Przewidywanie defektów w oprogramowaniu za pomocą narzędzia Git

Mikołaj Kapica

Politechnika Wrocławska,
Wydział Informatyki i Telekomunikacji W4N
272729@student.pwr.edu.pl

Streszczenie Celem pracy jest opracowanie klasyfikatora defektów w oprogramowaniu, który wykorzystuje dane z commitów pochodzących z repozytorium *microsoft/vscode*[3] na platformie GitHub [3]. Analiza będzie uwzględniać zarówno zmiany w kodzie źródłowym, jak i metadane przypisane do poszczególnych commitów.

Słowa kluczowe: Git · Przewidywanie defektów · Narzędzia analizy kodu · Uczenie maszynowe w inżynierii oprogramowania · Analiza zmian w repozytorium · Jakość kodu · Prognozowanie błędów · Optymalizacja procesu wytwarzania oprogramowania · Automatyzacja weryfikacji kodu · Doskonalenie procesu deweloperskiego

1 Wprowadzenie

Zadanie polega na opracowaniu klasyfikatora przewidującego defekty w oprogramowaniu z wykorzystaniem narzędzia Git. Git jest systemem kontroli wersji, który umożliwia śledzenie zmian w kodzie źródłowym oraz wspomaga współpracę programistów nad rozwojem projektu. Każdy commit, czyli zatwierdzenie zmiany w repozytorium (katalogu zawierającym kod), zawiera informacje o dokonanych modyfikacjach oraz dodatkowe metadane, takie jak tytuł zmiany, autor zmiany oraz czas jej wprowadzenia. Korzystając z danych z repozytorium GitHub o nazwie *microsoft/vscode* [3], zidentyfikowałem commity wprowadzające błędy i opracowałem klasyfikator do ich przewidywania.

2 Zgromadzenie zbioru danych

2.1 Źródło danych

Postanowiono zgromadzić zbiór danych, korzystając z repozytorium *microsoft/vscode* [3] na platformie GitHub. Repozytorium to zawiera dużą liczbę commitów - 121 579 na dzień 12 maja 2024 roku. Pozwoliło to na

zebranie znacznej ilości commitów zawierających błędy, które odtąd będą nazywane *"buggy commits"*, zaś inne *"normal commits"*. W praktyce jednak skorzystano z pierwszych 10 000 commitów, aby zasymulować warunki dopiero rozwijającego się oprogramowania.

2.2 Pozyskiwanie błędnych commitów

Aby znaleźć *buggy commits*, użyto biblioteki *pydriller* [1], która umożliwia wygodne wyciąganie interesujących informacji z repozytorium Git.

Przeglądając dokumentację narzędzia *pydriller*, natrafiono na metodę pozwalającą na efektywne zgromadzenie zbioru danych. Metoda `get_commits_last_modified_lines()` dla danego obiektu commit zwraca zestaw commitów, które jako ostatnie zmieniły linie modyfikowane w tym commicie. Implementuje ona algorytm SZZ [2], który działa w następujący sposób dla każdego pliku w commicie:

1. Uzyskaj różnicę (git diff).
2. Uzyskaj listę usuniętych linii.
3. Zastosuj komendę *git blame* i uzyskaj commity, w których te linie zostały dodane.

Korzystając z powyższej metody, napisano procedurę umożliwiającą zebranie danych.

```
repo = pydriller.Git(GIT_REPO_PATH)

buggy_commits = {
    commit_hash
    for fix_commit in repo.get_list_commits()
    if RE_FIX_COMMIT.match(fix_commit.msg)
    for hashes in
    ↪  repo.get_commits_last_modified_lines(fix_commit).values()
    for commit_hash in hashes
}
```

Wycinek kodu z pliku `src/data_collecting/gather_buggy_commits.py`

Upraszczając, dla każdego commit'u w repozytorium sprawdzano, czy jego nazwa zgadza się z wybranym wyrażeniem regularnym (czy w tytule znajduje się słowo 'fix' lub 'bug'). Następnie, za pomocą metody

`get_commits_last_modified_lines()`, identyfikowano interesujące commity i zapisywano odpowiadające im hashe do zbioru, a następnie serializowane do pliku `raw_dataset/buggy_hashes.txt`

W ten sposób zgromadzono zbiór, który został zapisany do pliku zawierającego hashe odpowiadające błędnym commitom.

2.3 Serializowanie commit'ów

Następnie ponownie użyto biblioteki `pydriller`, aby pozyskać zbiór danych obejmujący pierwsze 10 000 commitów w repozytorium. Do każdego z tych commitów dodano atrybut `is_bug` o wartości logicznej, wskazującej, czy dany commit wprowadzał błąd. Dane zapisano do dwóch plików: commitów z atrybutem `is_bug` do pliku `raw_dataset/buggy_commits.jsonl`, a pozostałe commity do pliku `raw_dataset/normal_commits.jsonl`.

2.4 Commit'y walidujące

Aby zweryfikować wyniki klasyfikatora, dodano także 10 commitów, które zostały specjalnie wybrane do celów walidacyjnych. Spośród tych commitów, 5 zostało celowo napisanych tak, aby zawierały błędy (tzw. *buggy commits*), a pozostałe 5 zawierało poprawne wpisy lub zmiany kodu (tzw. *normal commits*). Wyniki tej walidacji były kluczowe dla oceny skuteczności opracowanego modelu oraz dla dokonania ewentualnych poprawek i optymalizacji.

2.5 Wyekstrahowane dane

W ramach każdego commita zapisano poniższe metadane:

- ➔ **hash:** Unikalny identyfikator commita.
- ➔ **msg:** Opis zmian dokonanych w commicie.
- ➔ **author_name:** Imię i nazwisko autora commita.
- ➔ **author_date:** Data zatwierdzenia commita.
- ➔ **author_timezone:** Strefa czasowa autora commita.
- ➔ **merge:** Czy commit jest wynikiem zmergowania dwóch gałęzi.
- ➔ **deletions:** Liczba usuniętych linii kodu w commicie.
- ➔ **insertions:** Liczba dodanych linii kodu w commicie.
- ➔ **lines:** Całkowita liczba zmienionych linii kodu w commicie.
- ➔ **files:** Liczba plików zmienionych w commicie.

- **dmm_unit_size:** Wielkość jednostki miary według metryki DMM. (poprawa czytelności, biorąc pod uwagę długość zmodyfikowanych metod. Zakres [0;1], gdzie 1 oznacza niskie ryzyko, małe metody, 0 oznacza wysokie ryzyko, duże metody, null, jeśli język programowania nie jest obsługiwany)

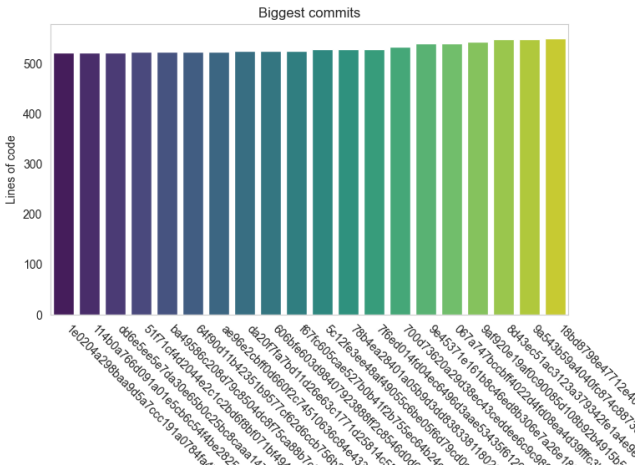
- **dmm_unit_complexity:** Złożoność jednostki miary według metryki DMM. (Wartość DMM (pomiędzy 0,0 a 1,0) dla złożoności cyklomatycznej metody w tym commicie, lub None, jeśli żaden z języków programowania w commicie nie jest obsługiwany.)

- **dmm_unit_interfacing:** Interfejs jednostki miary według metryki DMM. (Ilość parametrów metod: Wartość DMM (pomiędzy 0,0 a 1,0) dla interfejsu metody w tym commicie, lub None, jeśli żaden z języków programowania w commicie nie jest obsługiwany.)

- **is_bug:** Informacja, czy commit wprowadza poprawki błędów.

3 Exploratory data analysis

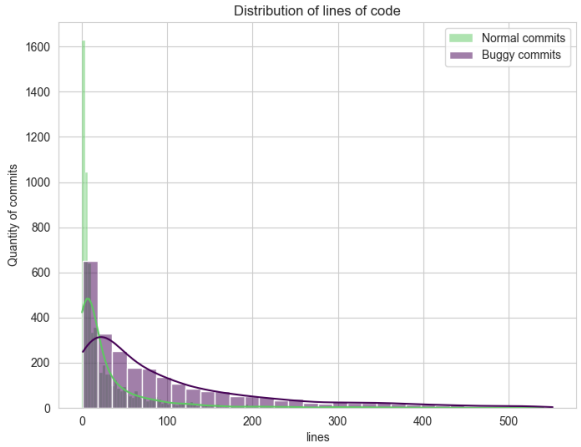
Po krótkiej analizie zauważono, że kilka z początkowych commitów w repozytorium jest bardzo dużych, a wiele commitów naprawia błędy, które powstały właśnie w tych commitach. Dlatego już na etapie eksploracyjnej analizy danych zdecydowano się wykluczyć commity o nietypowo dużej ilości zmienionych linii (dokładnie zostawiając tylko commity, których atrybut lines jest mniejszy niż wartość 95 kwantyla w rozkładzie). Po tej operacji największe commity pod względem ilości zmienionych linii kodu wyglądały następująco:



Rysunek 1. Największe commity

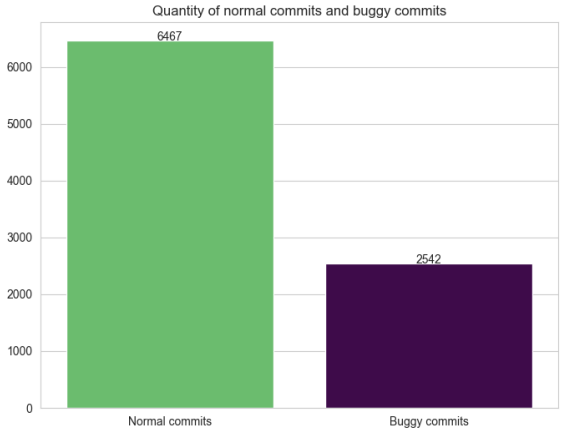
Wcześniej największe commity miały około 10 000 zmienionych linii, teraz ta wartość zmniejszyła się do około 500.

Rozkład commitów według liczby zmienionych linii przedstawia poniższy wykres:



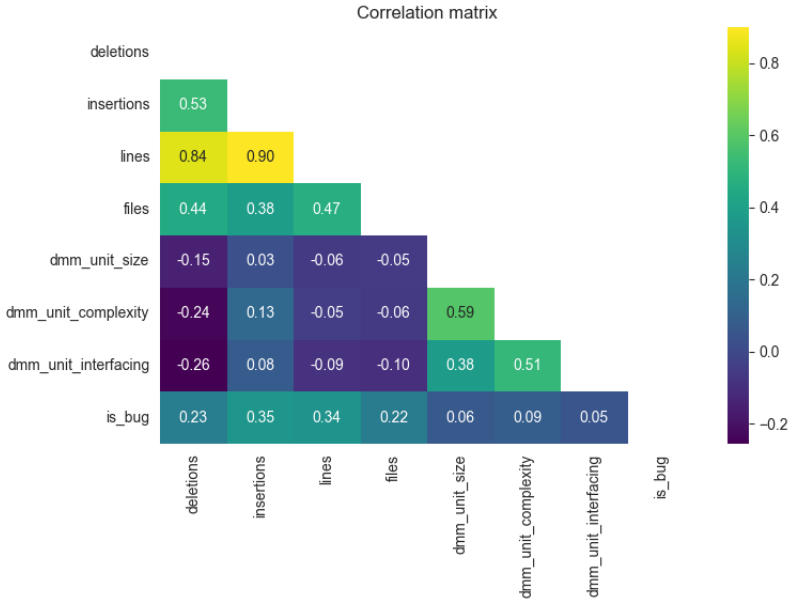
Rysunek 2. Rozkład zmienionych linii

Stosunek liczby commitów błędnych i normalnych przedstawiono na poniższym wykresie:



Rysunek 3. Liczba commitów

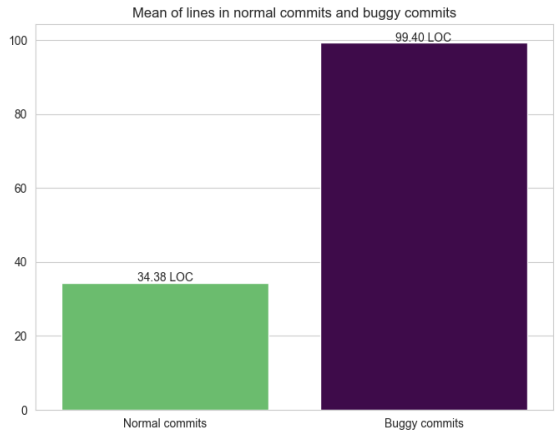
Następnie przeprowadzono analizę korelacji Pearsona. Macierz korelacji przedstawia poniższy wykres:



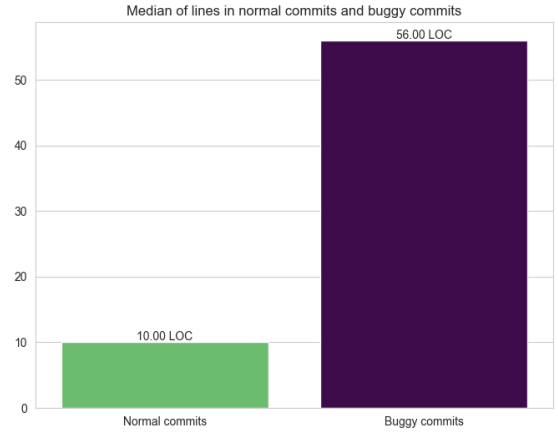
Rysunek 4. Macierz korelacji

Najbardziej interesujący jest ostatni wiersz macierzy, gdzie można zauważyć pewne zależności między byciem *"buggy commit"*, a liczbą zmienionych linii przez commit.

Aby to dogłębnie zbadać, stworzono poniższe wykresy prezentujące średnią i medianę liczby zmienionych linii:



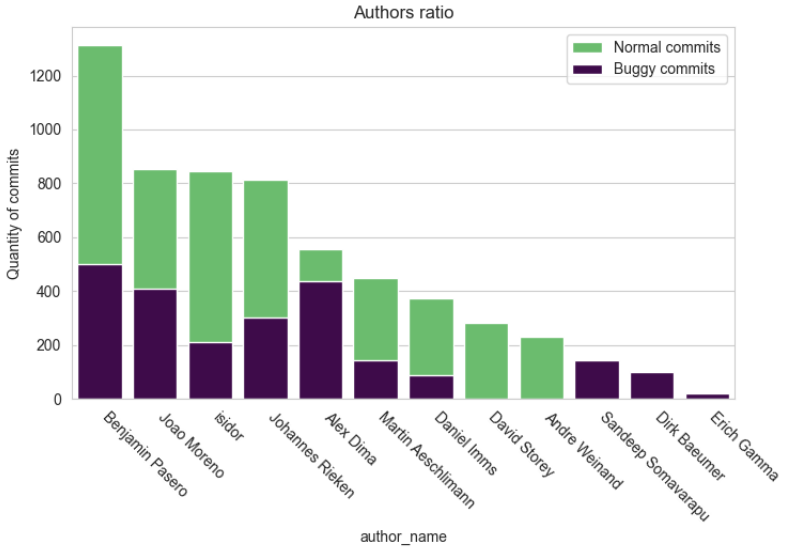
Rysunek 5. Średnie



Rysunek 6. Mediany

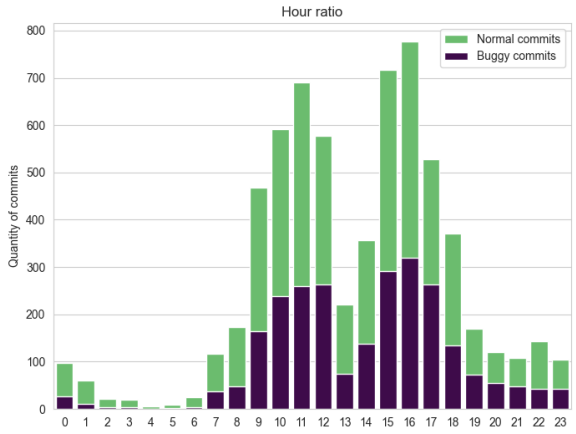
Jak widać, *"buggy commits"* mają znacznie większą średnią i medianę liczby zmienionych linii niż *"normalne commity"*. Ta cecha może być istotna dla klasyfikatorów, które zostaną wykorzystane w dalszej analizie.

Kolejną cechą, która może mieć wpływ na predykcję, jest autor commita. Stosunek *"buggy commitów"* do *"normalnych"* dla najbardziej aktywnych autorów przedstawiono na poniższym wykresie:

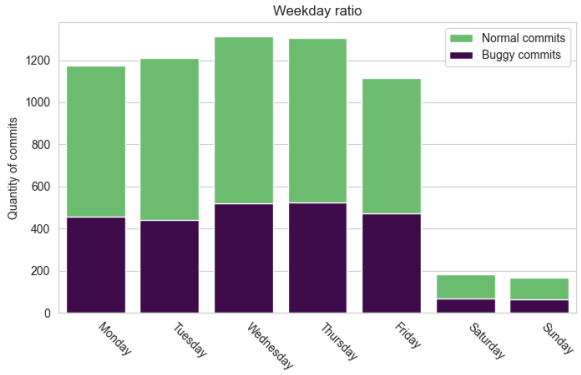


Rysunek 7. Stosunek *"buggy commitów"* do *"normalnych"* dla najbardziej aktywnych autorów

Ciekawy też jest rozkład commitów w zależności od godziny oraz od dnia tygodnia



Rysunek 8. Rozkład commitów w zależności od godziny



Rysunek 9. Rozkład commitów w zależności od dnia tygodnia

4 Preprocessing

Dane zostały poddane obróbce, przeprowadzając kolejne operacje:

- ➔ Usunięto atrybut *hash*, ponieważ jest to unikalny identyfikator każdego commita i nie powinien zależeć w żaden sposób od innych parametrów.
- ➔ Usunięto odstające dane (ang. outliers) na podstawie atrybutu *lines*, usuwając wszystkie rekordy, gdzie *lines* przekraczał wartość 99 kwantyla rozkładu *lines*.
- ➔ Przeprowadzono operacje tworzenia nowych danych ze starych (ang. feature engineering):
- ➔ Stworzono stosunek usuniętych linii do całkowitej liczby zmienionych linii oraz dodano informację o długości wiadomości commita.
- ➔ Zamieniono atrybuty *merge* oraz *is_bug* z wartości boolowskich na wartości całkowitoliczbowe 0 i 1.
- ➔ Za pomocą biblioteki *pandas* oraz funkcji `get_dummies()` przekształcono zmienne kategoryczne na zmienne wskaźnikowe (one-hot encoding), tworząc binarne kolumny dla każdej unikalnej wartości w oryginalnych kolumnach.
- ➔ Przetransformowano również wartość daty, którą zastano w formacie `%Y-%m-%d %H:%M:%S%z`, na dwa atrybuty: `day_of_the_week` oraz `hour_of_day`, reprezentujące odpowiednio dzień tygodnia i godzinę dnia.
- ➔ Atrybut `msg` został rozdzielony na kolumny reprezentujące różne cechy tekstowe za pomocą `TfidfVectorizer` z biblioteki *scikit-learn*[4], który przekształca tekst na wektory cech poprzez obliczenie wartości TF-IDF (Term Frequency-Inverse Document Frequency) dla każdego terminu w wiadomości.
- ➔ Następnie wszystkie dane zostały znormalizowane za pomocą `StandardScaler`'a z biblioteki *scikit-learn*[4], co zapewnia, że średnia wartość wynosi 0, a wariancja 1.
- ➔ Zauważono także, że czasami atrybuty wyliczające złożoność commita pozostają puste (null), a wiele modeli nie radzi sobie z brakującymi wartościami. Zastosowano więc wypełnienie brakujących wartości za pomocą średniej wartości dla każdej kategorii (osobno dla *buggy* i *normal*).
- ➔ Ostatecznie usunięto wszystkie commity typu *merge*, ponieważ te zatwierdzenia jedynie łączą poprzednie commity i zwykle nie wprowadzają nowego kodu.

5 Rozwiązywanie problemu

5.1 Dobór metod do rozwiązania problemu

Do rozwiązania zadania użyto czterech klasyfikatorów:

- `SVM (SVC) (support vector machine)`
- `LogisticRegression`
- `DecisionTreeClassifier`
- `RandomForestClassifier`

Te klasyfikatory uznano za właściwe z poniższych powodów:

5.1.1 Support vector machine (SVM) (maszyna wektorów nośnych) charakteryzuje się efektywnością na wysokowymiarowych przestrzeniach danych oraz elastycznością. SVM może dobrze generalizować na nowych, nieznanych danych.

5.1.2 Logistic Regression (regresja logistyczna) jest prosta w zrozumieniu i szybka w trenowaniu, co jest korzyścią przy dużych zbiorach danych, oraz często daje bardzo dobre wyniki i jest mniej podatna na przeuczenie w porównaniu do bardziej złożonych modeli. Interpretowalność współczynników regresji logistycznej pozwala na bezpośrednią interpretację wpływu poszczególnych cech na prawdopodobieństwo klasyfikacji jako bug.

5.1.3 Decision tree classifier (klasyfikator drzewa decyzyjnego) charakteryzuje się łatwością w wizualizacji i interpretacji, co umożliwia zrozumienie wpływu poszczególnych cech commitów. Klasyfikator ten radzi sobie całkiem dobrze z danymi nieliniowymi, co umożliwia wykrycie skomplikowanych zależności między cechami. Dodatkowo, jest on relatywnie szybki w procesie trenowania i predykcji, dlatego postanowiono przetestować jego skuteczność.

5.1.4 Random forest classifier (klasyfikator Lasu Losowego) zapewnia zwiększoną dokładność, redukując problem przeuczenia i zazwyczaj daje lepsze wyniki niż pojedyncze drzewo decyzyjne. Ponieważ jest to metoda baggingu, jest bardziej odporna na zmienności w danych treningowych. Ponadto, dostarcza narzędzi do oceny znaczenia poszczególnych

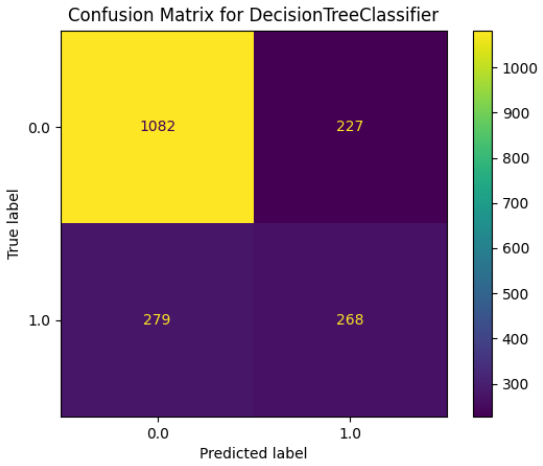
cech, co ułatwia identyfikację kluczowych atrybutów commitów wpływających na powstawanie błędów. Już na tym etapie podejrzewano, że klasyfikator `RandomForestClassifier` będzie najbardziej skuteczny, jednakże przeprowadzono również testy z innymi klasyfikatorami.

Do projektu użyto implementacji modeli z pakietu *scikit-learn* [4].

5.2 Wyniki modeli

Analizę wyników przeprowadzono dla wszystkich czterech modeli, tj. Klasyfikatora Drzewa Decyzyjnego, Klasyfikatora Lasu Losowego, Maszyny Wektorów Nośnych oraz Regresji Logistycznej. Testowane dane stanowiły 20% całego zbioru danych, co jest standardowym podejściem w uczeniu maszynowym. Dodatkowo, wyniki zostały porównane dla niezmienionych danych oraz danych po zastosowaniu technik oversamplingu i undersamplingu, aby ocenić wpływ tych metod na skuteczność modeli. Poniżej przedstawione są macierze pomyłek i raporty klasyfikatorów, pochodzące z pakietu *scikit-learn* [4], co pozwala na dokładną ocenę wyników każdego z modeli.

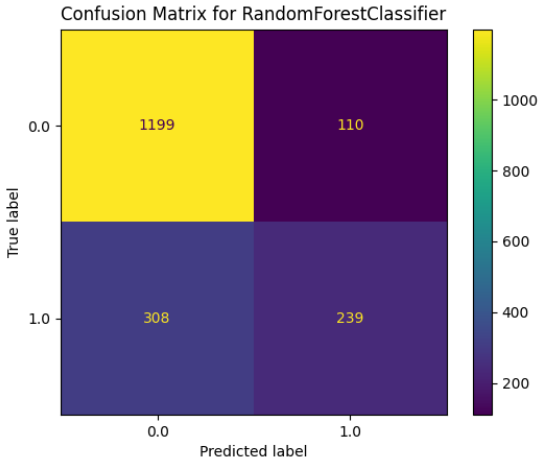
5.2.1 Imbalanced :



Rysunek 10. DecisionTreeClassifier confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.80	0.83	0.81	1309
1.0	0.54	0.49	0.51	547
Accuracy			0.73	1856
Macro Avg	0.67	0.66	0.66	1856
Weighted Avg	0.72	0.73	0.72	1856

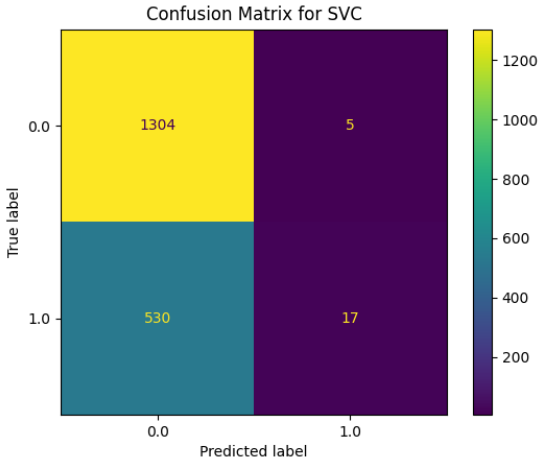
Tabela 1. Classification Report for DecisionTreeClassifier



Rysunek 11. RandomForestClassifier confusion matrix

	Precision	**Recall**	**F1-Score**	**Support**
0.0	0.80	0.92	0.85	1309
1.0	0.68	0.44	0.53	547
Accuracy			0.77	1856
Macro Avg	0.74	0.68	0.69	1856
Weighted Avg	0.76	0.77	0.76	1856

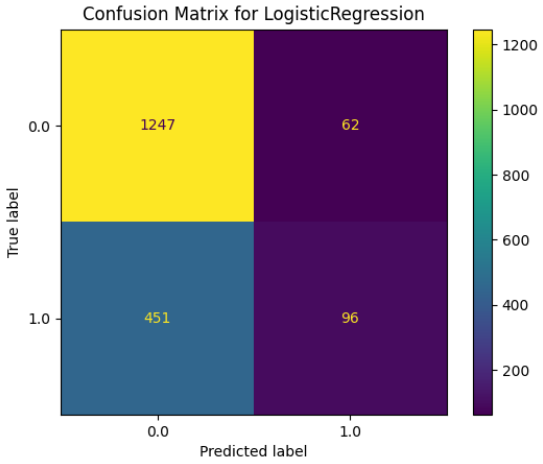
Tabela 2. Classification Report for RandomForestClassifier



Rysunek 12. SVC confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.71	1.00	0.83	1309
1.0	0.77	0.03	0.06	547
Accuracy			0.71	1856
Macro Avg	0.74	0.51	0.44	1856
Weighted Avg	0.73	0.71	0.60	1856

Tabela 3. Classification Report for SVC

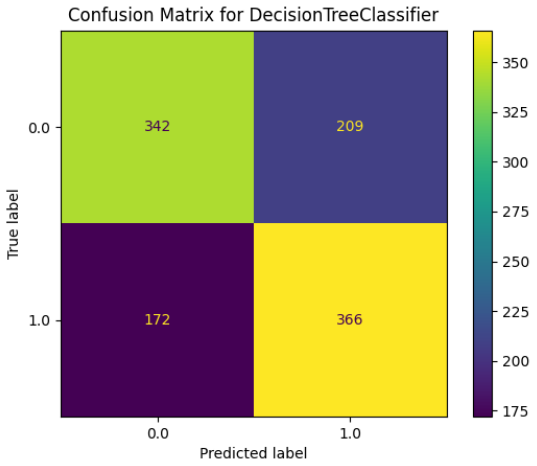


Rysunek 13. LogisticRegression confusion matrix

	Precision	**Recall**	**F1-Score**	**Support**
0.0	0.73	0.95	0.83	1309
1.0	0.61	0.18	0.27	547
Accuracy			0.72	1856
Macro Avg	0.67	0.56	0.55	1856
Weighted Avg	0.70	0.72	0.67	1856

Tabela 4. Classification Report for LogisticRegression

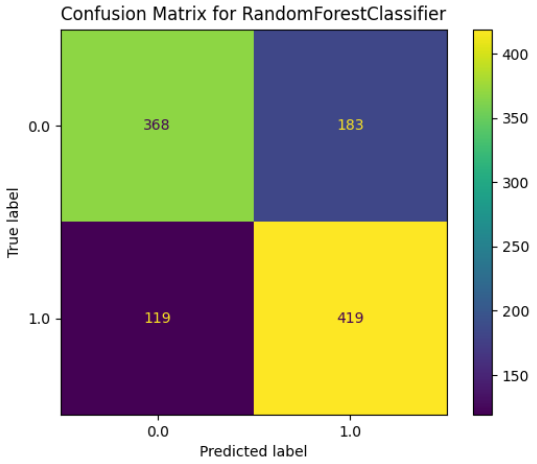
5.2.2 Undersampled :



Rysunek 14. DecisionTreeClassifier confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.67	0.62	0.64	551
1.0	0.64	0.68	0.66	538
Accuracy			0.65	1089
Macro Avg	0.65	0.65	0.65	1089
Weighted Avg	0.65	0.65	0.65	1089

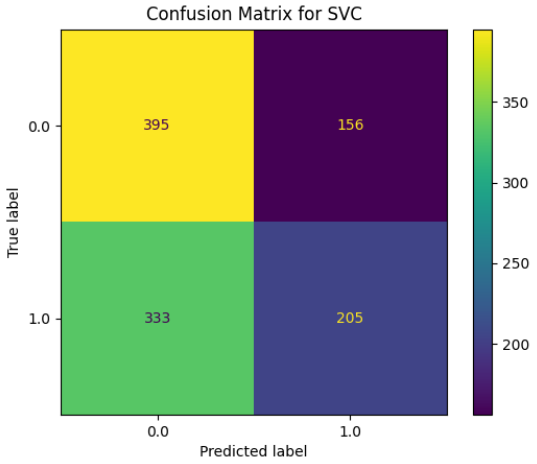
Tabela 5. Classification Report for DecisionTreeClassifier



Rysunek 15. RandomForestClassifier confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.76	0.67	0.71	551
1.0	0.70	0.78	0.74	538
Accuracy			0.72	1089
Macro Avg	0.73	0.72	0.72	1089
Weighted Avg	0.73	0.72	0.72	1089

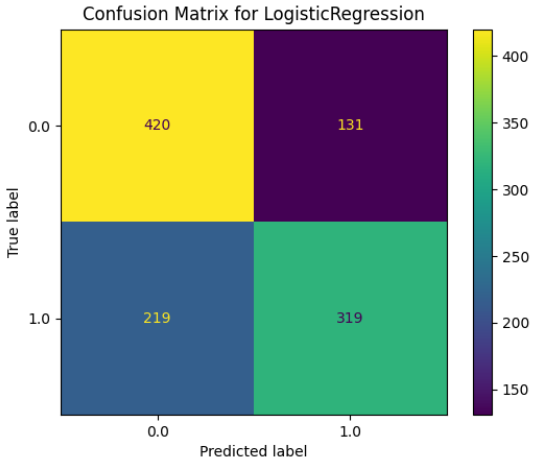
Tabela 6. Classification Report for RandomForestClassifier



Rysunek 16. SVC confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.54	0.72	0.62	551
1.0	0.57	0.38	0.46	538
Accuracy			0.55	1089
Macro Avg	0.56	0.55	0.54	1089
Weighted Avg	0.56	0.55	0.54	1089

Tabela 7. Classification Report for SVC

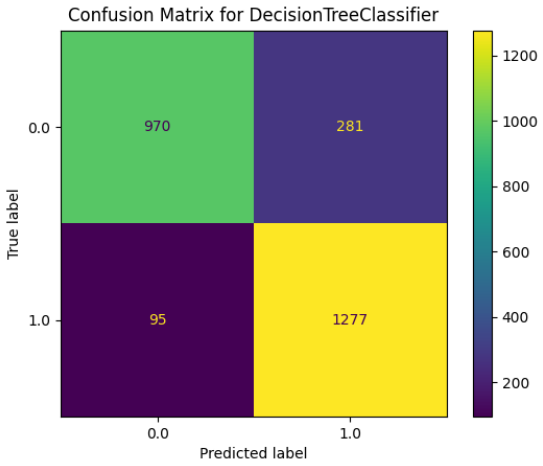


Rysunek 17. LogisticRegression confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.66	0.76	0.71	551
1.0	0.71	0.59	0.65	538
Accuracy			0.68	1089
Macro Avg	0.68	0.68	0.68	1089
Weighted Avg	0.68	0.68	0.68	1089

Tabela 8. Classification Report for LogisticRegression

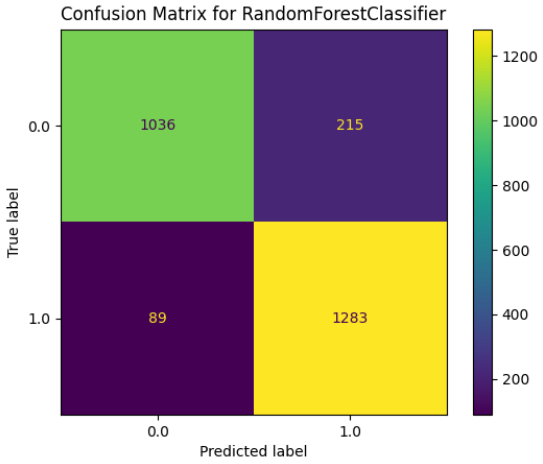
5.2.3 Oversampled :



Rysunek 18. DecisionTreeClassifier confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.67	0.62	0.64	1251
1.0	0.64	0.68	0.66	1372
Accuracy			0.86	2623
Macro Avg	0.87	0.85	0.85	2623
Weighted Avg	0.86	0.86	0.86	2623

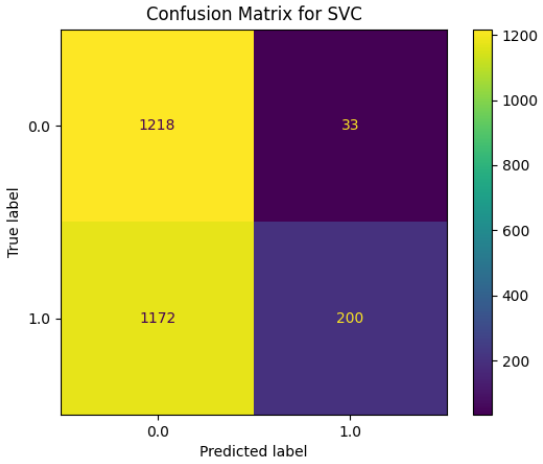
Tabela 9. Classification Report for DecisionTreeClassifier



Rysunek 19. RandomForestClassifier confusion matrix

	Precision	**Recall**	**F1-Score**	**Support**
0.0	0.92	0.83	0.87	1251
1.0	0.86	0.94	0.89	1372
Accuracy			0.88	2623
Macro Avg	0.89	0.88	0.88	2623
Weighted Avg	0.89	0.88	0.88	2623

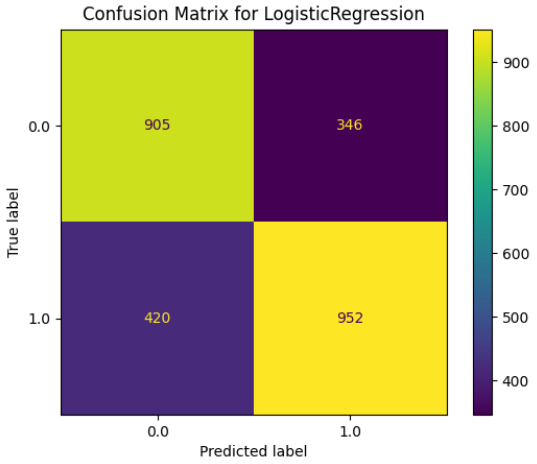
Tabela 10. Classification Report for RandomForestClassifier



Rysunek 20. SVC confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.51	0.97	0.67	1251
1.0	0.86	0.15	0.25	1372
Accuracy			0.54	2623
Macro Avg	0.68	0.56	0.46	2623
Weighted Avg	0.69	0.54	0.45	2623

Tabela 11. Classification Report for SVC



Rysunek 21. LogisticRegression confusion matrix

	Precision	Recall	F1-Score	Support
0.0	0.68	0.72	0.70	1251
1.0	0.73	0.69	0.71	1372
Accuracy			0.71	2623
Macro Avg	0.71	0.71	0.71	2623
Weighted Avg	0.71	0.71	0.71	2623

Tabela 12. Classification Report for LogisticRegression

5.3 Wnioski z wyników modeli (normal, undersampling, oversampling)

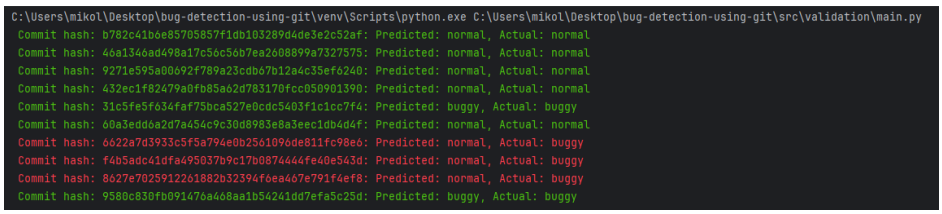
Najlepsze wyniki uzyskano dla modelu RandomForestClassifier z zastosowaniem techniki oversamplingu. Z tej przyczyny zdecydowano się wykorzystać ten model do przeprowadzenia walidacji.

Warto zauważyć, że w przypadku klasyfikatora drzewa decyzyjnego kluczowe znaczenie miały następujące cechy:

1. dmm_unit_size
2. msg_length
3. deletions
4. files
5. day_of_the_week

6 Walidacja

Walidacja modelu na podstawie nowych commitów jest kluczowym etapem dalszego rozwoju badania, ponieważ umożliwia ocenę, jak dobrze model radzi sobie z danymi, które nie były uwzględnione podczas treningu. Ten proces pozwala na oszacowanie praktycznej skuteczności modelu w realnym środowisku deweloperskim, co jest istotne dla jego ewentualnego wdrożenia i wykorzystania w praktyce.



Rysunek 22. Wyniki walidacji 10 commit'ów

Spośród 10 przygotowanych commit'ów 7 zostało poprawnie sklasyfikowanych przez model RandomForestClassifier.

Większość commit'ów z tych oznaczonych jako „normalne" stanowiło wprowadzenie jednej małej funkcji. Natomiast commitów oznaczonych jako „buggy" dotyczyły zmian zawierających błędy, często charakteryzujących się większą liczbą zmienionych linii oraz dłuższymi wiadomościami commitów.

7 Podsumowanie

W ramach niniejszego badania opracowano klasyfikator przewidujący defekty w oprogramowaniu, wykorzystując dane z repozytorium *microsoft/vscode* [3] na platformie GitHub. Zebrano informacje z pierwszych 10,000 commitów, identyfikując te, które wprowadzały błędy. Utworzono również zbiór 10 commitów ręcznie przygotowanych, aby zasymulować rzeczywiste wykorzystanie klasyfikatora. Następnie przeprowadzono wstępną obróbkę danych, w tym usunięcie anomalii i stworzenie nowych cech na podstawie istniejących atrybutów.

Testowano cztery różne modele klasyfikacyjne: drzewo decyzyjne (DecisionTreeClassifier), las losowy (RandomForestClassifier), maszynę wektorów nośnych (Support Vector Machine) oraz regresję logistyczną (Logistic Regression). Spośród tych modeli, najlepsze wyniki uzyskano dla klasyfikatora RandomForestClassifier z zastosowaniem techniki oversamplingu. Wyniki te były zgodne z wcześniejszymi oczekiwaniami, zakładającymi, że model ten poradzi sobie najlepiej.

Walidacja modelu na podstawie nowych commitów, które nie były uwzględnione podczas treningu, wykazała, że model poprawnie sklasyfikował 7 na 10 commitów. Większość commitów oznaczonych jako „normalne" dotyczyło wprowadzenia małych funkcji, natomiast commity oznaczone jako „buggy" zawierały błędy, często charakteryzujące się większą liczbą zmienionych linii oraz dłuższymi wiadomościami.

Warto jednak zauważyć, że model wykrywa głównie cechy meta dane związane z commitami, a nie bezpośrednie właściwości samego kodu. To ograniczenie może wpływać na skuteczność modelu w niektórych przypadkach. W przyszłości można rozważyć zastosowanie bardziej zaawansowanych technik, takich jak modele językowe o dużej skali (LLM), które mogłyby lepiej analizować rzeczywiste zmiany w kodzie i ich wpływ na stabilność oprogramowania.

Podsumowując, model RandomForestClassifier okazał się być najbardziej skutecznym narzędziem do przewidywania defektów w oprogramowaniu w oparciu o analizowane dane. Walidacja modelu w kontekście rzeczywistego środowiska deweloperskiego potwierdziła jego praktyczną użyteczność, co stanowi obiecujący krok w kierunku bardziej niezawodnego zarządzania jakością kodu w projektach programistycznych.

Literatura

1. ishepard. pydriller. https://github.com/ishepard/pydrillerLink to Repository.
2. Kristian Berg Daniel Hansson Markus Borg, Oscar Svensson. Szz unleashed: An open implementation of the szz algorithm. *MaLTeSQuE 2019: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019.
3. microsoft. vscode. https://github.com/microsoft/vscodeLink to Repository.
4. scikit learn. scikit-learn. https://github.com/scikit-learn/scikit-learnLink to Repository.