danych wejściowych. Ulepszenie polegało na wybieraniu tylko takich zmian, które poszukiwane słowa kluczowe zawierały w pierwszej linii *commit message*, z ang. nazywaną także jako *commit topic*. Programiści powinni umieszczać w niej główny powód wprowadzenia danej zmiany do projektu. Skoro głównym powodem wprowadzenia zmiany była refaktoryzacja kodu źródłowego – to dokładnie taka zmiana jest potrzebna przy budowaniu wejścia dla jakościowego modelu kodu.

Listing 4.3 zawiera dwa przykłady *commit messages*, które zostały zaklasyfikowane po optymalizacji skryptu jako zmiany wprowadzające refaktoryzację.

```
commit 8ec4c58eb37880a812bb2e23e8b3ec2a5e3b7f33
Author: asdf2014 <1571805553@qq.com>
Date: Sun Jun 25 15:07:06 2017 -0700
    Using try clause to close resource

    MINOR:
    * Using try clause to close resource;
    * Others code refactoring for PERSISTENCE module.
```

Listing 4.2: Przykład komentarza do zmiany, który został niepoprawnie sklasyfikowany jako refaktoryzacja kodu

```
commit 01ab972572c1cc00111e28e512ec746117bded09
Author: wenshao <szujobs@hotmail.com>
Date: Sat May 13 11:40:55 2017 +0800
    improved readablity of oracle pl/sql parser.

commit 5aa1c753c8ac4be59a70a35d16cf53614d17badc
Author: jfarcand <jfarcand@apache.org>
Date: Fri Oct 22 12:53:41 2010 -0400
    Refactor to improve the integration of Atmosphere within other
        ↪ framework
```

Listing 4.3: Przykład komentarzy do zmian, które zostały sklasyfikowane jako refaktoryzacje kodu (po optymalizacji)

4.1.3. Pozyskanie danych z wybranych repozytoriów

Pobranie repozytoriów

Repozytoria wybrane zgodnie z opisem w sekcji 4.1.1 zostały kolejno pobrane za pomocą operacji klonowania z platformy GitHub. Adresy repozytoriów zostały zbudowane na podstawie ich nazw. Aby wykonać to zadanie dla wszystkich wybranych repozytoriów, został napisany prosty skrypt `1_fetch_repos.sh`, którego źródła można podglądnąć w repozytorium [92].

Identyfikacja *commitów* wprowadzających refaktoryzację

Kolejnym krokiem była identyfikacja *commitów* refaktoryzacyjnych w danym repozytorium zgodnie z procedurą opisaną w sekcji 4.1.2. Zadanie to wykonano za pomocą skryptu `2_extract_refactor_commits.php` z repozytorium [92]. W celu odnalezienia zmian zawierających w *commit message* zadane słowa kluczowe wykorzystano komendę `git log --grep`. Została ona wykonana dla każdego słowa kluczowego sugerującego refaktoryzację (zob. Tabela 4.2).

Parametr `--grep` wyszukuje zadane słowo w całym *commit message*, dlatego zawężenie wyszukiwania do samego *commit topic* zostało zrealizowane na późniejszym etapie, przy ekstrakcji metod (zob. sekcję 4.3)[4].

Każdy *commit* w repozytorium Git jest identyfikowany przez unikalny ciąg znaków nazywany jako *commit hash*. Parametr `-pretty=format:%H` z Listingu 4.4 wskazuje komendzie, by z historii repozytorium zwróciła wyłącznie identyfikatory *commitów*. Przykładowy rezultat odnalezienia identyfikatorów zmian wprowadzających refaktoryzacje w repozytorium został zaprezentowany na Listingu 4.4.

```
$ git log --pretty=format:%H --grep=refactor
fc19c6d9d58543db652b956b96188033461db086
b42ba989469c79512b4e1955fa6db91f494fa207
d91b39ca6eeabeced11421bc0f5bd96b4bf09aa0
cd333f70365952602a9ec35ac97cff0c20e3a132
745e29b7832c3cc37e7b6d395aded79a908c3bbc
184578c24c4d4acea14b32c6031b67e5dd30ae50
```

Listing 4.4: Przykłaodowy rezultat wyszukania w repozytorium zmian z zadanym słowem kluczowym w *commit message*

[4]tym bardziej, że problem zbyt optymistycznego wyszukiwania słów kluczowych w *commit message* zidentyfikowano dopiero po pobraniu danych ze wszystkich repozytoriów

Rozpoznanie plików zmienionych przy refaktoryzacji

Każda klasa w Javie[5] musi być zdefiniowana w osobnym pliku, którego nazwa odpowiada nazwie klasy (o klasach napisano szerzej w kolejnej sekcji 4.2). Dzięki temu można łatwo zidentyfikować, które klasy analizowanej zmiany refaktoryzacyjnej zostały po niej pozostawione, oraz – które zostały dodane i usunięte. Wystarczy przeanalizować listę plików posiadających rozszerzenie `.java` przed i po danej zmianie.

Dokładnie taką operację wykonuje skrypt `2_extract_refactor_commits.php` z repozytorium [92]. Dla każdej zmiany wprowadzającej refaktoryzację zidentyfikowaną zgodnie z opisem w sekcji 4.1.2 wykonywana jest komenda `git diff-tree`, pokazująca listę zmienionych plików w danej zmianie.

Komenda ta została wykonana dla każdego *commit hasha* zidentyfikowanej zmiany refaktoryzacyjnej. Przykładowa lista plików dla jednego z takich *commitów* została zaprezentowana na Listingu 4.5.

```
$ git diff-tree --no-commit-id --name-only -r 745
    ↪ e29b7832c3cc37e7b6d395aded79a908c3bbc
retrofit/src/main/java/retrofit/RetrofitError.java
retrofit/src/test/java/retrofit/RestAdapterTest.java
```

Listing 4.5: Przykładowy wynik komendy prezentującej listę zmienionych plików w zadanym *commicie*

Ze zwróconej listy wspomniany wcześniej skrypt odfiltrowuje pliki które posiadają inne rozszerzenie niż `.java`. Może zdarzyć się tak, że *commit* który został zidentyfikowany jako refaktoryzacja nie zmienia żadnych plików z takim rozszerzeniem pomimo odpowiedniego wyboru projektów zgodnie z opisem w sekcji 4.1.1. W takim wypadku zmiana jest całkowicie odrzucana i nie jest brana pod uwagę w dalszej analizie.

Uzyskanie zawartości plików przed i po refaktoryzacji

Dla każdego odnalezionego pliku o odpowiednim rozszerzeniu w zmianie refaktoryzacyjnej wykonywana jest dwukrotnie komenda `git show`. Ogólna postać wykonywanych komend została zaprezentowane na Listingu 4.6.

```
git show $COMMIT_HASH^:$FILEPATH
git show $COMMIT_HASH:$FILEPATH
```

Listing 4.6: Komendy pozwalające na otrzymanie zawartości pliku przed i po wykonaniu danej zmiany

[5]każda publiczna klasa, która nie jest wewnętrzna lub anonimowa

Pierwsza z komend zwraca zawartość pliku przed wprowadzeniem analizowanej zmiany, a druga – po niej. Zmienna `$COMMIT_HASH` powinna zawierać *commit hash* przetwarzanej obecnie zmiany. `$FILEPATH` to pełna ścieżka ze zwróconej wcześniej listy plików zmienionych w danym *commicie*. Znak ^ w pierwszym wariancie komendy pozwala na cofnięcie się o jeden *commit* w celu pozyskania zawartości pliku przed zmianą (dzięki temu nie trzeba wyszukiwać *commit hasha* modyfikacji poprzedzającej zmianę refaktoryzacyjną). Wynikiem działania powyższych komend jest cała treść pliku pochodząca ze wskazanego miejsca w historii projektu.

Jeśli dany plik nie istniał przed lub przestał istnieć po analizowanej refaktoryzacji, odpowiednia komenda z Listingu 4.6 zwracała błąd. Skrypt w takiej sytuacji zakładał pustą zawartoć danego pliku, co w dalszym przetwarzaniu umożliwiało identyfikację klas dodanych lub usuniętych w ramach danej zmiany.

Przechowanie zgromadzonych danych

Dane pozyskane w ten sposób były przechowywane w systemie plików według struktury zaprezentowanej na Listingu 4.7. Tak przygotowane dane znajdują się w repozytorium [92] w katalogu `results-java`.

```
repozytorium_1/
  COMMIT_HASH_1/
    after/
      Class1.java
      Class2.java
    before/
      Class1.java
      Class2.java
    README.txt
  COMMIT_HASH_2/
    ...
  README.txt
repozytorium_2/
  ...
```

Listing 4.7: Struktura danych przechowujących kod poddany refaktoryazji po przetworzeniu zidentyfikowanych *commitów* wprowadzających refaktoryzację

Każde przetworzone repozytorium posiada swój katalog w pozyskanych danych (`repozytorium_1`, `repozytorium_2`). W tym katalogu znajdują się podkatalogi z nazwami odpowiadającymi identyfikatorom zidentyfikowanych zmian refaktoryzacyjnych

(`COMMIT_HASH_1`, `COMMIT_HASH_2`) oraz plik `README.txt` zawierający zagregowane informacje o danych zebranych z danego repozytorium (dzięki nim dokonano późniejszej kalkulacji rozmiaru zebranych danych). W każdym katalogu zmiany znajduje się podkatalog `before` zawierający zawartość plików przed oraz `after` zawierający zawartość plików po wykonanej refaktoryzacji.

Każda klasa została przechowana w pliku z oryginalną nazwą z analizowanego projektu, ale z pominiętą oryginalną strukturą katalogów. Dzięki takiemu spłaszczeniu struktury na tym etapie łatwiej przeglądnąć pozyskane dane. Teoretycznie takie działanie może prowadzić do powstania konfliktów nazw, gdy tak samo nazwane klasy w projekcie znajdowały się w różnych pakietach, a skrypt próbuje przenieść je do jednego folderu. W analizowanych danych jednak taka sytuacja nie wystąpiła.

Lista plików w katalogach `before` i `after` zawsze jest taka sama. W katalogu każdej zmiany skrypt zamieścił także plik `README.txt`, który zawiera jej pełny *commit message*.

4.2. Analiza pozyskanych zmian refaktoryzacyjnych

W programowaniu obiektowym klasa jest podstawową jednostką kodu systemu. Jest ona logiczną całością, która powinna realizować jedną funkcjonalność (z ang. SRP – zasada pojedynczej odpowiedzialności). Klasa jest definicją, według której tworzone są obiekty, które w trakcie działania systemu wchodzą ze sobą w interakcję i w konsekwencji spełniają wspólnie zadania nałożone na oprogramowanie. Dzięki takiemu zorganizowaniu kodu źródłowego, jest on uporządkowany, a co za tym idzie – łatwiej nim zarządzać. Co więcej, umieszczenie kodu źródłowego w odpowiednich klasach powoduje, że raz napisana funkcjonalność systemu nie jest powtarzana w innym miejscu. W przypadku zaistnienia potrzeby powtórzenia danej funkcjonalności – języki zorientowane obiektowo pozwalają na stworzenie obiektu istniejącej już klasy w innym miejscu. Dzięki temu kod jest „reużywalny". Nawet sama zasada SRP jest niekiedy przedstawiana jako „nie powinien istnieć więcej niż jeden powód do modyfikacji danej klasy" [93].

Gdyby SRP było zawsze stosowane na poziomie klas – byłyby one idealną jednostką kodu, która powinna zostać poddana analizie przy budowie jakościowego modelu kodu. Java jest językiem programowania bardzo silnie zorientowanym obiektowo. Oznacza to, że nie możemy napisać w Javie kodu, który nie jest częścią jakiejś klasy. Niestety, z drugiej strony ta cecha sprawia, że często klasy w Javie są źle rozumiane, szczególnie przez początkujących programistów. W kodzie programów możemy często

natknąć się na „klasy wszechwiedzące” [5] (z ang. *God Class*). Jest to typowy przykład zapachu kodu (zob. sekcja 2.3). Jak sama nazwa wskazuje, klasy te „wiedzą” wszystko, realizują zbyt wiele funkcjonalności, zdecydowanie nie spełniają zasady pojedynczej odpowiedzialności.

Niektóre refaktoryzacje kodu źródłowego prowadzą do rozdzielenia funkcjonalności danej klasy na kilka innych [20]. Często też zdarza się, że oryginalna klasa jest całkowicie usuwana w takiej sytuacji na rzecz zupełnie nowej, dokładniej przemyślanej struktury kodu. Dzięki temu kod staje się czytelniejszy i czystszy, ale taka refaktoryzacja sprawia spore trudności w jej automatycznej analizie. Trudno bowiem jednoznacznie stwierdzić, które części danej klasy zostały przeniesione do innych, jak zostały logicznie pogrupowane i czy w ogóle kod wewnątrz tych klas został zmieniony czy tylko przeniesiony w inne miejsce.

Ta, ręcznie wykonana, analiza sprawiła, że w trakcie pracy nad rozprawą odrzucono pomysł analizowania jakości kodu na poziomie klas i postanowiono na zawężenie zakresu prac i dzięki temu – zwiększeniu dokładności modelu.

4.2.1. Metody klas

Podstawowym składnikiem klas w obiektowych językach programowania są metody. Metoda reprezentuje zachowanie, które potrafi dostarczyć obiekt danej klasy. Składa się ona z sygnatury (nazwa, lista parametrów zawierająca ich nazwy oraz typy, typ zwracany) oraz ciała (kod źródłowy zawierający implementację danego zachowania). Metoda również powinna spełniać zasadę SRP, ale na poziomie niższym niż klasa. Nadal powinna realizować jedną funkcjonalność, ale powinna ona być mniejsza niż funkcjonalność klasy. Przykład: klasa realizuje funkcjonalność wyświetlenia okna dialogowego. Zawiera w sobie trzy metody, odpowiadające za wyświetlenie tytułu, treści i przycisku w oknie dialogowym, który pozwala na jego zamknięcie.

Zdarza się, że metoda w trakcie implementacji systemu też przestaje spełniać zasadę pojedynczej odpowiedzialności. Pierwszym zapachem kodu sugerującym takie naruszenie jakości jest często zbyt duża liczba linii kodu w ciele metody. W takiej sytuacji, z dużą dozą prawdopodobieństwa realizuje ona więcej niż jedną odpowiedzialność (R. Martin w [5] wskazał nawet niezobowiązujący limit długości metody jako *Functions [methods] should hardly ever be 20 lines long.*). Taki zapach kodu nazywany jest po prostu „Long Method” [5] (z ang. długa metoda), a jego refaktoryzacja polega na podzieleniu metod na mniejsze. Co ważne, jeśli klasa wewnątrz której znajduje się zbyt długa metoda mimo wszystko spełnia SRP, nowo powstałe w ten sposób metody w większości przypadków pozostają w tej samej klasie. Sama metoda, która zostaje

poddana takiemu przekształceniu również pozostaje na swoim miejscu bez zmiany nazwy – jest tylko mniejsza. To powoduje, że możliwe jest trafne wykrycie takiej zmiany automatycznie oraz dokładne przeanalizowanie zmian definicji danej metody.

Oczywiście, istnieje zdecydowanie więcej zapachów kodu rozwiązywanych na poziomie metod: przykładowo parametr flagowy (ang. *Flag Argument*) lub zbyt wiele parametów metody (z ang. *Too Many Parameters*). Wszystkie refaktoryzacje dotyczące wyrażeń używanych w kodzie źródłowego – a nie jego struktury – również przeprowadzane są w ciele metod: upraszczanie warunków logicznych, próby zmniejszania złożoności cyklomatycznej, ekstrakcja lokalnych zmiennych tłumaczących (ang. *explaining variables*), wydzielanie odpowiednich poziomów abstrakcji – by wymienić kilka pierwszych. Co ważne – wszystkie te przekształcenia poprawiają jakość i czytelność kodu, jednocześnie pozostawiając ten kod w tej samej klasie w i tej samej metodzie.

Omówione wyżej cechy metod klas w programowaniu obiektowym oraz mnogość różnych refaktoryzacji kodu, które są przeprowadzane na ich poziomie skłoniły autora rozprawy do użycia metod z klas istniejących w pozyskanych zmianach refaktoryzacyjnych jako jednostek kodu, z których pozyskane zostaną dane wejściowe dla modelu SCQM.

4.2.2. Metody przeciążone

Rozpoznawanie zmian refaktoryzacyjnych wewnątrz metod wymaga znalezienia tej samej metody w dwóch analizowanych zmianach. W tym celu wyszukiwana jest metoda o tej samej nazwie w tej samej klasie. Jeśli taka istnieje – a w ich ciele lub sygnaturze została wprowadzona jakaś zmiana – taka sytuacja tworzy próbkę refaktoryzacji używaną w dalszym przygotowaniu danych dla jakościowego modelu kodu źródłowego.

To podejście staje się problematyczne, gdy weźmiemy pod uwagę mechanizm przeciążania metod w języku Java. Przeciążanie metod umożliwia zdefiniowanie dowolnej liczby metod o tej samej nazwie w jednej klasie, pod warunkiem że każda z nich ma inną listę argumentów pod względem ich typu lub liczby. Jeśli przed refaktoryzacją w klasie istnieje inna liczba metod przeciążonych niż po refaktoryzacji, lub gdy lista argumentów dowolnej przeciążonej metody w analizowanej zmianie uległa modyfikacji. Nie da się automatycznie stwierdzić, która z analizowanych metod została przekształcona w którą. W związku z tym, skrypt dokonujący rozbioru syntaktycznego metod (opisany w sekcji 4.3) pomija takie przykłady.

4.3. Rozbiór syntaktyczny i ekstrakcja metod

Drzewa składniowe AST są metodą reprezentacji konstrukcji w kodzie źródłowym. Jest to uniwersalny zapis dowolnego języka programowania za pomocą odpowiednio zagnieżdżonych tokenów. Jak sama nazwa wskazuje – reprezentacja przyjmuje strukturę drzewiastą. Każdy węzeł tego drzewa reprezentuje pewną konstrukcję języka, a jego potomkowie składowe tej konstrukcji.

Rozbiór syntaktyczny jest wierną reprezentacją konstrukcji językowych i wyrażeń użytych w kodzie źródłowym [94]. Niemniej jednak, decyzja o jego wykorzystaniu przy dalszych przekształceniach zgromadzonych próbek kodu źródłowego niesie za sobą kilka konsekwencji.

Po pierwsze, dzięki AST niezwykle łatwo będzie zidentyfikować metody w kodzie źródłowym i wyekstrahować je z wybranych już klas – wystarczy odnaleźć odpowiedni token reprezentujący tę część językową. Wszystkie węzły, poczynając od znalezionego aż do samego dołu hierarchii, będą reprezentować pojedynczą metodę danej klasy, czyli dokładnie tą część kodu źródłowego na podstawie której ma być zbudowany SCQM (zob. sekcja 4.2).

Po drugie, ograniczenie się do możliwych elementów, które mogą wystąpić w języku programowania ogranicza liczbę tokenów, które będą składać się na alfabet wejściowy modelu (zob. sekcja 4.5). To powinno znacząco podnieść skuteczność uczenia.

Po trzecie, porównywanie zmian, które zaszły w rozbiorze syntaktycznym kodu źródłowego pomija nieistotne zmiany, takie jak formatowanie kodu czy białe znaki. Owszem, te aspekty kodu źródłowego wpływają na jego czytelność, ale mnogość narzędzi potrafiących nie tylko wykryć problemy z formatowaniem, ale także automatycznie je naprawić, skłania autora rozprawy do opinii, że programiści podczas przeglądów kodu w ogóle nie powinni tymi aspektami się zajmować (zob. sekcja 2.5). Co za tym idzie – jakościowy model kodu źródłowego również nie powinien ich brać pod uwagę. Tym bardziej, że analiza formatowania kodu mogłaby wprowadzić do procesu uczenia zbędny szum, który zaburzyłby istotne informacje o rzeczywistej jakości i czytelności wykorzystanych konstrukcji.

Oczywiście decyzja o wykorzystaniu AST niesie też za sobą kolejne zawężenie zakresu prowadzonych badań. Jak zaznaczono wyraźnie w sekcji 3.2, ważną informacją, która jest tracona podczas rozbioru syntaktycznego kodu źródłowego są nazwy poszczególnych elementów: klas, metod, parametrów, zmiennych. Bardzo często odpowiednio dobrana nazwa jest kluczowa przy próbie zrozumienia kodu przez programistę w [5] wydzielono całą grupę zapachów kodu związaną z nazewnictwem elementów, np. *Choose descriptive names* lub *Avoid encodings*. Ich refaktoryzacja polega wyłącznie

na wymyśleniu takiej nazwy, która trafniej opisuje rolę danego elementu. Zdecydowano, że ten problem wykracza poza ramy rozprawy oraz budowanego modelu jakościowego.

Budowa drzewa AST dla języka Java

Do rozbioru syntaktycznego zgromadzonych klas i ekstrakcji metod użyto biblioteki JavaParser[6]. Jest to otwarty parser języka Java w wersjach od 1.0 do 10 (w momencie pisania rozprawy). Dla zadanego kodu źródłowego potrafi zbudować AST, którego każdy typ węzła posiada swoją implementację w klasie rozszerzającej klasę `com.github.javaparser.ast.Node`.

W celu wykrycia i rozbioru metod w wybranych klasach ze zmian refaktoryzacyjnych napisano prosty program `java-parser/src/main/java/MethodTokenizer.java`[92], który wykorzystuje możliwości wspomnianego wyżej parsera. Za pomocą wzorca projektowego wizytator, program reaguje na pojawienie się węzła typu `com.github.javaparser.ast.body.MethodDeclaration` i przechwytuje wszystkie węzły, które znalazły się pod nim. Na tym etapie nie są już potrzebne instancje danych węzłów, więc są one automatycznie przetwarzane na ciąg znaków reprezentujący rozbiór syntaktyczny danej metody, używając nazwy klasy węzła (bez nazwy pakietowej). W rezultacie po pojawieniu się metody w trakcie budowania drzewa składniowego przez JavaParser, program generuje drzewo AST danej metody i przechowuje je w zmiennej typu `String`. Na koniec, z węzła `MethodDeclaration` pobierana jest nazwa analizowanej metody tak, że wynikiem przetworzenia jednej klasy jest zestaw par wartości składających się z nazwy metody oraz jej rozbioru syntaktycznego.

W ten sposób przetwarzana jest kolejno każda klasa z katalogów `before` i `after` (zob. Listing 4.7) każdej zidentyfikowanej zmiany refaktoryzacyjnej. Wynikiem działania programu są pliki zawierające kod źródłowy metody oraz drzewo AST – przed i po zmianie refaktoryzacyjnej. Części tego pliku są rozdzielone ustalonym separatorem, zgodnie ze strukturą zaprezentowaną na Listingu 4.8. Nazwy plików z rozbiorem syntaktycznym są tworzone według schematu `$NAZWA_PLIKU$NAZWA_METODY.txt` i zapisywane w katalogu danej zmiany w podkatalogu `diffs` (na tym samym poziomie co katalogi `before` i `after`). Listing 4.9 zawiera zaktualizowaną (w stosunku do Listingu 4.7) strukturę katalogów, która przechowuje także rozbiory syntaktyczne metod.

[6] `https://javaparser.org`

```
$KOD_PRZED
<separator>
$KOD_PO
<separator>
$AST_PRZED
<separator>
$AST_PO
```

Listing 4.8: Struktura pliku przechowującego drzewa AST przed i po refaktoryzacji

```
repozytorium_1/
  COMMIT_HASH_1/
    after/
    before/
    diffs/
      Class1.javamethodA.txt
      Class1.javamethodB.txt
      Class2.javamethodC.txt
      Class2.javamethodD.txt
  COMMIT_HASH_2/
    ...
repozytorium_2/
  ...
```

Listing 4.9: Struktura danych przechowujących rozbiór syntaktyczny metod

Rozbiór syntaktyczny jest zapisywany wyłącznie dla metod, które się zmieniły w analizowanym *commicie*. Jeśli któraś metoda jest usuwana lub dodawana – nie jest ona brana pod uwagę w dalszym przetwarzaniu. Nie można bowiem założyć, że każda usunięta metoda była kodem niskiej jakości, a każda dodana metoda – wysokiej, nawet jeśli bierzemy pod uwagę zmianę refaktoryzacyjną. Takie założenia mogłyby znacznie osłabić budowany model SCQM.

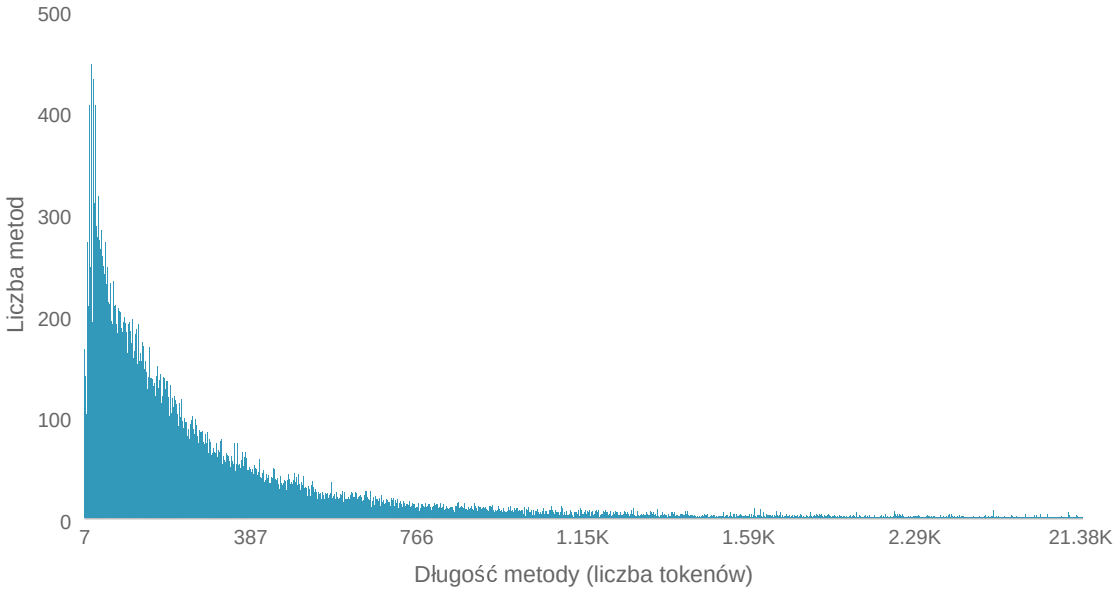
W dodatku B umieszczono szczegółowy opis przygotowania danych na przykładzie jednej ze zidentyfikowanych zmian refaktoryzacyjnych.

4.4. Usuwanie szumu z danych treningowych

Za pomocą opisanego sposobu pozyskiwania kodu źródłowego metod klas poddawanych refaktoryzacji zgromadzono 60764 metod. Jeszcze przed zebraniem opinii o jakości kodu od programistów (zob. opis badania w rozdziale 5), postanowiono przeanalizować zgromadzone atuomatycznie metody w celu zidentyfikowania potencjalnych problemów i dokładniejsze przefiltrowanie danych wejściowych dla modelu SCQM. W kolejnych podsekcjach przedstawiono wprowadzone pomysły na usunięcie szumu ze zbioru danych treningowych i modelu wraz z uzyskanymi rezultatami.

4.4.1. Ograniczenie długości metod

Aby lepiej poznać profil zgromadzonych danych, za pomocą skryptu `8_token_lengths_histogram.php` z repozytorium [92] zliczono długości zgromadzonych metod. To z kolei pozwoliło na naszkicowanie histogramu przedstawiającego rozkład zebranych wartości – przedstawiono go na Rysunku 4.1.



Rys. 4.1: Histogram długości zebranych metod (w tokenach)

Bez zaskoczenia – przeważająca większość refaktoryzowanych metod jest krótka – dużo krótsza od maksymalnej długości. Moda dla zbioru długości metod jest równa 29, mediana – 168.

W przypadku zbyt długich metod zmiany w nich przeprowadzane dotyczą często tylko jednego z wielu fragmentów kodu i nawet jeśli jest poprawiana jego jakość –

w trakcie uczenia modelu bezwzględnego model będzie źle instruowany o jakości kodu całej metody. Z tego powodu postanowiono ograniczyć długość metod, które będą składać się na zbiór treningowy.

Aby znaleźć odpowiednią maksymalną długość metody, do której powinno zostać ograniczone wejście, przeanalizowano jaka część zbioru danych pozostaje przy jej zmniejszaniu. W ten sposób oczyszczono zbiór danych z przykładów, które na pewno nie prezentują kodu wysokiej jakości oraz – docelowo – przyspieszono obliczenia modelu. Tabela 4.3 przedstawia zależność maksymalnej długości metody od liczby próbek, która pozostaje po nałożeniu takiego ograniczenia. Ostatecznie do obliczeń przyjęto ograniczenie długości metody do 200 tokenów. To ograniczenie wydaje się już spore, jednakże 200 tokenów w rozbiorze syntaktycznym metody przekłada się na około 30-40 linii kodu, co według książki [5] oraz autora rozprawy i tak jest jeszcze zbyt długą metodą.

4.4.2. Zliczanie zmienionych tokenów

Analizując przypadkowo wybrane przykłady zebranych refaktoryzacji, zwrócono uwagę na powtarzające się sytuacje, których nie wzięto wcześniej pod uwagę, a które pojawiały się w danych wejściowych. Ich jakość byłaby trudna do rozstrzygnięcia nawet przez programistę bez znajomości dodatkowego kontekstu. Niektóre przykłady to:

1. metody w interfejsach, do których została dodana domyślna implementacja (jest to możliwe od Javy w wersji 8),
2. przeniesienie implementacji w górę hierarchii klas – zastąpienie metody, która dotychczas była abstrakcyjna na metodę nieabstrakcyjną posiadająca implementację,
3. usunięcie całego ciała metody,
4. wykomentowanie lub usunięcie części kodu w ciele metody bez zamiany na np. wywołanie wydzielonej metody prywatnej,
5. dodanie nowego kodu w ciele metody,
6. minimalna zmiana w ciele metody, np. zmiana operatora `<` na `<=` w warunku logicznym.

Tabela 4.3: Liczba próbek pozostałych w zbiorze danych wejściowych po wprowadzeniu poszczególnych ograniczeń na maksymalną długość metody

Maksymalna długość metody	**Liczba pozostałych próbek**	**Część oryginalnego zbioru wejściowego**
33000	60764	100%
32000	60763	100%
30000	60763	100%
20000	60759	100%
10000	60747	100%
5000	60690	100%
2000	60087	99%
1000	57918	95%
800	56397	93%
700	55254	91%
600	53589	88%
500	51358	85%
400	47910	79%
300	42613	70%
200	34204	56%
100	20226	33%
50	10382	17%

Odkrycie takich przykładów w zebranych przykładach refaktoryzacji odsłoniło potrzebę dokładniejszego ich przefiltrowania. Oprócz ograniczenia długości metod, należy także sprawdzić czy kod wewnątrz nich rzeczywiście jest zmieniany. Należy wykluczyć przypadki, w których kod jest tylko dodawany bądź usuwany. W takiej sytuacji nie tylko programista nie byłby w stanie określić, czy kod po danej zmianie jest czytelniejszy, ale także podawanie takiego przykładu w trakcie uczenia modelu bezwzględnego wprowadza go w błąd. W ten sposób przy usuwaniu ciała metody model uczy się, że metoda bez ciała jest metodą z kodem wysokiej jakości, a przy dodaniu implementacji – odwrotnie.

W celu przygotowania danych, przetworzono raz jeszcze zgromadzone rozbiory syntaktyczne metod i na czas przetwarzania umieszczono każdy węzeł drzewa w osobnej linii (zob. skrypt `15_prepare_rnn_input_by_diff.php` z repozytorium [92]). Następnie porównano tak przygotowane rozbiory syntaktyczne za pomocą prostej implementacji porównywania zawartości ciągów znaków `Diff`[7]. Wynikiem takiego porównania jest tablica, z której każdy element jest parą wartości informującą o tym co stało się odpowiednio z danym tokenem przed i po refaktoryzacji. Informacje te zostały przekazane jako stałe:

1. `Diff::UNMODIFIED` jeśli token przed i po przeprowadzonej zmianie jest niezmieniony,
2. `Diff::INSERTED`, jeśli token został dodany w trakcie refaktoryzacji,
3. `Diff::DELETED`, jeśli token został usunięty w trakcie refaktoryzacji.

W celu wyeliminowania niejednoznacznych przykładów wymienionych na początku tej sekcji, ze zbioru danych wejściowych odrzucono przykłady które w wyniku porównania nie zwróciły obydwu stałych oznaczających dodanie lub usunięcie danego tokenu. Odfiltrowano w ten sposób przekształcenia, które tylko dodają bądź tylko usuwają kod z metody.

Ponadto, dla każdej zmiany wyznaczono liczbę zmienionych tokenów, zliczając pary zawierające wartość `Diff::INSERTED` lub `Diff::DELETED`.

4.4.3. Decyzje usprawniające przebieg uczenia

Najlepszą kombinacją reguł i parametrów opisanych w tej sekcji okazało się:

- ograniczenie długości metod do 200 tokenów (do 30-40 linii kodu)
- odfiltrowanie zmian, które wyłącznie dodają lub usuwają kod
- odfiltrowanie zmian, które zmieniają mniej niż 5 lub więcej niż 100 tokenów w rozbiorze syntaktycznym

Przyjęcie nowych reguł filtrowania danych wejściowych spowodowało kolejne zmniejszenie ich liczby. Po nałożeniu nowych warunków na dane wejściowe w zbiorze pozostaje 9925 zmian refaktoryzacyjnych. Porównując tę liczbę do liczby wszystkich pozyskanych metod o długości do 200 tokenów z tabeli 4.3 można stwierdzić, że nowe metody filtrowania odrzuciły niemal $\frac{3}{4}$ analizowanych dotąd zmian jako niejednoznaczne.

[7] `http://code.iamkate.com/php/diff-implementation/`

4.5. Format danych wejściowych modelu

Oczekiwanym formatem danych wejściowych dla rekurencyjnej sieci neuronowej jest sekwencja. Z tego powodu kolejnym krokiem było takie przekształcenie posiadanych rozbiorów syntaktycznych, by można było reprezentować je jako ciągi kolejno następujących po sobie węzłów.

Zbiór tokenów, które mogą zostać zwrócone przez omówiony w sekcji 4.3 parser w sposób naturalny tworzy alfabet, którym będzie posługiwać się model SCQM w trakcie trenowania oraz przy późniejszych klasyfikacjach. Po zliczeniu klas implementujących poszczególne rodzaje węzłów AST okazało się, że rozmiar alfabetu wejściowego jest równy 74. Pełny alfabet wejściowy znajduje się w pliku `java-parser/tokens-java.txt` w repozytorium [92].

Aby zbliżyć istniejącą reprezentację do sekwencji, zamieniono wcięcia symbolizujące zagnieżdżenie poszczególnych węzłów na nawiasowanie. W ten sposób uzyskiwano jeden długi ciąg znaków, w którym zachowane są wszystkie informacje z oryginalnego rozbioru syntaktycznego, łącznie ze strukturą kodu źródłowego. Listing 4.10 przedstawia przekształcony w ten sposób rozbiór syntaktyczny z listingu B.4.

```
(MethodDeclaration((BlockStmt(ReturnStmt(
    ↪ MethodCallExprNullLiteralExprNameExpr(SimpleName)SimpleName)))
    ↪ ClassOrInterfaceType(SimpleName)SimpleNameMarkerAnnotationExpr(
    ↪ Name)))
```

Listing 4.10: Reprezentacja rozbioru syntaktycznego metody w postaci ciągu znaków

Wprowadzenie nawiasowania wymagało dodania kolejnych dwóch słów do alfabetu wejściowego: nawiasu otwierającego „`(`” oraz zamykającego „`)`”. W rezultacie, liczba możliwych tokenów wzrosła do 76.

Na tym etapie zdecydowano też o zamianie tokenów składających się na alfabet wejścia na liczby w celu ich łatwiejszej reprezentacji w kodzie źródłowym modelu. Każdemu tokenowi przypisano liczbę naturalną, będącą numerem linii z pliku zawierającego alfabet wejściowy (`java-parser/tokens-java.txt` w repozytorium [92]), przy czym pierwsza linia to 0, druga to 1 itd. W ten sposób zdefinowano alfabet wejścia jako liczby naturalne od 0 do 75. Na listingu 4.11 przedstawiono przekształcony rozbiór syntaktyczny z listingu 4.10. Liczby, będące tokenami, rozdzielono przecinkami, które służą wyłącznie odseparowaniu kolejnych wyrazów wejścia od siebie. Brak takiej separacji w poprzedniej reprezentacji rozwiązano za pomocą zamiany tokenów na liczby w malejącym porządku według długości tokenów. W ten sposób najpierw zamieniono podstring `NameExpr` a dopiero potem `Name`.

```
74,56,74,74,20,74,32,74,73,63,36,74,4,75,4,75,75,75,38,74,4,75,
    ↪ 4,60,74,10,75,75,75
```

Listing 4.11: Reprezentacja rozbioru syntaktycznego metody w postaci sekwencji liczb

Reprezentacja rozbioru syntaktycznego jako ciągu liczb spowodowała naturalny wybór formatu danych wejściowych dla budowanego modelu – plików CSV.

W celu ułatwienia wczytania danych treningowych do docelowej implementacji sieci neuronowej podjęto decyzję o reprezentacji wejścia jako dwuwymiarowej macierzy. Oczywiście, zebranych metod nie cechuje ta sama liczba tokenów. Przed przygotowaniem danych należy więc znaleźć najdłuższą z nich i ustalić w ten sposób pożądany rozmiar każdej sekwencji danych wejściowych. Krótsze metody które mają znaleźć się w wejściowym zbiorze danych należy dopełnić następnie zerami tak, by liczba tokenów w każdym wierszu danych była jednakowa. Liczbę znaczących tokenów w każdej sekwencji wejściowej będzie przekazywana osobno do sieci neuronowej.

Poza rozbiorami syntaktycznymi i ich długością, w trakcie uczenia modelu należy także dostarczyć oczekiwaną klasyfikację danego przykładu. Zgodnie z projektem (zob. sekcje 3.5 oraz 3.6), klasyfikacją przykładu jest para liczb $[P_G, P_B]$, gdzie P_G oznacza prawdopodobieństwo pozytywnej próbki a P_B – prawdopodobieństwo negatywnej próbki. W trakcie uczenia dla każdej sekwencji wejściowej należy przekazać jako oczekiwaną klasyfikację wektor $[1, 0]$ dla przykładu pozytywnego i $[0, 1]$ dla przykładu negatywnego.

Aby zminimalizować wpływ sposobu pozyskania danych na wyniki uczenia, przygotowywane sekwencje były układane w losowej kolejności. Dzięki temu metody pobrane z kolejnych projektów nie znajdowały się obok siebie w macierzy wejściowej.

Dane w kolejnych etapach uczenia modelu (zob. rozdział 6) były przygotowywane za pomocą skryptów `10_*` – `15_*` z repozytorium [92]. Gotowe dane wejściowe zostały umieszczone w katalogu `input` w repozytorium [95]. Poza kilkoma wyjątkami, nazwy katalogów z danymi wejściowymi były tworzone według reguły `M-D-java`, gdzie M to maksymalna długość wiersza wejścia a D to skrótowy opis metody przygotowania zbioru. Przykładowo `200-diff5to100noonlyaddeddel-java` to katalog z danymi wejściowymi, które dały najlepszy rezultat uczenia obydwu modeli przed wprowadzeniem wiedzy ekspertowej (maksymalna długość metody to 200 tokenów, odrzucono zmiany które wprowadzały mniej niż 5 i więcej niż 100 zmienionych tokenów, porzucono zmiany które tylko dodawały i usuwały kod; zob. sekcję 4.4). W zależności od modelu dla którego zbiór danych był przygotowywany, foldery z plikami wejściowymi zawierają pliki dla modelu aSCQM lub rSCQM (zob. kolejne sekcje 4.5.1 i 4.5.2).

4.5.1. Format danych wejściowych dla modelu aSCQM

Zgodnie z projektem modelu bezwzględnego aSCQM (zob. sekcja 3.5), na wejście powinien on otrzymać kolejno wszystkie metody w wersji przed i po refaktoryzacji. W tym przypadku metody przed zmianą powinny być klasyfikowane jako kod niskiej jakości a metody po zmianie – wysokiej.

Zgodnie z wymaganiami opisanymi wcześniej w tej sekcji, na wejście do uczenia modelu bezwzględnego składają się trzy pliki:

1. `input.csv` – zawiera kolejne sekwencje reprezentujące rozbiory syntaktyczne pozyskanych metod (jedna sekwencja wejściowa zajmuje jedną linię w pliku),
2. `lengths.csv` – zawiera rozdzielone przecinkami długości kolejnych rozbiorów syntaktycznych z pliku `input.csv`; zawiera zawsze jeden wiersz danych; liczba elementów w tym wierszu odpowiada liczbie rozbiorów syntaktycznych (liczbie linii) w pliku `input.csv`,
3. `labels.csv` – oczekiwane klasyfikacje kodu źródłowego z pliku `input.csv`; każda linia w tym pliku to wektor $[1, 0]$ lub $[0, 1]$, oznaczający, że kod, którego rozbiór syntaktyczny jest w tej samej linii w pliku `input.csv` jest odpowiednio wysokiej lub niskiej jakości.

Liczba sekwencji w pliku `input.csv` jest zawsze dwukrotnie większa od przedstawianych w rozprawie liczby analizowanych próbek refaktoryzacji. Wynika to z faktu, że w modelu bezwzględnym każdy przypadek refaktoryzacji daje dwie sekwencje wejściowe – rozbiór AST przed zmianą (przykład negatywny) oraz rozbiór AST po zmianie (przykład pozytywny).

4.5.2. Format danych wejściowych dla modelu rSCQM

Zgodnie z opisem modelu względnego rSCQM w sekcji 3.6, w każdej iteracji uczenia wymaga on podania na wejściu kodu przed i po wykonanej zmianie. Zakłada się, że podanie kodu przed refaktoryzacją w pierwszej kolejności powinno być sklasyfikowane jako pozytywna zmiana (kod poprawia się, gdyż jest poddawany refaktoryzacji). Natomiast podanie w pierwszej kolejności rozbioru po refaktoryzacji powinno być sklasyfikowane jako przykład negatywny (jakość kodu spada, gdy wprowadzona do kodu refaktoryzacja jest wycofywana).

Decyzja o tym, czy daną sekwencję wejściową należy przyjąć tak jak została podana i potraktować ją jako przykład pozytywny lub czy należy ją odwrócić i potraktować jako przykład negatywny zostanie podjęta na etapie wczytywania danych do sieci

neuronowej, w trakcie uczenia. Dzięki temu w kolejnych iteracjach ten sam przykład może być potraktowany na różne sposoby, prezentując w odwróconych przypadkach niejako zmianę wycofującą refaktoryzację. To pozwala na przedstawienie modelowi różnych punków widzenia na tę samą próbkę i pokazania, że nie każda zmiana podnosi jakość kodu – nawet w danych treningowych. Z tego też powodu w plikach będących wejściem dla modelu względnego nie jest podawana oczekiwana klasyfikacja.

Uwzględniając powyższe wymagania, dane treningowe dla modelu względnego złożone są z czterech plików:

1. `input-before.csv` – zawiera kolejne sekwencje reprezentujące rozbiory syntaktyczne pozyskanych metod; jedna sekwencja wejściowa zajmuje jedną linię w pliku; umieszczone tu rozbiory syntaktyczne opisują metody przed wykonaniem refaktoryzacji, czyli próbki o lepszej jakości kodu,

2. `lengths-before.csv` – zawiera rozdzielone przecinkami długości kolejnych rozbiorów syntaktycznych z pliku `input-before.csv`; zawiera zawsze jeden wiersz danych; liczba elementów w tym wierszu odpowiada liczbie rozbiorów syntaktycznych (liczbie linii) w pliku `input-before.csv`,

3. `input-after.csv` – zawartość analogiczna do pliku `input-before.csv`, przy czym zawarte tu rozbiory syntaktyczne opisują metody po wykonaniu refaktoryzacji, czyli próbki o gorszej jakości kodu; liczba sekwencji wejściowych w tym pliku jest taka sama jak w `input-before.csv`,

4. `lengths-after.csv` – zawartość analogiczna do pliku `lengths-before.csv`, przy czym liczby znaczących tokenów dotyczą sekwencji wejściowych z pliku `input-after.csv`.

Liczba sekwencji danych w plikach `input-*.csv` jest zawsze równa przedstawianej w rozprawie liczbie analizowanych próbek refaktoryzacji. Wynika to z faktu, że na jeden przykład analizowany przez model danych składa się zmiana wprowadzona do kodu źródłowego, czyli zarówno sekwencja reprezentująca rozbiór syntaktyczny z pliku `input-before.csv` oraz sekwencja z pliku `input-after.csv`.

4.6. Podsumowanie

Jak zaznaczono w sekcji 4.1.1, **350 otwartych projektów z GitHuba** zostało przeszukanych pod kątem zmian refaktoryzacyjnych. Według zadanych słów kluczowych (zob. Tabela 4.2) odnaleziono w nich **17967 zmian wykonujących refaktoryzację** kodu źródłowego. Commity te obejmowały zmiany w **163013 klasach Javy**. Spośród nich zidentyfikowano 121696 metod, które były modyfikowane podczas znalezionych refaktoryzacji.

Po zastosowaniu bardziej rygorystycznego podejścia w identyfikacji zmian refaktoryzacyjnych tylko po temacie *commita*, w danych pozostaje **60764 metody** (50% zbioru wyjściowego), w których potencjalnie udało się zidentyfikować zmiany poprawiające jakość kodu.

Po analizie zebranych próbek podjęto próby usunięcia z nich szumu danych, nakładając ograniczenia na ich długość oraz liczbę zmienianych tokenów. W ten sposób na zbiór danych treningowych najlepszego uzyskanego wariantu jakościowego modelu kodu źródłowego składa się **9925 metod poddanych refaktoryzacji**.

Rozdział 5

Badanie opinii programistów o jakości kodu

Uzyskane w sposób automatyczny próbki kodu o różnej jakości stanowią solidną bazę wyjściową, która powinna umożliwić przekazanie do jakościowego modelu kodu źródłowego odpowiedniej wiedzy. Jak jednak już podkreślono kilkakrotnie, jakość kodu źródłowego jest niezwykle subiektywnym pojęciem, rozumianym różnie przez różnych programistów. W celu poprawienia trafności i zbudowania wiarygodnego zbioru walidacyjnego opracowano i przeprowadzono i opisano w niniejszym rozdziale badanie opinii programistów na temat jakości kodu źródłowego.

5.1. Klasyfikacja przez programistów

W celu poprawienia jakości danych wejściowych i w konsekwencji – uzyskania lepszej trafności tworzonego modelu jakościowego – postanowiono przeprowadzić badanie pozwalające sklasyfikować zebrane przykłady refaktoryzacji lepiej niż automatyczne reguły zastosowane podczas gromadzenia danych (zob. sekcję 4.1.2).

Podstawowym ryzykiem obranej metodyki gromadzenia próbek kodu źródłowego o różnej jakości jest błędna identyfikacja zmian wprowadzających refaktoryzację. Zaproponowana w sekcji 4.1.2 metoda wyszukiwania zmian po słowach kluczowych może się czasem nie sprawdzać, a co za tym idzie – dostarczać do modelu dane, które wprowadzają szum informacyjny lub – co gorsze – wprowadzają model w błąd. Co więcej, te obawy zostały potwierdzone gdy podczas uczenia modelu zidentyfikowano szereg kolejnych problemów w danych wejściowych takich jak zmiany dotyczące dodawania lub usuwania całego ciała metody lub wykomentowywania całej implementacji (zob.

sekcja 4.4). Nawet jeśli poprawnie wybrano *commit* wprowadzający refaktoryzację, nie można być pewnym że wszystkie zmieniane w nim metody rzeczywiście podnoszą jakość swojego kodu.

W celu wyeliminowania problematycznych przykładów w danych wejściowych oraz uzyskania ich poprawnej klasyfikacji, zaprojektowano i przeprowadzono badanie mające na celu zebranie od programistów opinii na temat pojmowania przez nich jakości i czytelności kodu źródłowego.

5.2. Projekt badania

W celu zebrania opinii, zostaje przygotowana specjalna platforma działająca online, prezentująca kod przed i po zmianie. Respondent ma możliwość oceny tej zmiany na trzy sposoby:

1. kod przed zmianą jest lepszy,
2. kod po zmianie jest lepszy,
3. jakość kodu w tej zmianie nie zmienia się.

Każdy przykład zostaje danej osobie przedstawiony tylko jeden raz – powtórzenia są wykluczane na poziomie przeglądarki internetowej. Przykłady są podawane użytkownikom w kolejności losowej. Czas na udzielenie odpowiedzi jest ograniczony do 60 sekund. Limit ten został ustalony w oparciu o kilkanaście klasyfikacji testowych na etapie projektowania badania wykonanych przez różnych programistów. Po upływie tego czasu przykład klasyfikowany jest jako niezmieniający jakości kodu i wyświetlana jest kolejna próbka. Nie można też udzielić odpowiedzi szybciej niż po 5 sekundach od zaprezentowania danego przykładu (przeciwdziała to nagminnemu klasyfikowaniu bez zastanowienia).

Aby dany przykład został uznany za sklasyfikowany, co najmniej 3 osoby muszą podjąć tę samą decyzję na jego temat. Jeśli przykład zebrał więcej niż 5 głosów i powyższy warunek nie został spełniony, jest on klasyfikowany jako niezmieniający jakości kodu. Przykłady oznaczone jako sklasyfikowane nie są już przedstawiane kolejnym respondentom.

Eksperyment w każdej chwili można przerwać i wrócić do niego w dowolnym momencie. Przeglądarka internetowa zapisuje liczbę sklasyfikowanych przez danego użytkownika metod tak, by można było ocenić swój wkład w budowę modelu SCQM.

5.2.1. Ograniczenie losowych lub nieprzemyślanych opinii

W celu przeciwdziałania losowym decyzjom udzielanym przez niedoświadczonych lub złośliwych respondentów, wprowadzono dodatkowy mechanizm weryfikacji odpowiedzi. Raz na kilka/kilkanaście przykładów[1] osobie biorącej udział w badaniu przedstawiany jest już jednoznacznie sklasyfikowany przez innych przykład kodu (tzn. wcześniej pojawiły się 3 takie same głosy). Prezentacja tego przykładu nie różni się niczym w interfejsie użytkownika od takiego, który jeszcze nie został sklasyfikowany. Jeśli respondent podejmie inną decyzję niż poprzednie 3 osoby – otrzymuje komunikat o tym, że platforma stworzona do badania sprawdziła jego czujność i niestety test nie został zdany (zob. rysunek 5.4). W ostrzeżeniu jest również informacja o tym, że kolejna niezaliczona próba będzie skutkować wykluczeniem danego uczestnika z eksperymentu na 10 minut. Takie właśnie zachowanie zostało zaimplementowane.

Podobny komunikat otrzyma respondent, który dwa przykłady z rzędu przekroczy limit czasu przeznaczonego na odpowiedź. Do momentu potwierdzenia ostrzeżenia, eksperyment zostanie zatrzymany by przykłady nie klasyfikowały się samoczynnie. W tym przypadku nie są nakładane żadne kary czasowe. Wyświetlenie komunikatu ma na celu jedynie wstrzymanie wykonywania eksperymentu w momencie, gdy respondent bez zamknięcia okna przeglądarki z badaniem odszedł od stanowiska pracy.

5.2.2. Oszacowanie doświadczenia respondentów

Przy pierwszym udziale w eksperymencie, platforma wyświetla także formularz wymagający określenia swojego doświadczenia w programowaniu (zob. rysunek 5.2).

Osoba, która programuje mniej niż 1 rok, prawdopodobnie nie ma jeszcze wykształtowanej opinii na temat tego, co znaczy „czysty" kod. Z tego powodu wszystkie głosy od osób, które wybrały ten okres zostały zignorowane na etapie analizowania zebranych danych.

Osoba, która w formularzu zaznaczyła opcję *I'm not a coder* (z ang. *nie jestem programistą*), po kliknięciu przycisku *Begin!* rozpoczynającego eksperyment była przekierowywana do strony z prostą aplikacją do nauki programowania[2].

[1] odstęp pomiędzy weryfikacjami był zmienny; każda udzielona klasyfikacja podnosiła prawdopodobieństwo pojawienia się weryfikacji przy następnym pytaniu o 5%; po udzieleniu poprawnej klasyfikacji prawdopodobieństwo to malało do zera

[2] `https://www.google.com/doodles/celebrating-50-years-of-kids-coding` lub `https://fracz.github.io/phd/links/5.2.html`

Opcja *I'd rather not say* (z ang. *Nie chcę określać*) została dodana, by respondenci nie czuli się osaczeni przez platformę lub nie odnieśli wrażenia, że wumusza ona podanie zbyt wielu informacji. Głosy takich osób zostały wzięte pod uwagę.

Zebrane dane pozwoliły wyciągnąć charakterystykę osób biorących udział w badaniu. Została ona zaprezentowana na rysunku 5.5.

5.2.3. Próbki kodu poddane analizie w badaniu

Metody, które stworzyły zbiór ocenianych w ramach badania próbek, pochodziły ze zbioru danych wejściowych, który dał najlepszy rezultat przy uczeniu przed wprowadzeniem wiedzy ekspertowej (zob. sekcja 4.4). Dzięki temu w większości przypadków można było spodziewać się różnego poziomu jakości pomiędzy prezentowanymi przykładami oraz podobnej realizowanej funkcjonalności. Zbiór ten składał się z niemal 10 tys. przykładów refaktoryzacji. W celu klasyfikacji ich wszystkich – w najbardziej optymistycznym przypadku wymagałoby to zebrania 30 tys. zgodnych głosów.

Podejrzewano, że ten wynik jest niemożliwy do osiągnięcia, dlatego aby nie zostać po badaniu ze sporą liczbą przykładów sklasyfikowanych tylko przez jedną osobę – przykłady refaktoryzacji posortowano arbitralnie i przekazywano do respondentów przykład wylosowany tylko z pierwszych 100 metod, które nie zostały jeszcze sklasyfikowane. Po sklasyfikowaniu danego przykładu, został on odrzucany z puli i do „przesuwającego się okna" 100 metod dostępnych dla respondentów dochodziła kolejna. To pozwoliło założyć, że po oddaniu 300 głosów w zbiorze danych powinno być około 100 sklasyfikowanych metod, zamiast 300 przykładów z jednym głosem.

5.3. Platforma gromadząca opinie o jakości kodu

Platformę zaimplementowane zgodnie z projektem opisanym w sekcji 5.2. Wykorzystano do tego język PHP[3] ze wsparciem *frameworka* Slim[4]. Część frontendową aplikacji wykonano za pomocą Vue.js[5]. Dane zostały zebrane w bazie danych MariaDB[6]. Całość infrastruktury została skonteneryzowana za pomocą Dockera[7] i uruchomiona na prywatnym serwerze wirtualnym (VPS).

[3] `http://www.php.net`

[4] `https://www.slimframework.com`

[5] `https://vuejs.org`

[6] `https://mariadb.org`

[7] `https://www.docker.com`

Na rysunkach 5.1, 5.2, 5.3 oraz 5.4 przedstawiono kolejno: stronę główną zapraszającą do wzięcia udziału w eksperymencie, wstępną ankietę przed oddaniem pierwszego głosu, ekran prezentujący zmianę w kodzie poddawaną ocenie oraz komunikat o oddaniu niepoprawnego głosu dla przypadku weryfikującego poprawność odpowiedzi respondenta.

Interfejs platformy został opracowany w języku angielskim ze względu na udostępnienie platformy publicznie i zachęcanie do wzięcia udziału w eksperymencie również programistów spoza Polski (zob. sekcja 5.5). Ze względu na powszechność posługiwania się tym językiem wśród programistów, decyzja ta nie utrudniła skorzystania z platformy również polskojęzycznym respondentom.

Kod źródłowy platformy został umieszczony w repozytorium [96] pod nazwą *Code Assessor*. Platforma była udostępniona pod adresem `https://code.fracz.com`. Mimo zakończenia badania do celów rozprawy, nadal jest uruchomiona i przyjmuje głosy respondentów, co w przyszłości być może pozwoli na powiększenie jakościowego *benchmarku* kodu źródłowego oraz zbudowanie jakościowego modelu, który jeszcze trafniej będzie rozpoznawać czysty kod.

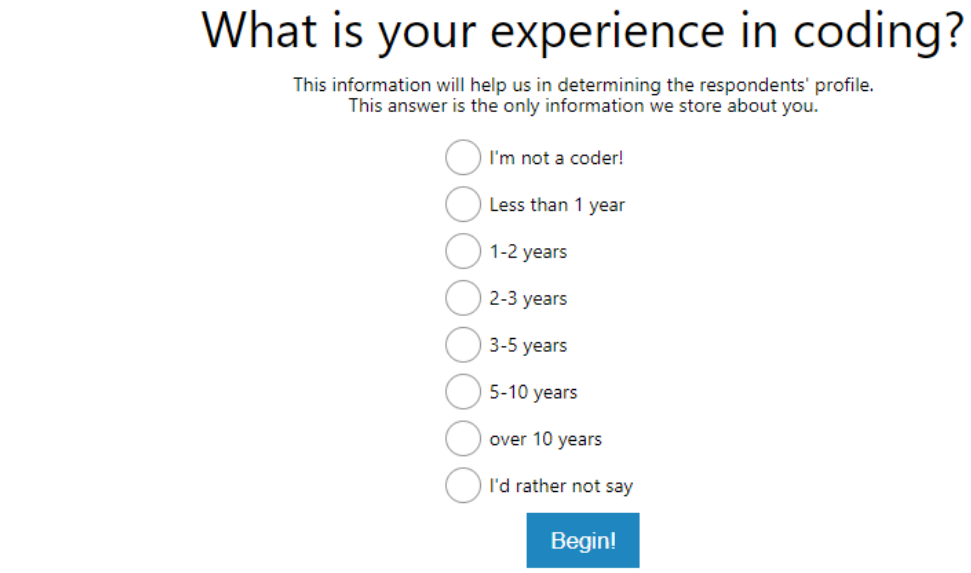
Ciekawym spostrzeżeniem była uwaga od kilku respondentów biorących udział w jednych z pierwszych sesji badania, że prezentowanie zmian w kodzie źródłowym z domyślnymi kolorami używanymi na ekranie porównywania zmiany (czerwony i zielony) zbytnio sugerowało, że kod podświetlony na kolor zielony jest kodem wyższej jakości. W rzeczywistości kolor czerwony oznacza kod usunięty, a zielony – dodany w ramach zmiany. Jednakże w celu ustosunkowania się do tej (słusznej) uwagi respondentów – kolory podświetlenia kodu na tym ekranie zostały zmienione na neutralne – niebieskie (zob. rysunek 5.3).

5.4. Przebieg eksperymentu

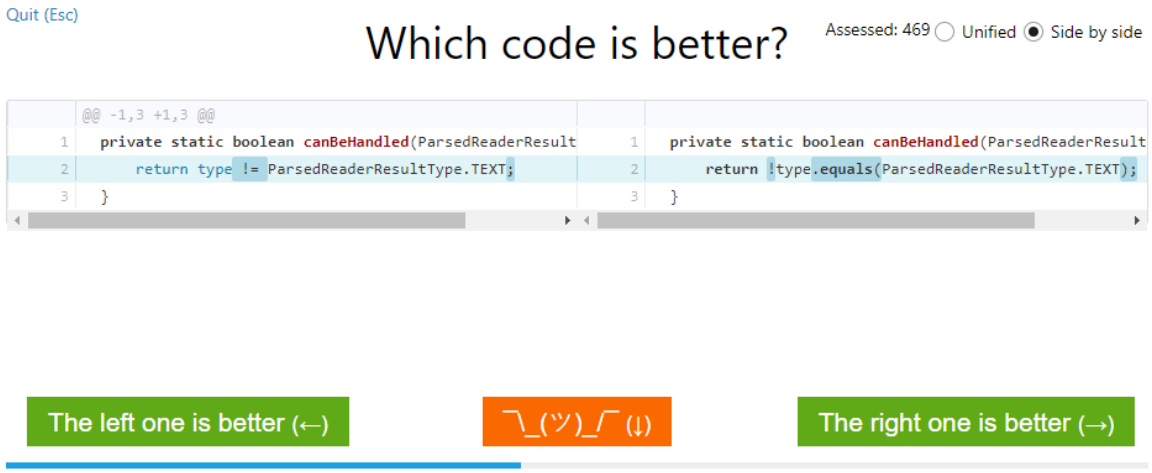
Dokładniejszego komentarza wymaga rysunek 5.3 z ekranem przedstawiającym analizowaną zmianę i pozwalającym na podjęcie decyzji na temat opinii o jakości kodu.

Na samej górze przez cały czas trwania eksperymentu wyświetlone jest pytanie, które mu przyświeca, tj. *Which code is better?*, czyli *Który kod jest lepszy?*. Sposób wyboru danych (metody ze zmian refkatoryzacyjnych) pozwala przypuszczać, że obydwie zaprezentowane metody realizują podobną odpowiedzialność, ale w nieco inny sposób. Celowo tutaj nie zadano pytania *Który kod jest wyższej jakości?*, ponieważ celem badania było poznanie ogólnej opinii programistów na temat tego, *który kod*

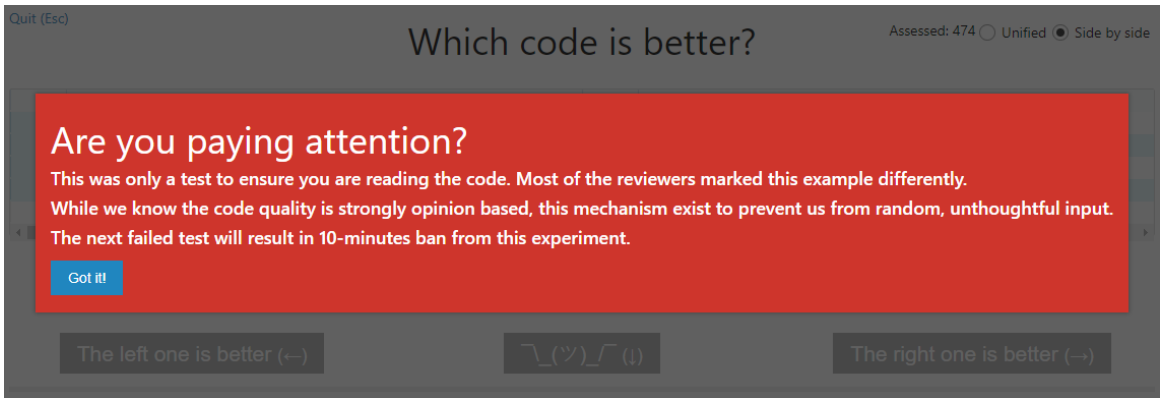
Rys. 5.1: Strona główna platformy stworzonej do zebrania opinii o jakości kodu źródłowego



Rys. 5.2: Ankieta wyświetlana respondentom badania opinii o jakości kodu przed oddaniem pierwszego głosu



Rys. 5.3: Ekran pozwalający respondentowi podjąć decyzję o jakości zaprezentowanej zmiany refaktoryzacyjnej



Rys. 5.4: Komunikat po wybraniu opinii dla przykładu weryfikującego niezgodnej z większością oddanych do tej pory głosów

wolą, lub – idąc dalej – *który kod woleliby zaakceptować w trakcie przeglądu kodu*. Podczas przeglądu kodu programista nie zastanawia się do końca, który kod ma wyższą jakość, ale który lepiej realizuje powierzone zadanie – który jest *lepszy*.

Poniżej pytania, główną część ekranu zajmuje zawsze porównywany kod źródłowy wyselekcjonowanych metod. Wykorzystano tutaj ten sam interfejs, jaki stosowany jest przy przeglądach kodu w portalu GitHub. Początkowo ekran porównania (ang. *diff*) używał standardowych kolorów spotykanych w platformach do przeglądu kodu źródłowego – kolorem czerwonym oznaczano kod który został usunięty w trakcie zmiany a kolorem zielonym – kod, który został dodany. W trakcie jednej z pierwszych sesji badania jeden ze studentów słusznie wskazał, że takie barwy mogą sugerować odpowiedź na zadawane pytanie (naturalnie zielony kolor jest raczej „lepszy"), dlatego zmieniono kolorystykę na niebieską tak, by po lewej i prawej stronie porównania była ona identyczna oraz nie sugerowała jakości kodu.

Warto tutaj zaznaczyć, że kod po lewej stronie nie zawsze był kodem przed refaktoryzacją a kod po prawej – po. Taka sytuacja jest naturalna w trakcie wykonywania przeglądu kodu. Pomimo że respondenci nie byli dokładnie zaznajomieni ze sposobem pozyskania prezentowanych metod, część z nich mogła założyć że kod po prawej stronie jest kodem po zmianie, więc jest lepszy. Z tego powodu opisywany ekran losowo przedstawiał kolejne zmiany, czasem prezentując je tak jak zostały pozyskane ze zmiany refaktoryzacyjnej, a czasem na odwrót – symulując niejako wycofywanie refaktoryzacji. Takie zachowanie minimalizuje ryzyko pozyskania szumu danych od osób, które mogłyby przyjąć regułę „zaznaczam zawsze prawy kod jako lepszy". W połączeniu z metodami próbującymi zidentyfikować osoby klasyfikujące przykłady metod bez zastanowienia (zob. sekcja 5.2) uzyskano platformę, która wymagała od respondentów rzeczywiście poprawnej klasyfikacji próbek.

Pod wyświetloną próbką kodu do oceny wyświetlono trzy przyciski. W lewym dolnym rogu umieszczono zielony przycisk *The left one is better*, *Kod po lewej jest lepszy*. W prawym dolnym rogu – analogicznie – zielony przycisk *The right one is better*, tj. *Kod po prawej jest lepszy*. Na środku znalazł się pomarańczowy przycisk z emotikonką *shruga*, dobrze znanej w środowisku programistów (¯_(ツ)_/¯), oznaczającą „nie wiem", „nie mam pojęcia" lub „nie mam zdania". Po kliknięciu przycisku, głos respondenta jest zapisywany w bazie danych, a ekran porównania pokazuje kolejną próbkę.

Niebieski pasek pod przyciskami „napełniał się" w miarę upływu czasu, prezentując w ten sposób ustalony 60-sekundowy limit oczekiwania na udzielenie odpowiedzi. Gdy limit został osiągnięty, zmiana była klasyfikowana przy użyciu środkowego przycisku (nawet jeśli nie został on wybrany) i respondent otrzymywał kolejną próbkę do oceny. Zgodnie z zaprojektowanymi zasadami opisanymi w sekcji 5.2, przez pierwsze 5 sekund respondent nie może dokonać klasyfikacji, co w interfejsie użytkownika jest zaznaczone wyszarzeniem przycisków tak, by wyglądały one na nieaktywne.

W prawym górym rogu użytkownik może zdecydować o sposobie wyświetlania porównania kodu. Może to być sposób *Side by side* (wybrany domyślnie), prezentujący dwie metody jednocześnie po lewej i prawej stronie ekranu. Drugi sposób to *Unified*, prezentujący jeden kod źródłowy, a usunięte i dodane linie są oznaczone odpowiednio przez znaki − i + (ten sposób wyświetlania zmiany jest znany choćby z linii poleceń i komendy `git diff`). Dzięki temu respondent może wybrać widok zmiany do którego jest przyzwyczajony i maksymalnie skupić się na prezentowanych przykładach a nie na ich formie. Obok przełącznika widoku ekran przedstawia liczbę sklasyfikowanych dotąd metod.

Naśladując styl używania IDE przez programistów oraz by dostarczyć łatwiejsze w użytkowaniu narzędzie, opisywaną funkcjonalnością można sterować także przy użyciu klawiatury. Wciskając strzałkę w lewo, w prawo bądź na dół, można sklasyfikować zaprezentowaną próbkę odpowiednio jako „lewy kod jest lepszy", „prawy kod jest lepszy" lub „nie wiem". Klawisz *Esc* przerywa eksperyment i przenosi użytkownika do strony wprowadzającej do badania (rysunek 5.1). Podpowiedzi sugerujące możliwość wykorzystania omówionych skrótów klawiaturowych zostały zaprezentowane na ekranie umożliwiającym klasyfikację przykładu bezpośrednio na przyciskach wykonujących poszczególne akcje.

5.5. Czas trwania eksperymentu i respondenci

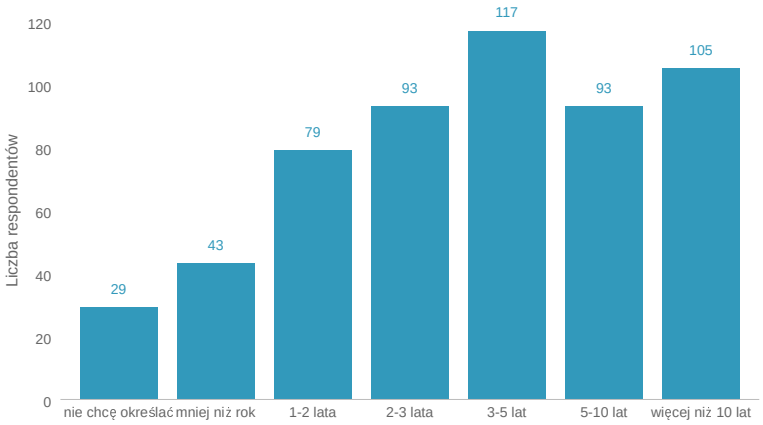
Badanie trwało 6 miesięcy (styczeń – czerwiec 2018). Głównymi respondentami byli studenci i absolwenci studiów Informatyki na wydziale Informatyki, Elektroniki i Telekomunikacji, AGH. W trakcie zajęć z przedmiotów Technologie Obiektowe oraz Inżynieria Oprogramowania poświęcono 10-15 minut na udział w badaniu. Tematem laboratoriów przeprowadzanych w ramach tych przedmiotów była jakość kodu, dlatego taka aktywność doskonale wpisywała się w tok zajęć. Studenci przed badaniem zostali poinformowani o celu gromadzenia danych oraz zostali wprowadzeni do obsługi platformy w formie krótkiej prezentacji. W swoim toku nauczania odbyli już kursy dotyczące programowania zorientowanego obiektowo oraz wzorców projektowych i refaktoryzacji. Dzięki temu można było oczekiwać, że mieli już wykształcone własne opinie na temat jakości i czytelności kodu źródłowego, co czyniło ich idealnymi kandydatami do wzięcia udziału w badaniu.

Dodatkowo, część klasyfikacji udało się zebrać od użytkowników platformy z zadaniami do nauki systemu kontroli wersji Git – `https://gitexerecises.fracz.com`. Jest to jeden z wcześniejszych projektów doktoranta [97], który zyskał sporą popularność w Internecie tak, że witryna notuje co najmniej kilkadziesiąt unikalnych odwiedzin dziennie. Ze względu na naturę aplikacji głównymi odbiorcami są programiści, dlatego wstawienie propozycji pomocy autorowi platformy w jego kolejnym projekcie polegającej na ocenie kilku przykładów refaktoryzacji wydawało się dobrym pomysłem – i rzeczywiście tak było.

Wykres na rysunku 5.5 zawiera zebrane statystyki przedstawiające doświadczenie respondentów na podstawie ankiety prezentowanej przed przystąpieniem do badania. Większość osób posiadała co najmniej dwuletnie doświadczenie w programowaniu. Tabela 5.1 przedstawia natomiast statystyki opinii zebranych za pomocą opisanej platformy.

Tabela 5.1: Charakterystyka respondentów biorących udział w badaniu opinii na temat jakości kodu źródłowego

Liczba respondentów	559
Liczba klasyfikacji (głosów)	5263
Średnia liczba głosów / osobę	9.4
Czas konieczny na podjęcie decyzji (średnia)	20s



Rys. 5.5: Doświadczenie respondentów, którzy wzięli udział w badaniu opinii o jakości kodu źródłowego

5.6. Jakościowy *benchmark* kodu źródłowego

Ze wszystkich metod sklasyfikowanych przez programistów w ramach eksperymentu utworzono *benchmark*, tj. zestaw przypadków testowych umożliwiających przetestowanie skuteczności dowolnej metody automatycznie oceniającej jakoś kodu źródłowego w języku Java. Dotąd w literaturze nie odnotowano zbudowania podobnych zestawów danych dla problemu jakości kodu źródłowego.

W trakcie pracy nad jakościowym modelem kodu źródłowego, przykłady z tego zbioru zostały podzielone na dwie części. Pierwsza została użyta w trakcie trenowania modelu. Dzięki temu do SCQM zostały przekazane rzeczywiste opinie programistów o jakości kodu źródłowego. Druga część została użyta w trakcie ewaluacji, by zmaksymalizować wiarygodność analizy porównawczej.

Benchmark został umieszczony w repozytorium [98]. Zgodnie z informacjami w pliku `README.md`, katalog `src` zawiera 645 przykładów metod o lepszej i gorszej jakości, co w sumie daje 1290 przykładów. Aby kod wewnątrz tych plików był poprawny syntaktycznie, każda sklasyfikowana metoda została opakowana w deklarację klasy, której nazwa zawiera liczebnik porządkowy przykładu oraz suffix `Better` dla kodu o wyższej jakości i `Worse` – dla kodu o jakości niższej. Ponadto, w komentarzu nad klasą umieszczono informację z której próbki z danych wejściowych pochodzi dany przykład. Wpisano tam także słowo `before` lub `after` w zależności od tego, czy dany kod w oryginalnej zmianie był kodem przed czy po zmianie refaktoryzacyjnej.

Listing 5.1 zawiera przykład klasy pochodzącej z omawianego *benchmarku*.

```
// original filename: 00005779.txt
// after
public class Class00000010Better {
    public VirtualFile getProjectFile() {
        if (myProjectFile == null)
            return null;
        return myProjectFile.getVirtualFile();
    }
}
```

Listing 5.1: Przykładowa klasa wchodząca w skład jakościowego *benchmarku* kodu źródłowego

5.7. Podsumowanie

Charakterystyka zebranych odpowiedzi została przedstawiona na wspomnianej już wcześniej tabeli 5.1. Natomiast liczba sklasyfikowanych poszczególnych przykładów została przedstawiona w tabeli 5.2.

Tabela 5.2: Uzyskane klasyfikacje w badaniu opinii o jakości kodu

Kod przed zmianą jest lepszy	180
Kod po zmianie jest lepszy	465
Zmiana nie wpływa na jakość kodu	439

Ciekawym spostrzeżeniem jest fakt, że znaczna część zmian refaktoryzacyjnych została sklasyfikowana jako „anty-refaktoryzacje". Niemal połowa z przeanalizowanych przez uczestników badania przykładów została odrzucona jako niezmieniające jakości. Większość z nich – zgodnie z oczekiwaniami – została sklasyfikowana jako „kod po zmianie jest lepszy".

Dzięki badaniu uzyskano zbiór **645 metod, których jakość została sklasyfikowana przez programistów**. Za ich pomocą został stworzony wzorcowy zbiór treningowy oraz testowy, który następnie wykorzystano w trakcie uczenia i ewaluacji modelu SCQM.

Rozdział 6

Uczenie SCQM

Zgromadzony zbiór danych wejściowych złożony z próbek kodu źródłowego o różnej jakości pozwolił na rozpoczęcie prac nad samym modelem. W tym rozdziale opisano projekt i implementację jakościowego modelu źródłowego opartego o dwukierunkową rekurencyjną sieć neuronową oraz przybliżono przebieg jej trenowania, wraz z przedstawieniem najlepszego rezultatu na zbiorze testowym.

6.1. Implementacja sieci neuronowej

W sekcji 3.3.4 przedstawiono motywację stojącą za wyborem modelu opartego o dwukierunkową rekurencyjną sieć neuronową z komórkami LSTM.

W celu dostarczenia implementacji sieci neuronowej wykorzystano *framework* TensorFlow [99]. Jest to implementacja różnych metod uczenia maszynowego w języku Python[1] na wystarczająco wysokim poziomie abstrakcji, by osoba niebędąca specjalistą w tej dziedzinie mogła wykorzystać ich potencjał.

Kod źródłowy dla modeli aSCQM oraz rSCQM znajduje sie odpowiednio w plikach `model2.py` oraz `model4.py` w repozytorium [95]. Szczegóły przedstawionych implementacji zostały umieszczone w dodatku C.

Obydwa skrypty pozwalają na uruchomienie ich w dwóch trybach – trenowania oraz klasyfikacji. W trakcie trenowania możliwe jest przekazanie do skryptu zadanego zbioru danych treningowych, pożądanej liczby iteracji, liczby ukrytych jednostek LSTM oraz innych parametrów pracy sieci neuronowej. Jeżeli skrypt uruchamiany jest w trybie klasyfikacji, wymaga on podania ścieżki do wytrenowanego wcześniej modelu.

[1] `https://www.python.org`

Na początku każdej implementacji znajduje się kod odpowiedzialny za wczytanie wejścia sieci neuronowej. Dane te, w zależności od skryptu, posiadają opisany wcześniej format (zob. sekcja 4.5). Następnie budowane są struktury dwukierunkowej rekurencyjnej sieci neuronowej za pomocą konstrukcji dostarczanych przez narzędzie TensorFlow. W końcowej części skryptu uruchamiana jest sesja TensorFlow, która wykorzystuje sieć neuronową budując, lub wykorzystując zbudowany wcześniej jakościowy model kodu źródłowego.

6.2. Rozmiar zbioru testowego

Obie implementacje modelu – względna i bezwzględna – każdy zbiór sekwencji wejściowych dzielą w stosunku 85% : 15%. Większy zbiór stawał się danymi uczącymi (treningowymi). Drugi zaś – danymi testowymi, które nie brały udziału w uczeniu.

Wszystkie trafności przedstawione w rozprawie zostały oparte o wyniki dla zbioru testowego.

6.3. Wykorzystane zasoby obliczeniowe

Uczenie jakościowego modelu kodu źródłowego na podstawie zebranych danych nie byłoby możliwe, gdyby nie zasoby obliczeniowe posiadane przez AGH. Teoretycznie model mógłby być obliczony na sprzęcie domowym (według szacunków trwałoby to około 3-4 tygodni). Jednakże mnogość prób i różnych konfiguracji, które zostały podjęte podczas uczenia (por. np. tabele 6.1 i 6.2) musiałaby być w znaczny sposób ograniczona, co z pewnością odbiłoby się na skuteczności uzyskanego modelu jakościowego.

Dzięki superkomputerowi Prometheus AGH udostępnianemu przez Cyfronet i portal PLGrid[2] oraz wsparciu GPU możliwe było skrócenie czasu obliczeń jednej konfiguracji do około 8 godzin dla modelu aSCQM oraz około 12 godzin dla modelu rSCQM (czasy były oczywiście zależne od zadanych parametrów modelu oraz rozmiaru danych wejściowych). Taki czas oczekiwania na wyniki umożliwił eksperymentowanie z konfiguracją sieci neuronowej i uzyskanie najlepszych możliwych wyników.

W repozytorium [95] można odnaleźć skrypty używane do zlecania zadań obliczeń w trybie wsadowym (pliki `batch*.sh`). Wykorzystują one parametry skryptów uczących poszczególne modele (zob. dodatek C oraz listingi C.1 i C.2). Ponadto, poza

[2] `http://www.plgrid.pl`

konfiguracją samego zadania wynikającą z dostarczonej dokumentacji [100], można z nich także odczytać użyte w trakcie obliczeń dodatkowe moduły. Są to:

1. `plgrid/apps/cuda/8.0`[3] – dostarcza biblioteki umożliwiające wykorzystanie procesorów graficznych GPU na klastrze Prometheus w architekturze CUDA[4]

2. `plgrid/tools/python/3.6.0`[5] – umożliwia korzystanie z języka Python, w którym zaimplementowany jest użyty *framework* TensorFlow

Po uprzedniej konfiguracji odpowiednich nazw katalogów zawierających dane użyte do kolejnych etapów uczenia modelu wewnątrz skryptów wsadowych (wszystkie dane wejściowe wraz z uzyskanymi wynikami są zamieszczone w repozytorium [95]), zadania były zlecane przy użyciu polecenia `sbatch`. Dzięki trybowi wsadowemu możliwe było obliczanie kilku konfiguracji jednocześnie, a przy długo trwających obliczeniach możliwe było opuszczenie maszyny i powrócenie do wyników uczenia w późniejszym czasie. Ze szczegółowymi informacjami na temat wspomnianej komendy można zapoznać się bezpośrednio z dokumentacji klastra [100].

Każda sesja uczenia pozostawiała za sobą dwa artefakty:

1. Plik z rozszerzeniem `.log` zawierający informacje o trafności i błędzie podczas uczenia (na ich podstawie zaprezentowano w rozprawie wykresy przebiegów uczenia w kolejnych próbach). Wszystkie pliki z logami zostały umieszczone w katalogu `logs` w repozytorium [95].

2. Zapisany stan sieci neuronowej na zakończenie uczenia. Pliki te umożliwiają późniejsze zaimportowanie nauczonego modelu i użycie go do sklasyfikowania kolejnych przykładów lub dalszego douczenia. Funkcjonalność ta zostanie wykorzystana przy ewaluacji stworzonego rozwiązania (zob. rozdział 7). Ze względu na duży rozmiar plików, w repozytorium nie udostępniono wszystkich wytrenowanych modeli. W repozytorium [101] znajduje się model, który dał najlepsze rezultaty.

[3] `https://apps.plgrid.pl/module/plgrid/apps/cuda/8.0.61`

[4] `https://www.nvidia.pl/object/cuda-parallel-computing-pl.html`

[5] `https://apps.plgrid.pl/module/plgrid/tools/python`

6.4. Trenowanie za pomocą danych zgromadzonych automatycznie

Próbki kodu źródłowego metod klas zgromadzone i przygotowane zgodnie z opisem w rozdziale 4 zostały już wstępnie przefiltrowane. Nałożone ograniczenia mające na celu usunięcie szumu z danych treningowych (zob. sekcja 4.4) pozwoliły przypuszczać, że posiadane rozbiory syntaktyczne metod zawierają całkiem sensowne refaktoryzacje.

Na tym etapie wykonano szereg prób uczenia modelu, manipulując sposobem filtrowania danych wejściowych oraz parametrami sieci neuronowej. Jedynym parametrem, który spowodował, że odnotowano zmiany w wynikach uczenia (przy użyciu tych samych danych), była liczba ukrytych jednostek LSTM. Liczba ta w teorii reprezentuje liczbę aspektów, które model jest w stanie wykryć w analizowanych danych. Ile jest aspektów, po których można rozpoznać kod wysokiej jakości? Początkowo wybrano arbitralnie liczbę 128, jednak próba uczenia z większą liczbą ukrytych jednostek pokazała, że model zachowuje się lepiej po zwiększeniu tego parametru.

Tabele 6.1 i 6.2 przedstawiają końcową trafność dla zbioru testowego osiąganą po 50 tys. iteracji dla modelu bezwzględnego i względnego dla poszczególnych prób kombinacji parametrów i metody filtrowania danych treningowych.

Tabela 6.1: Trafność modelu dla zbioru testowego przy różnych konfiguracjach po 50 tys. iteracji – model bezwzględny aSCQM

Maks. długość metody	**Min. liczba zmienionych tokenów**	**Maks. liczba zmienionych tokenów**	**Liczba jednostek LSTM**	**Trafność**
300	-	-	128	53%
200	10	-	128	54%
100	10	-	128	50%
100	10	-	256	53%
100	10	50	256	56%
200	10	50	256	54%
200	10	100	256	**56%**

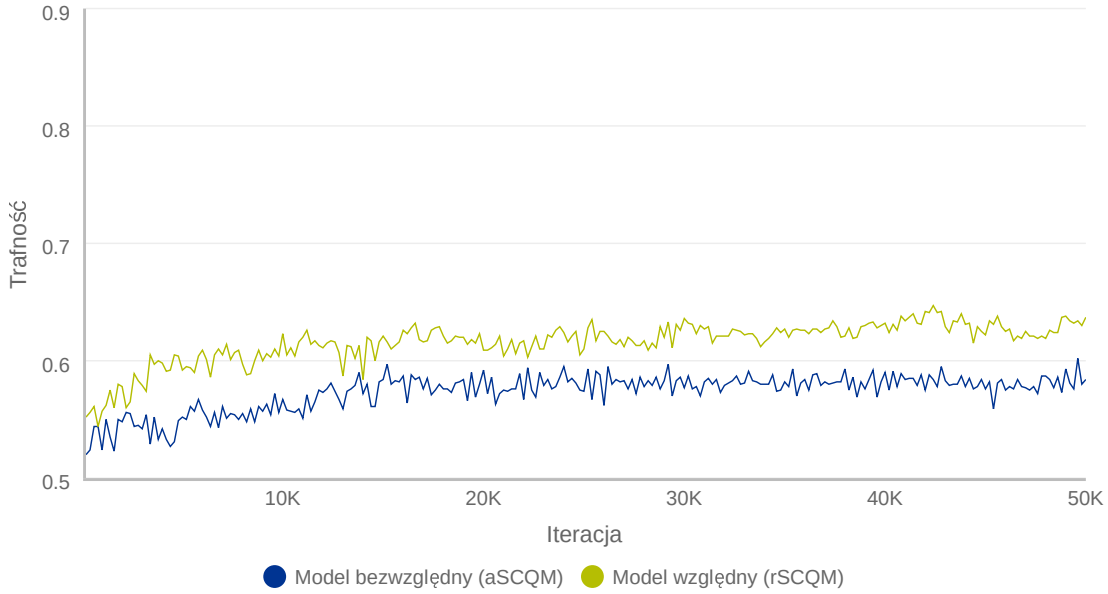
Rezultat uczenia modelu za pomocą danych zgromadzonych i przygotowanych zgodnie z opisem w rozdziale 4 i dla najlepszej uzyskanej konfiguracji parametrów sieci neuronowej przedstawiono na rysunku 6.1.

Tabela 6.2: Trafność modelu dla zbioru testowego przy różnych konfiguracjach po 50 tys. iteracji – model względny rSCQM

Maks. długość metody	**Min. liczba zmienionych tokenów**	**Maks. liczba zmienionych tokenów**	**Liczba jednostek LSTM**	**Trafność**
300	-	-	128	53%
200	10	-	128	54%
100	10	-	128	58%
100	10	-	256	59%
100	10	-	512	58%
100	5	100	256	59%
100	5	100	256	60%
100	10	50	256	62%
200	10	50	256	55%
200	10	100	256	63%
200	5	100	256	**64%**
300	5	100	256	62%

Uzyskano trafność modelu względnego rSCQM na poziomie 64% i bezwzględnego aSCQM na poziomie 56%. Model prezentuje wyższą trafność niż losowy klasyfikator, jednakże wyniki na tym etapie nie są jeszcze zadowalające. Znaczna różnica pomiędzy osiąganymi trafnościami modelu dla różnych metod filtrowania danych treningowych pokazuje, że w pozyskanych automatycznie próbkach rzeczywiście znajduje się sporo szumu danych.

Przed kolejnymi próbami uczenia postawiono także sprawdzić, czy implementacja sieci neuronowej jest zbudowana poprawnie i czy jest w stanie przyjąć jakąkolwiek wiedzę z tak przygotowanych danych treningowych. Przygotowano trzy proste metryki klasyfikowania jakości kodu i z powodzeniem podjęto próbę przekazania wiedzy o nich do jakościowego modelu kodu źródłowego. Szczegóły tej ewaluacji zamieszczono w dodatku D.



Rys. 6.1: Przebieg trenowania modelu za pomocą danych zgromadzonych automatycznie

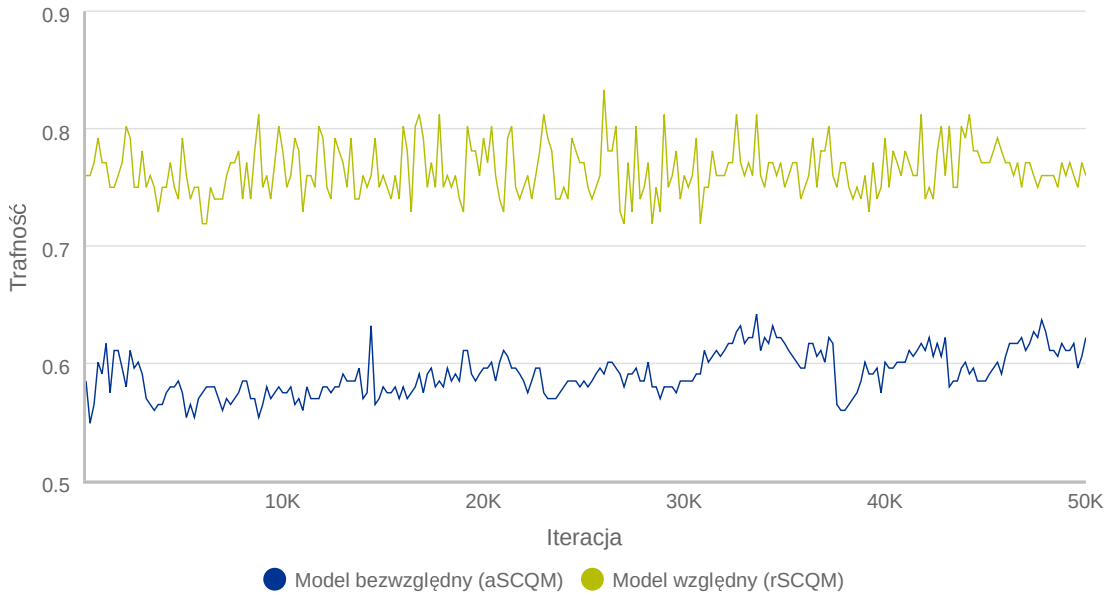
6.5. Trenowanie za pomocą próbek sklasyfikowanych przez programistów

Trafność 64% w rozpoznawaniu zmiany podnoszącej jakość kodu źródłowego dla modelu rSCQM jest lepsza niż trafność losowej klasyfikacji. Niemniej jednak, nie można jeszcze na tym etapie mówić o sukcesie prezentowanego modelu. Kolejnym krokiem usprawnienia modelu było więc wykorzystanie wiedzy zebranej przy użyciu badania opisanego w rozdziale 5.

Ważną korzyścią płynącą z faktu posiadania zbioru poprawnie sklasyfikowanych metod i ich jakości była możliwość zbudowania pewnego zbioru danych testowych nieposiadających szumu. Trafność obliczana na jego podstawie może być rzeczywiście traktowana jako podobieństwo działania zaprojektowanego modelu jakościowego do decyzji podejmowanych przez programistę w trakcie przeglądu. Zbiór testowy – podobnie jak poprzednio – stworzono wybierając losowo 15% próbek z pozyskanych danych, co dało 96 metod ocenionych przez programistów, które nie zostały wzięte pod uwagę w dalszym uczeniu (por. Tabela 5.2). Również model nauczony w oparciu o reguły z sekcji 4.4 wykazuje dla tak skonstruowanego zbioru testowego wyższą trafność (zob. trafność osiąganą w początkowych iteracjach na rysunku 6.3).

Przed stworzeniem modelu w oparciu o obydwa zestawy danych (to podejście opisano w sekcji 6.6), pierwszą próbą było nauczenie modelu tylko przy użyciu zmian sklasyfikowanych przez programistów jako zmieniających jakość kodu. Tak zbudowany model powinien rzeczywiście posiąść przekazaną w platformie wiedzę na temat jakości kodu.

Przebieg uczenia zaprezentowano na rysunku 6.2. Model bezwzględny aSCQM nie poprawił się znacząco w stosunku do poprzedniego wyniku po dokładniejszym przefiltrowaniu danych wejściowych. Zdecydowaną poprawę widać jednak w modelu względnym rSCQM. Dla tak zadanych danych wejściowych osiąga on trafność rozpoznawania zmiany poprawiającej jakość kodu na poziomie 78%.



Rys. 6.2: Przebieg uczenia modelu przy danych treningowych zbudowanych na podstawie metod sklasyfikowanych przez programistów

6.6. Najlepszy uzyskany rezultat trenowania modelu

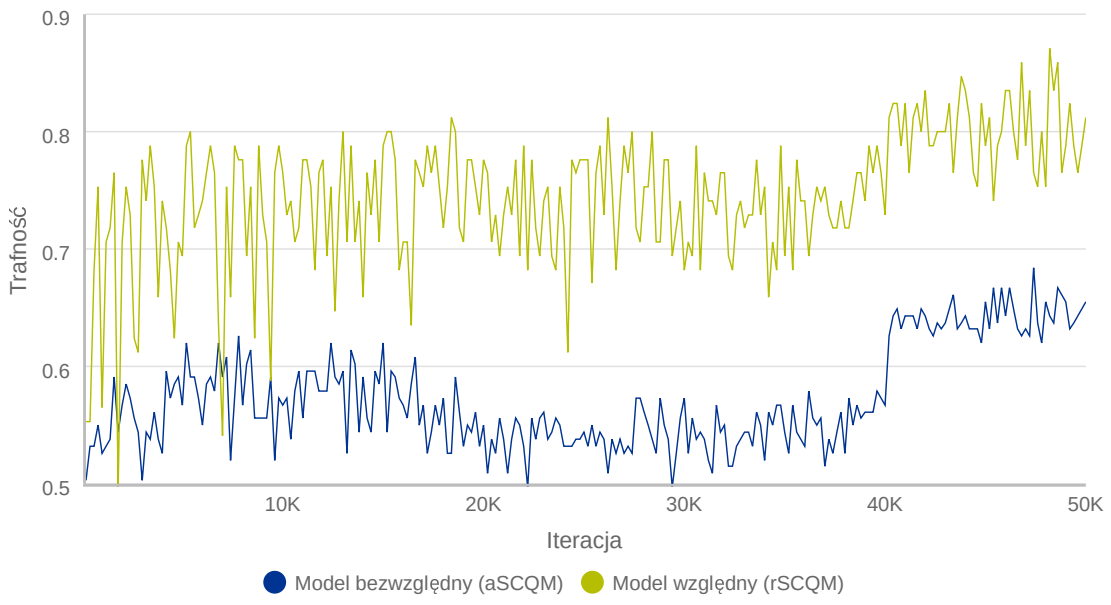
Ze względu na niewielką liczbę metod sklasyfikowanych przez programistów (zob. Tabela 5.2) sieć neuronowa zbyt dobrze dopasowywała się do danych treningowych. Przeuczanie się modelu widać także po szybkim osiągnięciu wysokiej trafności dla modelu rSCQM. To spostrzeżenie doprowadziło do pomysłu połączenia rozwiązań opisanych w poprzednich sekcjach 4.4 oraz 6.5. W tym wariancie sieć neuronowa jest

uczona początkowo za pomocą danych treningowych, które nie zostały sklasyfikowane przez programistów ze względu na ograniczenia eksperymentu.

Pozostając przy dotychczasowym limicie 50 tys. iteracji uczenia ustalono, że model będzie uczony przez 80% z nich (40 tys.) za pomocą rozbiorów syntaktyczynch metod pozyskanych za pomocą ustalonych reguł opisanych w sekcji 4.4. Następnie, przez kolejne 20% iteracji (10 tys.), sieć neuronowa zostanie „douczona" poprawnie sklasyfikowanymi metodami przez programistów.

Takie podejście powinno pozostawić w modelu wiedzę o jakości kodu źródłowego wyciągniętą bezpośrednio z otwartych projektów oraz doprecyzować ją za pomocą pozyskanej opinii programistów o jakości kodu źródłowego.

Przebieg uczenia zaprezentowano na rysunku 6.3. W momencie wprowadzenia do danych treningowych próbek sklasyfikowanych przez programistów następuje zauważalny wzrost trafności klasyfikacji modelu.



Rys. 6.3: Przebieg uczenia modelu przy danych treningowych zbudowanych na podstawie metod wybranych automatycznie (40 tys. iteracji) oraz metod sklasyfikowanych przez programistów (10 tys. kolejnych iteracji)

6.7. Podsumowanie

Kolejne próby uczenia zaprojektowanego jakościowego modelu kodu źródłowego opisane w tym rozdziale pozwoliły na uzyskanie zadowalających rezultatów, a w konsekwencji na kontynuowanie prac nad rozprawą. Proces ten był najbardziej czasochłonną czynnością w ramach rozprawy ze względu na czas oczekiwania na uzyskanie kolejnych wyników oraz mnogość możliwych rozwiązań.

Na tym etapie uzyskano **trafność dla modelu aSCQM na poziomie 65%** oraz **trafność dla modelu rSCQM na poziomie 80%**. Uznano, że taka trafność uzyskana dla metod sklasyfikowanych w ramach platformy jest wystarczająca, by podjąć próbę porównania stworzonego modelu z istniejącymi narzędziami oceniającymi kod, opartymi o statyczną analizę.

Rozdział 7

Ewaluacja SCQM

Trafność uzyskana w procesie uczenia modelu SCQM opisana w poprzednim rozdziale skłoniła autora rozprawy do podjęcia próby porównania skuteczności stworzonego modelu i dostępnych obecnie metod, które do oceny jego jakości wykorzystują statyczną analizę. W tym celu wybrano istniejące jakościowe metryki kodu źródłowego oraz popularne narzędzie klasyfikujące kod pod względem jego jakości: Checkstyle. Porównano ich skuteczność ze stworzonym modelem, a wyniki tego porównania zostały opisane w tym rozdziale. W ramach ewaluacji wykorzystano wybrane metody z jakościowego *benchmarku* kodu źródłowego stworzonego w ramach pracy nad rozprawą.

7.1. Narzędzia i metryki poddane analizie porównawczej

Teza rozprawy zakłada, że stworzony jakościowy model kodu źródłowego wykaże większą trafność w rozpoznawaniu kodu niskiej jakości niż dostępne obecnie narzędzia wykorzystujące jego statyczną analizę. Jednym z najpopularniejszych narzędzi powszechnie używanych do wykrywania zapachów w kodzie źródłowym w języku Java jest Checkstyle[1]. Porównanie rezultatów osiąganych przez to narzędzie oraz przez model SCQM jest więc niezbędne do stwierdzenia użyteczności powstałego rozwiązania w środowisku produkcyjnym.

[1] `http://checkstyle.sourceforge.net/`

Oprócz porównania modelu SCQM do wspomnianego narzędzia, wartościowe będzie także porównanie klasyfikacji otrzymywanych z nowego modelu do wartości jakościowych metryk kodu źródłowego, które są często wykorzystywane do automatycznego określania jakości kodu źródłowego. Zgodnie z opisem w sekcji 2.4 najlepszą skuteczność w wykrywaniu kodu źródłowego niskiej jakości ma złożoność cyklomatyczna. Nie można więc było jej pominąć w ewaluacji stworzonego modelu, by wykazać, że jest on rzeczywiście lepszy od istniejących rozwiązań.

Podobną metryką jest złożoność NPath (ang. *NPath complexity*). Jest to metryka, która określa możliwą liczbę różnych, acyklicznych ścieżek przejścia przez dany kod źrodłowy [102]. Podobnie jak przy złożoności cyklomatycznej, wyższa wartość metryki oznacza bardziej skomplikowany kod.

Kolejną metryką, do której autor rozprawy postanowił porównać predykcje SCQM jest metryka NCSS (ang. *Non Commenting Source Statements*, wyrażenia w kodzie źródłowym niebędące komentarzami). Metryka ta zlicza wyrażenia w kodzie źródłowym, które nie są komentarzami ani pustymi blokami kodu. Jej wartość więc można rozumieć jako liczbę operacji wykonywanych w zadanym kodzie źródłowym. Jak zasugerowano w [103], NCSS jest nieco bardziej wyszukanym określeniem rozmiaru kodu aniżeli liczba jego linii. Pomimo prostoty tej metryki, NCSS jest często zawierana w domyślnych konfiguracjach sprawdzeń dla narzędzia Checkstyle – a co za tym idzie – często jej wyniki są brane pod uwagę w środowiskach produkcyjnych. Podobnie jak w poprzednich metrykach – wyższa wartość oznacza kod niższej jakości.

Próbki kodu składające się na zbiór walidacyjny zostały poddanie ocenie przed wszystkie porównywane narzędzia i metryki. W dalszej części tej sekcji opisano parametry uruchomienia poszczególnych narzędzi.

7.1.1. Checkstyle

Do ewaluacji pobrano najnowszą w momencie tworzenia rozprawy wersję Checkstyle, tj. 8.12. Narzędzie to zostało pobrane w formie skompilowanego pliku `.jar`.

Checkstyle pozwala na konfigurację różnorakich sprawdzeń, którym ma zostać poddany kod źródłowy. Konfiguracji tej dokonuje się za pomocą pliku w formacie XML. W zależności od preferencji programistów w zespole można wybrać reguły, które mają być utrzymywane w projekcie oraz dodatkowo je skonfigurować, ustalając np. maksymalną długość linii kodu metody, sposób formatowania klamer w kodzie itp.

Aby zminimalizować ryzyko subiektywnego wyboru jakościowych reguł kodu, postanowiono skorzystać z domyślnych reguł udostępnianych przez Checkstyle: Sun

Code Conventions[2]. Jest to często wyjściowa, a czasem ostateczna postać konfiguracji Checkstyle w wielu projektach.

Jedyną modyfikacją wprowadzoną do wspomnianych reguł było usunięcie konfiguracji dotyczących formatowania kodu oraz usunięcie wymagania istnienia komentarzy nad klasami i metodami w kodzie źródłowym. Warunki te powinny być zapewniane przez stosowane środowisko programistyczne oraz serwery ciągłej integracji. Kod zawierający błędne formatowanie nigdy nie powinien dotrzeć do etapu przeglądu kodu, dlatego też stwierdzono że z punktu widzenia jakościowego modelu pozostawienie tych sprawdzeń wygenerowałoby zbyt wiele błędów, wynikających jedynie ze sposobu przygotowania danych walidacyjnych.

Listing 7.1 zawiera pełną listę reguł usuniętych z oficjalnego pliku `sun_checks.xml` przed przeprowadzeniem klasyfikacji.

```
<module name="JavadocPackage"/>
<module name="NewlineAtEndOfFile"/>
<module name="JavadocMethod"/>
<module name="JavadocType"/>
<module name="JavadocVariable"/>
<module name="JavadocStyle"/>
<module name="LineLength"/>
```

Listing 7.1: Reguły usunięte z Sun Code Conventions przed klasyfikacją kodu za pomocą Checkstyle

Listing 7.2 przedstawia komendę, którą uruchomiono w głównym katalogu projektu [98] w celu uzyskania klasyfikacji. W wyniku Checkstyle przedstawia szczegóły naruszeń reguł odnalezione w zadanym pliku wejściowym oraz ich sumaryczną liczbę na końcu raportu. Ta liczba właśnie została potraktowana jako wartość metryki pochodzącej z Checkstyle. W poniższym przypadku jest ona równa 1.

```
$ java -jar checkstyle-8.12-all.jar -c sun_checks_scqm.xml src/
    ↪ Class00000160Worse.java
[ERROR] D:\projects\code-quality-benchmark\scqm\src\Class00000160Worse
    ↪ .java:8:34: '33' is a~magic number. [MagicNumber]
Audit done.
Checkstyle ends with 1 error.
```

Listing 7.2: Przykład klasyfikacji klasy za pomocą Checkstyle

[2] `https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/sun_checks.xml` lub `https://fracz.github.io/phd/links/7.2.html`

7.1.2. Złożoność cyklomatyczna

Złożoność cyklomatyczna została obliczona również za pomocą narzędzia Checkstyle. W tym celu stworzono plik konfiguracyjny dla Checkstyle zawierający tylko jedną regułę obliczającą złożoność cyklomatyczną przedstawionego kodu (zob. listing 7.3). Ustalenie wartości `max` na -1 pozwoliło na zgłaszanie dowolnej złożoności cyklomatycznej jako naruszenie ustalonej reguły, co z kolei umożliwiło wykorzystanie tego narzędzia do obliczenia złożoności cyklomatycznej dla każdej próbki ze zbioru walidacyjnego. Przykładową komendę obliczającą tę metrykę przedstawiono na listingu 7.4. Należy zaznaczyć, że w tym wypadku ignorowano sumaryczną liczbę naruszeń, a jako wartość metryki uznawano liczbę zwróconą przez wskazaną komendę w wyrażeniu *Cyclomatic Complexity is X*.

```
<name="CyclomaticComplexity">
    <property name="max" value="-1"/>
</module>
```

Listing 7.3: Konfiguracja Checkstyle dla obliczania złożoności cyklomatycznej

```
$ java -jar checkstyle-8.12-all.jar -c cyclomatic.xml src/
    ↪ Class00000160Worse.java
Starting audit...
[ERROR] D:\projects\code-quality-benchmark\scqm\checkstyle\src\
    ↪ Class00000160Worse.java:4:5: Cyclomatic Complexity is 1 (max
    ↪ allowed is -1). [CyclomaticComplexity]
Audit done.
Checkstyle ends with 1 error.
```

Listing 7.4: Przykład uzyskania wartości złożoności cyklomatycznej za pomocą Checkstyle

7.1.3. Złożoność NPath

Analogicznie jak w przypadku złożoności cyklomatycznej, metryka NPath również może być obliczona za pomocą Checkstyle. Konfiguracja w tym przypadku jest tożsama z tą przedstawioną na listingu 7.3, z zamienioną wartością `name` na `NPathComplexity`. W ten sposób na wyjściu zawsze uzyskiwano naruszenie ustalonej reguły, a wartość metryki uzyskiwano z wyrażenia *NPath Complexity is X*.

7.1.4. Metryka NCSS

Tę metrykę również obliczono za pomocą Checkstyle. Konfiguracja w tym przypadku wygląda nieco inaczej, ponieważ metryka NCSS może być obliczana z punktu widzenia metody, klasy a nawet całego pliku. Zamiast ustalania wartości `max`, należało tutaj jednoznacznie wskazać metodę jako cel obliczania metryki (zob. listing 7.5). Komenda obliczająca metrykę nie zmieniła się i była analogiczna do tej przedstawionej na listingu 7.4. Wartość metryki pozyskiwano z wyjścia tej komendy, z wyrażenia *NCSS for this method is X*.

```
<module name="JavaNCSS">
    <property name="methodMaximum" value="-1"/>
</module>
```

Listing 7.5: Konfiguracja Checkstyle dla obliczania metryki NCSS z punktu widzenia metody klasy

7.1.5. aSCQM

Po uruchomieniu aplikacji udostępniającej model SCQM, kolejne próbki kodu ze zbioru walidacyjnego były przesyłane do endpointa `POST /ascqm`, zgodnie z opisem w sekcji F. Uzyskiwano w ten sposób dwie wartości P_B oraz P_G oznaczające prawdopodobieństwo, że dana próbka jest odpowiednio niskiej bądź wysokiej jakości. Jako wartość metryki pochodzącej z modelu wykorzystano wartość P_B pomnożoną przez 100 i zaokrągloną do najbliższej liczby całkowitej. Uzyskiwano w ten sposób procentową wartość prawdopodobieństwa, że dany kod jest niskiej jakości. Wykorzystanie liczby P_B zamiast P_G pozwoliło uzyskać zachowanie metryki podobne do pozostałych metryk, tj. im wyższa wartość tym kod uznawany jest za kod niższej jakości.

7.1.6. rSCQM

Analogicznie do aSCQM, każda para próbek była wysyłana do endpointa `POST /rscqm`. Uzyskiwano w ten sposób dwie wartości P_B oraz P_G oznaczające prawdopodobieństwo, że dana zmiana odpowiednio obniża lub podnosi jakość kodu. Tutaj z kolei wykorzystano wartość P_G, również mnożąc ją przez 100 i zaokrąglając do najbliższej liczby całkowitej. Ze względu na sposób klasyfikacji zmiany w kodzie (a nie jak wcześniej – próbki kodu) taki wynik był łatwiejszy do porównania (zob. sekcja 7.3.2).

7.2. Dane walidacyjne

Do porównania skuteczności modelu SCQM i omówionych w poprzedniej sekcji 7.1 narzędzi i metryk wykorzystano podzbiór utworzonego jakościowego *benchmarku* kodu źródłowego (zob. sekcja 5.6). Niesprawiedliwe byłoby wykorzystanie całego *benchmarku* ze względu na fakt, że większość przypadków kodu w nim zawartych była użyta w trakcie trenowania modelu SCQM. To skłoniło autora rozprawy do podjęcia decyzji o przetestowaniu trafności wskazywania kodu niższej jakości z użyciem 96 metod, które w trakcie uczenia stanowiły zbiór testowy. Ponieważ skrypt uczący model SCQM odcinał początkowe metody jako zbiór testowy (zob. dodatek C), *benchmark* dla ewaluacji tworzą próbki z liczbami porządkowymi od 0 do 95.

Pomimo tego, że jest to dość mały zbiór, nie ma podstaw by zanegować jego poprawność. Wszystkie te metody zostały sklasyfikowane przez programistów tak, jakby były klasyfikowane podczas przeglądu kodu źródłowego. Oczekuje się więc, że metoda która będzie najtrafjniej wskazywać wysoką lub niską jakość kodu z użyciem tych przykładów będzie mogła być uznana za taką, która najlepiej naśladuje subiektywne pojęcie czytelności kodu.

Brano pod uwagę również ewaluację przedstawionych rozwiązań z użyciem zupełnie nowych metod, wyekstrahowanych z innych repozytoriów według reguł opisanych w rozdziale 4. Jak jednak wykazał eksperyment z gromadzeniem wiedzy ekspertowej od programistów – nawet bardzo precyzyjne reguły filtrowania *commitów* refaktoryzacyjnych i zmienianych metod dawały zbiór treningowy ze sporym szumem danych. Ewaluacja na takim zbiorze byłaby narażona na duże ryzyko błędu, co z kolei mogłoby prowadzić do zbyt pochopnych wniosków. Z tego powodu porzucono pomysł pobierania i filtrowania *commitów* refaktoryzacyjnych z nowych repozytoriów.

7.3. Wynik analizy porównawczej

Dane uzyskane z klasyfikacji próbek kodu ze zbudowanego zbioru walidacyjnego za pomocą omówionych metod zostały poddane analizie. Wyniki osiągane przez każdą metodę zostały zestawione i przedstawione w tej sekcji. Ze względu na odmienną naturę modeli aSCQM i rSCQM, poddano je osobnym analizom, ustalając odpowiednie kryteria porównania z istniejącymi rozwiązaniami.

7.3.1. Analiza porównawcza aSCQM

Bezwzględny jakościowy model kodu źródłowego stara się odpowiedzieć na pytanie:

Czy przedstawiony kod jest kodem wysokiej jakości?

W celu porównania klasyfikacji aSCQM i innych metryk wymagane było określenie, co to znaczy według każdej z nich, że dany kod jest kodem wysokiej jakości. Należało określić stałe progi, od których wartość danej metryki będzie oznaczać kod niskiej jakości, a poniżej których będzie oznaczać kod jakości odpowiedniej.

Dla modelu aSCQM sytuacja jest prosta. Skoro mamy do czynienia z liczbą P_B, oznaczającą *prawdopodobieństwo, że dana próbka jest niskiej jakości*, to można stwierdzić że model aSCQM sugeruje niską jakość kodu gdy $P_B > 50$.

Dla pozostałych metryk określono progi tak, by metryki wypadały jak najlepiej dla zadanych danych Wartości, od których kod według poszczególnych metryk był uznawany za kod niskiej jakości zostały przedstawione w tabeli 7.1.

Tabela 7.1: Wartości poszczególnych metryk, od których kod był uznawany za kod niskiej jakości w modelu bezwzględnym

Metryka	**Próg**
Checkstyle	1
Złożoność cyklomatyczna	2
Złożoność NPath	2
Metryka NCSS	3

Tabela 7.2 przedstawia porównanie trafności wskazywania kodu wysokiej i niskiej jakości dla 192 próbek kodu pochodzących z opisanego w sekcji 7.2 *benchmarku*.

Tabela 7.2: Rezultat ewaluacji modelu bezwzględnego

Metryka	**Poprawne klasyfikacje**	**Błędne klasyfikacje**	**Trafność**
Checkstyle	99	93	52%
Złożoność cyklomatyczna	107	85	56%
Złożoność NPath	107	85	56%
Metryka NCSS	109	83	57%
aSCQM	152	40	79%

7.3.2. Analiza porównawcza rSCQM

Względny jakościowy model kodu źródłowego stara się odpowiedzieć na pytanie:
Czy wprowadzana zmiana podnosi jakość kodu źródłowego?

W przypadku modelu względnego nie jest konieczne określanie wartości progowych metryk. Przy analizowaniu zmiany w kodzie mamy do czynienia z dwoma próbkami. Wystarczy więc założyć, że dana metryka mówi o poprawie jakości kodu, gdy jej wartość dla kodu sprzed zmiany jest wyższa niż wartość dla kodu po zmianie. Analogicznie – metryka sugeruje pogorszenia jakości, gdy jej wartość dla kodu po zmianie jest wyższa niż przed.

Przyjęcie takiej reguły ewaluacji powoduje sytuacje, że dana metryka może nie mieć nic do powiedzenia o danej zmianie, tj. jej wartość dla kodu przed i po jest taka sama. Jest to możliwe dla wszystkich porównywanych metryk poza rSCQM. Stworzony model ma opinię o każdej zmianie, wyrażoną w prawdopodobieństwie P_G oznaczającym, że zmiana podnosi jakość kodu. W tym przypadku jednak wątpliwe jest, czy można uznać np. $P_G = 51\%$ jako sugestię, że dana zmiana jest poprawna w kontekście czytelności kodu. Z tego powoduj przyjęto, że rSCQM sugeruje zmianę poprawiającą jakość kodu gdy $P_G > 60\%$. Natomiast zmiana obniżająca jakość kodu to taka, dla której klasyfikator zwróci $P_G < 40\%$. Dzięki temu wyniki otrzymane ze stworzonego modelu również mogą być sklasyfikowane jako nieoznaczone gdy $40\% \leq P_G \leq 60\%$. Takie podejście powinno zminimalizować ryzyko przekłamania analizy porównawczej przez zbyt optymistyczne akceptowanie klasyfikacji pochodzących z modelu rSCQM.

Tabela 7.3 przedstawia porównanie trafności wskazywania zmiany podnoszącej jakość kodu dla 96 próbek zmian pochodzących z opisanego w sekcji 7.2 *benchmarku*.

Tabela 7.3: Rezultat ewaluacji modelu względnego

Metryka	**Poprawne klasyfikacje**	**Błędne klasyfikacje**	**Brak zdania**
Checkstyle	6	5	85
Złożoność cyklomatyczna	18	7	71
Złożoność NPath	17	7	72
Metryka NCSS	42	18	36
rSCQM	70	20	6

7.4. Próbki sprawiające problemy przy klasyfikacji

Niektóre przykłady zmian ze zbioru metod, które były poddane ewaluacji przykuły uwagę autora rozprawy. Poniżej przedstawiono i omówiono kilka z nich. Ich analiza potwierdziła, jak bardzo subiektywnym pojęciem jest jakość kodu źródłowego. Ponadto, zaprezentowane przykłady sugerują, że zebranie opinii programistów w przeprowadzonym badaniu było kluczowym etapem prac, który pozwolił na przekazanie odpowiedniej wiedzy do stworzonego jakościowego modelu kodu źródłowego.

7.4.1. Każda z metryk nie ma zdania

Listingi 7.6 i 7.7 przedstawione na kolejnej stronie zawierają zmianę, dla której wartość wszystkich analizowanych metryk była jednakowa dla kodu w lepszej i gorszej wersji.

Checkstyle w obydwu przypadkach nie odnalazł żadnego naruszenia reguł. Złożoność cyklomatyczna jest równa 1, NPath – 0, NCSS – 4. Pomimo braku zmiany tych metryk, co najmniej trzech programistów w badaniu oznaczyło kod z listingu 7.6 jako lepszy. Czy chodziło tu o brak tworzenia kolejnego obiektu klasy `ComponentName`? a być może mieli oni jakąś wiedzę dziedzinową, pozwalającą na rozstrzygnięcie tego przypadku? Bez wątpienia kod był tworzony dla platformy Android. Jest to przypadek rzeczywiście trudny do rozstrzygnięcia bez znajomości szerszego kontekstu.

```
public class Class00000020Better {
    /**
     * Dismiss the Keyboard Shortcuts screen.
     */
    public final void dismissKeyboardShortcutsHelper() {
        Intent intent = new Intent(Intent.
            ↪ ACTION_DISMISS_KEYBOARD_SHORTCUTS);
        intent.setPackage(KEYBOARD_SHORTCUTS_RECEIVER_PKG_NAME);
        sendBroadcastAsUser(intent, UserHandle.SYSTEM);
    }
}
```

Listing 7.6: Każda z metryk nie ma zdania – przykład kodu – wersja lepsza według programistów

```
public class Class00000020Worse {
    /**
     * Dismiss the Keyboard Shortcuts screen.
     */
    public final void dismissKeyboardShortcutsHelper() {
        Intent intent = new Intent(Intent.
            ↪ ACTION_DISMISS_KEYBOARD_SHORTCUTS);
        intent.setComponent(new ComponentName(
            ↪ KEYBOARD_SHORTCUTS_RECEIVER_PKG_NAME,
            ↪ KEYBOARD_SHORTCUTS_RECEIVER_CLASS_NAME));
        sendBroadcast(intent);
    }
}
```

Listing 7.7: Każda z metryk nie ma zdania – przykład kodu – wersja gorsza według programistów

7.4.2. Tylko model rSCQM wskazał poprawnie

Listingi 7.8 i 7.9 zawierają zmianę, dla której wszystkie metryki wskazały pogorszenie kodu – poza modelem rSCQM, który podał klasyfikację zgodną z opinią programistów.

```
public class Class00000031Better {
    private boolean jj_3R_31() {
        if (jj_3R_32())
            return true;
        if (jj_scan_token(MATCHES))
            return true;
        if (jj_scan_token(STRING_LITERAL))
            return true;
        return false;
    }
}
```

Listing 7.8: Tylko rSCQM wskazał poprawę jakości – przykład kodu – wersja lepsza według programistów

```
public class Class00000031Worse {
    private boolean jj_3R_31() {
        Token xsp = jj_scanpos;
        if (jj_3R_45()) {
            jj_scanpos = xsp;
            if (jj_3R_46())
                return true;
        }
        return false;
    }
}
```

Listing 7.9: Tylko rSCQM wskazał poprawę jakości – przykład kodu – wersja gorsza według programistów

Checkstyle dla kodu gorszego wskazuje jednokrotne naruszenie reguły `NeedBraces`, podczas gdy dla kodu lepszego – reguła jest naruszona trzykrotnie (reguła dotyczy braku klamer w ciele warunku `if`). Złożoność cyklomatyczna – kod gorszy ÷ lepszy: $3 \div 4$, NPath: $3 \div 8$. NCSS: $7 \div 8$.

Tymczasem model rSCQM poprawnie rozpoznał prostotę zaprezentowanego przykładu i pomimo wielu wyrażeń warunkowych i kilkukrotnego wystąpienia instrukcji `return` określił, że ta zmiana poprawia jakość kodu z prawdopodobieństwem $P_G = 100\%$. Tym samym potwierdził opinię programistów biorących udział w badaniu oraz autora rozprawy o tym przykładzie refaktoryzacji. Dla potwierdzenia – w oryginalnym repozytorium ta zmiana była właśnie refaktoryzacją przedmiotowej metody.

7.4.3. Wszystkie metryki popełniły błąd

Listingi 7.10 i 7.11 zawierają zmianę, dla której wszystkie metryki wskazały pogorszenie jakości kodu, pomimo odmiennego jej sklasyfikowania przez programistów w przeprowadzonym badaniu.

Checkstyle: kod gorszy – brak naruszeń, kod lepszy – jedno naruszenie. Podobnie do poprzedniego przykładu – został wykryty zapach kodu pochodzący z reguły `NeedBraces`. Złożoność cyklomatyczna oraz NPath przyjęły wartościo odpowiednio $1 \div 2$, NCSS: $2 \div 4$. W końcu – rSCQM określił zmianę jakości kodu źródłowego na poziomie $P_G = 0$.

```
public class Class00000033Better {
    @Override
    public List<ByteBuffer> values(QueryOptions options) throws
        ↪ InvalidRequestException {
        if (!isPartitionKey)
            throw new UnsupportedOperationException();
        return toByteBuffers(valuesAsClustering(options));
    }
}
```

Listing 7.10: Wszystkie metryki popełniły błąd – przykład kodu – wersja lepsza według programistów

```
public class Class00000033Worse {
    @Override
    public List<ByteBuffer> values(QueryOptions options) throws
        ↪ InvalidRequestException {
        return Composites.toByteBuffers(valuesAsComposites(options));
    }
}
```

Listing 7.11: Wszystkie metryki popełniły błąd – przykład kodu – wersja gorsza według programistów

Dlaczego więc, wbrew wszystkim analizowanym metodom automatycznej oceny jakości kodu, programiści ocenili przedstawioną zmianę jako poprawiającą jakość kodu? Nie zmieniła się sygnatura metody. Doszedł jedynie warunek sprawdzający stan w polu klasy i zastąpiono użycie narzędziowej klasy za pomocą wywołania metody z klasy obecnej bądź nadrzędnej. Być może przesądziło przekonanie, że kod ten jest bardziej przemyślany ze względu na wyrzucenie wyjątku w konkretnej sytuacji, co być może zostało w pierwotnej wersji przeoczone?

To jest przykład zmiany, którą na prawdę trudno sklasyfikować nie znając szerszego kontekstu i nie posiadając wiedzy na temat projektu, do którego ta zmiana jest wprowadzana. To z kolei pokazuje, jak rzeczywiście nietrywialny jest problem jakości kodu źródłowego.

7.5. Możliwe integracje

Implementacja modelu po zakończonym trenowaniu wymagała wprowadzenia kilku zmian przed przystąpieniem do opisanej ewaluacji stworzonego rozwiązania. Długi czas inicjalizacji struktur wykorzystywanego *frameworka* TensorFlow był sporym ograniczeniem utrudniającym jej przeprowadzenie. Co więcej, sam kod źródłowy modelu na tym etapie nie był gotowy do analizy dowolnej zadanej próbki. Oczekiwał on bowiem zbioru danych wejściowych, które dzielił na zbiór testowy i treningowy, a następnie mógł z zapisanego wcześniej wyniku uczenia obliczyć trafność dla zbioru testowego.

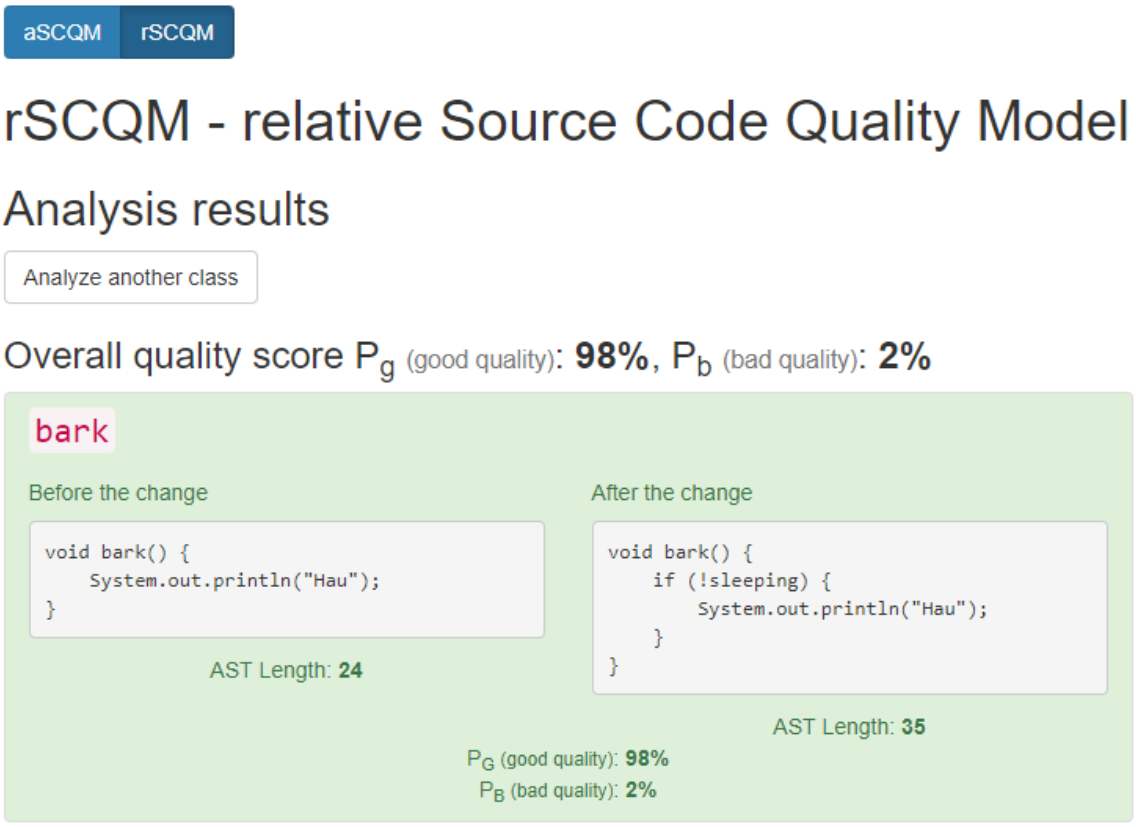
Ponadto, aby umożliwić potencjalną integrację modelu z innymi rozwiązaniami – należało przygotować jego implementację i uzyskane wyniki tak, by można je było łatwo uruchomić i uzyskać z nich klasyfikację dowolnego kodu źródłowego. Niepoprawnym byłoby wymagać od użytkowników zainteresowanych jakościowym modelem wykonywania wielu kroków przygotowania kodu w celu dostarczenia go do stworzonej sieci neuronowej.

Pierwszym usprawnieniem implementacji było wprowadzenie możliwości jednokrotnej inicjalizacji struktur sieci neuronowych, po której model był gotowy do klasyfikacji dowolnej liczby zadanych próbek kodu źródłowego. Dzięki temu nie było konieczne długie oczekiwanie na wynik klasyfikacji. Co więcej, umożliwiono przesłanie kodu źródłowego i otrzymanie wyniku za pomocą protokołu HTTP. Stało się to możliwe po ukryciu szczegółów implementacyjnych modelu za prostym interfejsem mikrousługi, z którą komunikacja jest możliwa za pomocą REST API.

Przygotowano także konfigurację pozwalającą uruchomić wprowadzoną mikrousługę w kontenerach Docker[3]. Dzięki temu uruchomienie wygodnego w użyciu klasyfikatora jakości kodu źródłowego opartego o SCQM jest bardzo prostą operacją. Co więcej, wykorzystanie kontenerów umożliwia przenośność implementacji pomiędzy różnymi platformami.

Warto zwrócić uwagę, że poza API, mikrousługa oferuje także przejrzysty interfejs użytkownika w formie aplikacji internetowej, która udostępnia użytkownikowi dwa formularze umożliwiające przesłanie kodu źródłowego w celu sklasyfikowania jego jakości. Przykładowy rezultat tak przeprowadzonej klasyfikacji przedstawiono na rysunku 7.1.

[3] `https://www.docker.com`



Rys. 7.1: Rezultat klasyfikacji kodu klasy przez model rSCQM

W dodatku E zamieszczono szczegóły wprowadzonych do implementacji modelu zmian oraz listę funkcjonalności oferowanych przez mikrousługę i sposób ich konfiguracji. Dodatek F zawiera instrukcję uruchomienia modelu w dowolnym systemie operacyjnym.

Przykładowa integracja

Aby zaprezentować w jaki sposób mogłoby wyglądać wdrożenie modelu SCQM do procesu implementacji oprogramowania, zaimplementowano przykładowy skrypt współpracujący z systemem kontroli wersji Git, sprawdzający jakość wprowadzanej do repozytorium zmiany. Jest to *hook* `pre-commit`, czyli skrypt odpowiedzialny za dodatkowe sprawdzenia kodu przed stworzeniem *commita*. Na tym etapie *commit* może zostać odrzucony a programista otrzyma informację o powodzie odrzucenia. Skrypt został zaprojektowany jako lokalny, czyli każdy programista w zespole musi odpowiednio skonfigurować swoją instancję projektu tak, by przed wykonaniem zmiany model

SCQM był użyty do sprawdzenia jakości nowego kodu. Nic nie stoi jednak na przeszkodzie, by zamiast zdarzenia `pre-commit` wykorzystać zdarzenie `pre-receive` na serwerze i dodać podobne sprawdzenie przed zaakceptowaniem kodu źródłowego od dowolnego członka zespołu w repozytorium zdalnym. Więcej o hookach w systemie kontroli wersji Git można przeczytać w jego oficjalnej dokumentacji [90]. Szczegóły implementacji integracji oraz przykład jej wykorzystania zamieszczono w dodatku G.

7.6. Podsumowanie

Podczas ewaluacji opracowanego modelu przygotowano aplikację, która w prosty sposób umożliwia uzyskanie klasyfikacji dla dowolnego kodu źródłowego w języku Java. Dzięki temu stworzone rozwiązanie może być łatwo zintegrowane z istniejącymi aplikacjami.

Z punktu widzenia rozprawy ważne jest, że takie przygotowanie modelu umożliwiło łatwe porównanie otrzymywanych z niego klasyfikacji z innymi dostępnymi rozwiązaniami. Po ich starannym wyselekcjonowaniu i przygotowaniu zbioru walidacyjnego, opisana w tym rozdziale ewaluacja modelu potwierdziła, że stworzony model SCQM wykazuje większą trafność od znanych narzędzi i metryk klasyfikujących jakość kodu na podstawie jego statycznej analizy.

Rozdział 8

Uwagi końcowe

Staranne zebranie próbek kodu o znanej jakości oraz zebranie opinii o jakości kodu od programistów pozwoliło na skuteczne wytrenowanie jakościowego modelu kodu źródłowego. W tym rozdziale odniesiono się do metodyki i ewaluacji opisanej we wcześniejszych rozdziałach by wykazać prawdziwość postawionej tezy oraz potwierdzić realizację założonych celów rozprawy.

8.1. Weryfikacja tezy

Dla przypomnienia, teza niniejszej rozprawy doktorskiej jest następująca.

Opracowanie jakościowego modelu kodu źródłowego z użyciem technik uczenia maszynowego na podstawie danych gromadzonych w systemach kontroli wersji pozwoli na wykazanie większej trafności w rozpoznawaniu kodu niskiej jakości niż dostępne obecnie narzędzia wykorzystujące jego statyczną analizę.

Pierwsza część tezy zakłada zbudowanie modelu kodu źródłowego o odpowiednich cechach. Stworzony jakościowy model kodu źródłowego SCQM jest oparty o dwukierunkową rekurencyjną sieć neuronową z komórkami LSTM. Jest to metoda uczenia maszynowego, która przechowuje wiedzę w postaci odpowiednich wag neuronów, z których jest złożona. Co więcej, tak jak zakładano – model ten został wytrenowany danymi pochodzącymi z systemów kontroli wersji. W rozdziale 4 dokładnie opisano sposób automatycznego pozyskania zmian refaktoryzacyjnych z wielu projektów. Także próbki kodu, które zostały wykorzystane przy badaniu opinii programistów o kodzie

źródłowym pochodziły z tego samego źródła, co potwierdza przekonanie autora rozprawy o niezwykle dużej przydatności systemów kontroli wersji we wszelkich analizach dotyczących jakości kodu źródłowego.

W drugiej części tezy zostało postawione założenie, że stworzony w ten sposób model będzie trafniej wskazywał kod niskiej lub wysokiej jakości niż dostępne aktualnie narzędzia, które umożliwiają jego statyczną analizę. Zgodnie z opisem w rozdziale 7, do porównania wykorzystano popularne i szeroko stosowane narzędzie Checkstyle oraz szereg jakościowych metryk, które według różnych publikacji mogą być wskaźnikami odpowiedniej jakości kodu źródłowego. Model SCQM, zarówno w wersji bezwzględnej jak i względnej, wykazał lepszą trafność w identyfikowaniu niskiej jakości kodu źródłowego. Aby weryfikacja tezy była możliwa, zbudowano także *benchmark* jakości kodu źródłowego ze względu na niedostępność odpowiedniego zbioru danych walidacyjnych. Zebranie opinii od programistów o jakości kilkuset próbek kodu źródłowego i użycie ich przy analizie porównawczej pozwala założyć, że zrealizowany model jest skuteczniejszym narzędziem w ocenie jakości kodu źródłowego niż istniejące obecnie narzędzia i metody.

Szczegółowe porównanie skuteczności rozpoznawania kodu niskiej jakości przez poszczególne narzędzia i metody zostało przedstawione w tabelach 7.2 i 7.3. Model bezwzględny (aSCQM) poprawnie sklasyfikował jakość kodu dla 79% przykładów ze zbioru walidacyjnego, podczas gdy druga pod względem skuteczności metryka NCSS osiągnęła jedynie 57%.

Podobny rezultat został uzyskany dla modelu względnego (rSCQM), oceniającego jakość zmiany w kodzie źródłowym. Stworzony jakościowy model kodu źródłowego poprawnie sklasyfikował 73% próbek kodu. Metryka NCSS również tutaj uplasowała się na drugiej pozycji, poprawnie klasyfikując 44% zmian[1].

Powyższe rezultaty uzyskane w trakcie pracy nad rozprawą **potwierdzają założoną tezę**. Zbudowano jakościowy model kodu źródłowego z wykorzystaniem danych gromadzonych w systemach kontroli wersji, który trafniej klasyfikuje jakość kodu źródłowego niż istniejące dotąd rozwiązania wykorzystujące jego statyczną analizę.

[1]w ewaluacji modelu względnego metody klasyfikacji mogły zgłosić „brak zdania" o danej zmianie; jeśli by brać pod uwagę tylko sklasyfikowane zmiany, to skuteczność rSCQM wynosiłaby 78%, a NCSS – 70%; zob. tabelę 7.3

8.2. Osiągnięcia rozprawy

Weryfikacja tezy rozprawy była możliwa dzięki zrealizowaniu poszczególnych celów pracy, które zostały zdefiniowane we wstępie. W niniejszej sekcji odniesiono się do każdego z nich, wskazując jednocześnie miejsce w rozprawie zawierające szczegóły jego realizacji.

1. **Analiza istniejących rozwiązań pozwalających na utrzymanie kodu źródłowego wysokiej jakości.** Rozdział 2 zawiera szczegółową analizę literatury i istniejących rozwiązań zapewniających utrzymanie kodu źródłowego wysokiej jakości w tworzonym oprogramowaniu. Dzięki tej analizie autor rozprawy dokładnie znał aktualny stan wiedzy o jakości kodu i wykazał ważność swojego rozwiązania na tle tych, które już są stworzone i używane.

2. **Opracowanie jakościowego modelu kodu źródłowego**, który będzie w sposób automatyczny klasyfikował jakość kodu zostało – jako główny cel rozprawy – zrealizowane. Wyniki analizy porównawczej potwierdzającej lepszą skuteczność w rozpoznawaniu kodu niskiej jakości od dostępnych obecnie technik zostały opisane przy wykazywaniu prawdziwości tezy rozprawy w poprzedniej sekcji 8.1.

3. **Zgromadzenie danych treningowych dla jakościowego modelu.** W rozdziale 4, opisano sposób pozyskania próbek kodu o różnej jakości z GitHuba, czyli jednej z najpopularniejszych platform przechowujących repozytoria projektów z otwartym kodem źródłowym. Projekty zostały przefiltrowane tak, by zawierały docelowy język programowania – Javę. Początkowo wybrano prawie 18 tysięcy zmian refaktoryzacyjnych zmieniających kod w ponad 150 tysiącach klas. Dalsze ich filtrowanie według coraz bardziej precyzyjnych reguł pozwoliło na zebranie odpowiednich danych treningowych, które umożliwiły wytrenowanie jakościowego modelu kodu źródłowego.

4. **Zaprojektowanie i implementacja modelu.** W sekcjach 3.5 oraz 3.6 przedstawiono projekt budowy modelu bezwzględnego i względnego (aSCQM i rSCQM). Obydwa modele zostały zaimplementowane za pomocą dwukierunkowej rekurencyjnej sieci neuronowej z komórkami LSTM. Motywację stojąca za wyborem tej reprezentacji przedstawiono w sekcji 3.3.4. Sposób implementacji oraz format danych wejściowych i rezultatu wyjściowego został szczegółowo opisany w sekcjach 4.5 i 6.1.

5. **Zebranie opinii o jakości kodu źródłowego od programistów.** W ramach rozprawy zostało przeprowadzone badanie, w którym programiści wykonali przegląd kodu źródłowego zaprezentowanych przykładów fragmentów oprogramowania o różnej jakości. W ten sposób, oprócz danych pozyskanych automatycznie ze zmian refaktoryzacyjnych, do modelu zostały także dostarczone rzeczywiste opinie programistów o jakości kodu źródłowego. Eksperyment trwał 6 miesięcy i wzięło w nim udział ponad 500 osób, zostawiając ponad 5000 opinii i klasyfikując w ten sposób 645 przykładów zmian refaktoryzacyjnych. Badanie zostało szczegółowo opisane w rozdziale 5.

6. **Zbudowanie zbioru danych walidacyjnych pozwalających na ewaluację jakościowych modeli kodu źródłowego.** Na podstawie danych pozyskanych od programistów został stworzony *benchmark* [98], który umożliwił ewaluację tworzonego modelu. Jest to zbiór próbek kodu o różnej, ale znanej jakości. Taki *benchmark* pozwala na sprawdzenie efektywności dowolnych innych metod starających się w przyszłości poprawnie klasyfikować jakość kodu źródłowego.

7. **Przygotowanie narzędzia oferującego funkcjonalność klasyfikacji kodu za pomocą stworzonego modelu.** Przygotowany jakościowy model kodu źródłowego został osadzony w wygodnej do uruchomienia i wykorzystania mikrousłudze za pomocą kontenerów Dockerowych. W sekcji 7.5 opisano sposób konfiguracji, uruchomienia i korzystania z narzędzia. Dzięki udostępnieniu REST API, dowolne narzędzie, które skorzystałoby na automatycznej klasyfikacji jakości kodu źródłowego (zob. sekcję 8.3) może być w prosty sposób zintegrowane z modelem SCQM. Pozostawienie głównego rezultatu rozprawy w formie prostej do użycia było bardzo istotne, by pomysł mógł być dalej rozwijany.

8.3. Rozwój koncepcji

Rozprawa nie wyczerpuje tematu rozwoju narzędzi analizujących jakość i czytelność kodu źródłowego lub metod wspierających wykonywanie jego przeglądów. W tej sekcji zaproponowano różne kierunki rozwoju przedstawionego rozwiązania.

8.3.1. Integracja modelu z istniejącymi narzędziami

Dzięki dostarczeniu mikrousługi udostępniającej klasyfikacje realizowane za pomocą modelu SCQM za pomocą REST API (zob. sekcja 7.5), dowolne narzędzie, które

potrafi wykonać żądanie HTTP może skorzystać ze zbudowanej wiedzy, przesyłając kod źródłowy i w odpowiedzi otrzymując klasyfikację jego jakości.

W dodatku G zaprezentowano przykładowe wdrożenie polegające na kontroli nowo wprowadzanych do repozytorium zmian. Przy jego zastosowaniu, każda modyfikacja kodu źródłowego, która zostanie przez SCQM sklasyfikowana jako pogarszająca jakość kodu w projekcie zostanie automatycznie odrzucona, a programista który chciał ją dodać otrzyma komunikat wskazujący na problematyczny kod źródłowy. Na przykładzie tej integracji zademonstrowano, że do skorzystania z jakościowego modelu kodu źródłowego stworzonego w ramach pracy nad tą rozprawą wystarczy dodanie jednego prostego skryptu do konfiguracji repozytorium kodu źródłowego.

Oczywiście, takie wdrożenie nie wyczerpuje w pełni potencjału dostarczanego rozwiązania. Sugestie o jakości poszczególnych metod modyfikowanych lub dodawanych w kodzie źródłowym Javy mogłyby być prezentowane w interfejsie użytkownika dowolnej aplikacji wspierającej przeglądy kodu źródłowego, np. Gerrit Code Review lub podczas przeglądania *pull requestów* na platformie GitHub. W obydwu przypadkach wymagałoby to uruchomienia modelu SCQM w lokalizacji dostępnej z serwera integrowanej aplikacji oraz dostarczenia rozszerzenia platformy, który po komunikacji z jakościowym modelem dodawałby odpowiednie informacje do kodu podczas przeglądu.

Ponadto, klasyfikacje jakości kodu źródłowego mogą być także wykorzystane podczas pisania kodu w środowiskach programistycznych IDE. Pozwolą one na wskazanie miejsc w istniejącym systemie, które wymagają refaktoryzacji. Takie fragmenty kodu mogłyby być specjalnie oznaczane jak ostrzeżenia o istnieniu w projekcie długu technologicznego. Stanowiłoby to bardzo użyteczny dodatek do już wyświetlanych ostrzeżeń pochodzących z kompilatorów lub *linterów*.

Wykorzystanie modelu w gamifikacji przeglądów kodu

Automatyczna klasyfikacja jakości przesyłanego kodu źródłowego byłaby idealnym dodatkiem do rozszerzeń wprowadzających gamifikację do przeglądów kodu źródłowego (zob. sekcję 2.5.4). Programiści byliby nagradzani nie tylko za osiąganie odpowiednich statystyk takich jak liczba zmian czy wykonanych przeglądów kodu. Opinia modelu SCQM o tworzonym przez nich kodzie źródłowym mogłaby być także brana pod uwagę w ogólnej ocenie programisty. W efekcie, osoby tworzące bardziej czytelny kod powinny plasować się wyżej w rankingu od osób, które o jakość kodu dbają w mniejszym stopniu.

8.3.2. Rozbudowa jakościowego *benchmarka* kodu źródłowego

Jakościowy *benchmark* kodu źródłowego [98], którego skompletowanie jest jednym z osiągnięć rozprawy, zawiera 645 przykładów zmian refaktoryzacyjnych ocenionych przez programistów w trakcie przeprowadzonego badania. Część z tych przykładów została użyta do wytrenowania stworzonego jakościowego modelu kodu. Pozostałe zostały użyte przy ewaluacji rozwiązania.

Pomimo zakończenia badania w czerwcu 2018 roku, autor rozprawy nie zamknął platformy przyjmującej opinie programistów. Aplikacja cały czas działa i jest dostępna pod dotychczasowym adresem `https://code.fracz.com`.

W momencie zakańczania niniejszej rozprawy, na platformie jest sklasyfikowanych już 674 metody przez 599 osób. Po porównaniu tych danych do rezultatu badania przedstawionego w sekcji 5.7, w trakcie kolejnych 6 miesięcy bez żadnego aktywnego werbowania respondentów ze strony autora rozprawy okazuje się, że na stronę z badaniem trafiło 40 nowych osób zainteresowanych projektem. Zostawili oni ponad 200 oddanych głosów, klasyfikując w ten sposób 29 następnych metod.

Przy niewielkim zaangażowaniu (np. przeprowadzaniu badania wśród studentów na kolejnych semestrach), zbiór ocenionych zmian refaktoryzacyjnych w stworzonym *benchmarku* może się znacząco powiększać z roku na rok. W ten sposób, opracowany w ramach niniejszej rozprawy mechanizm pozyskiwania opinii od programistów, gromadzenia i udostępniania ich publicznie w formie repozytorium będzie aktywny i użyteczny jeszcze długo po zakończeniu opisywanych tu prac.

8.3.3. Zniesienie ograniczeń modelu

W sekcji 3.2 dokładnie opisano ograniczenia modelu SCQM oraz wyznaczono zakres jego funkcjonalności i użyteczności. Głównym ograniczeniem jest wybór tylko jednego języka programowania.

Pierwszym badaniem, którego rezultaty mogą okazać się ciekawe, jest sprawdzenie jak zachowa się model wytrenowany za pomocą Javy dla innego języka programowania. Czy wiedza, która została przekazana do sieci neuronowej jest uniwersalna, czy dotyczy tylko i wyłącznie wybranego języka programowania? Aby to zweryfikować, należałoby dla innego języka programowania przygotować podobny rozbiór syntaktyczny (zob. sekcja 4.3), w którym tokeny będą oznaczać te same konstrukcje językowe. Teoretycznie po takim przygotowaniu danych model powinien poprawnie klasyfikować jakość kodu pod warunkiem, że nowo wspierany język też będzie językiem zorientowanym obiektowo.

Nie jest to zadanie trywialne, ponieważ różne języki posiadają różne konstrukcje, nawet jeśli realizują ten sam paradygmat programowania. Z tego powodu bardziej sensownym wydaje się przygotowanie danych dla innego języka programowania zgodnie z opisem w rozdziale 4 i ponowne wytrenowanie modelu za ich pomocą. Niestety, by osiągnąć rezultaty podobne do prezentowanych w niniejszej rozprawie, wymagane by także było przeprowadzenie kolejnego badania gromadzącego opinie od programistów, co wymaga dotarcia do osób posługujących się danym językiem programowania i zachęcenia ich do udzielenia odpowiedzi w badaniu.

Poza wsparciem innego języka programowania, prace rozwojowe nad stworzonym modelem mogłyby także dotyczyć wyjścia poza ustalony zakres metody w klasie. Można podjąć próbę nauczenia modelu podając rozbiór syntaktyczny całych klas jako jego wejście. Trudno powiedzieć, jak wpłynie to na szybkość uczenia się modelu oraz na trafność końcowej klasyfikacji. Ta ścieżka ewolucji modelu jest jednak prostsza, gdyż zestaw danych treningowych jest w dużej mierze przygotowany. Należy jedynie wyszukać pochodzenie sklasyfikowanych przez programistów metod w odkrytych zmianach refaktoryzacyjnych tak, aby można było wziąć pod uwagę całe klasy do których oryginalnie należały.

8.4. Podsumowanie

Wprowadzony w niniejszej rozprawie doktorskiej jakościowy model kodu źródłowego wykazuje dużo lepszą trafność w klasyfikacji czytelności kodu niż znane dotąd narzędzia, które przeprowadzają jego statyczną analizę. Autor rozprawy wykazał, że pomimo subiektywnego pojmowania jakości kodu źródłowego, metody uczenia maszynowego są w stanie wyodrębnić cechy wspólne, które taki kod charakteryzują. Wdrożenie osiągnięć rozprawy w praktyce przyczyni się do skrócenia czasu wymaganego na przeglądy kody źródłowego, a co za tym idzie – do wzrostu efektywności pracy programistów. Ponadto, wykorzystanie jakościowego modelu podczas pisania kodu źródłowego pozwoli na bieżąco kontrolować jego jakość, co z kolei ułatwi dalszy rozwój dostarczanych przez programistów rozwiązań.

Oprócz rzeczywistego wkładu w inżynierię oprogramowania, ważnym rezultatem rozprawy jest także zestaw ocenionych przez programistów próbek kodu źródłowego o zróżnicowanej jakości. Jest to niezwykle wartościowy wynik pracy w kontekście dalszych badań ze względu na fakt, że może on zostać użyty do ewaluacji dowolnej innej metody której zadaniem jest klasyfikacja jakości lub czytelności kodu źródłowego. Aktualnie nie istnieje podobny *benchmark* jakości kodu źródłowego.

Spis rysunków

Spis tabel

Spis listingów

Spis akronimów

API Application Programming Interface.

aSCQM Absolute Source Code Quality Model.

AST Abstract Syntax Tree.

CSV Comma Separated Values.

CUDA Compute Unified Device Architecture.

GPU Graphics Processing Unit.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

LSTM Long short-term memory.

NLP Natural Language Processing.

REST Representational State Transfer.

RNN Recurrent neural network.

rSCQM Relative Source Code Quality Model.

SCQM Source Code Quality Model.

SRP Single Responsibility Principle.

VCS Version Control System.

VPS Virtual Private Server.

Bibliografia

[1] Emilio Collar Jr i Ricardo Valerdi. *Role of software readability on software development cost*. Spraw. tech. 2006.

[2] Yahya Tashtoush i in. „Impact of programming features on code readability". W: (2013).

[3] Raymond PL Buse i Westley R Weimer. „Learning a metric for code readability". W: *IEEE Transactions on Software Engineering* 36.4 (2010), s. 546–558.

[4] Sławomir Sobótka. *Nie koduj, pisz prozę - lingwistyczne techniki wychodzące poza Clean Code*. [online; odwiedzono 17.01.2019]. Lip. 2014. URL: `https://www.youtube.com/watch?v=CKONKZLmMwk`.

[5] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[6] Moritz Beller i in. „Modern code reviews in open-source projects: Which problems do they fix?" W: *Proceedings of the 11th working conference on mining software repositories*. ACM. 2014, s. 202–211.

[7] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[8] Flemming Nielson, Hanne R Nielson i Chris Hankin. *Principles of program analysis*. Springer, 2015.

[9] *Lint (material) - Wikipedia description*. [online; odwiedzono 18.02.2019]. URL: `https://en.wikipedia.org/wiki/Lint_(material)r`.

[10] Stephen C Johnson. *Lint, a C program checker*. Citeseer, 1977.

[11] Stacy Nelson i Johann Schumann. „What makes a code review trustworthy?" W: *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*. IEEE. 2004, 10–pp.

[12] Robert C Martin. „The open-closed principle". W: *More C++ gems* 19.96 (1996), s. 9.

[13] Chris F Kemerer i Mark C Paulk. „The impact of design and code reviews on software quality: An empirical study based on psp data". W: *IEEE transactions on software engineering* 35.4 (2009), s. 534–550.

[14] Jason Cohen i in. *Best kept secrets of peer code review*. Smart Bear Somerville, 2006.

[15] Tobias Baum i in. „Factors influencing code review processes in industry". W: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, s. 85–96.

[16] StackOverflow. *Developer Survey Results 2018*. [online; odwiedzono 17.01.2019]. Grud. 2018. URL: `https://insights.stackoverflow.com/survey/2018/`.

[17] M. E. Fagan. „Design and code inspections to reduce errors in program development". W: *IBM Systems Journal* 15.3 (1976), s. 182–211.

[18] Robert Baggen i in. „Standardized code quality benchmarking for improving software maintainability". W: *Software Quality Journal* 20.2 (2012), s. 287–307.

[19] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[20] Mateusz Drożdż. „Ocena przekształceń refaktoryzacyjnych z wykorzystaniem metryk kodu źródłowego". Prac. mag. Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, 2016.

[21] Danilo Silva, Nikolaos Tsantalis i Marco Tulio Valente. „Why we refactor? confessions of github contributors". W: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, s. 858–870.

[22] Panita Meananeatra, Songsakdi Rongviriyapanish i Taweesup Apiwattanapong. „Using software metrics to select refactoring for long method bad smell". W: *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*. IEEE. 2011, s. 492–495.

[23] Panita Meananeatra. „Identifying refactoring sequences for improving software maintainability". W: *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE. 2012, s. 406–409.

[24] Baishakhi Ray i in. „A large scale study of programming languages and code quality in github". W: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, s. 155–165.

[25] Stefan Hanenberg. „An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time". W: *ACM Sigplan Notices*. T. 45. 10. ACM. 2010, s. 22–35.

[26] Stefan Endrikat i in. „How do API documentation and static typing affect API usability?" W: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, s. 632–642.

[27] Emerson Murphy-Hill, Chris Parnin i Andrew P Black. „How we refactor, and how we know it". W: *IEEE Transactions on Software Engineering* 38.1 (2012), s. 5–18.

[28] Philippe Kruchten, Robert L Nord i Ipek Ozkaya. „Technical debt: From metaphor to theory and practice". W: *Ieee software* 29.6 (2012), s. 18–21.

[29] Bart Du Bois, Serge Demeyer i Jan Verelst. „Does the "Refactor to Understand" reverse engineering pattern improve program comprehension?" W: *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*. IEEE. 2005, s. 334–343.

[30] Mäntylä Mika, Jari Vanhanen, Casper Lassenius i in. „A taxonomy and an initial empirical study of bad smells in code". W: *null*. IEEE. 2003, s. 381.

[31] Eva Van Emden i Leon Moonen. „Java quality assurance by detecting code smells". W: *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE. 2002, s. 97–106.

[32] Foutse Khomh, Massimiliano Di Penta i Yann-Gael Gueheneuc. „An exploratory study of the impact of code smells on software change-proneness". W: *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE. 2009, s. 75–84.

[33] Aiko Yamashita i Leon Moonen. „Do code smells reflect important maintainability aspects?" W: *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE. 2012, s. 306–315.

[34] Francesca Arcelli Fontana i in. „An experience report on using code smells detection tools". W: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE. 2011, s. 450–457.

[35] Panagiotis Louridas. „Static code analysis". W: *IEEE Software* 23.4 (2006), s. 58–61.

[36] Aiko Yamashita i Leon Moonen. „Do developers care about code smells? an exploratory survey". W: *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE. 2013, s. 242–251.

[37] Henrike Barkmann, Rüdiger Lincke i Welf Löwe. „Quantitative evaluation of software quality metrics in open-source projects". W: *2009 International Conference on Advanced Information Networking and Applications Workshops*. IEEE. 2009, s. 1067–1072.

[38] Thomas J McCabe. „A complexity measure". W: *IEEE Transactions on software Engineering* 4 (1976), s. 308–320.

[39] G. Ann Campbell. *Cognitive Complexity. A new way of measuring understandability.* [online; odwiedzono 8.03.2018]. Mar. 2018. URL: `https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html`.

[40] Matthew James Munro. „Product metrics for automatic identification of "bad smell" design problems in java source-code". W: *Software Metrics, 2005. 11th IEEE International Symposium*. IEEE. 2005, s. 15–15.

[41] Shane McIntosh i in. „An empirical study of the impact of modern code review practices on software quality". W: *Empirical Software Engineering* 21.5 (2016), s. 2146–2189.

[42] Nargis Fatima, Suriayati Chuprat i Sumaira Nazir. „Challenges and Benefits of Modern Code Review-Systematic Literature Review Protocol". W: *2018 International Conference on Smart Computing and Electronic Enterprise (ICSCEE)*. IEEE. 2018, s. 1–5.

[43] Lisa Wells. *9 Reasons to Review Code.* [online; odwiedzono 22.08.2018]. 2010. URL: `http://blog.smartbear.com/software-quality/9-reasons-to-review-code/`.

[44] Valerio Cosentino, Javier Luis Cánovas Izquierdo i Jordi Cabot. „Assessing the bus factor of Git repositories". W: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE. 2015, s. 499–503.

[45] Michael Fagan. „Design and code inspections to reduce errors in program development". W: *Software pioneers*. Springer, 2002, s. 575–607.

[46] Ronald A Radice. *High quality low cost software inspections*. Paradoxicon Publishing, 2001.

[47] Bernd Freimut, Lionel C Briand i Ferdinand Vollei. „Determining inspection cost-effectiveness by combining project data and expert opinion". W: *IEEE Transactions on Software Engineering* 31.12 (2005), s. 1074–1092.

[48] Alberto Bacchelli i Christian Bird. „Expectations, outcomes, and challenges of modern code review". W: *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, s. 712–721.

[49] Achyudh Ram i in. „What makes a code change easier to review: an empirical investigation on code change reviewability". W: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, s. 201–212.

[50] Eduardo Witter dos Santos i Ingrid Nunes. „Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data". W: *Journal of Software Engineering Research and Development* 6.1 (2018), s. 14.

[51] Alessandra Devito Da Cunha i David Greathead. „Does personality matter?: an analysis of code-review ability". W: *Communications of the ACM* 50.5 (2007), s. 109–112.

[52] Shane McIntosh i in. „The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects". W: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, s. 192–201.

[53] Olga Baysal i in. „The influence of non-technical factors on code review". W: *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE. 2013, s. 122–131.

[54] Luca Milanesio. *Learning Gerrit Code Review*. Packt Publishing, 2013.

[55] Vipin Balachandran. „Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation". W: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, s. 931–940.

[56] Patanamon Thongtanunam i in. „Who should review my code? A file location-based code-reviewer recommendation approach for modern code review". W: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE. 2015, s. 141–150.

[57] Vipin Balachandran. *Automatic code review and code reviewer recommendation*. US Patent 9,898,280. Lut. 2018.

[58] Cheng Yang i in. „RevRec: A two-layer reviewer recommendation algorithm in pull-based development model". W: *Journal of Central South University* 25.5 (2018), s. 1129–1143.

[59] Jakub Lipcak i Bruno Rossi. „A Large-Scale Study on Source Code Reviewer Recommendation". W: *arXiv preprint arXiv:1806.07619* (2018).

[60] Mohammad Masudur Rahman i in. „CORRECT: Code reviewer recommendation at GitHub for Vendasta technologies". W: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, s. 792–797.

[61] Amiangshu Bosu, Michaela Greiler i Christian Bird. „Characteristics of useful code reviews: An empirical study at microsoft". W: *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press. 2015, s. 146–156.

[62] Vladimir Kovalenko i in. „Does reviewer recommendation help developers?" W: *IEEE Transactions on Software Engineering* (2018).

[63] Wojciech Frącz i Jacek Dajda. „Experimental Validation of Source Code Reviews on Mobile Devices". W: *International Conference on Computational Science and Its Applications*. Springer. 2017, s. 533–547.

[64] Wojciech Frącz i Jacek Dajda. „Source code reviews on mobile devices". W: *Computer Science* 17.2 (2016), s. 143.

[65] Sebastian Deterding i in. „Gamification. using game-design elements in non-gaming contexts". W: *CHI'11 extended abstracts on human factors in computing systems*. ACM. 2011, s. 2425–2428.

[66] Juho Hamari, Jonna Koivisto i Harri Sarsa. „Does gamification work?–a literature review of empirical studies on gamification". W: *2014 47th Hawaii international conference on system sciences (HICSS)*. IEEE. 2014, s. 3025–3034.

[67] Wojciech Frącz i Jacek Dajda. „Developers' Game: A Preliminary Study Concerning a Tool for Automated Developers Assessment". W: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, s. 695–699.

[68] Naomi Unkelos-Shpigel i Irit Hadar. „Gamifying software engineering tasks based on cognitive principles: The case of code review". W: *Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on*. IEEE. 2015, s. 119–120.

[69] Shivam Khandelwal, Sai Krishna Sripada i Y Raghu Reddy. „Impact of gamification on code review process: An experimental Study". W: *Proceedings of the 10th Innovations in Software Engineering Conference*. ACM. 2017, s. 122–126.

[70] Georgios Gousios, Eirini Kalliamvakou i Diomidis Spinellis. „Measuring developer contribution from software repository data". W: *Proceedings of the 2008 international working conference on Mining software repositories*. ACM. 2008, s. 129–132.

[71] Ninus Khamis, René Witte i Juergen Rilling. „Automatic quality assessment of source code comments: the JavadocMiner". W: *International Conference on Application of Natural Language to Information Systems*. Springer. 2010, s. 68–79.

[72] Anne Veenendaal i in. „Sentiment Analysis in Code Review Comments". W: *Computer Science and Emerging Research Journal* 3 (2015).

[73] Toufique Ahmed i in. „SentiCR: a customized sentiment analysis tool for code review interactions". W: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, s. 106–111.

[74] Yuan Huang i in. „Salient-class location: help developers understand code change in code review". W: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, s. 770–774.

[75] Miltiadis Allamanis i in. „A survey of machine learning for big code and naturalness". W: *ACM Computing Surveys (CSUR)* 51.4 (2018), s. 81.

[76] Zhenmin Li i Yuanyuan Zhou. „PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code". W: *ACM SIGSOFT Software Engineering Notes*. T. 30. 5. ACM. 2005, s. 306–315.

[77] Song Wang i in. „Bugram: bug detection with n-gram language models". W: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, s. 708–719.

[78] Pavol Bielik, Veselin Raychev i Martin Vechev. „Learning a static analyzer from data". W: *International Conference on Computer Aided Verification*. Springer. 2017, s. 233–253.

[79] Wang, Song. „Leveraging Machine Learning to Improve Software Reliability". Prac. dokt. 2019. URL: `http://hdl.handle.net/10012/14334`.

[80] Miltos Allamanis i in. „Bimodal modelling of source code and natural language". W: *International Conference on Machine Learning*. 2015, s. 2123–2132.

[81] Daniel Stefan Tarlow i Christopher Joseph Maddison. *Source code generation, completion, checking, correction*. US Patent 9,928,040. Mar. 2018.

[82] Joshua Charles Campbell, Abram Hindle i José Nelson Amaral. „Syntax errors just aren't natural: improving error reporting with language models". W: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, s. 252–261.

[83] Sahil Bhatia i Rishabh Singh. „Automated correction for syntax errors in programming assignments using recurrent neural networks". W: *arXiv preprint arXiv:1603.06129* (2016).

[84] Anshul Gupta i Neel Sundaresan. „Intelligent code reviews using deep learning". W: (2018).

[85] Tobias Baum, Steffen Herbold i Kurt Schneider. „An Industrial Case Study on Shrinking Code Review Changesets through Remark Prediction". W: *arXiv preprint arXiv:1812.09510* (2018).

[86] Junji Shimagaki i in. „Why are commits being reverted?: a comparative study of industrial and open source projects". W: *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE. 2016, s. 301–311.

[87] Li Dong i in. „Adaptive recursive neural network for target-dependent twitter sentiment classification". W: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. T. 2. 2014, s. 49–54.

[88] Mike Schuster i Kuldip K Paliwal. „Bidirectional recurrent neural networks". W: *IEEE Transactions on Signal Processing* 45.11 (1997), s. 2673–2681.

[89] Felix A Gers, Jürgen Schmidhuber i Fred Cummins. „Learning to forget: Continual prediction with LSTM". W: (1999).

[90] Ben Straub Scott Chacon. *Pro Git, Second Edition.* [online; odwiedzono 17.01.2019]. Apress, 2019. URL: `https://git-scm.com/book/en/v2`.

[91] *GitHub - dokumentacja platformy.* [online; odwiedzono 17.01.2019]. URL: `https://help.github.com`.

[92] Wojciech Frącz. *Refactor Extractor - zestaw skryptów napisanych w języku PHP, stworzonych w celu przygotowania danych wejściowych do modelu SCQM.* [online; odwiedzono 20.08.2018]. URL: `https://github.com/fracz/refactor-extractor`.

[93] Robert C Martin. „The single responsibility principle". W: *The principles, patterns, and practices of Agile Software Development* 149 (2002), s. 154.

[94] Iulian Neamtiu, Jeffrey S Foster i Michael Hicks. „Understanding source code evolution using abstract syntax tree matching". W: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), s. 1–5.

[95] Wojciech Frącz. *Code Quality TF model - implementacja sieci neuronowej reprezentującej jakościowy model kodu źródłowego wraz z danymi przygotowanymi do uczenia.* [online; odwiedzono 20.08.2018]. URL: `https://github.com/fracz/code-quality-tensorflow`.

[96] Wojciech Frącz. *Code Assessor - platforma, która umożliwiła przeprowadzenie badania mającego na celu zebranie od programistów opinii na temat jakości kodu źródłowego.* [online; odwiedzono 10.01.2018]. URL: `https://github.com/fracz/code-assessor`.

[97] Wojciech Frącz. „An empirical study inspecting the benefits of gamification applied to university classes". W: *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th.* IEEE. 2015, s. 135–139.

[98] Wojciech Frącz. *Source Code Quality Benchmark - zbiór metod sklasyfikowanych przez programistów.* [online; odwiedzono 17.01.2019]. URL: `https://github.com/fracz/code-quality-benchmark`.

[99] *TensorFlow - dokumentacja użytej biblioteki TensorFlow.* [online; odwiedzono 13.08.2018]. URL: `https://www.tensorflow.org/api_docs/python/tf`.

[100] Cyfronet AGH. *Instrukcja uruchamiania zadań bezpośrednio przez system kolejkowy (SSH) na klastrze Prometheus AGH.* [online; odwiedzono 20.07.2018]. URL: `https://kdm.cyfronet.pl/portal/Prometheus:Podstawy`.

[101] Wojciech Frącz. *SCQM - implementacja jakościowego modelu kodu źródłowego.* [online; odwiedzono 17.01.2019]. URL: `https://github.com/fracz/scqm`.

[102] Brian A Nejmeh. „NPATH: a measure of execution path complexity and its applications". W: *Communications of the ACM* 31.2 (1988), s. 188–200.

[103] Nathaniel Ayewah i in. „Evaluating static analysis defect warnings on production software". W: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* ACM. 2007, s. 1–8.

[104] *NumPy - dokumentacja użytej biblioteki NumPy.* [online; odwiedzono 13.08.2018]. URL: `https://docs.scipy.org/doc/numpy-1.13.0/index.html`.

[105] Diederik P Kingma i Jimmy Ba. „Adam: A method for stochastic optimization". W: *arXiv preprint arXiv:1412.6980* (2014).

Dodatki

A. Pełna lista źródłowych repozytoriów

Poniższa lista zawiera wszystkie repozytoria kodu źródłowego (350) pobrane z portalu GitHub na etapie przygotowywania danych uczących dla modelu SCQM. Zgodnie z opisem w rozdziale 4, wyszukano w nich zmiany poprawiające jakość kodu, a następnie użyto ich do zbudowania bazy wiedzy jakościowego modelu kodu źródłowego.

W celu przejrzenia kodu źródłowego repozytorium, należy użyć jego nazwy z poniższej listy i poprzedzić ją adresem URL portalu GitHub, np. `https://github.com/ReactiveX/RxJava`.

Aby sprawdzić, jakie *commity* refaktoryzacyjne zostały w niej zidentyfikowane, należy przejść do katalogu `results-java` w repozytorium [92] i odnaleźć tam katalog zawierający zmiany z pobranego repozytorium. Ze względu na fakt, że znak `/` jest specjalnym znakiem w systemie plików, przy tworzeniu nazw katalogów zamieniono go na dwa myślniki `--`. W związku z tym zmian pozyskanych z projektu `ReactiveX/RxJava` należy szukać w katalogu `results-java/ReactiveX--RxJava`[1].

ReactiveX/RxJava
elastic/elasticsearch
iluwatar/java-design-patterns
square/retrofit
square/okhttp
google/guava
PhilJay/MPAndroidChart
JakeWharton/butterknife
JetBrains/kotlin
bumptech/glide
spring-projects/spring-boot
spring-projects/spring-framework
square/leakcanary
airbnb/lottie-android
greenrobot/EventBus
zxing/zxing
kdn251/interviews
nostra13/Android-Universal-Image-Loader
google/iosched
square/picasso
Blankj/AndroidUtilCode
ReactiveX/RxAndroid
facebook/fresco
alibaba/dubbo
libgdx/libgdx
chrisbanes/PhotoView

[1] `https://github.com/fracz/refactor-extractor/tree/master/results-java/ReactiveX--RxJava` lub `https://fracz.github.io/phd/links/A.1.html`

netty/netty
afollestad/material-dialogs
Netflix/Hystrix
alibaba/fastjson
CymChad/BaseRecyclerViewAdapterHelper
jfeinstein10/SlidingMenu
Tencent/tinker
lgvalle/Material-Animations
android10/Android-CleanArchitecture
nickbutcher/plaid
loopj/android-async-http
androidannotations/androidannotations
JakeWharton/ViewPagerIndicator
daimajia/AndroidSwipeLayout
pockethub/PocketHub
nathanmarz/storm
greenrobot/greenDAO
facebook/stetho
realm/realm-java
googlesamples/android-UniversalMusicPlayer
alibaba/druid
liaohuqiu/android-Ultra-Pull-To-Refresh
daimajia/AndroidViewAnimations
mikepenz/MaterialDrawer
navasmdc/MaterialDesignLibrary
chrisbanes/Android-PullToRefresh
hdodenhof/CircleImageView
WhisperSystems/Signal-Android
SeleniumHQ/selenium
google/ExoPlayer
ksoichiro/Android-ObservableScrollView
EnterpriseQualityCoding/FizzBuzzEnterpriseEdition
winterbe/java8-tutorial
orhanobut/logger
HannahMitt/HomeMirror
scwang90/SmartRefreshLayout
deeplearning4j/deeplearning4j
bazelbuild/bazel
roughike/BottomBar
JakeWharton/ActionBarSherlock
chrisbanes/cheesesquare
wasabeef/recyclerview-animators
chrisjenx/Calligraphy
apl-devs/AppIntro
openzipkin/zipkin
aosp-mirror/platform_frameworks_base
umano/AndroidSlidingUpPanel
eclipse/vert.x
google/agera
perwendel/spark
clojure/clojure
florent37/MaterialViewPager
Freelander/Android_Data
prestodb/presto
LMAX-Exchange/disruptor
apache/kafka
alibaba/vlayout
junit-team/junit4
square/dagger
JakeWharton/RxBinding
JackyAndroid/AndroidInterview-Q-A
Bearded-Hen/Android-Bootstrap
astuetz/PagerSlidingTabStrip
Curzibn/Luban
jeasonlzy/okhttp-OkGo
Yalantis/uCrop
81813780/AVLoadingIndicatorView
Tencent/VasSonic
mybatis/mybatis-3
dropwizard/dropwizard
cymcsg/UltimateRecyclerView
Bilibili/DanmakuFlameMaster
permissions-dispatcher/PermissionsDispatcher
skylot/jadx
google/physical-web
Netflix/SimianArmy
shuzheng/zheng
kaushikgopal/RxJava-Android-Samples
hongyangAndroid/okhttputils
xetorthio/jedis
google/guice
google/auto
gradle/gradle
Bigkoo/Android-PickerView
druid-io/druid
hongyangAndroid/Android-AutoLayout

nhaarman/ListView-Animations
mockito/mockito
code4craft/webmagic
brettwooldridge/HikariCP
JakeWharton/hugo
koush/ion
zhihu/Matisse
lingochamp/FileDownloader
lipangit/JiaoZiVideoPlayer
geeeeeeeeek/WeChatLucky-Money
aritraroy/UltimateAndroid-Reference
koral–/android-gif-drawable
dropwizard/metrics
alibaba/atlas
pedrovgs/EffectiveAndroid-UI
iBotPeaches/Apktool
DroidPluginTeam/Droid-Plugin
facebook/buck
rzwitserloot/lombok
pedant/sweet-alert-dialog
googlesamples/android-architecture-components
H07000223/FlycoTabLayout
rey5137/material
jhy/jsoup
wix/react-native-navigation
JakeWharton/u2020
futuresimple/android-floating-action-button
evant/gradle-retrolambda
JetBrains/intellij-community
lecho/hellocharts-android
emilsjolander/StickyList-Headers
springside/springside4
lucasr/twoway-view
amlcurran/ShowcaseView
wasabeef/glide–transformations
JakeWharton/timber
tbruyelle/RxPermissions
wyoufff/xUtils
google/j2objc
singwhatiwanna/dynamic-load-apk
square/otto
didi/VirtualAPK
ogaclejapan/SmartTab-Layout
ChrisRM/material-theme-jetbrains
OpenRefine/OpenRefine
facebook/rebound
naver/pinpoint
hankcs/HanLP
rengwuxian/MaterialEdit-Text
googlesamples/android-testing
antoniolg/androidmvp
gabrielemariotti/cardslib
trello/RxLifecycle
claritylab/lucida
koush/AndroidAsync
daimajia/AndroidImage-Slider
googlesamples/easy-permissions
apache/storm
etsy/AndroidStaggeredGrid
ikew0ng/SwipeBackLayout
sparklemotion/nokogiri
jgilfelt/SystemBarTint
thinkaurelius/titan
Trinea/android-common
wyoufff/xUtils3
swagger-api/swagger-core
GcsSloop/AndroidNote
dmytrodanylyk/circular-progress-button
YoKeyword/Fragmentation
daimajia/NumberProgressBar
pardom/ActiveAndroid
traex/RippleEffect
mcxiaoke/android-volley
vinc3m1/RoundedImageView
go-lang-plugin-org/go-lang-idea-plugin
eclipse/che
frogermcs/InstaMaterial
yixia/VitamioBundle
laobie/StatusBarUtil
apache/hadoop
square/okio
neo4j/neo4j
square/javapoet
CyberAgent/android-gpuimage
square/sqlbrite
Devlight/InfiniteCycleView-Pager
youth5201314/banner
Yalantis/Side-Menu.Android
facebook/facebook-android-sdk
Netflix/eureka

aporter/coursera-android
AndroidBootstrap/android-bootstrap
JakeWharton/NineOldAndroids
google/android-classyshark
daniulive/SmarterStreaming
commonsguy/cw-omnibus
kickstarter/android-oss
alibaba/freeline
scribejava/scribejava
facebook/litho
lwansbrough/react-native-camera
LitePalFramework/LitePal
danielzeller/Depth-LIB-Android-
Nightonke/BoomMenu
izzyleung/ZhihuDailyPurify
chentao0707/SimplifyReader
shwenzhang/AndResGuard
AsyncHttpClient/async-http-client
ACRA/acra
sockeqwe/mosby
crazycodeboy/TakePhoto
leolin310148/ShortcutBadger
stanfordnlp/CoreNLP
square/android-times-square
JakeWharton/DiskLruCache
apache/cassandra
ybq/Android-SpinKit
motianhuo/wechat
makovkastar/Floating-ActionButton
jdamcd/android-crop
roboguice/roboguice
Qihoo360/RePlugin
castorflex/SmoothProgress-Bar
Raizlabs/DBFlow
pxb1988/dex2jar
mission-peace/interview
processing/processing
Graylog2/graylog2-server
weibocom/motan
lzyzsd/JsBridge
google/closure-compiler
careercup/ctci
gitpitch/gitpitch
redisson/redisson
CarGuo/GSYVideoPlayer
Clans/FloatingActionButton
serge-rider/dbeaver
h6ah4i/android-advanced-recyclerview
gocd/gocd
naman14/Timber
alibaba/dexposed
aa112901/remusic
alibaba/ARouter
robolectric/robolectric
wequick/Small
evernote/android-job
prolificinteractive/material-calendarview
apache/zookeeper
Yalantis/Phoenix
markzhai/Android-PerformanceMonitor
XRecyclerView/XRecycler-View
NLPchina/ansj_seg
JustWayward/BookReader
Devlight/NavigationTabBar
hanks-zyh/HTextView
alibaba/canal
GrenderG/Toasty
orhanobut/dialogplus
diogobernardino/William-Chart
vondear/RxTools
openhab/openhab1-addons
JessYanCoding/MVPArms
Flipboard/bottomsheet
haifengl/smile
drakeet/Meizhi
dm77/barcodescanner
davemorrissey/subsampling-scale-image-view
b3log/solo
k9mail/k-9
grpc/grpc-java
java-native-access/jna
DreaminginCodeZH/Douya
Netflix/zuul
andkulikov/Transitions-Everywhere
romannurik/muzei
bingoogolapple/BGARefresh-Layout-Android
wasabeef/richeditor-android
hackware1993/MagicIndicator
apereo/cas
orfjackal/retrolambda
joelittlejohn/jsonschema2pojo
pedrovgs/AndroidWiFiADB
BoltsFramework/Bolts-Android
Vedenin/useful-java-links
hongyangAndroid/baseAdapter

mikepenz/Android-Iconics
chrisbanes/ActionBar-PullToRefresh
ben-manes/caffeine
square/moshi
spring-projects/spring-mvc-showcase
MyCATApache/Mycat-Server
kyleduo/SwitchButton
amitshekhariitbhu/Android-Debug-Database
JoanZapata/android-iconify
medcl/elasticsearch-analysis-ik
apache/zeppelin
JodaOrg/joda-time
orientechnologies/orientdb
alibaba/jstorm
Tapadoo/Alerter
airbnb/epoxy
Alluxio/alluxio
NanoHttpd/nanohttpd
saiwu-bigkoo/Android-ConvenientBanner
actorapp/actor-platform
codecentric/spring-boot-admin
antlr/antlr4
knightliao/disconf
bauerca/drag-sort-listview
grantland/android-autofittextview
google/error-prone
Yalantis/Context-Menu.Android
mcxiaoke/packer-ng-plugin
baoyongzhang/SwipeMenu-ListView
zaproxy/zaproxy
lightbend/config
avast/android-butterknife-zelezny
KeepSafe/TapTargetView
johncarl81/parceler
yanzhenjie/NoHttp
shardingjdbc/sharding-jdbc
dropbox/hackpad
Ramotion/folding-cell-android
yangfuhai/afinal
react-community/react-native-image-picker
yarolegovich/DiscreteScrollView
timehop/sticky-headers-recyclerview
hongyangAndroid/FlowLayout
HotBitmapGG/bilibili-android-client
udacity/Sunshine-Version-2
rengwuxian/RxJavaSamples
bluelinelabs/LoganSquare
real-logic/aeron
keyboardsurfer/Crouton
stephanenicolas/robospice
Atmosphere/atmosphere
rockerhieu/emojicon
ivacf/archi

B. Przykład przygotowania danych

W tym dodatku zaprezentowano na przypadkowo wybranym przykładzie jak przebiegł opisany w poprzednich rozdziale 4 proces przygotowywania danych i jaki jest dokładnie jego rezultat.

B.1. Projekt

Do przykładu wybrano projekt `JetBrains/kotlin`, który w trakcie pisania rozprawy zajmował 9. miejsce w rankingu projektów w języku Java według liczby gwiazdek zgromadzonych na Githubie (zob. Tabela 4.1). Repozytorium projektu znajduje się pod adresem `https://github.com/JetBrains/kotlin`.

B.2. Zmiany refaktoryzacyjne w projekcie

Po sklonowaniu projektu, komenda wyszukująca *commity* wprowadzające refaktoryzację (Listing 4.4) odnalazła 775 zmian poddających refaktoryzacji 4691 plików z rozszerzeniem `.java`. Kod klas przed i po każdym zidentyfikowanym *commicie* został zapisany w katalogu `results-java/JetBrains--kotlin`[1] zgodnie ze strukturą plików przedstawioną na Listingu 4.7.

B.3. Analiza wybranej zmiany

Jednym z *commitów* zidentyfikowanych jako refaktoryzacja była zmiana posiadająca *commit message* `Big refactoring of CallTranslator.` i *commit hash* rozpoczynający się od `003182f49`[2]. Wprowadza ona zmiany do pięciu plików z rozszerzeniem `.java`. Zgodnie ze strukturą plików przedstawioną na Listingu 4.7, kod przed i po wykonaniu tej zmiany został zapisany w katalogu z nazwą odpowiadającemu mu identyfikatorowi *commita*[3]. *commit message* tej zmiany zawiera poszukiwane słowo *refactoring* w pierwszej linii, więc jest ona użyta do dalszej analizy.

---

[1] `https://github.com/fracz/refactor-extractor/tree/master/results-java/JetBrains--kotlin` lub `https://fracz.github.io/phd/links/B.1.html`

[2] `https://github.com/JetBrains/kotlin/commit/003182f499651388aa3ca629752ef0207d52a412` lub `https://fracz.github.io/phd/links/B.2.html`

[3] `https://github.com/fracz/refactor-extractor/tree/master/results-java/JetBrains--kotlin/003182f499651388aa3ca629752ef0207d52a412` lub `https://fracz.github.io/phd/links/B.3.html`

B.4. Rozbiór syntaktyczny

Trzy z pięciu plików składających się na tę zmianę są plikami nowymi. Co za tym idzie – wszystkie metody w nich zawarte nie istniały wcześniej w kodzie, więc zgodnie z założeniami poczynionymi w sekcji 4.3 nie są one brane pod uwagę.

W dwóch pozostałych plikach – `CallTranslator.java` oraz `InlinedCallExpressionTranslator.java` zostało w sumie zmienionych 10 metod. Dla każdej z nich w katalogu `diffs` wewnątrz katalogu zmiany[4] powstał zgodnie z informacjami na Listingach 4.8 oraz 4.9 plik zawierający kod źródłowy oraz drzewo AST przed i po refaktoryzacji.

Na listingach B.1, B.2, B.3 oraz B.4 przedstawiono kolejno: kod przed, kod po, AST przed, AST po wykonaniu omawianej zmiany refaktoryzacyjnej dla metody `expressionAsFunctionCall` z klasy `CallTranslator`[5].

```
private JsExpression expressionAsFunctionCall() {
    assert callee != null;
    CallParameters expressionAsFunctionParameters = new CallParameters(
        ↪ null, callee);
    return methodCall(expressionAsFunctionParameters);
}
```

Listing B.1: Przykładowa metoda poddana refaktoryzacji: kod przed zmianą

```
@NotNull
private JsExpression expressionAsFunctionCall() {
    return methodCall(null, callee);
}
```

Listing B.2: Przykładowa metoda poddana refaktoryzacji: kod po zmianie

[4] `https://github.com/fracz/refactor-extractor/tree/master/results-java/JetBrains--kotlin/003182f499651388aa3ca629752ef0207d52a412/diffs` lub `https://fracz.github.io/phd/links/B.4.html`

[5] `https://github.com/fracz/refactor-extractor/blob/master/results-java/JetBrains--kotlin/003182f499651388aa3ca629752ef0207d52a412/diffs/CallTranslator.javaexpressionAsFunctionCall.txt` lub `https://fracz.github.io/phd/links/B.5.html`

```
MethodDeclaration
  BlockStmt
    AssertStmt
      BinaryExpr
        NameExpr
          SimpleName
        NullLiteralExpr
    ExpressionStmt
      VariableDeclarationExpr
        VariableDeclarator
          ObjectCreationExpr
            NullLiteralExpr
            NameExpr
              SimpleName
            ClassOrInterfaceType
              SimpleName
          SimpleName
          ClassOrInterfaceType
            SimpleName
    ReturnStmt
      MethodCallExpr
        NameExpr
          SimpleName
        SimpleName
  ClassOrInterfaceType
    SimpleName
  SimpleName
```

Listing B.3: Przykładowa metoda poddana refaktoryzacji: rozbiór syntaktyczny przed zmianą

```
MethodDeclaration
  BlockStmt
    ReturnStmt
      MethodCallExpr
        NullLiteralExpr
        NameExpr
          SimpleName
        SimpleName
  ClassOrInterfaceType
    SimpleName
  SimpleName
  MarkerAnnotationExpr
    Name
```

Listing B.4: Przykładowa metoda poddana refaktoryzacji: rozbiór syntaktyczny po zmianie

C. Szczegóły implementacji sieci neuronowej

Niniejszy dodatek zawiera szczegółowy opis implementacji dwukierunkowej rekurencyjnej sieci neuronowe, za pomocą której został zbudowany jakościowy model kodu źródłowego. Implementacje sieci neuronowych dla modeli aSCQM i rSCQM są podobne, dlatego pierwsza sekcja opisująca model bezwzględny jest szczegółowa, a druga – opisująca model względny – jedynie wskazuje różnice w stosunku do pierwszej.

C.1. Implementacja modelu aSCQM

Kod źródłowy, który został użyty do uczenia, testowania oraz ewaluacji modelu bezwzględnego jest umieszczony w pliku `model2.py`[1] w repozytorium [95]. W tej sekcji został on szczegółowo omówiony.

Na początku skryptu zdefiniowane są możliwe argumenty wykonania, które w prosty sposób umożliwiły dostosowanie parametrów uczenia sieci neuronowej bez konieczności modyfikowania kodu źródłowego. Ich lista została przedstawiona poniżej.

- `datasetName` – określa zbiór danych wejściowych (z katalogu `input`) do analizy w danym wykonaniu (zbiory danych opisano w sekcji 4.5).
- `--numHidden` – określa rozmiar stanu ukrytego w komórce LSTM, tj. liczbę jednostek ukrytych. Parametr ten określa zdolność modelowania cech sekwencji podawanych na wejściu. Im większa jest liczba jednostek tym więcej cech model będzie w stanie zapamiętać. Przesadne jej zwiększenie grozi jednak zbyt dobrym dopasowaniem się modelu do danych treningowych (przeuczeniem). Warto też zauważyć, że zwiększanie tego parametru znacząco wpływa na wydłużenie czasu trenowania modelu.
- `--steps` – liczba iteracji algorytmu optymalizującego model. Po jej osiągnięciu skrypt zakończy działanie i zapisze osiągnięty rezultat.
- `--batchSize` – liczba sekwencji wejściowych podawana na sieć neuronową w trakcie jednej iteracji. Domyślna wartość to 64.
- `--displayStep` – liczba iteracji, po której postęp uczenia – aktualna trafność (ang. *accuracy*) i koszt (ang. *loss*) – jest wyświetlana na standardowym wyjściu skryptu. Wartości te posłużyły do stworzenia wykresów przebiegu uczenia w rozdziale 6.

[1] `https://github.com/fracz/code-quality-tensorflow/blob/master/model2.py` lub `https://fracz.github.io/phd/links/C.1.html`

- `--tokensCount` – rozmiar alfabetu wejściowego. Parametr ten był przydatny podczas początkowych testów, gdy ustalano jeszcze rodzaj wejścia. Podczas dalszych prób uczenia modelu wartość ta zawsze była zgodna z liczbą możliwych tokenów rozbioru syntaktycznego i była równa 76 (zob. sekcję 4.6).

Przykładowe uruchomienia modelu zaprezentowano na listingach C.1 i C.2. Na listingu C.2 dodatkowo pokazano, jak zapisywano postępy przebiegu uczenia do pliku, by móc je następnie wykorzystać do przygotowania wizualizacji w rozprawie.

```
python model2.py 100-diff10-java --steps 50000 --numHidden 128 --
    ↪ tokensCount=76 --batchSize=32
```

Listing C.1: Uruchomienie uczenia modelu aSCQM – przykład 1

```
python model2.py 200-diff10to50-java --steps 20000 --numHidden 256 --
    ↪ tokensCount=76 &> logs/model2-200-diff10to50-java.log
```

Listing C.2: Uruchomienie uczenia modelu aSCQM – przykład 2

Po odczytaniu parametrów, na podstawie nazwy zbioru wejściowego, liczby jednostek w komórce LSTM oraz liczby iteracji tworzono katalog, do którego po zakończeniu uczenia trafi wytrenowany model. Przykładowo, dla wykonania z listingu C.1 był to katalog `trained/model2/100-diff10-java/50000-128`, a dla wykonania z listingu C.2 – `trained/model2/200-diff10to50-java/20000-256`.

Następnie do działania wkracza klasa `RefactorDataset`, która jest odpowiedzialna za odczytanie danych wejściowych i przygotowanie ich do użycia w procesie uczenia modelu. Zgodnie z opisem danych wejściowych dla modelu bezwzględnego (sekcja 4.5.1), w konstruktorze odczytuje ona kolejno pliki `input.csv`, `labels.csv` oraz `lengths.csv` ze wskazanego parametrem zbioru danych wejściowych. Za pomocą funkcji `genfromtxt` z biblioteki NumPy [104] tworzona jest z nich macierz.

Kolejnym krokiem jest rozpoznanie długości sekwencji wejściowych – w tym celu sprawdzana jest długość pierwszej z nich z pliku `input.csv`. Uzyskany rezultat reprezentuje długość wszystkich sekwencji, ponieważ na etapie przygotowania danych zadbano o dopełnienie krótszych rozbiorów syntaktycznych zerami tak, aby wszystkie miały tą samą długość. Liczba znaczących tokenów w każdej sekwencji jest przekazana w pliku `lengths.csv`. Ostatnią operacją wykonywaną podczas tworzenia reprezentacji danych wejściowych w pamięci jest podzielenie danych wejściowych na zbiór treningowy i zbiór testowy w stosunku 85%:15%. Jeśli wszystkie powyższe operacje zostały zakończone powodzeniem, proces wczytywania danych zostaje zakończony, co zostaje

potwierdzone odpowiednim komunikatem a skrypt przechodzi do budowy sieci neuronowej.

Za pomocą funkcji `placeholder` z *frameworka* TensorFlow tworzone są tensory, które przechowają kolejno: dane wejściowe (sekwencje), oczekiwane klasyfikacje wyjściowe oraz długości kolejnych wierszy w wejściowym zbiorze danych dla aktualnej iteracji. Tworzone są tu też osadzenia tokenów (ang. *token embeddings*), których rozmiar zależy od wielkości przetwarzanego alfabetu rozbioru syntaktycznego.

Kolejnym elementem w kodzie źródłowym modelu aSCQM jest funkcja `BiRNN`, która jest odpowiedzialna za stworzenie struktury dwukierunkowej rekurencyjnej sieci neuronowej. Pierwszą operacją jest stworzenie dwóch komórek LSTM o zadanej (argumentem wykonania) liczbie jednostek ukrytych. Pierwsza z nich – `lstm_fw_cell` – będzie starać się odnaleźć zależności przy przeglądaniu tokenów zgodnie z kolejnością podaną w sekwencji wejściowej. Druga – `lstm_bw_cell` – będzie przeglądać tokeny w kolejności odwróconej. Obydwie komórki są instancją klasy `BasicLSTMCell` z TensorFlow i służą do zbudowania modelu dwukierunkowej rekurencyjnej sieci neuronowej, która jest główną składową klasyfikatora. Jest ona tworzona w następnej linii kodu za pomocą funkcji `static_bidirectional_rnn`.

Wynik przetworzenia sekwencji wejściowych (`inputs`) jest tutaj zapisywany jako tensor `outputs`. Oprócz przekazania stworzonych wcześniej komórek LSTM, wykorzystany tutaj jest także parametr `sequence_length`, który umożliwia przekazanie długości kolejnych sekwencji w danych wejściowych tak, by dołożone sztucznie dopełnienie zerami nie było brane pod uwagę w trakcie trenowania. Po kolejnych przekształceniach, otrzymywany jest tensor reprezentujący wyjście dwukierunkowej sieci neuronowej.

Wyjście z sieci rekurencyjnej jest mnożone przez macierz wag warstwy klasyfikującej, a następnie zapisywane w tensorze `logits`. Kolejną operacją jest przekształcenie logitów do rozkładu prawdopodobieństwa obu klas. Realizuje to funkcja aktywacji `softmax`. Dzięki temu wartości w tensorze wyjściowym sumują się do 1 i reprezentują klasyfikację zadanego na wejściu kodu źródłowego. Spełnia to założone w sekcji 3.5 wymagania rezultatu modelu, tj. $P_G + P_B = 1$. Wynikowy tensor zapisywany jest w zmiennej `prediction` i reprezentuje odpowiedź dla aktualnie analizowanego przykładu.

Wynik klasyfikacji jest następnie porównywany do oczekiwanego rezultatu za pomocą funkcji `softmax_cross_entropy_with_logits`. Uzyskany w ten sposób błąd uczenia zapisywany jest w tensorze `loss_op` i jest minimalizowany za pomocą optymalizatora Adam [105]. Końcowy rezultat zapisywany jest w zmiennej o nazwie `train_op` reprezentującej pojedynczy krok uczenia modelu.

Przed rozpoczęciem sesji TensorFlow, tworzony jest jeszcze tensor `accuracy`, który pozwoli na pomiar postępu trenowania modelu. Dodatkowo, tworzona jest tutaj instancja klasy `train.Saver`, która pozwoli zapisać, i docelowo odczytać, wytrenowany model.

Używając konstrukcji `with` oraz funkcji `Session` z TensorFlow w końcowej części skryptu rozpoczynana jest sesja trenowania zbudowanego wcześniej modelu. Najpierw sprawdzane jest, czy model ma być przywrócony z poprzedniego etapu uczenia. Jeśli tak, obiekt `Saver` jest wykorzystywany by go odczytać. W przeciwnym razie trenowanie rozpoczyna się od początkowego stanu modelu.

W trakcie uczenia, pętla podająca sekwencje wejściowe sieci neuronowej wykonywana jest ustaloną (parametrem wykonania skryptu) liczbę razy. W każdej iteracji ze zbioru danych wejściowych pobierany jest kolejny podzbiór sekwencji wejściowych, ich długości oraz oczekiwane rezultaty klasyfikacji. Stworzona wcześniej sesja jest uruchamiana i wykonuje zdefiniowaną operację optymalizacji modelu (`train_op`). Jako dane wejściowe (`feed_dict`) przekazywane są tensory wypełnione danymi przeznaczonymi dla danej iteracji. W ten sposób wykonywanie kolejnych iteracji doprowadza do minimalizacji błędu uczenia i w efekcie do zwiększenia dokładności przewidywań modelu.

Jeśli dana iteracja powinna także skutkować wypisaniem aktualnego postępu trenowania na standardowe wyjście, sesja wylicza wartości `loss_op` oraz `accuracy` dla zbioru testowego. Informacje te są wypisywane a skrypt kontynuuje trenowanie modelu.

Po zakończeniu trenowania lub po poprawnym odczycie nauczonego wcześniej modelu sesja TensorFlow uruchamiana jest ponownie. Uzyskana w ten sposób wartość `accuracy` dla zbioru testowego reprezentuje końcową trafność modelu dla użytych w danym wykonaniu sekwencji wejściowych. Jest ona wypisywana na standardowe wyjście programu, po czym skrypt kończy swoje działanie i zwalnia zajęte przez siebie zasoby.

C.2. Implementacja modelu rSCQM

Kod źródłowy, który został użyty do uczenia, testowania oraz ewaluacji modelu względnego jest umieszczony w pliku `model4.py`[2] w repozytorium [95]. Ze względu

[2] `https://github.com/fracz/code-quality-tensorflow/blob/master/model4.py` lub `https://fracz.github.io/phd/links/C.2.html`

na znaczne podobieństwo implementacji modeli aSCQM i rSCQM, w tej sekcji wskazano tylko różnice w stosunku do opisu modelu bezwzględnego z poprzedniej sekcji C.1.

Parametry wykonania skryptu uczącego model rSCQM są takie same jak w przypadku modelu aSCQM, więc przykłady wykonań z listingów C.1 oraz C.2 są również przykładami uruchomienia dla rSCQM. Należy zmienić jedynie nazwę skryptu z `model2.py` na `model4.py`.

Klasa `RefactorDataset` w katalogu z danymi wejściowymi oczekuje teraz plików `input-before.csv` i `input-after.csv` oraz `lengths-before.csv` i `lengths-after.csv`, zgodnie z opisem danych wejściowych dla modelu rSCQM w sekcji 4.5.2. Ponadto, przy zwracaniu kolejnych podzbiorów danych wejściowych metoda `next` w połowie przypadków odwraca próbki, podając na wejście sieci neuronowej najpierw rozbiór syntaktyczny metody po refaktoryzacji a następnie metody przed refaktoryzacją. Dzięki temu sieć neuronowa nie odkrywa błędnej informacji, że kod podawany na wejściu jako pierwszy zawsze jest kodem niższej jakości.

Zamiast jednego tensora przechowującego wejście dla sieci neuronowej, w modelu rSCQM są dwa tensory wejściowe: dla sekwencji przed i po zmianie. Taka sama sytuacja ma miejsce dla tensorów przechowujących długości poszczególnych sekwencji wejścia, ponieważ kod przed zmianą może być (i zazwyczaj jest) innej długości niż kod po zmianie.

Za pomocą funkcji `BiRNN` przedstawionej w poprzedniej sekcji C.1 tworzone są teraz dwie sieci neuronowe analizujące przykłady kodu przed i po zmianie. Ich rezultaty są zapisywane odpowiednio w tensorach `outputs_before` i `outputs_after`. Tensory te są następnie konkatenowane (łączone). Rezultat stanowi wejście do klasyfikatora skonstruowanego podobnie, jak w modelu aSCQM.

W ten sposób, z dwóch próbek rozbiorów syntaktycznych i z dwóch dwukierunkowych rekurencyjnych sieci neuronowych otrzymywana jest klasyfikacja przedstawionej zmiany kodu źródłowego. Oczywiście, zmienia się nieco sposób podawania danych wejściowych: dla tak zbudowanego modelu należy wypełnić większą liczbę tensorów. Jednakże sposób optymalizacji funkcji kosztu oraz sposób wykonywania sesji jest analogiczny do implementacji w modelu aSCQM.

D. Testowanie poprawności implementacji modelu sieci neuronowej

Od samego początku zakładano, że kod przed refaktoryzacją ma niższą jakość niż kod po refaktoryzacji. Tak w rzeczywistości powinno być, ale nawet rezultaty otrzymane z doświadczenia opisanego w rozdziale 5 pokazały, że nie zawsze jest to zgodne z prawdą.

Dlatego, po niezadowalających wynikach przy pierwszych próbach uczenia modelu postanowiono sprawdzić, czy zbudowana sieć neuronowa w ogóle jest w stanie przyjąć jakąkolwiek wiedzę z tak zaprojektowanego wejścia.

Podjęto trzy próby stworzenia coraz bardzie skomplikowanej, ale jednoznacznej metryki jakości kodu źródłowego i sprawdzono rezultat uczenia na tak przygotowanych danych. Do analizy były wykorzystywane te same metody, które były wybrane w poprzedniej części, ale ich klasyfikacja była przeprowadzona według zaproponowanych metryk, z porzuceniem zasady „kod przed refaktoryzacją jest niższej jakości niż kod po refaktoryzacji". Oczekiwano, że zaprojektowana sieć neuronowa będzie w stanie odkryć zaprojektowane metryki i poprawnie klasyfikować kod. To potwierdzi poprawność zaimplementowanego modelu i pozwoli skupić się na dobrze odpowiednich parametrów lub lepszym przygotowaniu danych wejściowych.

W testowych próbach modelu ograniczono się do metod o maksymalnej długości 100 tokenów i modelu bezwzględnego aSCQM. Zgodnie z tabelą 4.3, na wejściu modelu bezwzględnego uzyskano 40452 metody[1].

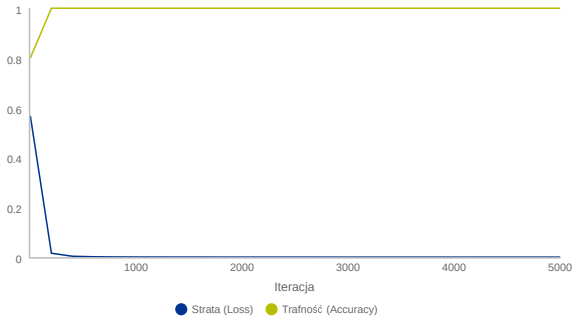
D.1. Negatywne traktowanie wyrażenia warunkowego

Pierwszą – trywialną – metryką klasyfikującą zebrane metody było negatywne traktowanie wyrażenia warunkowego.

> Jeśli rozbiór syntaktyczny zawiera węzeł `IfStmt` lub `ConditionalExpr`, reprezentujący kolejno wyrażenie warunkowe `if` oraz warunkowy operator potrójny (z ang. *ternary*) – traktuj kod jako niskiej jakości. W przeciwnym razie – traktuj kod jako wysokiej jakości.

[1] podwojona liczba refaktoryzacji, ponieważ w modelu bezwzględnym jedna zmiana tworzy dwie próbki

Według powyższej reguły 34526 metod zostało sklasyfikowane jako kod niskiej jakości (85%).

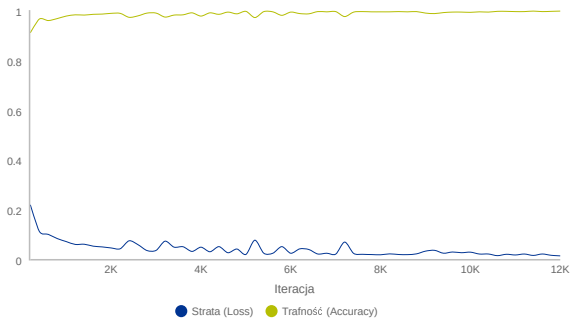


Rys. D.1: Rezultat uczenia modelu bezwzględnego w uproszczonej metryce jakości kodu traktującej negatywnie wyrażenia warunkowe

Okazuje się, że dla tak przygotowanych danych model potrzebuje tylko kilkadziesiąt iteracji uczenia by osiągnąć całkowitą skuteczność w rozpoznawaniu przygotowanej definicji kodu niskiej jakości. Wynik ten nie był zaskakujący – dla tak zdefiniowanego pojęcia jakości kodu wystarczy rzucić okiem na implementację by wyznaczyć klasyfikację, którą powinien zwrócić model. Ten krok potwierdził jednak wstępnie poprawność implementacji modelu. Pełny przebieg uczenia zaprezentowano na rysunku D.1.

D.2. Wirtualna metryka bezwzględna

Po uzyskaniu pozytywnego rezultatu przy zastosowaniu trywialnej metryki, podjęto próbę uczenia modelu klasyfikacją według bardziej złożonej reguły. W tym celu przypisano wybranym konstrukcjom językowym stały koszt (rozumiany jako „koszt zrozumienia danej konstrukcji przez programistę”, idąc za przykładem metryki Cognitive Complexity [39] wspomnianej w sekcji 2.4). Ustalone koszty poszczególnych konstrukcji językowych przedstawiono w tabeli D.1. Metryka ta jest bardziej skomplikowana niż wariant pierwszy. Jeśli tak byłaby rozumiana jakość kodu źródłowego, programiście wystarczyłaby krótka analiza implementacji by stwierdzić czy jest ona czytelna. Model nie powinien mieć więc żadnych problemów z rozpoznaniem nałożonych reguł.



Rys. D.2: Rezultat uczenia modelu bezwzględnego w uproszczonej metryce jakości kodu zliczającej koszt wybranych konstrukcji językowych

Druga metryka testowa jest zdefiniowana następująco.

> Jeśli koszt metody liczony według tabeli D.1 przekroczy 7, traktuj kod jako niskiej jakości. W przeciwnym razie – traktuj kod jako wysokiej jakości.

Tabela D.1: Koszt poszczególnych konstrukcji językowych w przyjętej wirtualnej metryce kodu źródłowego

Typ węzła AST	**Koszt**
`IntegerLiteralExpr`	1
`StringLiteralExpr`	1
`ConditionalExpr`	2
`ForeachStmt`	2
`IfStmt`	2
`CastExpr`	3
`ForStmt`	3
`WhileStmt`	3
`SwitchStmt`	4

Wartość 7 ustalono tak, aby reguła podzieliła posiadany zbiór danych w stosunku 40%:60%. W efekcie, 17181 metod (42%) zostało sklasyfikowanych jako kod niskiej jakości.

Według tak przygotowanych danych, na zbiorze testowym model wahał się bardziej niż przy pierwszej – trywialnej – metryce, ale i tak po kilkudziesięciu iteracjach był w stanie rozpoznać przygotowaną regułę klasyfikacji kodu i jego trafność nie spadała poniżej 98%. Przebieg uczenia zaprezentowano na rysku D.2.

D.3. Wirtualna metryka zależna od długości metody

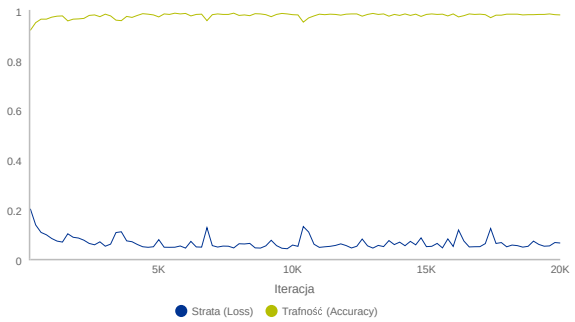
Metryka zaproponowana w sekcji D.2 nie traktowała sprawiedliwie dłuższych metod. Jeśli – przykładowo – metoda posiada 50 konstrukcji językowych, sprawiedliwa klasyfikacja powinna pozwolić jej na osiągnięcie wyższego koszt niż metodzie, która zawiera ich tylko 20. Dlatego finalną metryką rozstrzygająca o poprawnej implementacji modelu sieci neuronowej było uzależnienie kosztu, przy którym metoda była klasyfikowana jako niskiej jakości od jej długości. Poprawne wytrenowanie modelu

dla takiej metryki przekonało już autora rozprawy, że problemów z osiągnięciem pożądanej trafności jakościowego modelu należy szukać w innym miejscu.

Ostatnią testową metrykę można zdefiniować w następujący sposób.

> Jeśli stosunek kosztu metody liczonego według tabeli D.1 do długości metody wyrażonej w liczbie węzłów AST przekroczy 0.15, traktuj kod jako niskiej jakości. W przeciwnym razie – traktuj kod jako wysokiej jakości.

Analogicznie do poprzedniego przypadku – wartość 0.15 została ustalona empirycznie tak, by uzyskać zrównoważony podział danych wejściowych. Przy zastosowaniu tej reguły, 16533 metody (41%) zostało sklasyfikowane jako kod niskiej jakości.



Rys. D.3: Rezultat uczenia modelu bezwzględnego w uproszczonej metryce jakości kodu zliczającej wybrane konstrukcje językowe i klasyfikującej kod w zależności od jego długości

Również w tym przypadku model poradził sobie z odpowiednim klasyfikowaniem zbioru testowego. Przebieg uczenia przedstawiono na rysunku D.3.

Na tym etapie zakończono testowanie zaimplementowanego modelu i powrócono do prób przekazania wiedzy do modelu na podstawie danych sklasyfikowanych według oryginalnych założeń.

E. Opis implementacji mikrousługi udostępniającej klasyfikacje jakości kodu z użyciem SCQM

Pierwotna postać implementacji modelu SCQM, która pozwoliła wytrenować go za pomocą zgromadzonych danych została opisana w dodatku C. Dla produkcyjnej wersji modelu głównym celem jest możliwość przyjęcia dowolnej próbki kodu źródłowego lub zmiany i uzyskanie w odpowiedzi prawdopodobieństw P_G i P_B, świadczących o ich jakości.

Aby maksymalnie ukryć szczegóły implementacji, podjęto decyzję o udostępnieniu interfejsu modelu w postaci REST API. Jest to popularny sposób udostępniania funkcjonalności z różnego rodzaju mikroserwisów – i właśnie tak autor rozprawy chciałby postrzegać wdrożeniową wersję modelu SCQM. Dzięki temu dowolny klient potrafiący wykonać żądanie HTTP będzie w stanie sklasyfikować dowolny kod źródłowy.

E.1. Mikrousługa wykonująca rozbiór syntaktyczny

Zanim przystąpiono do modyfikacji modelu w języku Python, konieczne było dostarczenie implementacji parsera dla języka Java tak, by możliwe było przesyłanie w zapytaniu kodu źródłowego bez żadnych przekształceń. To uruchomiona aplikacja SCQM powinna być odpowiedzialna za odpowiednie przekształcenie kodu źródłowego i przekazanie uzyskanej postaci wejścia dalej do modelu w TensorFlow.

Dotychczasowa implementacja parsera, opisana w sekcji 4.3, została wzbogacona o *framework* Spring Boot[1], który umożliwia uruchomienie REST API w aplikacjach opartych o język Java. Z aplikacji parsera udostępniany jest tylko jeden endpoint:

- `POST /parse` – umożliwia przesłanie kodu źródłowego klasy w Javie; w odpowiedzi klient uzyskuje listę metod odnalezionych w klasie wraz z ich ztokenizowanym rozbiorem syntaktycznym.

Każdy element tablicy będącej odpowiedzią parsera zawiera klucz `methodName` z nazwą metody, `sourceCode` z kodem źródłowym oraz `tokens` zawierający tablicę zawierającą elementy jego rozbioru syntaktycznego, zamienione na liczbowe tokeny zgodnie z opisiem metodyki przygotowania wejścia modelu w sekcji 4.5.

Kod źródłowy parsera wraz z REST API znajduje się w repozytorium [101] w katalogu `java-parser`.

[1] `https://spring.io/projects/spring-boot`

Listing E.1 zawiera przykład poprawnego żądania do parsera. Listing E.2 zawiera odpowiedź na to żądanie.

```
POST http://localhost:8080/parse
{
  "source": "public class Dog {\n\tvoid bark() {\n\t\tSystem.out.println
     ↪ ("Hau");\n\t}\n\n\tvoid sleep() {\n\t\tSystem.exit();\n\t}\n}\n
     ↪ "
}
```

Listing E.1: Przykład poprawnego żądania do REST API wystawionego przez aplikację z parserem

```
[
  {
     "methodName": "sleep",
     "tokens": [74, 56, 74, 20, 74, 8, 74, 73, 4, 36, 4, 75, 75, 75, 74, 34,
        ↪ 75, 4, 75],
     "sourceCode": "void sleep() {\n System.exit();\n}"
  },
  {
    "methodName": "bark",
    "tokens": [74, 56, 74, 20, 74, 8, 74, 73, 37, 4, 74, 48, 4, 36, 4, 75,
       ↪ 75, 75, 75, 74, 34, 75, 4, 75],
    "sourceCode":"void bark() {\n System.out.println(\"Hau\");\n}"
  }
]
```

Listing E.2: Przykład odpowiedzi z REST API wystawionego przez aplikację z parserem

E.2. Mikrousługa klasyfikująca kod za pomocą modelu SCQM

Jednym z popularnych frameworków umożliwiających uruchomienie REST API w dowolnej aplikacji pisanej w języku Python jest Flask[2]. Ze względu na prostotę jego użycia, wdrożono go do implementacji modelu SCQM.

[2] http://flask.pocoo.org

Skrypt `scqm/scqm.py`[3] w repozytorium [101] jest połączeniem implementacji aSCQM i rSCQM, do której dołożono także wsparcie *frameworka* Flask i wystawiono z niego REST API umożliwiające przesłanie kodu źródłowego do klasyfikacji. W katalogu `trained/code-fracz-645`[4] znajdują się parametry wytrenowanego modelu dla najlepszej konfiguracji danych wejściowych i parametrów uczenia opisanych w rozdziale 6. W trakcie uruchamiania skryptu modele odczytują te parametry, inicjalizując nimi zbudowany w TensorFlow model. Jeśli to się uda, Flask uruchamia REST API, oferujące następujące punkty końcowe:

- `GET /` – udostępnia aplikację internetową zawierającą formularz do przesyłania danych i prezentacji odpowiedzi (zob. rysunki E.2, E.1, E.4 i E.3)
- `POST /ascqm` – umożliwia przesłanie kodu źródłowego jednej klasy; kod ten zostanie sklasyfikowany przy użyciu aSCQM i w odpowiedzi klient otrzyma klasyfikację przesłanego kodu oraz inne szczegółowe informacje (zob. listingi E.3 i E.4)
- `POST /rscqm` – umożliwia przesłanie kodu źródłowego dwóch klas, reprezentujących zmianę w kodzie; zmiana ta zostanie sklasyfikowana przy użyciu rSCQM i w odpowiedzi klient otrzyma klasyfikację przesłanej zmiany oraz inne szczegółowe informacje (zob. listingi E.5 i E.6)

Po otrzymaniu kodu źródłowego do sklasyfikowania na endpoincie `/ascqm` lub `/rscqm`, skrypt w Pythonie odpytuje endpoint parsera w celu otrzymania listy metod w zadanej klasie oraz ich rozbioru syntaktycznego. Następnie, każda z uzyskanych w ten sposób sekwencji z rozbiorem syntaktycznym jest dopełniana zerami tak, aby jej długość była równa maksymalnej długości sekwencji wejścia, która była ustalona w trakcie uczenia. Zależy więc ona od parametrów wytrenowanego modelu, z którym uruchomiono aplikację. Dłuższe metody na tym etapie są odrzucane. Ponadto, w przypadku wariantu względnego rSCQM, metody które istnieją zarówno w kodzie klasy przed i po zadanej zmianie są łączone w pary, które stworzą wejście dla modelu. Metody, które zostały usunięte lub dodane w ramach zmiany nie są uwzględniane w modelu względnym (zob. ograniczenia modelu opisane w sekcji 3.2).

[3] `https://github.com/fracz/scqm/blob/master/scqm-model/scqm.py` lub `https://fracz.github.io/phd/links/E.3.html`

[4] `https://github.com/fracz/scqm/tree/master/trained/code-fracz-645` lub `https://fracz.github.io/phd/links/E.4.html`

Następnie, dla każdej rozpoznanej przez parser i nieodrzuconej przez powyższe warunki metody lub pary w zadanym kodzie źródłowym, odpowiedni model jest uruchamiany w celu obliczenia wartości tensora `prediction` (zob. dodatek C). Dzięki temu dla każdej próbki uzyskiwane są wartości P_G i P_B i są zwracane w odpowiedzi.

E.3. Konfiguracja uruchomieniowa prototypu

Pomimo możliwości uruchomienia implementacji opisywanej mikrousługi na dowolnym systemie operacyjnym, wymagałoby to od użytkownika instalacji i konfiguracji odpowiednich wersji języków Python oraz Javy wraz ze wszystkimi wymaganymi zależnościami. Dlatego podjęto decyzję o dalszym uproszczeniu sposobu uruchomienia stworzonego modelu za pomocą kontenerów Docker[5].

Konfiguracja kontenerów znajduje się w katalogu `docker` w repozytorium [101]. Przygotowano dwa kontenery:

- `scqm-parser`, zawierający środowisko aplikacji udostępniającej funkcjonalność parsera w języku Java za pomocą Spring Boot, oraz
- `scqm-model`, zawierający aplikację udostępniającą model SCQM za pomocą *frameworka* Flask.

Obydwa kontenery zostały połączone w jedną aplikację za pomocą Docker Compose[6]. Kontener z parserem jest wykorzystywany tylko wewnątrz skryptu z modelem, dlatego nie wystawiono dla niego żadnych portów sieciowych. Kontener z modelem natomiast wystawia na zewnątrz port 7276[7], pod którym dostępne jest REST API uruchomione przez *framework* Flask.

E.4. API

Uruchomiona mikrousługa udostępnia REST API, które może być wykorzystane do otrzymania klasyfikacji kodu źródłowego dowolnej klasy napisanej w języku Java. W tym celu udostępniono dwa punkty końcowe `/ascqm` oraz `/rscqm`, które pozwalają na przesłanie odpowiednio jednej lub dwóch klas (zmiany) i uzyskania ich klasyfikacji. Przykłady żądania i odpowiedzi z klasyfikacją kodu w klasie dla modelu aSCQM

[5] `https://www.docker.com`

[6] `https://docs.docker.com/compose/`

[7] nie jest to port przypadkowy; pisząc na klawiaturze numerycznej telefonu „SCQM” otrzymamy dokładnie wartość 7276

przedstawiono na listingach E.3 i E.4. Analogicznie, listingi E.5 i E.6 przedstawiają żądanie i odpowiedź dla modelu rSCQM.

```
POST http://localhost:7276/ascqm
{
  "source": "public class Dog {\n\tvoid bark() {\n\t\tSystem.out.println
     ↪ ("Hau");\n\t}\n\n\tvoid sleep() {\n\t\tSystem.exit();\n\t}\n}\n
     ↪ "
}
```

Listing E.3: Przykład poprawnego żądania do REST API modelu aSCQM

```
[
  {
    "methodName": "sleep",
    "tokens": [74, 56, 74, 20, 74, 8, 74, 73, 4, 36, 4, 75, 75, 75, 74, 34,
       ↪ 75, 4, 75],
    "sourceCode": "void sleep() {\n System.exit();\n}",
    "prediction": [0.9346317648887634, 0.0653681755065918]
  },
  {
    "methodName": "bark",
    "tokens": [74, 56, 74, 20, 74, 8, 74, 73, 37, 4, 74, 48, 4, 36, 4, 75,
       ↪ 75, 75, 75, 74, 34, 75, 4, 75],
    "sourceCode": "void bark() {\n System.out.println(\"Hau\");\n}",
    "prediction": [0.949866771697998, 0.05013318732380867]
  }
]
```

Listing E.4: Przykład odpowiedzi z predykcją dla modelu aSCQM

```
POST http://localhost:7276/rscqm
{
  "sourceBefore": "public class Dog {\n\nvoid bark() {\nSystem.out.
     ↪ println(\"Hau\");\n}\n\nvoid sleep() {\nSystem.exit();}\n\n}",
  "sourceAfter": "public class Dog {\nprivate boolean sleeping = false;\
     ↪ nvoid bark() {\nif (!sleeping) {\n System.out.println(\"Hau\");\
     ↪ n}\n}\n\nvoid sleep() {\nSystem.exit();}\n\n}"
}
```

Listing E.5: Przykład poprawnego żądania do REST API modelu rSCQM

```
{
  "astBefore": [
    // Parser response for code "before"; skipped for clarity
  ],
  "astAfter": [
     // Parser response for code "after"; skipped for clarity
  ],
  "predictions": [
    {
      "methodName": "bark",
      "methodBefore": {
        "methodName": "bark",
        "tokens": [74, 56, 74, 20, 74, 8, 74, 73, 37, 4, 74, 48, 4, 36, 4,
            ↪ 75, 75, 75, 75, 74, 34, 75, 4, 75],
        "sourceCode": "void bark() {\n System.out.println(\"Hau\");\n}"
      },
      "methodAfter": {
        "methodName": "bark",
        "tokens": [74, 56, 74, 20, 74, 66, 74, 26, 36, 4, 75, 74, 20, 74, 8,
            ↪ 74, 73, 37, 4, 74, 48, 4, 36, 4, 75, 75, 75, 75, 75, 75, 74,
            ↪ 34, 75, 4, 75],
        "sourceCode": "void bark() {\n if (!sleeping) {\n System.out.
            ↪ println(\"Hau\");\n }\n}"
      },
      "prediction": [0.9768826961517334, 0.02311730943620205]
    }
  ]
}
```


Listing E.6: Przykład odpowiedzi z predykcją dla modelu rSCQM

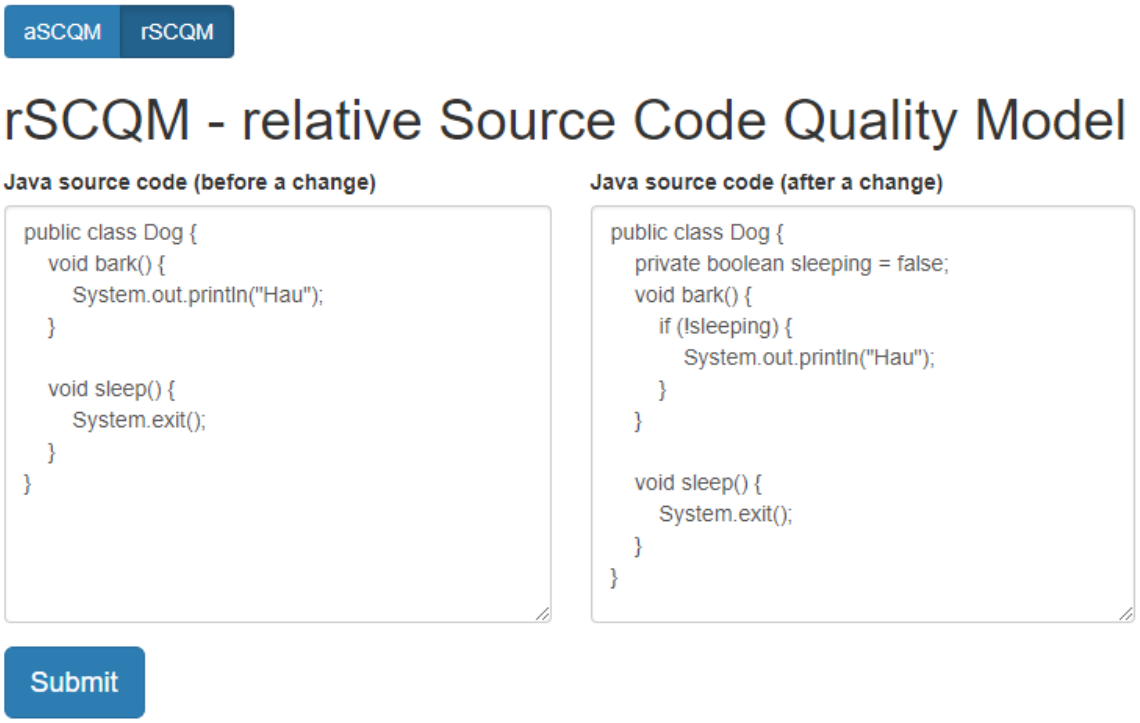
Warto zwrócić uwagę, że na listingach z odpowiedziami E.4 i E.6 są zawarte predykcje wyłącznie dla metod, które mogłybyć sklasyfikowane przez model (tj. nie były za długie, a w przypadku rSCQM także istniały przed i po zmianie). Odpowiedzi te generowane są bez zauważalnego opóźnienia (poniżej 500ms), więc predykcje te mogą być wykorzystywane w rozwiązaniach, które wymagają szybkiej klasyfikacji kodu.

E.5. Interfejs użytkownika

Poza API, do łatwego wykorzystania stworzonego modelu została dostarczona prosta aplikacja webowa, która umożliwia przesłanie kodu do sklasyfikowania i odczytania odpowiedzi w przejrzystej dla użytkownika formie.

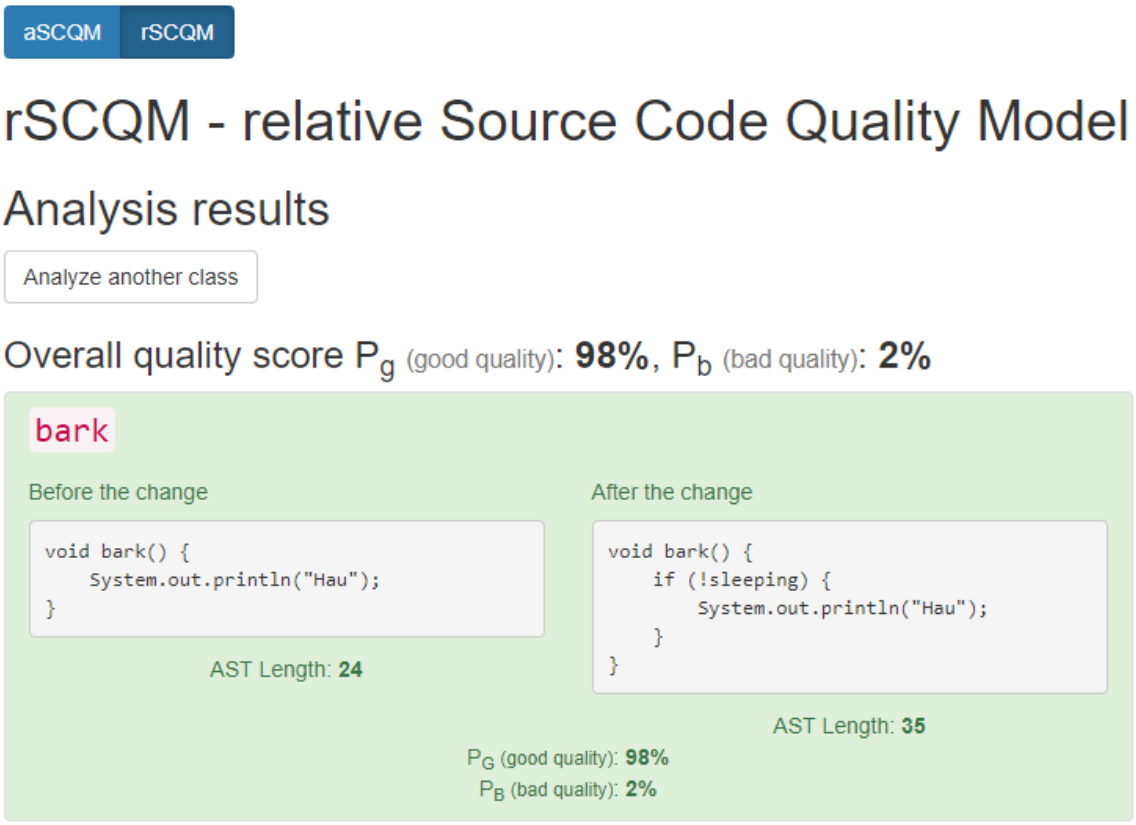
Aplikacja wykorzystuje te same punkty końcowe, które zostały opisane powyżej. Jest ona dostępna po uruchomieniu modelu pod adresem `http://localhost:7276`. Do jej otwarcia wystarczy użyć dowolnej przeglądarki internetowej.

Rysunki E.1 i E.2 przedstawiają formularze, do których można wstawić kod źródłowy. Kolejne rysunki E.3 i E.4 przedstawiają rezultaty takiej klasyfikacji po przesłaniu formularzy. Wartości P_G i P_B podane dla całej klasy są średnią arytmetyczną tych samych wartości dla poszczególnych metod.

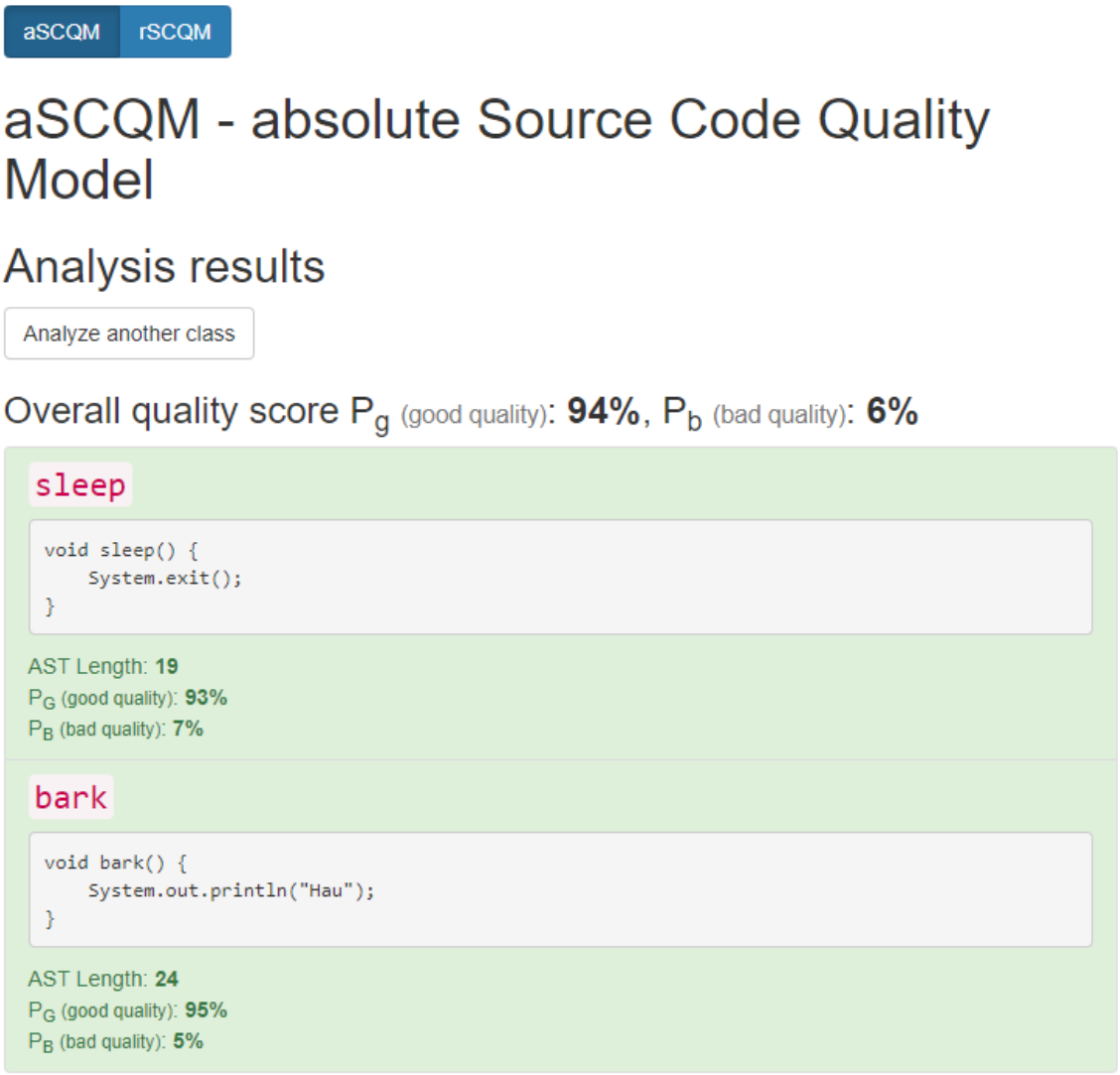


Rys. E.1: Formularz pozwalający na przesłanie kodu do klasyfikacji przez model rSCQM

Rys. E.2: Formularz pozwalający na przesłanie kodu do klasyfikacji przez model aSCQM



Rys. E.3: Rezultat klasyfikacji kodu klasy przez model rSCQM



Rys. E.4: Rezultat klasyfikacji kodu klasy przez model aSCQM

F. Instrukcja uruchomienia mikrousługi udostępniającej klasyfikacje jakości kodu z użyciem SCQM

Dzięki odpowiedniemu przygotowaniu aplikacji z modelem SCQM, które zostało szczegółowo opisane w dodatku E, do jej uruchomienia wystarczy dowolny komputer z zainstalowanym Dockerem. Wystarczy sklonować repozytorium z aplikacją, przejść do katalogu `docker` i uruchomić aplikację za pomocą narzędzia `docker-compose`. Lista komend, które należy wykonać, została przedstawiona na listingu F.1.

```
git clone https://github.com/fracz/scqm.git
cd scqm/docker
docker-compose up --build -d
```

Listing F.1: Komendy uruchamiające aplikację z modelem SCQM

Aby TensorFlow mógł zainicjalizować wszystkie struktury wytrenowanym modelem, wymagane jest posiadanie co najmniej 6GB pamięci operacyjnej. Warto też zwrócić uwagę, że aplikacja może uruchamiać się nawet 10 minut. Aby zweryfikować, czy wszystkie operacje inicjalizacji zostały już wykonane, można sprawdzić logi kontenera `scqm-model`. Informacja o działającej uruchomionej aplikacji od *frameworka* Flask ostatecznie oznacza poprawne uruchomienie modelu i gotowość na przyjmowanie kodu źródłowego do klasyfikacji (zob. listing F.2).

```
docker logs --tail=1 scqm-model
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Listing F.2: Logi kontenera scqm-model informujące o poprawnym uruchomieniu aplikacji

Od tego momentu aplikacja jest dostępna pod adresem `http://localhost:7276`. Do otwarcia interfejsu użytkownika aplikacji udostępniającej model SCQM można użyć dowolnej przeglądarki internetowej.

G. Przykładowa integracja SCQM i systemu kontroli wersji

Na końcu niniejszego dodatku, na listingu G.3 zaprezentowano przykładową implementację *hooka* typu `pre-commit`, który przed stworzeniem nowej zmiany w historii projektu odpytuje REST API mikrousługi SCQM (zob. sekcja F). Jeśli jakościowy model kodu źródłowego wykryje, że *commit* obniżałby jakość kodu źródłowego – będzie on zablokowany z odpowiednim komunikatem.

Przedstawione rozwiązanie na początku swojego działania pozyskuje listę dodanych lub zmienionych w danej zmianie plików. Nie ma sensu analizowanie kodu, który jest usuwany. Następnie sprawdzane jest, czy jest to kod napisany w Javie – poszukiwane jest po prostu rozszerzenie pliku `.java`. Dla każdego takiego pliku wykonywane jest żądanie HTTP do uruchomionej aplikacji SCQM. W zależności od sytuacji, wykonywane jest żądanie do modelu

- aSCQM, jeśli plik jest dodawany w analizowanym *commicie*, lub
- rSCQM, jeśli plik istniał już wcześniej i analizowany *commit* go modyfikuje.

Niezależnie od wybranego przypadku, dla każdego pliku w odpowiedzi otrzymujemy liczby P_G i P_B dla każdej odnalezionej w nim metody spełniającej wymagania modelu, oznaczające prawdopodobieństwa że dany kod lub dana zmiana jest odpowiedniej jakości. Skrypt wylicza na ich podstawie średnią jakość kodu lub zmiany w danymi pliku i prezentuje programiście wartość $P_G * 100$. Jest to procentowo wyrażona klasyfikacja kodu w danym pliku jako odpowiedniej jakości. Jeśli którykolwiek z plików nie spełnia zadanego warunku $P_G \geq 0.5$, skrypt odrzuca tworzony *commit*, zwracając programiście odpowiedni komunikat.

Oczywiście, programista zawsze ma możliwość pominięcia sprawdzenia przy wybranym *commicie*, podając parametr `-no-verify` przy komendzie `git commit`. Pozwala ona na wykonanie zmiany bez wyzwalania *hooka* `pre-commit`, co całkowicie pomija sprawdzenie kodu za jego pomocą. W ten sposób można wykonać zmianę pomimo identyfikowania jej niskiej jakości, jeśli model dla danego przypadku będzie się mylić.

Listingi G.1 i G.2 zawierają przykładowe tworzenie nowego *commita* w repozytorium, w którym wprowadzono integrację z modelem SCQM za pomocą opisanego *hooka*. Pierwszy z nich to przypadek gdzie *commit* został utworzony, ponieważ nie naurszał nałożonych na projekt reguł jakościowych. Drugi – pokazuje jak wygląda komunikat dla programisty, gdy jakość kodu została sklasyfikowana jako zbyt niska.

```
$ git \textit{commit} -am "Launching rockets working"
src/main/java/pl/edu/agh/rocket/MainLauncher.java: rSCQM: 100%
src/main/java/pl/edu/agh/rocket/LauncherHelper.java: rSCQM: 87%
src/test/java/pl/edu/agh/rocket/MainHelperTest.java: aSCQM: 95%

[rockets 3134794b6] Launching rockets working
 3 files changed, 35 insertions(+), 2 deletions(-)
 create mode 100644 src/test/java/pl/edu/agh/rocket/MainHelperTest.java
```

Listing G.1: Próba wprowadzenia zmiany kodu odpowiedniej jakości wraz z informacją pochodzącą z hooka integrującego projekt z SCQM

```
$ git \textit{commit} -am "Launching rockets working"
src/main/java/pl/edu/agh/rocket/MainLauncher.java: rSCQM: 100%
src/main/java/pl/edu/agh/rocket/LauncherHelper.java: rSCQM: 87%
src/test/java/pl/edu/agh/rocket/MainHelperTest.java: aSCQM: 5%

SCQM model detected that your \textit{commit} decreases the quality of the
    ↪ source code.
Pay attention to the files that has been marked with score lower than 50%
    ↪ and refactor your code.
```

Listing G.2: Próba wprowadzenia zmiany kodu zbyt niskiej jakości zablokowanej przez hook integrujący projekt z SCQM

W celu poprawnego działania *hooka* wymagana jest instalacja języka PHP w systemie operacyjnym (w tym języku właśnie został napisany przykładowy *hook*). Skrypt należy umieścić w katalogu projektu, w którym powinna być wdrożona kontrola jakości za pomocą stworzonego modelu, w pliku `.git/hooks/pre-commit`.

Kod źródłowy skryptu jest dostępny także na platformie GitHub[1].

```
#!/usr/bin/env php
<?php

exec('git diff --cached --name-status | awk \'$1 != "D" { print $2 }\'',
    ↪ $changedFiles);
```

[1] `https://gist.github.com/fracz/a3d3a5e6d12a5538f2858a79fd568fd1` lub `https://github.com/fracz/phd/links/G.1`

```
$allChangesWithGoodQuality = true;

foreach ($changedFiles as $changedFile) {
    if (!preg_match('#\.java$#i', $changedFile)) {
        continue;
    }
    $sourceCurrent = file_get_contents($changedFile);
    $sourceBefore = '';
    exec('git show HEAD:' . $changedFile . ' 2>&1', $sourceBefore,
        ↪ $exitCode);
    if ($exitCode === 0 && $sourceBefore) {
        $sourceBefore = implode(PHP_EOL, $sourceBefore);
        $rscqm = getScqm('rscqm', ['sourceBefore' => $sourceBefore, '
            ↪ sourceAfter' => $sourceCurrent]);
        if ($rscqm) {
            $rscqmResult = array_column($rscqm['predictions'], '
                ↪ prediction');
            $rscqmResult = array_column($rscqmResult, 0);
            $averageResult = array_sum($rscqmResult) / count($rscqmResult
                ↪ );
            $rscqmResult = round($averageResult * 100);
            echo sprintf("%s: rSCQM: %d%%\n", $changedFile, $rscqmResult)
                ↪ ;
            $allChangesWithGoodQuality &= $rscqmResult >= 50;
        }
    } else {
        $ascqm = getScqm('ascqm', ['source' => $sourceCurrent]);
        if ($ascqm) {
            $ascqmResult = array_column($ascqm, 'prediction');
            $ascqmResult = array_column($ascqmResult, 0);
            $averageResult = array_sum($ascqmResult) / count($ascqmResult
                ↪ );
            $ascqmResult = round($averageResult * 100);
            echo sprintf("%s: aSCQM: %d%%\n", $changedFile, $ascqmResult)
                ↪ ;
            $allChangesWithGoodQuality &= $ascqmResult >= 50;
```

```
        }
    }
}

if (!$allChangesWithGoodQuality) {
    echo PHP_EOL . 'SCQM model detected that your \textit{commit}
        ↪ decreases the quality of the source code.' . PHP_EOL;
    echo 'Pay attention to the files that has been marked with score lower
        ↪ than 50% and refactor your code.' . PHP_EOL;
    exit(1);
}

function getScqm($scqm, $data) {
    $dataString = json_encode($data);
    $ch = curl_init('http://localhost:7276/' . $scqm);
    curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "POST");
    curl_setopt($ch, CURLOPT_POSTFIELDS, $dataString);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_HTTPHEADER, [
        'Content-Type: application/json',
        'Content-Length: ' . strlen($dataString)
    ]);
    $result = curl_exec($ch);
    return json_decode($result, true);
}
```

Listing G.3: Przykładowy *hook* `pre-commit` dla systemu kontroli wersji Git integrujący repozytorium z modelem SCQM

H. Biografia naukowa doktoranta

Dane bibliometryczne

- h-index: 1
- Wykaz publikacji BPP AGH
  `https://bpp.agh.edu.pl/autor/fracz-wojciech-20298`
- ORCiD
  `https://orcid.org/0000-0002-3613-6335`
- ResearchGate
  `https://www.researchgate.net/profile/Wojciech_Fracz`
- StackOverflow
  `https://stackoverflow.com/users/878514/fracz`

Tytuły naukowe

- inż., 2013, temat pracy: *System zamawiania posiłków wykorzystujący możliwości urządzeń mobilnych oraz mechanizmy lokalizacji użytkownika*, promotor: dr inż. Robert Marcjan
- mgr inż., 2014, temat pracy: *Wspieranie praktyki przeglądu kodu źródłowego na urządzeniach mobilnych*, promotor: dr inż. Jacek Dajda, praca dyplomowa obroniona z wyróżnieniem

Publikacje

- W. Frącz, J. Dajda. (2018). *Developers' Game: a Preliminary Study Concerning a Tool for Automated Developers Assessment.* 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018. p. 695-699. MNiSW=15
- M. Piech, W. Frącz, W. Turek, M. Kisiel-Dorohinicki, J. Dajda, A. Byrski. (2018). *Model for Dynamic and Hierarchical Data Repository in Relational Database.* Computer Science ; ISSN 1508-2806. — 2018 vol. 19 no. 4, s. 479–500. MNiSW=12

- W. Frącz, J. Dajda. (2017). *Experimental Validation of Source Code Reviews on Mobile Devices*. International Conference on Computational Science and Its Applications (pp. 533-547). Springer, Cham.
  MNiSW=15

- W. Frącz, J. Dajda. (2016). *Source code reviews on mobile devices*. Computer Science ; ISSN 1508-2806. — 2016 vol. 17 no. 2, s. 143–161.
  MNiSW=12

- W. Frącz. (2015). *An empirical study inspecting the benefits of gamification applied to university classes*. 7th Computer Science and Electronic Engineering Conference (CEEC) (pp. 135-139). IEEE.
  MNiSW=15

- W. Frącz, J. Dajda. (2014) *Can the source code be reviewed on a smartphone?*. Software engineering from research and practice perspectives / eds. Lech Madeyski, Mirosław Ochodek. — Poznan; Warszawa: Wydawnictwo Nakom, 2014. — ISBN: 978-83-63919-16-0. — S. 179–195.
  MNiSW=5

Nagrody i wyróżnienia

- Wyróżnienie Rektora AGH zespołowe III stopnia za osiągnięcia dydaktyczne, zaprojektowanie i przeprowadzenie bloku zajęć z użyciem grywalizacji, wspólnie z dr inż. Jackiem Dajda, 2017

Udział w projektach badawczo-rozwojowych

- *Centrum Mistrzostwa Informatycznego*, Akademia Górniczo – Hutnicza im. Stanisława Staszica w Krakowie, nr projektu POPC.03.02.00-00-0002/18, kierownik: dr hab. inż. Aleksander Byrski, prof. AGH, 2019

- *Wykorzystanie danych gromadzonych w repozytoriach kodu źródłowego w celu poprawy jakości pracy programistów i popularyzacji przeglądów kodu źródłowego*, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, grant dziekański (umowa nr 15.11.230.399), kierownik: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, 2018

- *Wsparcie praktyki przeglądu kodu źródłowego na urządzeniach mobilnych*, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, grant dziekański

(umowa nr 15.11.230.289), kierownik: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, 2016-2017

- *Europejskie dziedzictwo techniczne - upowszechnienie historycznych i współczesnych publikacji z zakresu nauk technicznych w innowacyjnym środowisku informatycznym.*, Programu Operacyjnego Polska Cyfrowa, Europejskiego Funduszu Rozwoju Regionalnego, kierownik: dr hab. inż. Aleksander Byrski, prof. AGH, 2016-2019

- *Zastosowanie technik gamifikacji w inżynierii oprogramowania*, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, grant dziekański (umowa nr 15.11.230.192), kierownik: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, 2015

- *System gromadzenia i generowania informacji na potrzeby analizy kryminalnej i koordynacji działań w Straży Granicznej*, Narodowe Centrum Badań i Rozwoju, kierownik: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, 2014-2017

- *System zarządzania informacjami w transmisji elektronicznej (radio, TV)*, Narodowe Centrum Badań i Rozwoju, kierownik: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, 2014-2017

- *Opracowanie systemu koordynacji kontroli „PORTY 24”*, Izba Celna w Gdyni, kierownik: dr inż. Jacek Dajda, 2014-2015

- *DiSSBy Systemy informacyjno-analityczny wspomagający planowanie działań BOR*, Ministerstwo Nauki i Szkolnictwa Wyższego, kierownik: dr hab. inż. Marek Kisiel-Dorohinicki, prof. AGH, 2013-2016

Staże

- Staż asystencki, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, Wydział Informatyki, Elektroniki i Telekomunikacji, Katedra Informatyki, Październik 2013 – Wrzesień 2014.

Aktywność jako recenzent

- Computing and Informatics, Institute of Informatics Slovak Academy of Sciences, 2018

- International Conference on Computational Science and Its Applications, 2017

- Computer Science, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, 2016

Udział w konferencjach

- 34th 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 26-28 września 2018, Madryt, Hiszpania
- 17th International Conference on Computational Science and Its Applications (ICCSA 2017), 3-6 lipca 2017, Trieste, Włochy
- 7th Computer Science and Electronic Engineering Conference (CEEC), 24-25 września 2015, Colchester, Wielka Brytania

Działalność dydaktyczna

- Tworzenie nowoczesnych aplikacji internetowych, 1. rok studiów stac. W ramach programu Prymusi AGH, 2018-2019
- Podstawy programowania w Javie, 1. rok studiów stac. W ramach programu Prymusi AGH, 2018
- Introduction to web programming, studenci z uczelni EFREI w Paryżu, 2018-2019
- Java 1, studenci z uczelni EFREI w Paryżu, 2018
- Programowanie aplikacji webowych 2, studia podyplomowe: Metody Wytwarzania Oprogramowania, 2018-2019
- Prelekcja *SUPLA – otwarta, polska automatyka budynkowa* w ramach Nocy Informatyka 1.1, 2018, `https://www.youtube.com/watch?v=5WobKP58YIk` lub `https://fracz.github.io/phd/links/H.1.html`
- Techniki obiektowe i komponentowe, studia podyplomowe: Projektowanie, Programowanie i Eksploatacja Systemów, 2017-2019
- Techniki programowania w Javie 3, wykład *Wprowadzenie do Javy*, studia podyplomowe: Metody Wytwarzania Oprogramowania, 2017-2019

- Prelekcja *Nowoczesny stos JS w 60 minut* w ramach cyklu spotkań *Epizody* (s17e02) organizowanego przez Koło Naukowe Epicentrum (UJ), 2017, `https://www.youtube.com/watch?v=nrCejJjxMe4` lub `https://fracz.github.io/phd/links/H.2.html`
- Inżynieria oprogramowania, 3. rok studiów stac. I stopnia na kierunku Informatyka, 2015-2019
- Wykład *Git demystified*, 2015, 2019
- Techniki wytwarzania oprogramowania, 2. rok studiów stac. II stopnia na kierunku Informatyka, 2014-2017
- Technologie obiektowe 2, 3. rok studiów stac. I stopnia na kierunku Informatyka, 2014-2018
- Technologie obiektowe 1, 2. rok studiów stac. I stopnia na kierunku Informatyka, 2014-2018

Umiejętności

- języki obce: angielski
- języki programowania: Java, PHP, JavaScript
- frameworki webowe (frontend): AngularJS, Vue.js, Aurelia, Bootstrap
- frameworki webowe (backend): Symfony, Slim, Spring
- zarządzanie projektami: Git, Maven, Gradle, Composer, NPM, JSPM, Webpack, Gulp