Politechnika Wrocławska

DZIEDZINA NAUK - NAUKI INŻYNIERYJNO-TECHNICZNE
DYSCYPLINA NAUKOWA - INFORMATYKA TECHNICZNA I TELEKOMUNIKACJA

ROZPRAWA DOKTORSKA

ZASTOSOWANIE METOD UCZENIA MASZYNOWEGO DZIAŁAJĄCYCH W CZASIE RZECZYWISTYM DO DIAGNOSTYKI I LOKALIZACJI BŁĘDÓW W STACJACH PRZEKAŹNIKOWYCH (BTS)

mgr inż. Wojciech Dobrowolski

Promotor:
prof. dr hab. inż. Olgierd Unold
Promotor pomocniczy:
dr inż. Maciej Nikodem

Słowa kluczowe: automatyczna analiza logów, lokalizacja błędu, sekwencja logów, segmentacja logów, sztuczna inteligencja

Wrocław 2024

Streszczenie

Lokalizacja i diagnostyka błędów w dużych systemach komputerowych nie mogą funkcjonować bez skutecznych metod wykrywania anomalii. Metody te są szeroko badane i zyskują na popularności, jednak ich zastosowanie w komercyjnych projektach pozostaje ograniczone. Wynika to z faktu, że wyniki osiągane na przykładowych zbiorach referencyjnych często znacząco odbiegają od wyników na zbiorach komercyjnych. Drugim powodem jest to, że metody lokalizacji i diagnostyki nierzadko opierają się na specyficznych cechach danego systemu.

Dążąc do zapewnienia uniwersalności rozwiązania, skoncentrowałem się na powszechnym źródle informacji na temat przebiegu wykonania, jakim są logi. Logi są szeroko wykorzystywane, jednak na drodze do ich efektywnego wykorzystania w wykrywaniu anomalii i lokalizowania błędu stoi brak regularnej struktury. Chociaż metody przetwarzania języka naturalnego (ang. *Natural Language Processing* - NLP) są szeroko stosowane do przetwarzania treści logów, metody nienadzorowanego segmentowania wyrazów nie zyskały jeszcze większej popularności i użycia do sekwencji logów.

W doktoracie zaproponowałem metody segmentacji i analizy logów, które pozwalają na lepsze wykrywanie anomalii i dokładniejsze lokalizowanie źródeł błędów w oprogramowaniu. Eksperymenty były przeprowadzone w dużej części na zbiorach logów produkcyjnych pochodzących z prawdziwych błędów firmy Nokia. Ich zanonimizowana postać jak i algorytm są dostępne online.

Summary

Localization and diagnosis of faults in large computer systems cannot function without effective anomaly detection methods. These methods are widely researched and are gaining popularity; however, their application in commercial projects remains limited. This is due to the fact that the results achieved on example benchmark datasets often significantly differ from those on commercial datasets. Another reason is that localization and diagnostic methods are often based on specific features of a given system.

Aiming to ensure the universality of the solution, I focused on a common source of information about execution, namely logs. Logs are widely used, but their lack of regular structure presents a barrier to their effective utilization in anomaly detection and fault localization. Although natural language processing (NLP) methods are widely applied to log content processing, unsupervised word segmentation methods have yet to gain wider popularity and use in log sequences.

In my doctoral work, I proposed log segmentation and analysis methods that enable better anomaly detection and more precise localization of software fault sources. Experiments were largely conducted on production log datasets from real faults provided by Nokia. Their anonymized form, as well as the algorithm, are available online.

Ad maiorem Dei gloriam.

Szczególne podziękowania należą się mojemu promotorowi
prof. dr. hab. inż. Olgierdowi Unoldowi, który wziął mnie
pod swoją opiekę w trudnym dla mnie czasie,
nie szczędząc czasu i mądrości.
Nie sposób nie wspomnieć nieocenionych uwag
i pomocy udzielonej przez promotora pomocniczego
dr. inż. Macieja Nikodema, który mimo wielości obowiązków
znajdował czas na bezcenne recenzowanie moich prac.
Podziękowania również kieruję do Macieja Gruszki.
Jego pomoc była kluczowa do rozpoczęcia całego procesu.

Pracę tę dedykuję mojej żonie Marcelinie,
oraz synom Józefowi i Karolowi.
Bez Waszej obecności praca ta nie mogłaby powstać
ani mieć ostatecznej wartości.

Spis treści

Rozdział 1
Wstęp

1.1 Cel rozprawy

Celem niniejszej rozprawy doktorskiej było opracowanie skutecznej metody ułatwiającej programistom odnalezienie błędu w oprogramowaniu. Źródłem informacji o błędzie są logi z wykonania, a do osiągnięcia celu wykorzystano metody oparte na sztucznej inteligencji. Cel ten osiągnąłem poprzez zastosowanie znanych w przetwarzaniu języka naturalnego metod segmentujących słowa do wykrywania znaczących segmentów w sekwencji logów oraz użycia tych segmentów w autorskiej metodzie lokalizacji błędu. Weryfikację rozwiązania przeprowadziłem na logach pochodzących ze zbiorów ogólnodostępnych, jednak nacisk kładłem na logi z błędnych wykonań w stacjach Base Transciever Stations (BTS). Na podstawie przeglądu literatury oraz doświadczenia w pracy z dużą liczbą logów z systemów Nokia oraz poprzednich firm, postawiłem następującą tezę:

Teza:
Zastosowanie metod uczenia maszynowego działających w czasie rzeczywistym pozwala na diagnostykę i lokalizację błędów w stacji przekaźnikowej o jakości nie gorszej od metod referencyjnych.

Na potrzebę udowodnienia tej tezy sformułowałem pięć celów, które należy osiągnąć:

C1 Rozszerzenie koncepcji sekwencji logów jako języka naturalnego oraz zastosowanie metod nienadzorowanej segmentacji słów do segmentowania sekwencji logów.

C2 Wykorzystanie uzyskanych segmentów do poprawy wykrywania anomalii w logach.

C3 Opracowanie algorytmu lokalizacji błędów działającego w czasie rzeczywistym bazującego na kontekście otrzymanym z segmentacji.

C4 Eksperymentalna ocena zaproponowanych metod.

C5 Opracowanie i implementację systemu opartego na powyższych badaniach, umożliwiającego pracownikom Nokii szybką analizę logów pochodzących z błędnego wykonania programu, zwracającego segmenty błędne oraz sugestie lokalizacji błędu.

1.2 Motywacja

Stacje BTS są nieodzownym elementem infrastruktury komunikacyjnej i stanowią część infrastruktury krytycznej. Ich niezawodne i poprawne działanie jest kluczowe nie tylko dla realizacji połączeń, ale także dla płatności online oraz zarządzanie urządzeniami internetu rzeczy (ang. *Internet of Things*). Problemy w działaniu BTS wynikają zarówno z działań intencjonalnych (ataków), jak i z niezamierzonych usterek, z których część wynika z błędów oprogramowania. Szybkie usuwanie tych błędów jest istotne, aby zapewnić niezawodną komunikację oraz uchronić firmy takie jak Nokia, dostarczające BTS wraz z oprogramowaniem, przed karami umownymi. Stacje BTS mają skomplikowaną architekturę składającą się z wielu komponentów, a ich liczba i rozproszenie sprawiają, że w przypadku wystąpienia błędu jego diagnoza i lokalizacja są często trudne.

Najpowszechniejszym źródłem informacji o błędach są logi. Jednakże ich liczba powoduje, że bardzo trudno jest uzyskać z nich oczekiwane informacje. Z drugiej strony logi nie są losową sekwencją zdarzeń systemu. Odzwierciedlają one jego wewnętrzną strukturę opracowaną przez wykwalifikowanych architektów. Odtworzenie tej struktury może przyczynić się do dokładniejszego określenia miejsca wystąpienia błędu. Dodatkowo istniejące rozwiązania osiągają wyśmienite wyniki na przykładowych zbiorach referencyjnych, ale zupełnie nie radzą sobie z logami pochodzącymi z systemów komercyjnych. Niezbędne jest zatem znalezienie metody, która na na podstawie logów produkcyjnych pozwoli osiągnąć oczekiwane rezultaty.

1.3 Architektura BTS

Stacje BTS to kluczowy element infrastruktury sieci komórkowych, odpowiedzialny za komunikację między urządzeniami mobilnymi a siecią rdzeniową. BTS, często określana jako stacja bazowa, stanowi podstawowy element w systemie komunikacji bezprzewodowej, takiej jak GSM, UMTS, LTE czy 5G, i jest niezbędna do zapewnienia pokrycia sygnałem radiowym danego obszaru. Architektura BTS jest złożona i obejmuje wiele różnych komponentów, które współpracują w celu efektywnej transmisji danych i zarządzania połączeniami.

Wraz z rozwojem technologii sieci komórkowych, architektura BTS również ewoluowała. W starszych sieciach GSM BTS były często dużymi, samodzielnymi jednostkami. W sieciach 4G LTE oraz 5G, BTS są bardziej zintegrowane i mogą być częścią większej infrastruktury, na przykład Distributed Antenna Systems (DAS) lub Cloud-RAN (C-RAN), gdzie zasoby radiowe są wirtualizowane i zarządzane centralnie.

1.3.1 Komponenty BTS

Architektura BTS składa się z kilku podstawowych elementów:

- Nadajniki/Odbiorniki (TRX) – Odpowiadają za nadawanie i odbiór sygnałów radiowych na określonych częstotliwościach. Każdy TRX może obsługiwać jedno lub więcej połączeń na różnych kanałach, co pozwala na równoczesną obsługę wielu użytkowników w jednym obszarze.

- Anteny – Anteny są odpowiedzialne za transmisję sygnału radiowego w określonym obszarze pokrycia. Mogą być kierunkowe, aby skupić sygnał w określonym

kierunku (sektorze), lub dookólne, pokrywając sygnałem cały obszar wokół BTS. Zazwyczaj BTS obsługuje kilka sektorów, każdy z własną anteną.

- Kontroler BTS (BSC) – BSC jest urządzeniem, które zarządza kilkoma BTS-ami i koordynuje ich działanie. BSC odpowiada za alokację zasobów, zarządzanie ręcznym przełączaniem użytkowników między stacjami BTS (ang. *handover*) i ogólną kontrolę jakości połączenia.

- Moduł zasilania – BTS wymaga stabilnego zasilania do działania. Moduł ten zapewnia odpowiednią moc dla wszystkich komponentów stacji bazowej. W wielu przypadkach BTS są również wyposażone w systemy zapasowe, takie jak baterie lub generatory, aby zapewnić ciągłość działania w razie awarii zasilania.

- System chłodzenia – Ze względu na intensywne zużycie energii oraz ciepło generowane przez elektronikę, BTS musi być wyposażona w odpowiedni system chłodzenia, który zapewnia optymalną temperaturę pracy.

- Połączenia transmisyjne – BTS musi mieć możliwość komunikacji z siecią rdzeniową. Zazwyczaj realizowane jest to poprzez światłowody, mikrofalowe łącza radiowe lub inne technologie transmisyjne, które umożliwiają przesyłanie danych do kontrolera (BSC) lub bezpośrednio do innych elementów sieci.

1.3.2 Funkcje BTS

Główną funkcją BTS jest zapewnienie interfejsu radiowego między siecią a użytkownikami końcowymi, czyli telefonami komórkowymi, smartfonami czy urządzeniami IoT. BTS przetwarza sygnały przychodzące i wychodzące, dbając o odpowiednią jakość połączeń głosowych i transmisji danych.

Stacja bazowa obsługuje także procedury związane z zarządzaniem połączeniami, takie jak inicjowanie i kończenie połączeń, utrzymywanie sygnału oraz realizowanie procedur związanych z przekazywaniem połączenia (ang. *handover*) między różnymi BTS-ami, gdy użytkownik się przemieszcza.

Podsumowując, Base Transceiver Station stanowi kluczowy element sieci komórkowej, którego zadaniem jest zapewnienie komunikacji radiowej. Dzięki swojej złożonej architekturze BTS skutecznie obsługuje połączenia mobilne. Rozwój BTS dostosowuje się do rosnących wymagań nowoczesnych sieci, takich jak LTE i 5G, oferując lepszą przepustowość, mniejszą latencję i bardziej niezawodną łączność.

Rozdział 2
Przegląd literatury

Analizie poddana zostanie literatura dotyczącą lokalizowania i diagnozowania błędów w oprogramowaniu, uwzględniając różne typy systemów. Po ogólnym wprowadzeniu skupię się na rozwiązaniach opartych wyłącznie na logach. Logi, jako najbardziej powszechne źródło informacji, i zarazem wymagające najmniejszej ingerencji w system, stanowią podstawę mojego podejścia. Pozwala to na niezależność od specyficznej architektury, inaczej niż ma to miejsce w przypadku rozwiązań dla mikro-serwisów.

Na początku warto określić, czym dokładnie jest błąd w oprogramowaniu. W jednym z najnowszych przeglądów literatury tematu mamy następujące rozróżnienie:

- "A failure is when a service deviates from its correct behavior.
- An error is a condition in a system that may lead to a failure.
- A fault is the underlying cause of an error, also known as a bug." [88]

W tłumaczeniu na polski oznacza to:

- Awaria ma miejsce, gdy usługa odbiega od swojego poprawnego działania.
- Błąd to stan w systemie, który może prowadzić do awarii.
- Usterka to podstawowa przyczyna błędu, znana również jako błąd (ang. *bug*).

W powyższych definicjach dwukrotnie pojawia się słowo "błąd". Pierwsze odnosi się do błędnego działania programu, drugie do usterki w programie. Niniejsza praca rozważa tę drugą sytuację, czyli próbuje zlokalizować usterkę w programie na podstawie symptomów - informacji na temat awarii, która wystąpiła w czasie działania systemu.

Diagnostyka błędów w oprogramowaniu składa się z obliczania miary anomalii komponentu oraz korelacji otrzymanych miar w celu lokalizacji błędu [53]. Taksonomia jednakże nie jest jednolita i często zdarza się, że metody opisane jako lokalizacja posiadają architekturę metod służących do diagnozy [102]. Lokalizacja błędu to ograniczenie ilości kodu do analizy w celu przyspieszenia znalezienia przyczyny i dokonania poprawki [88]. Lokalizacja stanowi ważny element Artificial Intelligence for IT Operations (AIOps) [64], czyli wykorzystania metod big data i uczenia maszynowego w celu wzmocnienia i zautomatyzowania operacji IT. AIOps to metodologia stosunkowo nowa, zapoczątkowana w 2017 roku, jednak jej elementy są znane od dawna. Nowość polega na ujednoliceniu i usystematyzowaniu znanych wcześniej technik. Jej rozwój łączy się z gwałtownym rozwojem rozwiązań opartych na sztucznej inteligencji i rozwojem sieci w architekturze Transformer. AIOps zyskuje na zainteresowaniu wśród dużych graczy, takich jak IBM [42], Alibaba [45] i Microsoft [16, 12].

AIOps składa się z wykrywania anomalii i symptomów związanych z błędem oraz ich lokalizacją poprzez analizę dużej ilości danych historycznych. Jest nastawiona na minimalizowanie fałszywych alarmów. Faza lokalizacji identyfikuje i analizuje potencjalną przyczynę błędu, badając przyczynowość i korelacje w kontekście topologii danego systemu. AIOps posiada też fazę akcji, w której incydenty są priorytetyzowane. Bardzo ważnym elementem AIOps jest faza interakcji, w której wyniki działania algorytmów uczenia maszynowego spotykają się z wiedzą ekspercką człowieka. Zaproponowany przeze mnie system czerpie z AIOps, jednak nie wymaga dużych ilości danych historycznych i jest niezależny od topologii systemu. Ograniczeniem tradycyjnego podejścia AIOPs jest złożoność wynikająca z dużej ilości danych, na których pracują i konieczności z dostosowania metod do topologii systemu. Rozwiązanie zaproponowane przeze mnie jest pozbawione tych wad. Dzięki temu system jest łatwy do zastosowania w każdej sytuacji, gdy logi są dostępne, nawet, gdy ich liczba jest ograniczona do jednego wykonania bez błędu i z błędem.

W literaturze spotykane są dwa podejścia: uni- i multimodalne. Zacznę od ogólnego przeglądu podejść multimodalnych, by przejść do podejść unimodalnych, które są podstawowym obszarem moich zainteresowań, w szczególności metody oparte na logach.

2.1 Dane multimodalne

Lokalizowanie błędów opiera się najczęściej na danych multimodalnych, czyli różnych informacjach związanych z przebiegiem wykonania/działania systemu. W ich skład najczęściej wchodzą: logi, ścieżki wykonania, metryki [103], wyniki testów, informacje tekstowe [88]. Często dodatkową informacją są zdarzenia i topologie. Gromadzenie tych informacji wymaga rozbudowania systemu o kod programu (tzw. instrumentację), który je generuje i zapisuje, albo wykorzystać istniejące rozwiązania (ang. *framework*) takie jak OpenTelemtry [60]. OpenTelemetry jest narzędziem, które pozwala poprawić obserwowalność wewnętrznego stanu systemu na podstawie informacji o przebiegu działania. Aby system spełniał założenia obserwowalności, musi generować: logi, ścieżki wykonania oraz metryki. Tak wzbogacony system jest dobrym podłożem do zastosowania metod automatycznej diagnozy i lokalizacji, takich jak AIOps [64].

Logi są generowane przez specjalne linie w kodzie źródłowym. Programiści umieszczają je w celu ułatwienia zrozumienia co się wydarzyło w czasie wykonania programu. Logi stanowią ślad po tym co wykonał program. Ten ślad ma być najmniejszy z możliwych, lecz wystarczający do odtworzenia przebiegu programu (ścieżki wykonania), tak by zrozumieć, co się wydarzyło. W zależności od porządku panującego w danej firmie, logi są bardziej lub mniej uporządkowane, dostarczając różną ilość dodatkowych danych. Niektóre firmy posiadają bardzo ścisłe reguły co i gdzie można logować, podczas gdy inne nie mają ustalonych jednolitych zasad. Powoduje to, że metody oparte na logach muszą wykazywać się poprawnym działaniem na różnorodnych, nieujednoliconych logach. Jeszcze niedawno takie zadanie wydawało się niemożliwe do realizacji, jednakże postęp, jaki nastąpił w ostatnich latach w obszarze przetwarzania danych generowanych przez człowieka sprawia, że przyszłość metod przetwarzających logi jest obiecująca. Logi generowane w wyniku wykonania systemu komputerowego mają szereg cech podobnych do języka naturalnego, co sugeruje, że narzędzia wykorzystywane do przetwarzania języka naturalnego można skutecznie dostosować do logów oprogramowania. Logi najczęściej rozpatruje się pod kątem części stałej i zmiennej. Część stała pokrywa się z tekstem umieszczonym w kodzie przez programistę, a część zmienna z tekstem generowanym

podczas działania programu.

Ścieżka wykonania to zapis przebiegu programu, który obejmuje wykonane operacje przez poszczególne serwisy i komponenty. Zapis przebiegu zawiera szereg informacji o jego wykonaniu i miar np.: znacznik czasu, identyfikator obiektu wywołującego i wywoływanego oraz parametry wywołania. Dzięki temu otrzymujemy szczegółowy przebieg obsługi programu, z którego można uzyskać informacje o potencjalnych wąskich gardłach, wizualizować drzewo wywołań i zrozumieć zależności między serwisami i komponentami programu. Ścieżki wywołania należą do podstawy algorytmów zwanych Slicing Based Fault Localization, w których ścieżki dynamiczne są otrzymywane w wyniku działania programu, a statyczne na podstawie statycznej analizy kodu programu.

Metryki to dane numeryczne dające wgląd w wewnętrzny stan systemu w danej chwili. Wartości metryk są wyznaczane na podstawie wybranych parametrów, takich jak obciążenie procesora, alokowana pamięć RAM, liczba obsłużonych żądań czy liczba błędów.

Zdarzenia to wpisy o szczególnej ważności logowane przez system i zapisywane w sytuacjach brzegowych istotnych z punktu widzenia architektów systemu. Topologie natomiast to zależności między komponentami systemu reprezentowane często jako graf. Taka topologia może być generowana automatycznie i pomaga w lepszym zrozumieniu błędu oraz bardziej precyzyjnym wskazaniu miejsca jego pochodzenia.

2.1.1 Podejścia niezależne od domeny

Istnieje szereg metod lokalizacji błędu, które nie wymagają wiedzy domenowej. Mają one tę przewagę, że łatwo jest je zastosować do każdego oprogramowania, natomiast są mniej precyzyjne od konkurentów wykorzystujących wiedzę domenową.

Podejście określane jako information retrieval based (IR-based) polega na wykorzystaniu tekstu z raportów o błędach do lokalizacji fragmentów oprogramowania, na przykład plików lub metod odpowiedzialnych za błąd [40, 76]. Technika ta wykorzystuje dwa źródła informacji: raport o błędzie i kod źródłowy. Raport o błędzie jest przygotowywany przez osobę, która dany błąd wykryła. Opisuje ona kroki, które wykonała i symptomy, które zauważyła. Może tam pojawić się również wycinek logów. Wykorzystując zaawansowane techniki osadzania słów w przestrzeni wektorowej, metoda IR kojarzy tekst raportu z kodem programu. Najbardziej podobny plik kodu jest potencjalnym źródłem błędu.

Rozwiązanie to zakłada dwie nieoczywiste rzeczy: że tekst raportu o błędzie będzie tak wysokiej jakości, że na jego podstawie będzie można wychwycić niuanse w podobnych błędach systemu. Drugim założeniem jest jakość kodu, a właściwie jego semantyczne właściwości. Kod taki, czyli nazwy zmiennych, metod, klas, powinien być pisany z użyciem zrozumiałych dla człowieka słów. Pierwsze założenie niestety nie zawsze jest spełnione, a nawet przy staraniach testera, może on nie być w stanie zobaczyć różnicy między dwoma różnymi przyczynami błędu tego samego typu - w złożonych systemach objawy kilku różnych błędów mogą być dokładnie takie same. Mimo inicjatyw takich jak Clean Code [52], kod w warstwie nazewnictwa zmiennych, metod i klas wciąż może znacząco odbiegać od semantycznie i syntaktycznie poprawnego tekstu. W konsekwencji może to istotnie utrudnić metodom IR-based pracę i spowodować, że będą bezużyteczne [83].

Metoda ReLink [90] zastępuje wykorzystanie kodu na rzecz zmian zarejestrowanych w systemie wersji. ReLink wykorzystuje szereg cech charakterystycznych dla

np. interwałów czasowych między momentem zgłoszenia naprawy błędu a zatwierdzeniem odpowiedniej zmiany w kodzie, zgodności między osobą odpowiedzialną za błąd a autorem zmiany oraz tekstowym podobieństwem między raportami o błędach a dziennikami zmian. Metoda ta stosuje techniki przetwarzania języka naturalnego, takie jak model przestrzeni wektorowej i miary podobieństwa kosinusowego, aby ocenić prawdopodobieństwo, że dany błąd i zmiana są powiązane.

W ClaFa [109] zamiast wiązać IR z poprawką, dane te są łączone w jeden wektor, by przewidywać dokładne miejsce w kodzie. Najpierw model tematyczny analizuje raport o błędzie, identyfikując w nim kluczowe tematy, które następnie są reprezentowane jako wektor tematyczny. ClaFa wykorzystuje Program Dependency Graph (PDG), strukturę danych używaną w informatyce do reprezentowania zależności między instrukcjami w programie komputerowym. PDG łączy zarówno dane, jak i przepływ sterowania w jednym grafie, co ułatwia analizę i optymalizację programów. W PDG wierzchołki reprezentują operacje lub instrukcje w programie, a krawędzie reprezentują zależności między tymi operacjami, zarówno przepływ danych (kiedy wynik jednej instrukcji jest używany w innej), jak i przepływ sterowania (określający kolejność wykonywania instrukcji). Równocześnie ClaFa ekstrahuje cechy z każdego wierzchołka w PDG kodu źródłowego, który jest tworzony zarówno dla działającej wersji jak i dla błędnej. Dzięki temu PDG odzwierciedla zmiany wprowadzone w ramach poprawek. Dla każdego wierzchołka w PDG tworzony jest wektor cech, który opisuje zarówno wierzchołek, jak i związane z nim zależności. Wektor tematyczny z raportu o błędzie jest następnie łączony z wektorem cech wierzchołka PDG, tworząc złożony wektor, który zawiera zarówno kontekstowe informacje o błędzie, jak i szczegółowe dane strukturalne o kodzie oraz jego poprawkach. Ten połączony wektor jest podawany do modelu klasyfikacyjnego, który przewiduje, czy dany wierzchołek jest błędny. Dzięki integracji informacji z raportów o błędach, struktury kodu oraz związanych z nim poprawek, ClaFa może dokładniej lokalizować błędy w programie.

Istnieją również rozwiązania, które idą krok dalej, zamieniają tekst z raportu o błędzie na zapytanie do bazy z kodem. Użytkownik ma wgląd w treść tego zapytania i jest ono zrozumiałe dla niego. Gdy wyniki zapytania nie zwracają poprawnej lokalizacji błędu, może on modyfikować jego treść tak, by otrzymać bardziej precyzyjne miejsce potencjalnego błędu [8, 56]. Zatem zamiast automatycznej reprezentacji raportu, użytkownik uzyskuje wgląd w tę reprezentację i może w nią ingerować używając swojej wiedzy eksperckiej.

2.1.2 Podejście zależne od domeny

Podejścia zależne od domeny wymagają najczęściej zaangażowania grupy ekspertów, aby wdrożyć proponowane rozwiązanie. Ich podstawową przewagą jest wyższa skuteczność od rozwiązań ogólnych. Przykładem takiego rozwiązania jest Groot [80], zaawansowany system analizy przyczyn awarii w systemach o architekturze mikrousług, który wykorzystuje różnorodne dane operacyjne do dokładnej diagnostyki problemów. Podstawowym źródłem danych są metryki operacyjne, które obejmują takie aspekty, jak zużycie CPU, opóźnienia w obsłudze zapytań, wykorzystanie pamięci, przepustowość sieci, liczbę błędów oraz inne wskaźniki wydajności mikrousług. Dane te są zbierane w czasie rzeczywistym z poszczególnych komponentów systemu, co pozwala na śledzenie ich stanu i identyfikację wszelkich nieprawidłowości. Dzięki tym danym Groot może wykrywać anomalie w zachowaniu systemu, które mogą prowadzić do awarii, a następnie

analizować je w celu ustalenia, które komponenty są odpowiedzialne za problem.

Wiedza domenowa odgrywa kluczową rolę w funkcjonowaniu Groot. System nie tylko analizuje surowe dane operacyjne, ale także korzysta z wiedzy na temat zależności i interakcji między mikrousługami. Dzięki zastosowaniu algorytmów odkrywania przyczynowości Groot jest w stanie tworzyć modele przyczynowe, które odzwierciedlają rzeczywiste relacje między różnymi metrykami operacyjnymi. W ten sposób system może precyzyjnie określić, które zmienne są bezpośrednio odpowiedzialne za zaistniałe problemy, a także przewidzieć potencjalne skutki tych problemów dla innych części systemu. Groot integruje tę wiedzę z danymi operacyjnymi, co umożliwia szybsze i dokładniejsze wykrywanie przyczyn awarii w złożonych środowiskach chmurowych.

2.1.3 Metody uczenia maszynowego

Metody uczenia maszynowego w podejściach opartych o dane multimodalne są wykorzystywane głównie do automatyzacji i poprawy dokładności lokalizowania błędów w kodzie źródłowym na podstawie analizy raportów o błędach oraz innych danych związanych z kodem. Każda z metod stosuje różne modele i techniki, które są dostosowywane do specyficznych aspektów problemu.

Metoda DeepLoc [92] jest przykładem modelu opartego na głębokim uczeniu, który służy do automatycznej lokalizacji plików zawierających błędy na podstawie raportów o błędach. DeepLoc wykorzystuje trzy konwolucyjne siecie neuronowe (ang. *Convolutional Neural Network* - CNN): dwie konwencjonalne do ekstrakcji cech z raportów o błędach i kodu źródłowego oraz trzecią, rozszerzoną, która uwzględnia historię napraw błędów. Różnica między DeepLoc a innymi metodami polega na unikalnym podejściu do przekształcania danych wejściowych (raporty i kod źródłowy) na wektory przy użyciu technik osadzania, takich jak Sent2Vec i Word2Vec, oraz na integracji historii napraw, co poprawia dokładność prognozowania.

W przeciwieństwie do DeepLoc, metoda BugTranslator [91] wykorzystuje sieć rekurencyjną (ang. *Recursive Neural Network* - RNN) z mechanizmem uwagi i komórkami pamięci (ang. *Long Short Term Memory* - LSTM) do tłumaczenia zgłoszeń błędów na odpowiadające im pliki kodu źródłowego. Kluczowym elementem jest tutaj zastosowanie mechanizmu uwagi, który pozwala na bardziej precyzyjne odwzorowanie kontekstu w zgłoszeniu błędu na kod źródłowy. BugTranslator różni się od DeepLoc tym, że nie koncentruje się na historii napraw, ale na bardziej dokładnym mapowaniu błędów na kod poprzez analizę abstrakcyjnych drzew składniowych (ang. *Abstract Syntax Tree* - AST) i zastosowanie mechanizmu uwagi.

Kolejna metoda, DeepFL [44], również wykorzystuje uczenie maszynowe do lokalizacji błędów, ale używa innych typów sieci neuronowych: wielowarstwowego perceptronu (ang. *Multi Layer Perceptron* - MLP) i RNN (w szczególności LSTM). Podczas gdy DeepLoc i BugTranslator koncentrują się na przetwarzaniu i analizie tekstu oraz raportów, DeepFL łączy różne grupy cech wyodrębnionych z kodu i testów, wykorzystując RNN do analizy sekwencji danych i integracji informacji z różnych źródeł. Cechą wyróżniającą DeepFL jest zastosowanie specyficznych funkcji aktywacji i hierarchicznych relacji między grupami cech, co umożliwia bardziej precyzyjne wykrywanie błędów w kodzie.

Z kolei metoda DEEPRL4FL [46] inspiruje się technikami śledztwa kryminalistycznego i składa się z trzech procesów w których model uczy się reprezentować trzy typy danych wejściowych: pokrycia kodu, zależności między instrukcjami oraz samego kodu źródłowego. DEEPRL4FL różni się od poprzednich metod tym, że kładzie nacisk na

reprezentację i analizę struktury kodu oraz zależności między instrukcjami, co pozwala na dokładniejsze zidentyfikowanie wadliwych fragmentów kodu. Metoda ta wykorzystuje CNN do nauki cech charakterystycznych i integruje różne źródła danych, takie jak grafy przepływu danych i struktury AST.

Wszystkie te metody mają na celu poprawę procesu lokalizacji błędów w kodzie, jednak różnią się podejściami do analizy danych oraz stosowanymi modelami uczenia maszynowego, co wpływa na ich specyfikę i zastosowania.

2.2 Dane unimodalne

Posiadanie tak szerokiej gamy danych, jak w podejściach multimodalnych, może być trudne do osiągnięcia ze względu na narzut w postaci dodatkowego oprogramowania, czasu i kosztów jego utrzymania oraz wymagań pamięciowych i obliczeniowych. Dlatego badane są też rozwiązania, które opierają się na jednym źródle informacji.

2.2.1 Dane z pokrycia kodu

Bardzo prężnie rozwijającą się dziedziną wykorzystującą dane unimodalne jest Spectrum Based Fault Localization (SBFL). Metody oparte na pokryciu kodu (spektrum) wykorzystują je do obliczania miary anomalii. Odbywa się to na podstawie informacji o tym, w ilu testach przechodzących i nieprzechodzących dana linia występuje. Następnie linie kodu z pokrycia są wiązane z wynikiem testu. Korelacja polega na tym, że te linie, które częściej występują w testach z błędem, są bardziej prawdopodobnym źródłem problemu. Stosowane są do tego metody statystyczne [1, 36, 87], jak i głębokie [84]. Wynikiem działania SBFL jest ranking linii, który programista może przejrzeć i analizować kolejne pozycje. W idealnym świecie linia z rzeczywistym błędem jest na szczycie propozycji.

Metody mutujące testy wzbogacają dane o spektrum pochodzące ze zmienionych przypadków testowych. Korzystając ze zbioru operatorów, zmieniane są linie z istniejącego spektrum i sprawdzany jest wynik testu. Mimo, że są lepsze niż SBFL, nie radzą sobie z błędami, które wymagają bardziej skomplikowanych zmian niż operatory mutujące.

2.2.1.1 Podejścia oparte na uczeniu maszynowym

Modele uczenia maszynowego również używają spektrum do przewidywania błędnych linii [89]. Najpierw sieć uczy się przewidywać rezultat testu na podstawie spektrum. Następnie, w czasie wnioskowania, na wejście podawane są tzw. wirtualne testy, czyli testy złożone tylko z jednej linii kodu. Zwrócony wynik wskazuje, czy dana linia jest potencjalnym źródłem błędu, czy nie. Metody głębokie używają pokrycia kodu reprezentowanego w postaci Coverage Matrix [107]. W przypadku stosowania sieci CNN, ważne jest, aby pokrycie kodu podobnych testów było umieszczone obok siebie w macierzy, ponieważ sieci CNN dobrze radzą sobie z wykrywaniem lokalnych zależności. Daje to sieci możliwość nauczenia się, które zmiany w wykonaniu testu powodują błąd.

Jedną z pierwszych zaproponowanych metod jest wspomniana już wcześniej metoda oparta na wstecznej propagacji [89]. W tej metodzie, dane dotyczące pokrycia kodu przez poszczególne testy (reprezentowane jako wektory binarne) oraz wyniki testów (sukces lub porażka) są wykorzystywane do trenowania trójwarstwowej sieci

neuronowej z wsteczną propagacją (ang. *back propagation* - BP). Sieć ta przewiduje prawdopodobieństwo, że dana instrukcja programu zawiera błąd. Wartość ta, nazywana "podejrzliwością", pozwala na sortowanie instrukcji w kolejności od najbardziej do najmniej podejrzanych, co znacząco przyspiesza proces lokalizacji błędu. Dodatkowo, efektywność metody zwiększono poprzez analizę przecięcia wycinków wykonania testów, które zakończyły się porażką, co pozwala na zawężenie liczby instrukcji wymagających dalszej analizy. Metoda ta łączy w sobie precyzyjne modelowanie danych testowych z efektywnymi technikami uczenia maszynowego, oferując skuteczne narzędzie do szybkiego i dokładnego lokalizowania błędów w złożonych systemach oprogramowania.

Metoda XGB-FL [95] wykorzystuje z kolei macierz pokrycia, ale nie używa metod głębokich. Składa się z trzech głównych etapów: budowania macierzy pokrycia, trenowania modelu XGBoost oraz generowania listy podejrzanych instrukcji programu. Macierz pokrycia w tej metodzie nie wymaga specjalnego układu wierszy i kolumn. Model ten przetwarza macierz, sortując dane według wartości cech, a następnie oblicza istotność cech (instrukcji programu). Ostatecznie, na podstawie uzyskanej listy najistotniejszych cech, generowana jest wykaz podejrzanych instrukcji, które mogą zawierać błąd.

W metodzie CNN-FL [107], korzysta się w pełni z potencjału sieci głębokich. Jednak, krokiem wstępnym jest ułożenie danych w macierzy pokrycia, tak by umożliwić sieci CNN działanie. Dane w macierzy pokrycia są układane tak, aby lepiej odzwierciedlać relacje pomiędzy pokryciem testów a wynikami tych testów. Macierz pokrycia jest macierzą o wymiarach $M \times N$, gdzie M to liczba przypadków testowych, a N to liczba instrukcji w kodzie źródłowym. W macierzy tej, każdy wiersz odpowiada jednemu przypadkowi testowemu, a każda kolumna odpowiada jednej instrukcji w kodzie. Sieci CNN jest używana do wychwytywania i uczenia się złożonych, nieliniowych wzorców w danych, co pozwala na bardziej precyzyjne i efektywne przypisywanie podejrzliwości poszczególnym instrukcjom programu. Dzięki temu CNN-FL może lepiej identyfikować błędne fragmenty kodu, nawet w przypadku skomplikowanych systemów oprogramowania, gdzie tradycyjne metody mogą nie być wystarczająco dokładne.

2.2.2 Przekrój programu

Sama informacja o pokryciu kodu nie zawsze wiąże się z rzeczywistą przyczyną błędu. Wynika to z faktu, że na wynik lokalizacji ma wpływ częstotliwość wykonania danej linii kodu. Może się zdarzyć, że poprawne fragmenty programu są wykonywane częściej niż rzeczywiście błędne fragmenty, co zafałszuje wynik SBFL. Dlatego istnieją metody, bardziej kosztowne, ale rozwiązujące ten problem.

Metoda oparta na przekroju programu (ang. *slicing*) identyfikuje dokładnie te instrukcje, które odpowiadają za dane wyjściowe systemu. Punktem odniesienia jest dana instrukcja lub zmienna, której wartość chcemy śledzić. Obserwowany jest przepływ przez wszystkie linie kodu, od jej utworzenia, modyfikacji aż do zwrócenia. Można otrzymać przekrój programu w sposób statyczny [7, 86], jak i dynamiczny [3, 4, 104] oraz łącząc oba podejścia [51], aby ograniczyć wielkość statycznego przekroju programu i jednocześnie poprawić jakość dynamicznego.

Statyczny przekrój programu wymaga analizy całego kodu źródłowego, z którego wybiera wszystkie instrukcje, które mogą wpływać na wynik w określonym punkcie. Wymaga to osobnego skomplikowanego i trudnego do utrzymania kodu. Dynamiczny przekrój wymaga zaś instrumentowania kodu, co wpływa na czas jego wykonania, ale zwraca tylko instrukcje, które w znaczący sposób wpłynęły na wynik. Analiza w czasie

rzeczywistym powala sprawdzać nie tylko instrukcje, od których zależy wynik, ale i wartości. W obu przypadkach wymagane są narzędzia instrumentacji kodu, których dostosowanie może być problematyczne samo w sobie. Statyczny przekrój jest bardziej kosztowny pamięciowo i obliczeniowo, natomiast dynamiczny przekrój jest bardziej wydajny i skuteczniejszy, ponieważ analizuje tylko i dokładnie tę ścieżkę, która została wykonana w czasie powstania błędu [71].

2.2.3 Dane ze ścieżek

Metoda MEPFL (ang. *Microservice Error Prediction and Fault Localization*) [110] to zaawansowane podejście służące do przewidywania błędów i lokalizacji usterek w aplikacjach opartych na mikroserwisach. Kluczowym elementem tej metody jest wykorzystanie czterech modeli predykcyjnych, które umożliwiają analizę na dwóch poziomach: ścieżki (ang. *trace-level*) oraz mikroserwisu (ang. *microservice-level*). Modele te pozwalają na kompleksową diagnozę problemów, łącząc kontekst specyficzny dla aplikacji z bardziej uniwersalnymi cechami mikroserwisów.

Na poziomie ścieżki metoda MEPFL stosuje trzy różne modele: LE (ang. *Latent Error*), FM (ang. *Faulty Microservice*) oraz FT (ang. *Fault Type*). Model LE służy do klasyfikacji ścieżek jako zawierających błąd lub nie, model FM identyfikuje wadliwe mikroserwisy w ramach danego śledzenia, natomiast model FT określa, jakie rodzaje usterek występują. Modele te opierają się na cechach specyficznych dla danej aplikacji, co pozwala na uwzględnienie kontekstu aplikacji w procesie predykcji.

Model MS (ang. *Microservice Status*) działa na poziomie mikroserwisu i jest niezależny od specyfiki aplikacji. Ocena statusu mikroserwisu przez ten model jest bardziej uniwersalna, co umożliwia jego zastosowanie w różnych aplikacjach. Wszystkie modele MEPFL to modele klasyfikacyjne, które wykorzystują techniki uczenia maszynowego, takie jak losowy las (ang. *Random Forests*), K-najbliższych sąsiadów (ang. *K-Nearest Neighbors*) oraz MLP, co pozwala na skuteczne przewidywanie błędów i lokalizowanie usterek w mikroserwisach.

MEPFL działa w dwóch fazach: offline i online. W fazie offline metoda zbiera dane treningowe z logów ścieżek aplikacji, uruchamianej w środowisku testowym. Aby zebrać różnorodne dane, wprowadza się błędy za pomocą specjalnych strategii wstrzykiwania błędów. Dane te są następnie wykorzystywane do trenowania modeli predykcyjnych z zastosowaniem technik uczenia maszynowego. W fazie online MEPFL monitoruje aplikację w środowisku produkcyjnym, analizując nowe dane i wykorzystując modele do identyfikacji potencjalnych błędów, lokalizacji usterek oraz określenia ich rodzaju.

Metoda MEPFL pozwala na szybką identyfikację problemów w mikroserwisach, co jest kluczowe w złożonych aplikacjach rozproszonych, gdzie lokalizacja i naprawa usterek mogą być trudne i czasochłonne. Dzięki zastosowaniu zaawansowanych modeli uczenia maszynowego, MEPFL wspiera ciągłą i efektywną diagnostykę błędów, co może znacząco poprawić stabilność i niezawodność aplikacji opartych na mikroserwisach.

2.2.4 Dane z logów

W rzeczywistych systemach komputerowych dostęp do szerokiego zakresu informacji na temat działania systemu nie zawsze jest możliwy. Instrumentacja systemu, umożliwiająca rejestrowanie większej ilości danych o jego działaniu, wprowadza narzut, który może być nieakceptowalny dla architektów systemu. W efekcie, w wielu przypadkach

logi są jedynym źródłem informacji o działaniu systemu. Na takich sytuacjach koncentruje się moja praca, dlatego metody bazujące wyłącznie na logach zostaną opisane w sposób bardziej szczegółowy.

Logi są tradycyjną formą uzyskiwania informacji o działaniu programu. Otrzymywane są z wyrażeń logujących w kodzie źródłowym, takich jak *print*, które są wprowadzane tylko w tym celu i są pomocne przy odnajdywaniu błędów przez programistów [88]. Istnieje kilka powodów, dla których metody automatycznie analizujące logi bazują wyłącznie na nich. Pierwszym czynnikiem jest minimalna ingerencja. Narzędzia używane do automatycznej analizy nie powinny wprowadzać zmian w kodzie, ponieważ same mogą być źródłem błędu. Jeśli zmiany są konieczne, ingerencja powinna być jak najmniejsza. Po drugie, rozwiązanie powinno być niezależne od technologii wytwarzania oprogramowania. Bogactwo różnych technologii sprawia, że opracowanie rozwiązania dla jednej tylko technologii jest mocno ograniczające dla szerokiego zastosowania.

Istnieją różne podejścia do lokalizacji błędu na podstawie logów. Pierwsze to lokalizacja bezpośrednio linii związanej z błędem [17, 67], a drugie to wykrycie najpierw anomalii w sekwencji logów, a następnie lokalizacja przyczyny [82, 102].

Metody lokalizacji błędów na podstawie logów dzieli się również na podejścia zależne od domeny (ang. *domain dependent*), oparte na przyczynowości (ang. *causality based*) oraz podejścia oparte na uczeniu maszynowym (ang. *machine learning based*).

Każde z tych podejść ma swoje zalety i ograniczenia, a wybór odpowiedniego zależy od specyficznych wymagań i kontekstu systemu, w którym są stosowane.

2.2.4.1 Podejścia zależne od domeny

W podejściu nazywanym Kahuna [75], przeznaczonym do lokalizowania problemów wydajnościowych w systemach opartych na MapReduce [18], kluczową rolę odgrywa analiza logów i metryk systemowych z poszczególnych węzłów klastra - jednostek (komputerów lub serwerów) tworzących rozproszoną infrastrukturę do przetwarzania dużych zbiorów danych. Metoda Kahuna wykorzystuje zarówno podejście tzw. „czarnej skrzynki" (ang. *black-box*), jak i „białej skrzynki" (ang. *white-box*). W przypadku "czarnej skrzynki" wykorzystywane dane obejmują metryki systemowe zbierane na poziomie systemu operacyjnego, takie jak użycie procesora, liczba przełączeń kontekstu, czy obciążenie sieci. Z kolei w przypadku podejścia "białej skrzynki" dane pochodzą bezpośrednio z logów, i są analizowane za pomocą narzędzia, które przekształca logi w widoki maszyn stanowych, ułatwiające identyfikację problematycznych węzłów w klastrze.

Kahuna w pełni wykorzystuje wiedzę domenową, aby zidentyfikować wierzchołki, które zachowują się inaczej niż reszta w klastrze, co może wskazywać na problemy wydajnościowe. Kluczową obserwacją, na której opiera się Kahuna, jest „podobieństwo rówieśnicze" (ang. *peer-similarity*), czyli założenie, że wierzchołki w klastrze powinny zachowywać się podobnie w poprawnych warunkach. wierzchołki, które wykazują znaczące odstępstwa od tego wzorca, są identyfikowane jako potencjalne źródła problemów. Podejście to pozwala na skuteczne diagnozowanie problemów w dużych klastrach MapReduce, nawet w obecności zróżnicowanych obciążeń i przy minimalnym wpływie na działanie systemu.

W narzędziach Pattern-Oriented Discovery (POD-Discovery) i Pattern-Oriented Vizualization (POD-Viz) [85], logi stanowią fundamentalne źródło danych wykorzystywanych do modelowania i wizualizacji procesów operacyjnych. POD-Discovery

automatycznie przekształca surowe logi z systemów, takich jak operacyjne logi serwerów, w abstrakcyjne modele procesów. Proces ten obejmuje klasteryzację logów na wyższy poziom zdarzeń, co pozwala na uproszczenie skomplikowanych sekwencji operacyjnych do bardziej zrozumiałych jednostek procesowych. Logi są analizowane przy użyciu zaawansowanych technik klasteryzacji, które grupują linie logów w odpowiednie aktywności procesowe, a następnie są przekształcane w ścieżki zdarzeń, które mogą być analizowane i wizualizowane w czasie rzeczywistym przez POD-Viz. Dzięki temu logi nie tylko umożliwiają śledzenie bieżących procesów, ale również pozwalają na analizę retrospektywną, co jest kluczowe dla lokalizowania błędów i rozwiązywania problemów operacyjnych.

Wiedza domenowa odgrywa kluczową rolę w efektywnym wykorzystaniu narzędzi POD-Discovery i POD-Viz do modelowania i monitorowania procesów operacyjnych. POD-Discovery wykorzystuje wiedzę ekspertów w zakresie operacji do definiowania reguł klasteryzacji logów, co umożliwia dokładne odwzorowanie rzeczywistych procesów operacyjnych w abstrakcyjnym modelu procesowym. Wiedza ta jest szczególnie istotna w kontekście dostosowywania narzędzia do specyficznych wymagań danego systemu, co obejmuje dostosowanie metryk odległości między liniami logów oraz konfigurowanie reguł transformacji, które przekładają surowe dane logów na znaczące zdarzenia procesowe. POD-Viz z kolei wykorzystuje wiedzę domenową, aby umożliwić operatorom monitorowanie procesów w czasie rzeczywistym, identyfikację anomalii oraz analizę przyczyn błędów, poprzez interaktywną wizualizację procesów, która jest dostosowana do specyfiki działania konkretnego systemu. Dzięki integracji wiedzy domenowej, narzędzia POD są w stanie dostarczać bardziej precyzyjnych i użytecznych informacji, co przekłada się na zwiększenie niezawodności i efektywności operacyjnej systemów.

Dla oprogramowania grupy LAMP (Linux, Apache, MySQL, PHP) zaproponowano metodę która nazywa się LBFL [67], jako, że bazuje na metodzie SBFL używając logów zamiast spektrum. Jest to metoda oparta na wiedzy domenowej, ale już na tyle uniwersalna, że może być wykorzystana w różnych systemach tworzonych w oparciu o LAMP. Wspólną cechą tych rozwiązań jest generowanie wielu logów w różnych plikach, co utrudnia identyfikację przyczyn problemów. W metodzie LBFL zaproponowano podejście pozwalające na lokalizację logów związanych z awarią poprzez zastosowanie techniki SBFL (ang. *Spectrum-Based Fault Localization*). Podobną wersję tego podejścia wykorzystuję w moim rozwiązaniu, dlatego opiszę ją szerzej.

Metoda rozpoczyna się od parsowania logów, aby znaleźć i zamaskować zmienne części komunikatów, takie jak numery procesów czy wartości czasu. Dzięki temu, logi są bardziej jednolite i łatwiejsze do analizy. Następnie, logi są poddawane dwóm analizom: (A) zastosowaniu SBFL do logów oraz (B) identyfikacji powiązań między komunikatami z różnych warstw stosu LAMP. Analizy te wykonywane są niezależnie, mogą być łączone, co poprawia ich użyteczność w oprogramowaniu typu LAMP, natomiast mogą być też stosowane oddzielnie.

W analizie A, SBFL jest używany do identyfikacji logów, które mogą być związane z błędnym wykonaniem. Tradycyjnie, SBFL służy do lokalizowania błędów w kodzie na podstawie ścieżek wykonania testów, ale tutaj metoda ta jest dostosowana do analizy logów, które zastępują jednostki kodu, a pliki logów odpowiadają testom.

Analiza A zaczyna się od połączenia logów z różnych zestawów oprogramowania w jeden plik logów, a następnie oblicza znormalizowaną liczbę wystąpień każdego szablonu logu. Następnie obliczany jest wskaźnik podejrzanego logu, a komunikaty są sortowane od najbardziej do najmniej podejrzanych.

Druga analiza (B) polega na identyfikacji powiązań między logami z różnych warstw stosu LAMP. Chodzi tu o znalezienie wspólnych ciągów znaków w logach, które mogą sugerować, że są one powiązane ze sobą, np. komunikat z logu MySQL może pojawić się również w logu aplikacji. Analiza ta może być szczególnie przydatne w przypadkach, gdy tradycyjna analiza SBFL nie jest wystarczająca do lokalizacji problemu.

Obie analizy są stosowane niezależnie, ale mogą też być łączone, aby uzyskać pełniejszy obraz przyczyn powstania błędu. SBFL pozwala na wyłonienie najbardziej podejrzanych komunikatów, podczas gdy identyfikacja powiązań między warstwami może pomóc w potwierdzeniu tych wyników lub wskazać na inne przyczyny problemów. Metoda ta jest szczególnie przydatna w złożonych aplikacjach, gdzie wiele komunikatów logów pojawia się jednocześnie, co utrudnia ich analizę.

2.2.4.2 Podejścia oparte na wnioskowaniu przyczynowym

Podejście oparte na wnioskowaniu przyczynowym polega na wydobyciu w sposób automatyczny lub półautomatyczny grafu zależności między komponentami z dostępnych logów. Graf zależności posiada serwisy jako wierzchołki, a ich zależności są określone przez krawędzie. Dzięki temu, gdy wykrywana jest anomalia w kilku wierzchołkach, poszukiwany jest węzeł połączony z nimi relacją przyczynowo-skutkową [2, 33]. Dzięki odnalezieniu takiego wierzchołka możliwe jest wskazanie go jako potencjalnego źródła błędu wszystkich znalezionych anomalii.

Relacja przyczynowo-skutkowa jest również podstawą dla rozwiązania prezentowanego w [32], gdzie logi stanowią kluczowy element w procesie identyfikacji przyczyn awarii. Dane z logów są zbierane z mikroserwisów, a następnie analizowane w celu wykrycia anomalii, które mogą być uznane za przyczyny awarii. Logi dostarczają metryk i informacji o stanie mikroserwisów przed i po wystąpieniu awarii, co pozwala na stworzenie zestawu danych obserwacyjnych (dla popranego stanu) oraz anomalii. W ten sposób logi stają się podstawą do budowania graficznego modelu przyczynowości, który umożliwia identyfikację przyczyny źródłowej awarii poprzez analizę zmian w rozkładach metryk w okresie anomalii.

W artykule [34] główny nacisk położono na wykorzystanie logów wywołań API (ang. *Application Programming Interface*) do diagnozowania anomalii w mikroserwisach. Zamiast polegać na rozbudowanej infrastrukturze monitorowania, MicroCU analizuje minimalne dane diagnostyczne, jakie dostarczają logi API, rejestrując takie informacje jak ścieżki żądań, kody odpowiedzi i opóźnienia. Logi te, zbierane przez bramkę API, stanowią podstawę do wygenerowania macierzy opóźnień. W ten sposób, nawet przy dużym stopniu niekompletności danych, system jest w stanie śledzić dynamikę żądań i odpowiedzi w mikroserwisach, co umożliwia diagnozowanie problemów bez potrzeby zaawansowanej infrastruktury monitorującej.

Drugim kluczowym elementem opisanego rozwiązania jest tworzenie dynamicznych grafów zależności czasowych, które są oparte na analizie przyczynowej generowanej z logów. MicroCU dzieli dane na krótkie interwały czasowe i przeprowadza testy przyczynowości Grangera [28], aby zidentyfikować, które usługi mają wpływ na siebie nawzajem w danym okresie. Następnie z tych wyników budowany jest dynamiczny graf zależności, w którym wagi krawędzi odzwierciedlają siłę tych zależności. W celu zmniejszenia liczby fałszywych połączeń, autorzy zastosowali nową technikę przekształcania krawędzi, co pozwala na wyeliminowanie nieistotnych zależności i stworzenie bardziej precyzyjnego grafu, który następnie jest używany do śledzenia ścieżek propagacji błędów i identyfikacji źródeł problemów w systemie.

2.2.4.3 Podejścia oparte na uczeniu maszynowym

Zastosowanie metod uczenia maszynowego wymaga wcześniejszej reprezentacji danych źródłowych za pomocą wektora liczbowego. Proces ten zazwyczaj jest precyzyjnie zaprojektowany przez człowieka, który spodziewa się, że dana reprezentacja przyniesie pożądany zysk w procesie uczenia. LogDC [93] ekstrahuje cechy z prawidłowych deklaracji plików wdrożenia (ang. *deployment*) aplikacji oraz logi z przebiegu wdrożenia i trenuje na tym klasyfikator, którym jest naiwny klasyfikator Bayesa. Naturą tego modelu jest traktowanie wejściowych danych jako niezależne zmienne losowe. Dzięki temu model jest odporny na zaburzenia w chronologii, z drugiej strony taki model nie potrafi rozpoznać błędów, w których kolejność ma znaczenie.

W procesie identyfikacji błędów w logach testowych w firmie Ericsson zidentyfikowano dwie kluczowe trudności: skomplikowane środowisko testowe generujące wiele błędów niezwiązanych z produktem oraz olbrzymią ilość szczegółowych informacji zawartych w logach [5]. Aby przezwyciężyć te wyzwania, zastosowano metodę abstrakcji logów, która usuwa informacje kontekstowe, takie jak data uruchomienia testu i inne parametry. Następnie porównano logi niepowodzeń z ostatnimi logami poprawnych testów, aby usunąć linie, które nie są związane z awarią, co nazwano operacją DiffWithPass. Na końcu, zidentyfikowano najrzadsze linie logów i wykorzystano techniki przetwarzania informacji do określenia najbardziej prawdopodobnej przyczyny błędu.

SwissLog [43] również uczy się na danych historycznych (faza offline), ale ma również możliwość działania online, na logach pochodzących z działającego systemu. W fazie offline SwissLog buduje graf relacji identyfikatorów komponentów systemu (ID) oraz buduje bazę szablonów logów. Szablony logów są przechowywane jako zdania, a nie tylko jako identyfikatory zdarzeń. Następnie sekwencje logów są przekształcane w informacje semantyczne i czasowe. Do kodowania informacji semantycznych wykorzystywany jest model BERT, który generuje osadzenia (ang. *embeddings*) zarówno dla kontekstu, jak i dla informacji czasowych. Połączenie tych osadzeń jest następnie przekazywane do modelu Attention-based Bi-directional Long Short Term Memory (Attn-based Bi-LSTM), który uczy się cech poprawnych, błędnych oraz związanych z anomaliami wydajności.

W fazie online, gdy nowy log jest przekazywany na wejście do modelu, SwissLog kojarzy go z wcześniej przetworzonymi danymi na podstawie identyfikatorów, a następnie tworzy graf relacji dla danego logu. Log jest następnie przetwarzany przez etapy parsowania, kodowania zdań oraz przekazywany do wytrenowanego modelu z fazy offline. W przypadku wykrycia anomalii, system alarmuje o jej wystąpieniu i przechodzi do procesu lokalizacji anomalii, w którym analizowana jest instancja odpowiadająca za błąd.

Budowa grafu relacji identyfikatorów (ID) w SwissLog, który reprezentuje hierarchiczne zależności pomiędzy ID, opiera się na wykorzystaniu istniejących technik. Zależności te mogą być różne: brak zależności, 1:1, 1:n oraz m:n. SwissLog wykorzystuje te zależności do lokalizowania anomalii, bazując na założeniu, że anomalia w niskopoziomowych instancjach będzie propagować się do instancji wyższego poziomu. W przypadku wykrycia anomalii system przeszukuje graf w celu zlokalizowania dokładnej przyczyny błędu.

2.2.5 Wykorzystanie kontekstu

Kontekst odgrywa kluczową rolę w dwóch różnych metodach lokalizacji błędów: "Deep Learning with Contextual Information" [108] oraz "Context-based Cluster Fault Localization" (CBCFL) [99]. Chociaż obie metody wykorzystują dynamiczny przekrój programu (Rozdział 2.2.2) do identyfikacji instrukcji wpływających na błędne wyniki, robią to w odmienny sposób.

W przypadku metody z głębokim uczeniem [108], dynamiczny przekrój jest używany jako dodatkowe dane wejściowe do modelu uczenia maszynowego. Dynamiczny przekrój identyfikuje instrukcje, które mają bezpośredni wpływ na błędne wyniki testów. Te instrukcje tworzą kontekst, który wzbogaca dane, jakie sieć neuronowa wykorzystuje do oceny podejrzliwości instrukcji. Dzięki temu model jest w stanie lepiej zrozumieć zależności w programie, co prowadzi do dokładniejszego rankingu podejrzanych instrukcji i sprawniejszego lokalizowania błędu przez programistę.

Z kolei w CBCFL [99] dynamiczny przekrój służy do identyfikacji sytuacji, w których program wykonuje się poprawnie, to znaczy test kończy się sukcesem, mimo że zostały wykonane linie błędne (przypadkowa poprawność ang. *coincidental correctness* - CC). Po stworzeniu dynamicznego przekroju pochodzącego z testów błędnych, który redukuje zbiór instrukcji programu do tych mających wpływ na zwracaną wartość, CBCFL przeprowadza grupowanie testów z sukcesem. To pozwala na wykrycie testów z przypadkową poprawnością i oznaczenie ich jako błędnych. Dopiero wtedy, na tak poprawionych testach przeprowadzana jest standardowa lokalizacja bazująca na spektrum.

Podsumowując, choć oba podejścia korzystają z kontekstu, różnią się one w zastosowaniu. W głębokim uczeniu kontekst poprawia jakość danych wejściowych do modelu, podczas gdy w CBCFL kontekst służy do oczyszczenia danych, a w rezultacie poprawy jakości metody partej na spektrum. Obie metody pokazują, że kontekst może być wszechstronnym narzędziem, które odpowiednio zastosowane, znacząco poprawia skuteczność lokalizacji błędów.

2.3 Wykrywanie anomalii w logach

Wykrywanie anomalii w logach jest osobnym kierunkiem badań, jak i pierwszym krokiem w wielu metodach lokalizacji błędów. Pomimo to, nie zawsze osiągnięcia w wykrywaniu anomalii przekładają się na metody lokalizacji. Prezentuję zatem najpierw podejścia znane z lokalizacji, a następnie osiągnięcia w obszarze wykrywania anomalii z naciskiem na to jak można te osiągnięcia przenieść na lokalizację błędu.

2.3.1 Wykrywanie anomalii w czasie lokalizacji błędu

Wiele systemów lokalizacji błędu zaczyna swoje działanie od wykrycia anomalii, a następnie tworzy ich ranking. Systemy te można podzielić na kilka kategorii w zależności od tego, jakiego rodzaju informacje są używane do wykrywania anomalii: identyfikatory szablonów lub zawartość szablonów. Podział można też wprowadzić według tego czy algorytmy uwzględniają kolejność zdarzeń, czy tekst w logach. Aby lepiej zrozumieć, jak różne algorytmy radzą sobie z wykrywaniem anomalii, przyjrzyjmy się im w kontekście tych dwóch kryteriów.

Algorytmy takie jak *Log Based Fault Localization (LBFL)* oraz *LOGFAULTFLAGGER* opierają się na identyfikatorach szablonów, które reprezentują zgrupowane linie logów. Proces lokalizacji błędów w tych algorytmach polega na analizie statystycznej, której celem jest porównanie szablonów występujących w poprawnych i błędnych wykonaniach programu. LBFL zakłada, że błędy są związane z określonymi szablonami logów, a identyfikatory szablonów służą do przypisania wag podejrzliwości [67]. LOGFAULTFLAGGER z kolei usuwa linie wspólne dla poprawnych i błędnych wykonań, a następnie nadaje wagę pozostałym szablonom, co umożliwia określenie, które z nich mogą wskazywać na błąd [5].

Oba algorytmy bazują na tym, że logi z błędnych wykonań różnią się od logów z poprawnych wykonań jedynie w pewnych miejscach, a ich celem jest izolacja tych różnic poprzez analizę wystąpień szablonów. W tych podejściach nie jest uwzględniana dokładna zawartość logów, a kluczowa jest liczba wystąpień identyfikatorów szablonów.

Z kolei algorytmy takie jak *LogDC* oraz *SwissLog* operują na zawartości samych szablonów, a nie na ich identyfikatorach. LogDC tworzy wektory cech na podstawie pełnej zawartości szablonów, a następnie używa algorytmów takich jak k-średnich (ang. *k-means*) i regresja logistyczna do wykrywania anomalii [93]. SwissLog idzie o krok dalej, wykorzystując osadzenia słów angielskich (ang. *embeddings*) BERT do reprezentacji szablonów, co pozwala na bardziej zaawansowaną analizę semantyczną logów. Zawartość szablonów staje się kluczowa, ponieważ bardziej precyzyjnie oddaje to, co się wydarzyło w systemie.

Te algorytmy są szczególnie przydatne w sytuacjach, gdy struktura logów jest skomplikowana, a różnice między poprawnymi a błędnymi wykonaniami nie mogą być łatwo zidentyfikowane na podstawie samych identyfikatorów szablonów.

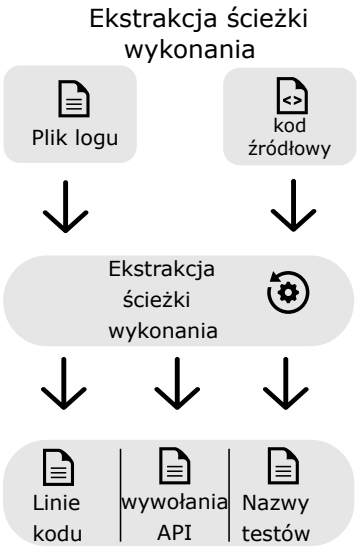
Niektóre algorytmy, jak *POD-Discovery* oraz *Kahuna*, oraz wykorzystujące wnioskowanie przyczynowo-skutkowe [2, 32, 33, 34] skupiają się na analizie sekwencji i przepływu logów. W Kahuna, charakterystyka logów MapReduce jest używana do tworzenia przepływów kontrolnych, które następnie są analizowane pod kątem różnic od normy. Błędy są zgłaszane, gdy przepływ logów różni się od oczekiwanego. POD-Discovery grupuje logi na podstawie odległości między elementami szablonów i umożliwia użytkownikowi interaktywne wybieranie klastrów, które najlepiej odpowiadają jego oczekiwaniom. Każdy klaster reprezentuje określoną czynność w procesie, co pozwala na analizę odstępstw od typowego przebiegu [93].

Podsumowując, algorytmy do lokalizacji błędów opartych na logach można podzielić na te, które używają identyfikatorów szablonów, i te, które operują na pełnej zawartości logów. Dodatkowo, niektóre z tych algorytmów uwzględniają kolejność zdarzeń, podczas gdy inne koncentrują się wyłącznie na zawartości logów. Wybór odpowiedniego algorytmu zależy od specyfiki analizowanego systemu i dostępnych logów.

2.3.2 Podejścia klasyczne

Przez podejście klasyczne mam na myśli metody oparte wyłącznie na logach, które są rozwiązaniem najbardziej uniwersalnym. Podejście to korzysta z metod statystycznych, uczenia maszynowego lub głębokiego. Wykrywanie anomalii na podstawie logów jest zadaniem trudnym, głównie dlatego, że oprogramowanie jest stale rozwijane, nowe logi są dodawane, a stare są modyfikowane lub usuwane. Większość metod uczy się na podstawie przeszłości, czyli poprzednich wersji. Natomiast, większość awarii występuje we wczesnych etapach rozwoju nowej wersji, gdzie może brakować wystarczającej ilości

historycznych danych do nauczenia się poprawnych i błędnych sekwencji.



Rysunek 1: Schemat procesu przetwarzania ścieżki wykonania programu na podstawie linii logów i odszukania najbardziej prawdopodobnej ścieżki wykonania. W wyniku można otrzymać dokładne linie kodu, wywołania API lub oznaczyć segmenty kodu najbardziej podobnymi testami widzianymi podczas treningu.

2.3.2.1 Podejścia statystyczne

Metody statystyczne mogą uchwycić związek między liniami logów w jednym pliku logu. Wymagają one jednak dużego zaangażowania ekspertów, aby wybrać ważne cechy. Po ich wybraniu, cechy te nie dają się łatwo przenieść na inne systemy i proces musi być powtórzony.

Podstawowym krokiem w większości metod jest parsowanie logów, które polega na rozpoznaniu części stałych komunikatów (tzw. szablonów logów) oraz zmiennych (np. identyfikatorów obiektów czy stanów operacji). W jednej z pierwszych metod [94] służy do tego analiza kodu źródłowego. Umożliwia ona dokładne określenie struktury każdego logu, co eliminuje konieczność stosowania heurystyk i zgadywania, jak to ma miejsce w innych metodach parsowania. Z drugiej strony wymaga bardzo precyzyjnych reguł wychwytujących linie kodu źródłowego generujące log, co w przypadku dużych systemów, może być bardzo trudne. Metoda [94] wykorzystuje obie części z parsowanych logów do stworzenia dwóch wektorów: wektora stanów (ang. *state ratio vector*) i wektora liczby szablonów (ang. *message count vector*). Wektor stanów przedstawia częstotliwość występowania różnych wartości stanu w danym oknie czasowym takich jak otwarcie pliku, zamknięcie pliku, ustanowienie połączenia i utracenie go. Wektor liczby szablonów z kolei zlicza liczbę wystąpień szablonów w tym samym oknie czasowym. Dla obu zestawów wektorów prowadzona jest oddzielna analiza kluczowych składowych (ang. *Primary Component Analysis* - PCA). Określenie co jest anomalią polega na obliczeniu SPE (ang. *squared prediction error*) i porównywaniu z progiem wyznaczonym statystycznie na podstawie danych treningowych. W ten sposób otrzymujemy algorytm, który potrafi wykryć anomalie na dwóch poziomach: stanów systemu i szablonów logów.

Największą słabością tego podejścia jest wymagana wiedza ekspercka do stworzenia wektora stanów, to ekspert bowiem określa, które stany będą brane pod uwagę. Powoduje to, że dla każdego nowego systemu, praca ta musi być powtórzona. Zależność od kodu źródłowego w czasie tworzenia szablonów logów to drugi powód, dla którego to rozwiązanie nie jest łatwe do przeniesienia.

Metoda LogLens [19] rozwiązuje problem zależności od kodu źródłowego stosując metody nienadzorowanego wykrywania szablonów. Szablony są tworzone na podstawie logów normalnych przebiegów systemu przy użyciu wstępnie określonych wyrażeń regularnych dopasowanych do struktury logów. Proces ten eliminuje potrzebę ręcznej interwencji, automatycznie tworząc szablony na podstawie grupowania podobnych logów.

Metoda LogLens wykrywa anomalie, korzystając z dwóch algorytmów: bezstanowego i stanowego. W algorytmie bezstanowym, system analizuje każdy log osobno, porównując go z wcześniej nauczonymi szablonami. Jeśli dany log nie pasuje do żadnego szablonu, zostaje oznaczony jako anomalia. Ten proces jest szybki i opiera się na analizie struktury pojedynczych logów, gdzie logi niespełniające oczekiwanego formatu lub zawierające nieznane dane są zgłaszane jako potencjalne anomalie.

W algorytmie stanowym, LogLens analizuje sekwencje logów w kontekście zdarzeń lub transakcji. System buduje model automatu stanowego, który śledzi sekwencję działań reprezentowanych przez logi, określając prawidłowy przebieg zdarzenia. Anomalie wykrywane są, gdy sekwencja logów odbiega od przewidywanego wzorca, na przykład brakuje pewnych logów, pojawiają się nieoczekiwane logi, lub czas trwania zdarzenia przekracza dopuszczalne granice. System wykrywa te odstępstwa od normy i raportuje je jako anomalie, co pozwala na identyfikację problemów w działaniu systemu.

2.3.2.2 Uczenie Maszynowe

Metody uczenia maszynowego najczęściej bazują na szablonach logów, tworzonych w podobny sposób co w LogLens. Metoda LogCluster [48] wykorzystuje treść szablonów i zamienia je na wektory korzystając z osadzeń słów (ang. *embeddings*). Następnie, wektorom z sekwencji logów nadawane są wagi obliczone przy pomocy Tf-IDF [65] bazującego na identyfikatorach szablonów. Pozwala to otrzymać reprezentację wektorową sekwencji. Wektory sekwencji są grupowane przy pomocy Agglomerative Clustering [38]. Centroidy każdego klastra są następnie używane do określenia, które sekwencje w błędnym logu powinny zostać przeanalizowane przez programistę.

Metoda opisana w [47] rezygnuje z szablonów na rzecz powiększenia liczbę klasyfikatorów. Wektory cech oparte są na częstości logów z 6 poziomów logowania (np. INFO, WARNING, ERROR), odległości czasowej między znanymi błędami, oraz częstością słów z tekstu logów. Klasyfikatorów natomiast jest aż cztery, klasyfikator oparty na regułach, SVM i najbliżsi sąsiedzi (ang. *Nearest Neighbours*) W dwóch wersjach: tradycyjnej i dostosowanej przez autorów. Modele trenowane są tak, by przewidzieć, czy następne okno będzie zawierało błąd.

Użycie drzew decyzyjnych [11] także korzysta z precyzyjnie wybranych cech z logów zamiast szablonów. Podejście to oferuje oprócz wykrycia anomalii również identyfikację komponentów systemu skorelowanych z błędem. Wykorzystywany jest model drzewa decyzyjnego, który klasyfikuje żądania jako zakończone sukcesem lub porażką na podstawie danych z okresy kiedy wystąpił błąd. Drzewo decyzyjne, choć nie zawsze najlepsze pod względem predykcji, ma istotną zaletę, że jego wyniki są łatwe do interpretacji przez ludzi. W procesie diagnozowania awarii analizuje się ścieżki prowadzące do węzłów

przewidujących błędy, co pozwala na wyciągnięcie wniosków na temat możliwych źródeł problemu.

Logi są przetwarzane w postaci wektorów cech, takich jak nazwa maszyny, typ żądania czy wersja oprogramowania. Dane są dzielone na podstawie wartości tych cech, a kryterium podziału maksymalizuje tzw. zysk informacyjny. Zysk informacyjny (ang. *information gain*) to miara używana w algorytmach uczenia maszynowego, szczególnie w budowaniu drzew decyzyjnych, do oceny, jak dobrze podział danych na podstawie określonej cechy poprawia ich klasyfikację. Mówiąc prościej, zysk informacyjny mierzy, o ile zmniejsza się niepewność (entropia) w danych po dokonaniu podziału na grupy według wartości wybranej cechy. Po wytrenowaniu drzewa, podgałęzie o niskiej wartości zysku są przycinane, aby uniknąć nadmiernego dopasowania do danych treningowych. Następnie stosuje się heurystyki, które ignorują nieistotne węzły, filtrowanie szumu oraz scalanie węzłów w celu uproszczenia diagnozy. Na końcu, zidentyfikowane przyczyny awarii są sortowane według liczby wystąpień awarii, aby wskazać najbardziej prawdopodobne źródła problemów.

Metoda LogEvent2Vec [81] powraca do reprezentowania logów za pomocą identyfikatorów szablonów. Każda linia logu jest zamieniana na szablon, co pozwala na uproszczenie i ustrukturyzowanie informacji. Następnie logi są dzielone na sekwencje szablonów na podstawie stałej wielkości okna.

Kolejny krok to ekstrakcja cech, gdzie zdarzenia logów są przekształcane na numeryczne wektory za pomocą modelu Word2Vec[55], który jest popularnym narzędziem w przetwarzaniu języka naturalnego. Wektory te reprezentują logi w przestrzeni euklidesowej, gdzie podobne szablony są położone blisko siebie. Z tych wektorów tworzone są reprezentacje całych sekwencji logów, które następnie są używane w algorytmach klasyfikacyjnych, takich jak lasy losowe (ang. *Random Forests*), Naive Bayes czy sieci neuronowe, w celu wykrycia anomalii. Proces wykrywania anomalii jest traktowany jako problem klasyfikacji binarnej, gdzie model decyduje, czy dana sekwencja logów jest normalna, czy anomalna.

2.3.2.3 Głębokie Uczenie

Podejścia statystyczne i uczenie maszynowe wymagają ponownego treningu przy każdej zmianie wielkości zbioru szablonów logów co jest częstym zdarzeniem w toku normalnego rozwoju oprogramowania. Dodatkowo często podejścia te często są uzależnione od precyzyjnie wybranych wektorów cech. Głębokie uczenie natomiast buduje reprezentację danych wejściowych w czasie treningu.

Metoda DeepLog [25] po zamianie sekwencji logów na identyfikatory szablonów używa stałego okna do zbudowania zbioru poprawnych sekwencji. Model LSTM uczy się następnie tego wzorca, by w czasie wnioskowania wykryć wszystkie podejrzane zmiany w sekwencji. Tą samą zasadę wykorzystuje LogGPT [29] zamieniając sieć LSTM na GPT, oraz CausalConvLSTM [98] używając kompozycji dwóch modeli CNN i LSTM.

Zamiast okna przesuwnego, niektóre modele przyjmują na wejście całe sekwencje. Sekwencje mogą mieć różną długość, dlatego krótsze sekwencje są uzupełniane zerami (ang. *padding*), a dłuższe są przycinane do ustalonej maksymalnej długości. Dzięki temu wszystkie sekwencje mają taką samą długość, co jest niezbędne do przetwarzania ich przez na przykład przez sieć CNN [49]. W podejściu tym każdy identyfikator szablonu w sekwencji jest kodowany w postaci wektora przy użyciu macierzy kodowej. Proces ten, nazywany logkey2vec, przekształca każdą liczbę reprezentującą klucz logu w wektor

o zadanym wymiarze, który następnie staje się częścią macierzy sesji. Tak zakodowana macierz jest przekazywana jako wejście do modelu CNN.

Modele głębokie dają większe możliwości jeśli chodzi o użycie semantyki logów. PyLogSentiment [72] w fazie treningowej najpierw przetwarza wstępnie logi ekstrahując z logu systemu operacyjnego informacje zawierające istotne treści, które będą analizowane pod kątem sentymentu. Do tego celu używa się narzędzia, które bazuje na technice rozpoznawania standardowych elementów logu, aby automatycznie rozpoznawać poszczególne pola w logach, takie jak daty, nazwy hostów, czy nazwy usług. Z tego procesu wydobywane są wiadomości, które mają kluczowe znaczenie dla analizy sentymentu.

Następnie każda wiadomość przechodzi proces osadzania słów. Po wydobyciu wiadomości, każde słowo w logach jest przekształcane w odpowiedni wektor liczbowy za pomocą wcześniej wytrenowanego modelu GloVe [61]. Wiadomości, które mają różną długość, są dopasowywane do stałego rozmiaru poprzez dodawanie zer (ang. *padding*) lub ucinanie zbyt długich wiadomości, aby umożliwić efektywne przetwarzanie w modelu.

Kolejnym krokiem jest rozwiązywanie problemu niezrównoważonych danych. Ponieważ logi systemowe często mają więcej wpisów o pozytywnym sentymencie, który oznacza normalne działanie, niż negatywnym, wskazującym na anomalie, stosuje się metodę Tomek Link [78]. Ta metoda pomaga zmniejszyć liczbę pozytywnych próbek poprzez usunięcie tych, które są najbliżej granicy klas, co pozwala na wyrównanie liczby próbek dla obu klas. Dzięki temu model ma lepszą zdolność do rozpoznawania rzadkich anomalii.

Po przetworzeniu danych, logi są przekazywane do modelu Gated Recurrent Unit (GRU) [14]. GRU to rodzaj sieci neuronowej, która uczy się długoterminowych zależności w sekwencjach danych, co jest istotne w przypadku analizy logów systemowych. GRU jest często preferowane ze względu na mniejszą liczbę parametrów w porównaniu do bardziej złożonej sieci LSTM. Na końcu zastosowana jest warstwa softmax, która klasyfikuje każdy wpis logu jako mający pozytywny lub negatywny sentyment.

2.3.2.4 Potencjalne zastosowanie do lokalizacji błędu

Głębokie modele generatywne mogą zostać wykorzystane do lokalizacji błędu przez wskazanie błędnych linii logów w sekwencji. Modele te przewidując następny znak na podstawie kontekstu mogą zwrócić informację, kiedy następnik odbiega od normy i jakie jest prawdopodobieństwo tego następnika. Te dane mogą służyć jako podstawa do szeregowania nieoczekiwanych linii jak w LBFL i LOGFAULTLOGGER.

Dyskryminatywne metody głębokie wykrywające anomalie, różnią się od znanych metod lokalizacji tym, że zbiór danych jest oznaczony binarnie: norma/anomalia. W momencie dostarczenia bogatszych oznaczeń, takich jak kategorii błędów, metody LogRobust i inne w łatwy sposób można dostosować do lokalizacji przez kategoryzowanie przyczyny błędu.

2.3.3 Metody odtwarzania ścieżki wykonania

Zwykłe procesy otrzymywania ścieżki wymagają instrumentacji systemu, ale często jest to niemożliwe. To motywuje do poszukiwania innych, nieinwazyjnych metod. Ponieważ logi są często dostępne i generowane przez kod, stanowią one część ścieżki

wykonania. Kilka prac stara się wykorzystać tę sytuację. W tej sekcji zostaną opisane metody, które odtwarzają całą lub część ścieżki wykonania na podstawie logów.

Metody odtwarzanie ścieżki zwracają wyniki na różnych poziomach szczegółowości. Od instrukcji wykonania [9, 100], plików [10] przez wywołania API [106], aż po nazwy testów jednostkowych z podobnym logiem wykonania [24]. Czasami mogą dodatkowo zawierać testy jednostkowe [106]. Odtwarzanie ścieżki wykonania odgrywa kluczową rolę podczas analizy awarii. Gdy jest wykonane z dużą precyzją, może pomóc programiście w znalezieniu przyczyny awarii. Z drugiej strony, dla długich wykonań ścieżka będzie składać się z tysięcy lub milionów operacji, co czyni ją niemożliwą do użycia. To jest znany problem SherLog [100].

Celem SherLog jest wywnioskowanie, co musiało, nie mogło i mogło się wydarzyć podczas błędnego uruchomienia produkcyjnego. Osiąga to poprzez łączenie informacji z logów i kodu źródłowego. Wynik składa się z dokładnej ścieżki oraz prawdopodobnej ścieżki. LogMap [9] działa podobnie, ale używa tylko fragmentów logów zawartych w raporcie o błędzie. W ten sposób przezwycięża problem zbyt dużej ilości zwracanych informacji dotyczący SherLog. Podobnie Pathidea [10] używając stosu i informacji tekstowych zawartych w raporcie o błędzie odtwarza ścieżkę wykonania na poziomie plików. Pensieve [106] wybiera tylko wywołania API ze ścieżki wykonania. Następnie dodatkowo zmniejsza liczbę wywołań API do minimalnego zbioru wywołań, w których awaria jest reprodukowana, i zwraca je opakowane w test jednostkowy. Moja wcześniejsza praca [24] wykorzystała inne podejście do opisu ścieżki wykonania. Główna idea polega na tym, że ścieżka wykonania jest odtwarzana na poziomie testów jednostkowych. Model ML jest nauczony rozpoznawać nazwę zbioru testów na podstawie logów. W czasie wnioskowania na logu produkcyjnym przetwarza on logi otrzymane z okna przesuwnego i nadaje im oznaczenia nazw testów. W ten sposób użytkownik otrzymuje opis scenariusza za pomocą nazw testów, co jest informacją z wyższego poziomu niż ścieżka wykonania lub nazwa pliku. Warto wspomnieć, że to przybliżenie nie jest tak dokładne jak SherLog, ale jest znacznie szybsze, ponieważ najbardziej wymagająca część obliczeniowa odbywa się podczas treningu.

2.3.3.1 Potencjalne zastosowanie do lokalizacji błędu

Wymienione tutaj metody pasują do metod lokalizacji opartych na wnioskowaniu przyczynowo-skutkowym. Budują one swojego rodzaju modele działania systemu. Mając do dyspozycji ich wynik z poprawnego wykonania i z wykonania błędnego można zastosować znane już techniki wnioskowania do otrzymania potencjalnych lokalizacji błędu.

2.3.4 Autoencodery

Użycie autoenkodera do wykrywania anomalii w logach jest znanym podejściem [39, 58]. W pierwszym przypadku [39] dane logów są zorganizowane w macierz, gdzie każda linia logu zawiera znaczniki czasu, komunikaty, kategorie oraz wartości numeryczne. Dane są odpowiednio przetwarzane by były dostosowane do wykorzystania przez sieci neuronowe. Autoenkoder jest trenowany na każdej linii logu minimalizując błąd średniokwaratowy rekonstruowanego wektora (ang. *mean square error* - MSE). Linia logu jest uznawana za anomalię, jeśli wartość MSE przekracza ustalony próg, który jest określany we współpracy z ekspertem domenowym. W celu zastosowania metody

do innych zbiorów danych konieczne jest dostosowanie liczby neuronów wejściowych i wyjściowych oraz ponowne trenowanie autoenkodera na nowych danych.

Metoda opisana w [58] rozpoczyna się od przekształcenia logów w stałej długości sekwencje, przy użyciu uzupełnienia i kodowania one-hot. Tak przygotowane dane są następnie wejściem do Autoenkodera, który uczy się minimalizując średni błąd kwadratowy pomiędzy wejściem a wyjściem. W trakcie treningu, do wejść dodawany jest szum, aby zapobiec przetrenowaniu. Po zakończeniu treningu Autoenkoder jest w stanie rozpoznawać anomalie w logach zdarzeń na podstawie błędów rekonstrukcji (ang. *reconstruction error*), czyli różnicy między oryginalnym wejściem a jego rekonstrukcją uzyskaną na wyjściu dekodera. Pozwala to na identyfikację zarówno nietypowych sekwencji, jak i pojedynczych, błędnych logów. Metoda ta wykorzystuje modele z ukrytą warstwą o zmiennym rozmiarze, dostosowanym do liczby różnych szablonów.

Podejście autoencoder wydaje się najbliższe lokalizacji błędu proponownej w LogDC. Mogłyby one zastąpić kodowanie one-hot i w sposób nienadzorowany pretworzyć logi oraz zwrócić ich osadzenie na wejście algorytmu k-means.

2.4 Nienadzorowana segmentacja logów

Segmentacja logów jest fundamentalnym krokiem w wykrywaniu anomalii, szczególnie dla głębokich modeli. Polega ona na dzieleniu ciągłego strumienia logów na mniejsze, łatwiejsze do zarządzania sekwencje na podstawie identyfikatorów takich jak ID bloków, ID zadań czy ID wątków. Ta segmentacja jest kluczowa dla efektywnego wprowadzania danych logów do modeli wykrywania anomalii, jednak nie zawsze jest konsekwentnie stosowana do różnych zestawów danych. Na przykład DeepLog używa identyfikatora bloków (block_ID) dla logów HDFS i identyfikatora instancji (instance_ID) dla zestawu danych OpenStack, podczas gdy LogGPT i LogBERT używają tylko identyfikatora bloków dla logów HDFS, pomijając segmentację ID dla zestawów danych Thunderbird i BGL (ang. *BlueGene/L Supercomputer System*).

Jeśli wynikowa sekwencja jest nadal zbyt długa, stosuje się okno przesuwne o stałym rozmiarze, obejmujące czas lub liczbę elementów. Badania empiryczne wykazały, że taka segmentacja może być niedokładna, gdyż logi związane z jedną awarią mogą zostać przypadkowo podzielone na podstawie znaczników czasowych [13].

Dodatkowo interpretacja segmentów otrzymanych za pomocą okna przesuwnego jest trudna dla programisty. Właściwa segmentacja powinna zapewnić, że logi pochodzące z jednej funkcjonalności są utrzymane razem, umożliwiając wykrywanie anomalii z niezbędnymi informacjami.

2.4.1 Nienadzorowana segmentacja słów

Nienadzorowana segmentacja słów (UWS) ma potencjał do skutecznego zastosowania w kontekście sekwencji logów, rozwiązując zarówno problemy segmentacji, jak i wykrywania anomalii. Ta metoda jest szczególnie odpowiednia dla sekwencji, gdzie uzyskanie "złotego" standardu segmentacji jest bardzo trudne. Sekwencje logów mają z natury strukturalny format, który odzwierciedla przemyślany projekt systemu. Ta strukturalna natura może być wykorzystana do ulepszenia jakości segmentacji używanej do wykrywania anomalii i lokalizacji błędu. Uzyskanie "złotego" standardu segmentacji dla logów jest trudne, ponieważ każdy zestaw danych logów wymaga własnego "złotego" standardu, który może szybko stać się przestarzały w normalnym procesie rozwoju.

2.4.2 Hipoteza Harrisa

Hipoteza Harrisa to koncepcja zaproponowana przez Zellig Harris'a w 1955 roku, która dotyczy segmentacji języka na podstawowe jednostki, takiej jak morfem czy słowo. Harris zauważył, że lingwistyczne granice, takie jak granice słów, mogą być identyfikowane przez zmiany w liczbie możliwych kontynuacji danej sekwencji fonemów lub liter.

Zgodnie z Hipotezą Harrisa, kiedy analizujemy ciąg fonemów (w mowie) lub liter (w tekście pisanym), różnorodność możliwych kontynuacji tej sekwencji (czyli jakie fonemy/litery mogą pojawić się następne) zwykle zmniejsza się, gdy poruszamy się w ramach jednego morfemu lub słowa. Jednakże, gdy zbliżamy się do końca morfemu lub słowa, nagle wzrasta liczba możliwych kontynuacji. To nagłe zwiększenie różnorodności wskazuje na potencjalną granicę lingwistyczną – na przykład, koniec jednego słowa i początek następnego.

Na przykład, w sekwencji "ca" w języku angielskim, prawdopodobieństwo, że następny znak to "t" (tworząc "cat"), jest wysokie. Jednakże, po dodaniu "t", różnorodność możliwych kontynuacji znacznie wzrasta, co może wskazywać na granicę słowa, ponieważ kończy się tu słowo "cat" i może zacząć inne słowo.

Harris użył tej hipotezy do opracowania procedur segmentacji mowy i tekstu na morfemy. W początkowej formie opierał się na intuicyjnych ocenach ludzi dotyczących ich języka ojczystego, pytając ich, jakie dźwięki mogą pojawić się przed lub po danym ciągu fonemów. Z czasem metodologia ta została zaadaptowana do analizy korpusów językowych, co pozwoliło na automatyczną segmentację na podstawie statystycznej analizy tekstów.

W nienadzorowanej segmentacji słów, Hipoteza Harrisa jest używana do wykrywania granic słów na podstawie zmian w entropii rozgałęzień (ang. *Branching Entropy*), które mogą być mierzone przez liczbę różnych kontekstów, w których dany ciąg znaków występuje. Współczesne podejścia, takie jak Variation of Branching Entropy (VBE), są rozwinięciami tej pierwotnej koncepcji.

2.4.3 Podejścia dyskryminatywne

Podejścia do nienadzorowanych systemów segmentacji słów można podzielić na modele dyskryminatywne i generatywne. Oba podejścia można przedstawić jako dwufazowe, gdzie kodowanie wydobywa niezbędne informacje z danych, a dekodowanie wykonuje segmentację. Modele dyskryminatywne używają różnych metryk, oraz algorytmu dekodowania do wykonania segmentacji na podstawie tych metryk (np. Viterbi'[31]). Spójność, to jedna z metryk, polegająca na ocenie, jak dobrze części segmentu (np. słowa) do siebie pasują. Informacja wzajemna (ang. *Mutual Information*) jest często używaną metryką do oceny prawdopodobieństwa współwystępowania dwóch lub więcej części w porównaniu do występowania niezależnie. Wyższa informacja wzajemna sugeruje silniejszą spójność. Stopień rozdzielenia między wynikowymi jednostkami, to metryka, która uwzględnia, jak bardzo dwa segmenty różnią się od siebie. Jej przykłady to entropia rozgałęzień (BE), wariacyjna entropia rozgałęzień (VBE) i długość minimalnego opisu. Entropia rozgałęzień, która jest główną koncepcją w wielu metrykach dyskryminatywne, analizuje różnorodność kontekstów, w których pojawia się segment, pomagając określić granice między segmentami. Jeśli segment pojawia się w szerokiej gamie kontekstów, jest prawdopodobnie samodzielną jednostką.

2.4.4 Podejścia generatywne

Generatywne modele nienadzorowanej segmentacji słów można opisać jako systemy używające prawdopodobieństwa segmentacji względem ciągu znaków. To podejście wykorzystuje probabilistyczne lub neuronowe modele językowe do określenia najbardziej prawdopodobnej segmentacji danego ciągu tekstów. Model językowy jest uczony bezpośrednio na podstawie danych. Można wyróżnić dwa kroki w konstrukcji takiego algorytmu: wybór rodziny modelu i funkcji straty. Modelem może być Ukryty model Markowa (ang. *Hidden Markov Model* - HMM), proces Pitmana-Yora, hierarchiczny procesy Dirichleta (ang. *Hierarchical Dirichlet Process* - HDP), a także neuronowy model LSTM lub Transformer. Definicja funkcji straty jest podobna do tej w modelach dyskryminatywnych, i głównie oparta jest na algorytmie Viterbi [31].

Generatywne metody segmentacji słów zazwyczaj polegają na wstępnej segmentacji na zdania, co działa dobrze w wielu językach, ale stawia wyzwania w innych dziedzinach, takich jak logi oprogramowania. Sekwencje logów oprogramowania często zawierają tysiące linii, nawet po zastosowaniu segmentacji według identyfikatorów takich jak identyfikator wątku lub bloku. Wysoka złożoność pamięciowa większości istniejących algorytmów generatywnych oznacza, że w przypadku zastosowania ich do tak długich sekwencji, zasoby pamięci zostaną szybko wyczerpane.

2.4.4.1 Nested Pitman-Yor Language Model (NPYLM)

Model NPYLM (ang. *Nested Pitman-Yor Language Model*) [57] jest jednym z generatywnych modeli. Rozszerza on tradycyjne modele oparte na procesie Dirichleta, łącząc modelowanie n-gramów słów i znaków. NPYLM skutecznie modeluje granice słów, integrując procesy Pitmana-Yora w strukturę hierarchiczną. Słowa są generowane jako ciągi znaków, co pozwala na precyzyjniejsze odwzorowanie rzeczywistych granic słów w językach, w których te granice nie są wyraźnie określone. Dostosowałem tę metodę do segmentowania logów. Jej bardziej szczegółowy opis znajduje się w Rozdziale 3.5.4.

2.4.4.2 Segmental Recurrent Neural Networks (SRNN)

Segmentalne sieci neuronowe (SRNN) [41] wprowadziły nowatorskie podejście do rozwiązywania problemu segmentacji i etykietowania danych sekwencyjnych, takich jak mowa, pismo odręczne czy sekwencje DNA. Zadanie to polega na podzieleniu sekwencji wejściowej na ciągłe segmenty o dowolnej długości oraz przypisaniu etykiety do każdego segmentu. SRNN łączą dwa potężne narzędzia uczenia maszynowego: uczenie reprezentacji oraz predykcję strukturalną. Wykorzystują dwukierunkowe sieci rekurencyjne (RNN) do tworzenia osadzeń w przestrzeni ciągłej dla każdego możliwego segmentu. Te osadzenia pozwalają modelowi ocenić zgodność każdego segmentu z daną etykietą. W odróżnieniu od wcześniejszych metod opartych na RNN, SRNN w sposób jawny reprezentują każdy kandydacki segment, co umożliwia ich zastosowanie w scenariuszach, gdzie dopasowanie między segmentami a etykietami jest częścią pożądanego wyniku, na przykład w predykcji struktury białek czy ekstrakcji informacji z tekstu.

Jedną z kluczowych innowacji SRNN jest ich zdolność do modelowania zależności statystycznych między sąsiadującymi etykietami oraz długościami segmentów. Osiąga się to przez ujęcie problemu w kontekście pół-Markowskich warunkowych pól losowych, co pozwala SRNN zdefiniować warunkowy rozkład prawdopodobieństwa nad przestrzenią wyjściową — segmentacją i etykietowaniem — dla danej sekwencji wejściowej.

Podejście SRNN różni się od tradycyjnych metod, które opierają się na symbolicznych oznaczeniach, poprzez wykorzystanie ciągłych reprezentacji i obliczanie cech za pomocą osadzeń. Pozwala to modelowi uchwycić złożone zależności i generować bardziej precyzyjne segmentacje i etykiety, co jest szczególnie wartościowe w zadaniach takich jak rozpoznawanie pisma odręcznego czy etykietowanie części mowy.

2.4.4.3 Segmental Language Model (SLM)

Segmentalne modele językowe (SLM) [73] są zaprojektowane tak, aby bezpośrednio generować zdania podzielone na segmenty oraz obliczać odpowiadające im prawdopodobieństwo generatywne.

SLM-y, podobnie jak segmentalne sieci neuronowe (SRNN), wykorzystują zaawansowane techniki uczenia maszynowego do modelowania danych sekwencyjnych, ale zamiast skupiać się na segmentacji i etykietowaniu, SLM-y uczą się modelowania języka poprzez uwzględnienie segmentacji znaków. W przetwarzaniu języka chińskiego, gdzie znaki są podstawowymi elementami, SLM-y uczą się wspólnej funkcji prawdopodobieństwa dla posegmentowanej sekwencji znaków. Segmentalne podejście w SLM-ach pozwala na bardziej precyzyjne modelowanie zależności między segmentami, co jest kluczowe dla poprawnego generowania zdań w języku chińskim.

Podobieństwo do SRNN polega na tym, że oba modele wykorzystują programowanie dynamiczne do efektywnego rozwiązywania problemów związanych z segmentacją. W przypadku SLM, programowanie dynamiczne umożliwia obliczanie funkcji straty oraz znajdowanie segmentacji o maksymalnym prawdopodobieństwie w czasie liniowym, co jest kluczowe dla przetwarzania długich sekwencji tekstu. Dzięki temu SLM-y mogą generować zdania segment po segmencie, aktualizując stan ukryty kontekstowego enkodera RNN i tworząc nowe słowa na tej podstawie.

W skrócie, segmentalne modele językowe (SLM) mogą efektywnie zastąpić tradycyjne modele językowe, oferując bardziej zaawansowane podejście do modelowania języka opartego na segmentach, podobnie jak SRNN-y oferują zaawansowane podejście do segmentacji i etykietowania sekwencji.

Każda z tych metod oferuje unikalne podejście do segmentacji i modelowania języka, z różnymi mocnymi stronami w zależności od specyfiki danych i celu analizy.

2.4.5 Segmentacja sekwencji logów

Znane są podejścia do segmentowania sekwencji logów wykorzystujące osiągnięcia z segmentacji słów języka naturalnego. Dotychczas segmentacja ta służyła tylko do lepszej prezentacji treści w nich zawartych. Autorzy algorytmów zauważają, że sekwencja logów nie jest losowa lecz ustrukturyzowana. Wydobycie tych struktur pozwala użytkownikowi na lepsze zrozumienie prezentowanych treści. Struktury bowiem łączą logi związane z funkcjonalnościami systemu. Poniżej prezentuję dwie najbardziej znane metody. Pierwsza, VotingExeprts, została użyta w części eksperymentalnej do segmentacji sekwencji związanych z ID wątków. Warto zwrócić uwagę, że sekwencje logów w sposób zasadniczy różnią się od sekwencji liter. Po pierwsze, nie da się określić w czasie treningu, jaki alfabet będzie potrzebny w fazie testów. Wynika to z faktu, że nowe linie są dodawane, stare usuwane w związku z normalnym procesem rozwoju oprogramowania. Jest to niespotykane utrudnienie dla metod bazujących na alfabetach języka naturalnego. Dodatkowo długość sekwencji logów jest dużo większa niż sekwencji liter. W segmentacji słów, zazwyczaj niepisanym założeniem jest, że podana

sekwencja liter pochodzi z jednego zdania. W przypadku logów nie musi być to prawdą. Często otrzymane sekwencje logów stanowią wycinek działania programu, trudno zatem mówić o początku i końcu. W konsekwencji, metody bazujące na charakterystycznych znacznikach początku i końca zdania mogę być nieodpowiednie do segmentowania logów.

2.4.5.1 Algorytm VotingExperts

Algorytm *VotingExperts* [15] jest narzędziem do segmentacji szeregów czasowych poprzez identyfikację znaczących granic epizodów. Jego podstawowym założeniem jest zastosowanie grupy ekspertów, którzy głosują nad potencjalnymi miejscami podziału szeregu czasowego. Algorytm przesuwa okno o stałej długości przez szereg czasowy i ocenia, czy w danym miejscu należy przeciąć szereg, na podstawie głosów oddanych przez ekspertów.

Pierwszym krokiem algorytmu jest zbudowanie drzewa n-gramów o określonej głębokości, gdzie każdy węzeł reprezentuje n-gram o długości zależnej od poziomu drzewa. Dla przykładu, sekwencja "abcabd" wygeneruje drzewo, w którym n-gramy o długości 2 i mniejszej są reprezentowane przez wierzchołki, a każde z dzieci wierzchołka reprezentuje przedłużenie n-gramu.

Po utworzeniu drzewa n-gramów, algorytm oblicza entropię granicy dla każdego n-gramu, która jest miarą niepewności co do możliwych rozszerzeń danego n-gramu. Entropia rozkładu dla zmiennej losowej X jest wyrażona jako:

$$H(X) = -\sum_{x \in X} p(x) \log p(x), \tag{2.1}$$

gdzie $p(x)$ jest prawdopodobieństwem wystąpienia tokenu x. Entropia granicy jest kluczowym wskaźnikiem, który pozwala ekspertom ocenić, gdzie może znajdować się granica epizodu.

Kolejnym krokiem jest standaryzacja obliczonych częstotliwości i entropii, co pozwala na porównywanie n-gramów różnej długości. Standaryzacja polega na odjęciu od każdej wartości wartości średniej wszystkich próbek i podzieleniu przez odchylenie standardowe, co pozwala wyrazić każdą wartość jako liczbę odchyleń standardowych od średniej.

Gdy eksperci przeanalizują wszystkie możliwe miejsca granic w oknie, każdy z nich oddaje głos na miejsce, które według niego jest najlepszym kandydatem na granicę epizodu. Miejsca z największą liczbą głosów są wybierane jako punkty podziału szeregu czasowego.

Z perspektywy niniejszej pracy najważniejszą właściwością VotingExperts jest jego oparcie na n-gramach a nie na zdaniach. Powoduje to, że jest on najbardziej odpowiedni do dziedziny sekwencji logów.

2.4.5.2 Metoda hierarchicznej akwizycji leksykonu z wykorzystaniem modelu multigramowego

Innym podejściem jest zastosowanie modelu multigramowego [68], który modeluje prawdopodobieństwo ciągów zdarzeń (czyli "zdania") jako konkatenacji niezależnie losowanych segmentów (czyli "słów"). Metoda ta opiera się na założeniu, że każdy segment ma swoje prawdopodobieństwo w leksykonie segmentów, a całkowite prawdopodobieństwo zdania jest iloczynem prawdopodobieństw jego segmentów.

Metoda, którą opisano, służy do segmentacji sekwencji zdarzeń w celu lepszego zrozumienia wysokopoziomowych procesów w domenach, takich jak serwery Microsoft Exchange i aplikacje sterowane interfejsem użytkownika. Zamiast polegać jedynie na statystykach niskopoziomowych zdarzeń, metoda ta umożliwia identyfikację bardziej złożonych wzorców i struktur w danych, co pozwala na bardziej precyzyjną analizę i optymalizację systemów.

Podstawą tej metody jest tworzenie hierarchicznego leksykonu, który opisuje sekwencje zdarzeń. Proces rozpoczyna się od najprostszych segmentów, składających się z pojedynczych zdarzeń, i stopniowo rozwija bardziej złożone segmenty poprzez łączenie tych podstawowych elementów. Kluczowym narzędziem w tym procesie jest informacja wzajemna, która mierzy siłę związku między sąsiadującymi segmentami. Gdy para segmentów ma wysoką wartość informacji wzajemnej, zostają one połączone, tworząc nowy segment. Ten proces jest iteracyjny, co oznacza, że z każdym krokiem leksykon staje się bardziej złożony i bardziej zbliżony do rzeczywistych wzorców zachodzących w danych.

Segmentacja sekwencji za pomocą tej metody pozwala na lepsze odwzorowanie wysokopoziomowych procesów, które generują zdarzenia. Dzięki temu można dokładniej symulować rzeczywiste interakcje użytkowników z systemami, co jest niezwykle ważne w kontekście optymalizacji wydajności serwerów lub analizy zgłoszeń o błędach w oprogramowaniu. Dodatkowo, metoda ta ułatwia klasyfikację danych, takich jak raporty o awariach, co przyspiesza i usprawnia proces rozwiązywania problemów.

Rozdział 3
Zaproponowane rozwiązanie

W pracy skupiłem się na opracowaniu metody lokalizacji błędu opartej na logach. Jednym z kryterium określonym w tytule pracy jest, by metoda działała w czasie rzeczywistym lub zbliżonym do rzeczywistego. Zastosowanie metod opartych na sieciach neuronowych daje w rezultacie metody działające w czasie zbliżonym do rzeczywistego [43, 105], gdyż większość kosztu obliczeniowego jest przeniesiona na proces treningu. Metody oparte o modele głębokie LogRobust [105] i SwissLog [43], działają w czasie zbliżonym do rzeczywistego, gdyż rozpoczynają działanie w momencie otrzymania sekwencji pochodziła z właśnie zakończonego wątku. Moja metoda rozpoczyna działanie po dostarczeniu jej pierwszych n logów z danego wątku, nie wymagając jego zakończenia by zwracać pierwsze rezultaty, zatem jest szybsza niż wymienione wcześniej podejścia.

Swoje badania prowadziłem w czterech obszarach. Pierwszym było opisanie scenariusza wykonania z pomocą nazw testów. Drugim była ocena nienadzorowanej segmentacji logów przez pryzmat oczekiwań eksperta systemu. Trzecim był wpływ nienadzorowanej segmentacji logów na wykrywanie anomalii, a czwartym wykorzystanie nienadzorowanej segmentacji do stworzenia dokładniejszej metody lokalizacji błędów. Badania w ramach pierwszego obszaru rozpocząłem od oceny dokładności klasyfikacji sekwencji logów pochodzących z wykonań testów jednostkowych i modułowych nazwami testów z których pochodzą. Otrzymane wysokie wartości F-score sugerowały, że można takiego modelu użyć do oznaczania segmentów logów produkcyjnych i otrzymać tym samym opis scenariusza nazwami testów. Ułatwia to zrozumienie scenariusza wykonania, a dodatkowo kojarzy log z miejscem w kodzie. W tym badaniu segmenty z logu produkcyjnego zostały wyznaczone przy wykorzystaniu okna przesuwnego. Ponieważ takie podejście ma znane ograniczenia [13], to postanowiłem skorzystać z dokładniejszej, nienadzorowanej segmentacji słów użytej na sekwencji identyfikatorów szablonów linii logów. Gdy wyniki tej segmentacji względem złotego wzorca określonego przez eksperta osiągnęły wystarczające wartości F-score, pozostało sprawdzić, czy nienadzorowana segmentacja ma wpływ na automatyczną analizę logów. Wybrałem dwa zadania: wykrywanie anomalii i lokalizację błędu. Do sprawdzenia wykrywania anomalii wykorzystałem algorytm LogGPT, opracowany przy współpracy Nokia Bell Labs. W przypadku lokalizacji, stworzyłem dwa algorytmy: statystycznej lokalizacji błędu oraz algorytm lokalizacji bazujący na wykrywaniu anomalii przez LogGPT. W obu przypadkach zastosowanie nienadzorowanej segmentacji okazało się korzystne dla poprawy wyniku lokalizacji. Efektem badań było stworzenie oprogramowania wykorzystującego logi z poprawnego wykonania do lokalizacji błędu w logu z błędnego wykonania. Oprogramowanie zostało wykorzystane do pilotażowego zastosowania w Nokia.

3.1 Ewaluacja

Do oceny jakości segmentacji sekwencji logów przez VotingExperts i wykrywania anomalii przez LogGPT wykorzystałem wskaźnik F-score:

$$\text{F-score} = 2 \times \frac{\text{Precyzja} \times \text{Czułość}}{\text{Precyzja} + \text{Czułość}}. \tag{3.1}$$

Czułość (ang. *Recall*) oblicza się ze wzoru:

$$\text{Czułość} = \frac{\text{PD}}{\text{PD} + \text{FU}}. \tag{3.2}$$

W przypadku segmentacji logów prawdziwie dodatni (PD) oznacza poprawnie wskazaną granicę między słowami, fałszywie dodatni (FD) to błędnie wskazana granica między słowami. Fałszywie ujemny (FU) to granica między słowami pominięta w czasie segmentacji, a prawdziwie ujemny (PU) to nie wskazanie granicy słów, tam, gdzie w rzeczywiście jej nie ma. Czułość określa, jaką część dodatnich przypadków wykrył model. W kontekście segmentacji logów oznacza to, jaką część faktycznych granic między słowami wykrył model.

Precyzja (ang. *Precision*) mierzy jaka część przypadków określonych jako dodatnie przez model, jest faktycznie dodatnich:

$$\text{Precyzja} = \frac{\text{PD}}{\text{PD} + \text{FD}}. \tag{3.3}$$

Do wyznaczenia tych metryk dla segmentacji, skonstruowałem dwie tablice, każda o długości równej liczbie szablonów w logu. Pierwsza tablica reprezentuje poprawną segmentację, a druga reprezentuje segmentację zaproponowaną przez model. Obie tablice początkowo są wypełnione zerami. Następnie każda granica między słowami jest oznaczona w tablicy jako 1 i umieszczona jest na pozycji za ostatnią literą należącą do słowa.

Segmentacja logów jest też oceniana przez odniesienie do entropii znakowej. Intuicyjnie segmentacja jest trudna dla zbiorów danych z dużą różnorodnością, a łatwa dla dobrze ustrukturyzowanych i przewidywalnych sekwencji. Metodą mierzenia tej zależności jest obliczanie entropii [69]. W swoim badaniu użyłem entropii znakowej (3.4) i odniosłem ją do jakości segmentacji. Entropię znakową obliczyłem jako:

$$H = -\sum_{x} p(x) \log p(x), \tag{3.4}$$

gdzie $p(x)$ jest prawdopodobieństwem wystąpienia znaku x w danym zbiorze danych, a suma jest liczona po wszystkich znakach x.

3.2 Normalizacja

Logi są tekstową informacją o przebiegu programu. Jako takie są zrozumiałe dla człowieka jednakże niezrozumiałe dla metod automatycznej analizy. Zatem pierwszym krokiem jest zawsze zamiana ich na reprezentację liczbową. Często, by ułatwić ten proces, stosowana jest normalizacja, czyli usunięcie zbędnych, nadmiarowych informacji z logów, by precyzyjniej i szybciej dokonać zamiany. W przypadku logów Nokii logi

zostały znormalizowane poprzez usunięcie znaczników czasowych oraz linii, w których opisywane były obiekty (Rysunek 2). Znaczniki czasowe mają spójny format w całym pliku logu, dlatego nie przyczyniają się do rozróżniania szablonów, a jedynie zwiększają obciążenie procesu ekstrakcji szablonów. Użyłem znaczników czasowych do sortowania linii, ale po tym etapie zostały one usunięte.

Obecność opisów obiektów w logu z kolei utrudnia opracowanie wyrażeń regularnych z powodu różnic w odstępach, znakach specjalnych i interpunkcyjnych. Elementy te nie zawierają informacji związanych z wykonaniem programu, a są w zasadzie jednoliniowymi informacjami na temat jednego obiektu zapisanymi w wielu liniach dla poprawy czytelności. Struktura i wartości wydrukowanych pól obiektów mogą być zweryfikowane jedynie w odniesieniu do specyfikacji, dlatego zdecydowałem się na wykluczenie tych linii z analizy.

```
inf component6:   operation_type: CREATE
inf component6:   object {
inf component6:     object_name: "ObjectName1"
inf component6:     class_name: "ClassName"
inf component6:     [ClassName.object] {
inf component6:       structure1 {
inf component6:         structure2{
inf component6:           field1: "text1"
inf component6:           field1: "text2"
inf component6:           field1: 8
inf component6:           field1: 24
inf component6:           field1: 12
inf component6:           field1: 20
inf component6:           field1: 0
inf component6:           field1: 1
inf component6:           field1: 2
inf component6:           field1: 3
inf component6:           field1: 4
inf component6:           field1: 5
inf component6:           field1: 6
inf component6:       }
inf component6:     }
inf component6:   }
inf component6: }
```

Rysunek 2: Opis obiektu w liniach logu. Takie opisy są usuwane podczas normalizacji, ponieważ nie dostarczają użytecznych informacji do lokalizacji błędów [23].

3.3 Parsowanie logów

Proces parsowania logów za pomocą narzędzia Drain [30] jest techniką stosowaną do ekstrakcji szablonów logów z surowych danych. Proces parsowania rozpoczyna się od wykorzystania zdefiniowanych wyrażeń regularnych do określenia szablonów logów na podstawie historycznych logów. Drain analizuje nowo pojawiające się logi, porównując je z istniejącymi szablonami. Jeśli log pasuje do istniejącego wzorca, zostaje do niego przypisany. W przeciwnym razie algorytm tworzy nowy wzorzec, odpowiednio aktualizując drzewo decyzyjne. Struktura drzewa pozwala na szybkie dopasowanie logów, dzięki czemu proces jest skalowalny i może być stosowany w dużych systemach z ogromną ilością danych. Parsowanie logów za pomocą Drain umożliwia zamianę logów na identyfikatory szablonów im odpowiadających, lub samą zawartość szablonów, co jest kluczowe dla umożliwienia dalszej automatycznej analizy.

Proces parsowania logów za pomocą narzędzia Drain można zilustrować na przykładzie systemu serwerów, który generuje logi w trakcie obsługi zapytań HTTP. Załóżmy,

że serwer produkuje różne logi, takie jak:

- `INFO 2024-08-12 10:15:32 GET /index.html 200 OK`
- `ERROR 2024-08-12 10:16:15 GET /unknown.html 404 Not Found`
- `INFO 2024-08-12 10:17:08 POST /submit-form 200 OK`
- `ERROR 2024-08-12 10:18:22 GET /index.html 500 Internal Server Error`

Surowe logi zawierają widoczną redundancję danych. Drain rozpoznaje wzorce i generuje szablony. W powyższym przykładzie Drain może zidentyfikować następujące wzorce:

- `INFO * GET * OK`
- `ERROR * GET * Not Found`
- `INFO * POST * OK`
- `ERROR * GET * Internal Server Error`

Każdy log zostaje zredukowany do odpowiedniego wzorca z unikalnymi atrybutami (takimi jak czas i nazwa pliku), które są przechowywane osobno. Na przykład, pierwsza linia:

`INFO 2024-08-12 10:15:32 GET /index.html 200 OK`

zostanie odwzorowana na szablon `INFO * GET * OK` z zapisanymi atrybutami `czas= 2024-08-12 10:15:32` i `ścieżka=/index.html`.

Dzięki temu procesowi Drain redukuje złożoność analizy logów, umożliwiając skupienie się na sekwencji zdarzeń, a nie na zmiennej treści. Z drugiej strony warto pamiętać, że takie podejście jest również mocnym ograniczeniem. Gdy zredukujemy powyższą sekwencję do szablonów, zupełnie stracimy informację o oczywistym błędzie 404 i 500.

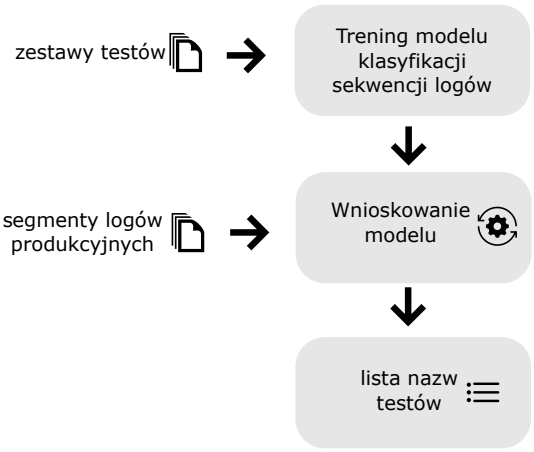
3.4 Klasyfikacja sekwencji logów z testów

3.4.1 Architektura

Architektura mojego rozwiązania [24] jest przedstawiona na Rys. 3. Model jest najpierw wytrenowany na logach zgromadzonych w czasie testów jednostkowych. Następnie model jest używany do oznaczenia segmentów logów produkcyjnych i zwraca sekwencję nazw zestawów testów. Sekwencja logów produkcyjnych jest przedstawiana modelowi we fragmentach, które są wybierane za pomocą okna przesuwnego o stałym rozmiarze.

3.4.2 Modele uczenia maszynowego

Spośród modeli nadzorowanego uczenia maszynowego rozwiązujących problem klasyfikacji (drzewa decyzyjne, metoda k-najbliższych sąsiadów, maszyny wektorów nośnych (SVM – support vector machine), naiwny klasyfikator Bayesa, las losowy, regresja logistyczna, sieci neuronowe) wybrałem do eksperymentów trzy: Naive Bayes Classifier (NBC) [35] jako najprostszy model, chcąc zweryfikować, czy kolejność logów ma duże



Rysunek 3: Diagram blokowy procedury oznaczania logów produkcyjnych przez nazwy testów. Logi z zestawów testów są wykorzystywane do trenowania modelu. Log produkcyjny jest segmentowany, a predykcje są dokonywane na segmentach.

znaczenie dla klasyfikacji, Random Forest Classifier (RFC) [37] jako model, który może pomóc w interpretowalności, oraz Gradient Boosting Classifier (GBC) [54], potencjalnie najmocniejszy, który wychwyci drobne różnice między podobnymi zestawami testów. Do zamiany sekwencji szablonów na wektor liczb zastosowałem TfIDF (ang. *Term Frequency Inverse Document Frequency*) co jest standardem w dziedzinie NLP [50]. Oznaczeniami tak otrzymanych wektorów były nazwy zestawów testów. Zastosowałem przeszukiwanie Grid Search with Cross-Validation, jako popularną metodę strojenia, w celu wybrania najlepszych parametrów dla modelu RFC i GBC. Dla RFC najlepsze parametry to liczba estymatorów równa 100 (spośród 11-2000 przedziału zaproponowanego w [62]), funkcja obliczająca maksymalną liczbę cech do rozważenia przy poszukiwaniu najlepszego podziału to logarytm o podstawie 2 z liczby cech macierzy danych wejściowych (wybrana spośród funkcji tożsamościowej i pierwiastka kwadratowego), maksymalna głębokość nieograniczona, a funkcja do mierzenia jakości podziału to *gini*. Dla GBC najlepsze parametry to głębokość 10, 100 estymatorów i współczynnik uczenia 0.1. Metoda ta, przez wykorzystanie okna przesuwnego i modeli ML, działa w czasie zbliżonym do rzeczywistego, oznacza to, że wytrenowany model zwraca rezultaty po otrzymaniu sekwencji logów równiej długości okna.

3.4.3 Predykcja

Wytrenowany model jest stosowany do segmentów logu produkcyjnego otrzymanych za pomocą okna przesuwnego. Prawdopodobieństwo z jakim model klasyfikuje dany segment musi wynosić ponad 0.5 i jest przechowywane wraz z nazwą zestawu testowego do wglądu przez programistę. W przypadku logu z HDFS wybrałem długość okna równą 100, podczas gdy dla danych Nokia najlepszy rozmiar okna wynosił 10. Kryterium wyboru długości okna było to, by jak najmniej linii logów pozostało bez oznaczenia. Zwracana sekwencja nazw zestawów testowych pomaga programiście zrozumieć scenariusz wykonania.

3.5 Segmentacja sekwencji logów

Po otrzymaniu zadowalających wyników klasyfikacji sekwencji logów przystąpiłem do badań nad bardziej dokładną metodą segmentowania logów niż zastosowanie okna przesuwnego. Zadania z wykrywania anomalii i lokalizacji błędu powinny dawać lepsze rezultaty po otrzymaniu segmentów dokładniej grupujących linie związane z tą samą funkcjonalnością [23]. Znane są podejścia do segmentacji oparte na częstotliwości występowania zdarzeń w logach [101], w których wykorzystywane są najczęstsze podciągi. Jednakże, samo mierzenie częstotliwości jest podatne na nieduże zmiany sekwencji, które odpowiadają np. odmianie słów w języku polskim, lub przedrostkom i przyrostkom w języku angielskim. Takie podejście pomija silne wewnętrzne korelacje między literami w słowach, co prowadzi do nadmiernie pociętego tekstu, gdzie każdy przedrostek i przyrostek są oddzielane. VotingExperts i Zagnieżdżony Model Językowy Pitman-Yor (NPYLM), to bardziej zaawansowane probabilistyczne podejścia, są inspirowane segmentacją języka naturalnego i stosują nienadzorowane metody probabilistycznej segmentacji słów. Posiadają one też tę zaletę, że nie są zależne od alfabetu znaków, co z punktu widzenia sekwencji logów umożliwia ich utrzymanie i przeniesienie na wiele produktów.

3.5.1 Założenia

Z powodu braku zbioru referencyjnego w tej dziedzinie segmentacji logów, przygotowałem zbiór "złotej segmentacji" dla logów CloudStack i Nokii, dzięki czemu efekt mojej pracy będzie mógł być wykorzystany jako punkt odniesienia dla innych metod i przyszłych rozwiązań. Segmenty wyznaczone w referencyjnym podziale logu (określane jako "złota segmentacja") zostały przygotowane ręcznie na podstawie surowych linii logów.

Po parsowaniu logów i zamianie na identyfikatory szablonów, kolejnym krokiem jest segmentacja. Wstępna segmentacja polega na podzieleniu sekwencji logów na podstawie identyfikatorów takich jak ID bloków, ID zadań lub ID wątków. Następnie wyodrębniane są znaczące segmenty metodą VotingExperts i NPYLM.

3.5.2 Algorytm VotingExperts

Algorytm VotingExperts jest jednym z pierwszych zastosowań segmentacji słów języka naturalnego do sekwencji logów. Składa się z następujących kroków:

1. **Budowa drzewa n-gramów**
2. **Obliczanie entropii granicy (ang. *Boundary Entropy*)**
3. **Obliczanie częstotliwości ciągów znaków (ang. *Boundary Entropy*)**
4. **Standaryzacja częstotliwości i entropii**
5. **Głosowanie nad granicami**.
6. **Segmentacja szeregu**.

Rozważmy sekwencję tekstu: `itwasacold`. Załóżmy, że używamy okna o długości 3. Algorytm VotingExperts wykonuje następujące kroki:

1. **Budowa drzewa n-gramów**: Z sekwencji `itwasabrig` tworzymy drzewo n-gramów do maksymalnej długości 3.

2. **Obliczanie entropii granicy**: Na przykład n-gram `i` ma dwie możliwe kontynuacje `t`, oraz `g`, zatem entropia dla tej litery wynosi $H(i) = -(p(it)\log_2(p(it)) - p(ig)\log_2(ig)) = -(0.5) \times (-1.0) - (0.5) \times (-1.0) = 1.0$. Z kolei n-gram `it` ma tylko jedną możliwą kontynację `w`, zatem jego entropia będzie równa 0.

3. **Obliczanie częstotliwości**: Dla n-gramu `i` i okna 3 częstotliwość obliczana przez eksperta to suma $z(i) = freq(i) + freq(tw) = 2 + 1 = 3$. Natomiast dla n-gramu `it` i okna 3 częstotliwość to $z(it) = freq(it) + freq(w) = 1 + 1 = 2$

4. **Standaryzacja entropii i częstotliwości**: Obie te wartości są standaryzowane przez odjęcie wartości średniej i podzielenie przez odchylenie standardowe liczonej dla każdej długości n-gramu osobno. Na przykład dla 1-gramu średnia częstości wynosi 1.25, a odchylenie standardowe 0.43. Stąd ustandaryzowana wartość częstości wynosi $z_std(i) = 2.73$

5. **Głosowanie**: Eksperci głosują nad granicami. Dla okna `itw`, obaj eksperci głosują na podział po `i`.

6. **Segmentacja**: Miejsca z liczbą głosów ponad określony próg są wybierane jako punkty podziału. W tym przykładzie próg może być ustalony na 1 i granica będzie ustawiona na `i` i `tw`.

3.5.3 Strojenie hiperparametrów dla VotingExperts

Wybór najlepszego zestawu hiperparametrów dla modelu jest skomplikowanym zadaniem. Chociaż przeszukiwanie siatki (ang. *grid search*) jest szeroko stosowane w tym celu [96], to bardziej efektywnym podejściem jest wykorzystanie procesu Gaussa (GP). Traktując dokładność modelu jako próbkę z GP, wcześniejsze wyniki pomagają zmniejszyć niepewność i kierować wyborem kolejnego hiperparametru. Badania wykazały, że GP może zidentyfikować lepsze hiperparametry przy znacznie mniejszej liczbie eksperymentów w porównaniu z przeszukiwaniem siatki [70].

3.5.4 Nested Pitman-Yor Language Model

Model NPYLM (ang. *Nested Pitman-Yor Language Model*) [57] rozszerza tradycyjne modele oparte na procesie Dirichleta [77], wprowadzając bardziej zaawansowane modelowanie językowe, które uwzględnia zarówno n-gramy słów, jak i n-gramy znaków, co pozwala na lepsze odwzorowanie rzeczywistych granic słów w tekstach.

Proces Pitmana-Yora (PY) jest procesem stochastycznym generującym rozkłady prawdopodobieństwa, które są podobne do pewnego rozkładu bazowego G_0. Proces ten jest opisany przez dwa parametry: parametr zniżki (ang. *discount parameter*) d i parametr koncentracji (ang. *concentration parameter*) θ. Kiedy parametr zniżki d jest mały (bliski 0), nowy rozkład G będzie bardziej podobny do G_0. Gdy d jest większy (bliższy 1), oznacza to mniejsze podobieństwo do rozkładu bazowego G_0. Mała wartość θ powoduje, że nowy rozkład generuje bardziej "skupione" wyniki, z większą liczbą powtórzeń już istniejących klas (kategorii). Oznacza to, że mniej nowych klas jest wprowadzanych, a więcej obserwacji przypisuje się do już istniejących klas. Duża

wartość θ prowadzi do większego zróżnicowania wyników, co oznacza, że proces będzie bardziej skłonny do tworzenia nowych klas i dystrybuowania obserwacji w bardziej rozproszony sposób.

Zatem oba parametry kontrolują, jak bardzo generowany rozkład G jest podobny do G_0:

$$G \sim PY(G_0, d, \theta). \tag{3.5}$$

W kontekście modelowania językowego, możemy zastosować proces PY do generowania rozkładów n-gramów. Na przykład, rozkład bigramów G_2 może być generowany z G_1, który jest rozkładem znaków, co można opisać jako:

$$G_2 \sim PY(G_1, d, \theta). \tag{3.6}$$

Parametry d i θ kontrolują jak wiele różnych bi-gramów będzie modelowanych. Podobnie, trigramy mogą być generowane na podstawie bigramów, tworząc strukturę drzewiastą, w której wyższe poziomy reprezentują bardziej złożone n-gramy.

Praktyczne zastosowanie procesów PY do modelowania języka wymaga integracji tych procesów w strukturę hierarchiczną, gdzie n-gramy są generowane z procesów PY, a każde drzewo reprezentuje rozkład n-gramów na różnych poziomach. Na przykład, model HPYLM (ang. *Hierarchical Pitman-Yor Language Model*) może być użyty do modelowania n-gramów, gdzie prawdopodobieństwo n-gramu jest obliczane rekurencyjnie:

$$p(w \mid h) = \frac{c(w \mid h) - d \cdot t_{hw}}{\theta + c(h)} + \frac{\theta + d \cdot t_{h\cdot}}{\theta + c(h)} p(w \mid h_0), \tag{3.7}$$

gdzie w to słowo, h jest kontekstem słowa (np. $n-1$ poprzednich słów), $c(h)$ to liczba wystąpień danego kontekstu h, t_{hw} to liczba różnych grup, które słowo w tworzyło po kontekście h, $t_{h\cdot}$ jest liczbą wystąpień kontekstu h dla wszystkich możliwych słów w, co oznacza ogólną różnorodność w rozkładzie słów po kontekście h. Z kolei $c(w \mid h)$ to liczba wystąpień słowa w w kontekście h, a $p(w \mid h_0)$ to prawdopodobieństwo słowa w w skróconym kontekście h_0 (czyli $n-2$ poprzednich słów).

Model NPYLM rozszerza HPYLM, łącząc modelowanie n-gramów znaków i słów. Słowa są generowane z modelu słów, a następnie dzielone na znaki, które są generowane przez model znaków. Rozkład bazowy dla modelu słów $G_0(w)$ jest obliczany na podstawie prawdopodobieństwa wystąpienia ciągu znaków reprezentujących dane słowo, co można zapisać jako:

$$G_0(w) = p(c_1 \dots c_k) = \prod_{i=1}^{k} p(c_i \mid c_1 \dots c_{i-1}), \tag{3.8}$$

gdzie $c_1 \dots c_k$ to ciąg znaków reprezentujących słowo w, a $p(c_i \mid c_1 \dots c_{i-1})$ jest obliczane za pomocą HPYLM dla znaków.

Model NPYLM można także reprezentować jako zagnieżdżony proces restauracji chińskiej (Nested Chinese Restaurant Process, CRP [77]). Kiedy słowo w jest generowane, jest ono pobrane z rozkładu bazowego, który jest modelem HPYLM dla znaków. Następnie słowo jest dzielone na znaki, które są traktowane jako „zdania" w modelu znaków. Proces ten tworzy zagnieżdżoną strukturę, w której segmentacja słowa jest jednocześnie segmentacją znaków.

Wnioskowanie w modelu NPYLM polega na próbkowaniu segmentacji słów przy użyciu algorytmu Gibbs sampling [27], wykorzystując do tego dynamiczne programowanie dzięki czemu obliczenia są bardziej wydajne. Proces ten uwzględnia wszystkie

możliwe segmentacje ciągu znaków, co pozwala na dokładniejsze modelowanie prawdopodobieństw. Dodatkowo, w modelu NPYLM stosujemy korektę długości słowa przy użyciu rozkładu Poissona, co pozwala na lepsze odwzorowanie rzeczywistej dystrybucji długości słów w danym języku.

Model Nested Pitman-Yor Language Model (NPYLM) stanowi zaawansowane podejście do nienadzorowanej segmentacji słów, łącząc modelowanie n-gramów znaków i słów w złożoną strukturę hierarchiczną. Dzięki zastosowaniu procesów Pitmana-Yora, NPYLM umożliwia dokładniejsze modelowanie granic słów, co jest szczególnie przydatne w przypadku języków bez wyraźnych granic między słowami. Model ten oferuje także zaawansowane metody wnioskowania, które pozwalają na efektywne i dokładne segmentacja tekstu.

3.6 Wykrywanie anomalii

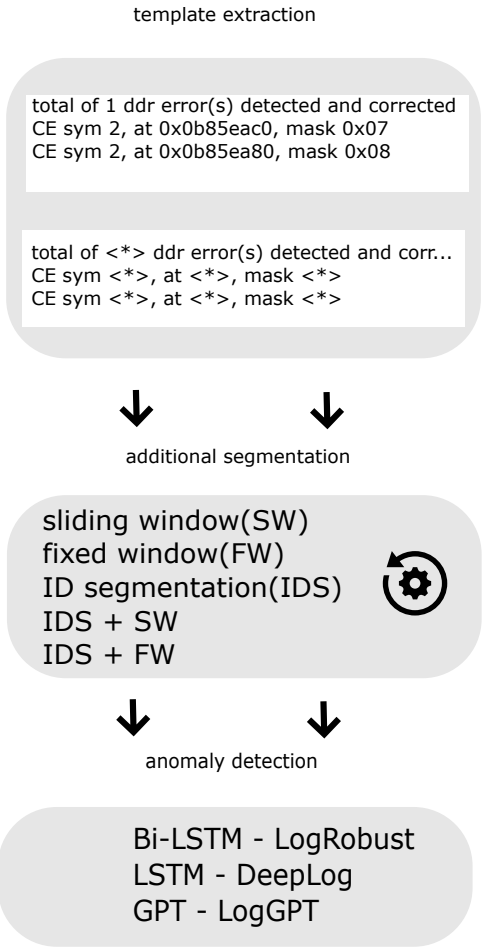
Mając do dyspozycji istotne segmenty wyodrębnione z sekwencji logów przystąpiłem do sprawdzenia tezy, czy zaproponowana segmentacja wpływała na wyniki analiz, które wykorzystują tę segmentację. W pierwszej kolejności sprawdziłem wpływ na wykrywanie anomalii. Zadanie to, w odróżnieniu od weryfikacji na złotym standardzie, jest bardziej obiektywne. Dużo łatwiej jest bowiem stwierdzić i oznaczyć, co jest w logu anomalią, niż, wyodrębnić segmenty logów stanowiące całość w sposób definitywny. Dotyczy to szczególnie sekwencji logów łączących krótkie funkcjonalności. W zależności od tego co preferuje programista, lepszym może być ich rozdrobnienie, albo połączenie.

Znane są w literaturze metody segmentowania sekwencji logów, takie jak Voting Experts i Hierarchical Probabilistic Segmentation i pokazały one swą przydatność do ręcznej analizy, natomiast ich wykorzystanie do analizy automatycznej pozostawało wciąż nieznane (Rysunek 4). Przeprowadzone przeze mnie badanie zapełnia tę lukę (Rysunek 5). Dodatkowo, rezultat wykrycia anomalii, czyli oznaczenie segmentu błędnego, zyskałby dodatkową wartość. Wynik byłby bardziej zrozumiały dla człowieka. Zamiast bowiem okna o stałej długości dostawałby segment zawierający pewną całość, opisujący część funkcjonalności. Wyniki eksperymentów przeprowadzonych na logach z BLG oraz Thunderbird potwierdzają tę tezę (Rozdział 6.3).

Wykrywanie anomalii z użyciem modeli autoregresyjnych polega na nauczeniu modelu przewidywania następnego szablonu na podstawie kontekstu. Dane treningowe składają się wyłącznie z poprawnych sekwencji. W ten sposób model uczy się, co stanowi poprawne wykonanie. Anomalie są wykrywane, jeśli napotkany następnik ma niskie prawdopodobieństwo. Jest to oparte na zdefiniowanym progu prawdopodobieństwa lub liczby najbardziej prawdopodobnych wyników traktowanych jako poprawne. Jest to zadanie identyczne z modelowaniem języka.

3.6.1 Wprowadzenie do modelu języka

Model języka (ang. *language model*) jest narzędziem szeroko stosowanym w dziedzinie przetwarzania języka naturalnego (NLP). Jego celem jest przewidywanie kolejnych elementów sekwencji tekstowych na podstawie wcześniejszych danych. Modele te znajdują zastosowanie w różnych zadaniach, takich jak automatyczne tłumaczenie, generowanie tekstu czy analiza sentymentu. Formalnie, model języka uczy się rozkładu prawdopodobieństwa nad sekwencjami tekstowymi S, gdzie $S = \{w_1, w_2, \ldots, w_T\}$, a w_i oznacza



Rysunek 4: Etapy typowej metody wykrywania anomalii.

słowo lub token na pozycji i. Zadanie modelu języka polega na estymacji warunkowego prawdopodobieństwa:

$$p(w_{t+1}|w_1, w_2, \ldots, w_t), \tag{3.9}$$

czyli prawdopodobieństwa wystąpienia słowa w_{t+1}, jako kolejnego słowa w sekwencji, znając kontekst poprzednich słów $w_1, w_2, \ldots, w_t$.

3.6.1.1 Transformer: nowoczesna architektura modeli języka

Jedną z najnowocześniejszych architektur do modelowania języka jest Transformer [79], który opiera się na mechanizmie uwagi (ang. *attention*). Transformer składa się z warstw enkodera i dekodera, gdzie każda warstwa przetwarza dane równolegle, co pozwala na lepsze uchwycenie zależności w długich sekwencjach tekstowych. W przeciwieństwie do starszych modeli sekwencyjnych, takich jak LSTM (ang. *Long Short-Term Memory*), Transformer lepiej radzi sobie z modelowaniem zależności na odległych pozycjach w sekwencji, co wynika z jego zdolności do jednoczesnego przetwarzania wszystkich pozycji w sekwencji.

Transformer działa na wejściowych sekwencjach S, przekształcając je w ukryte reprezentacje h_t za pomocą warstw uwagi (ang. *self-attention layers*) i sieci neurono-

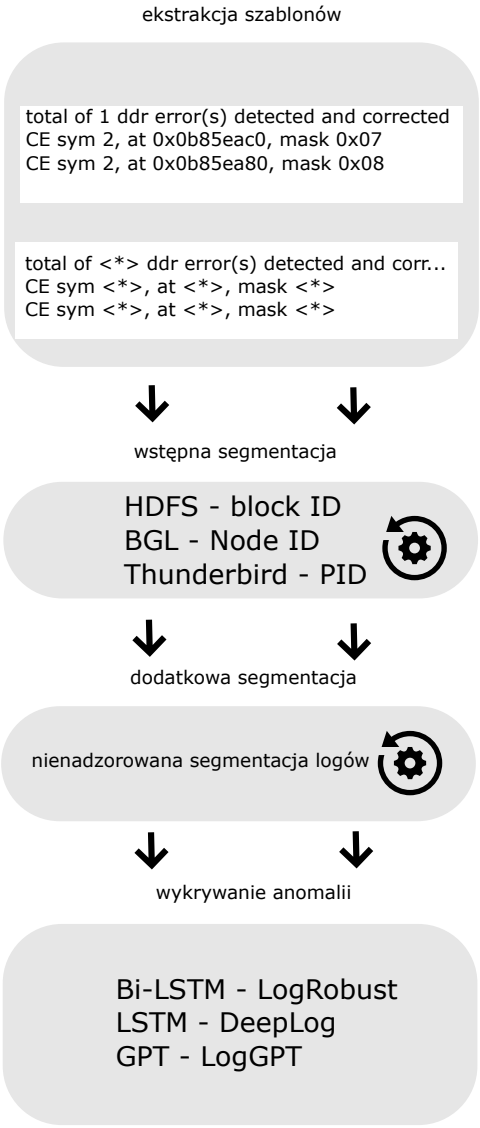
wych. Warstwy uwagi w modelu Transformer są kluczowym elementem, który pozwala na efektywną pracę z sekwencjami danych, takimi jak tekst czy sekwencje czasowe. W dużym uproszczeniu, działają one poprzez przypisywanie wag do różnych elementów sekwencji wejściowej, co umożliwia modelowi skupienie się na najważniejszych fragmentach informacji przy tworzeniu ukrytych reprezentacji.

$$h_t = \text{Transformer Encoder}(w_1, w_2, \ldots, w_t). \tag{3.10}$$

Następnie, Transformer przewiduje kolejne słowo w_{t+1} jako najbardziej prawdopodobne na podstawie rozkładu prawdopodobieństwa generowanego przez dekoder:

$$p(w_{t+1}|w_1, w_2, \ldots, w_t) = \text{Softmax}(h_t W), \tag{3.11}$$

gdzie W to macierz wag modelu.



Rysunek 5: Fazy procedury wykrywania anomalii, zaproponowane w niniejszej pracy doktorskiej.

3.6.2 DeepLog: detekcja anomalii za pomocą modelu języka logów

DeepLog [25] to model języka zaprojektowany specjalnie do analizy logów systemowych i detekcji anomalii. Bazuje on na architekturze LSTM i jest trenowany na zbiorze poprawnych sekwencji logów $D = \{S_i\}_{i=1}^{N}$, gdzie S_i to sekwencja logów, a N to liczba sekwencji w zbiorze treningowym.

Model DeepLog uczy się przewidywać następny klucz logu k_{t+1} w sekwencji $S_{1:t} = \{k_1, k_2, \ldots, k_t\}$ na podstawie poprzednich kluczy logu $S_{t-m:t}$, gdzie m oznacza rozmiar okna. Formalnie, zadanie DeepLog polega na estymacji warunkowego prawdopodobieństwa

$$p(k_{t+1}|S_{t-m:t}). \tag{3.12}$$

Celem treningu modelu jest maksymalizacja prawdopodobieństwa L przewidywań modelu dla całego zbioru treningowego:

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{t-m-1}\log p(k_{t+1}|S_{t-m:t};\theta), \tag{3.13}$$

gdzie θ to parametry modelu LSTM. Podczas detekcji anomalii, DeepLog klasyfikuje sekwencje jako błędne, jeśli rzeczywisty klucz logu nie pojawia się wśród $Top - K$ najbardziej prawdopodobnych przewidywań modelu wielokrotnie w różnych oknach sekwencji.

3.6.3 LogGPT: generatywny model językowy do detekcji anomalii

LogGPT to model detekcji anomalii w logach, oparty na architekturze GPT-2 [63], który wykorzystuje sieci Transformer do modelowania odległych zależności w logach. W przeciwieństwie do DeepLog, LogGPT nie wymaga dzielenia sekwencji logów na mniejsze okna, co pozwala na bardziej efektywne i dokładne modelowanie.

LogGPT jest uczony modelowania sekwencji logów w czasie procesu podobnego do DeepLog z tą różnicą, że na wejście otrzymuje całe sekwencje, a rekonstruuje sekwencje przesunięte o 1. Po pretrenowaniu, LogGPT jest zdolny do generowania kolejnych kluczy logów na podstawie częściowej sekwencji logów. LogGPT wykorzystuje następnie uczenie przez wzmocnienie (ang. *reinforcement learning*) do dostrojenia modelu do zadania detekcji anomalii. Faza dostrajania jest czasochłonna i nie przynosi znaczących korzyści w zestawach danych BGL i Thunderbird [29]. Dlatego ograniczyłem trening LogGPT do fazy wstępnego treningu. Zmiana ta jest bez istotnego wpływu na skuteczność metody, ponieważ zmiana F-Score dla Thunderbird wynosi 0.001%, a dla BGL 0.003%.

Po zakończeniu procesu dostrajania, LogGPT jest używany do detekcji anomalii w nowych sekwencjach logów. Dla każdej sekwencji logów $S_{1:t}$, LogGPT iteracyjnie przewiduje następny klucz logu k_{t+1} i porównuje go z $Top - K$ przewidywaniami modelu. Jeśli rzeczywisty klucz logu nie znajduje się w $Top - K$ przewidywaniach w którymkolwiek kroku, cała sekwencja jest oznaczana jako błędna.

3.6.4 Miara istotności statystycznej

Zastosowałem statystyczny test t Studenta (ang. *t-test*) do oceny istotności statystycznej różnic między średnimi dwóch zestawów danych. Test t Studenta jest szeroko uznawaną metodą statystyczną, która pomaga określić, czy obserwowane różnice są prawdopodobnie wynikiem losowej zmienności, czy odzwierciedlają rzeczywisty efekt. W tym badaniu użyłem tego testu dla porównania dwóch grup dziesięciu eksperymentów LogGPT i LogGPT używającego segmentów z VE, aby ocenić, czy ich średnie F-score różnią się istotnie. Wynik testu t Studenta został obliczony jako:

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}, \tag{3.14}$$

gdzie $\overline{X_1}$ i $\overline{X_2}$ to średnie próbek, s_1^2 i s_2^2 to wariancje próbek, a n_1 i n_2 to rozmiary dwóch grup próbek. Próbki to zbiory danych, które reprezentują wyniki dwóch różnych grup eksperymentalnych. Porównując wynik testu t Studenta z wartością krytyczną z rozkładu t, określiłem wartość p, która wskazuje prawdopodobieństwo zaobserwowania wyników, zakładając prawdziwość hipotezy zerowej. Różnice uznaje się za statystycznie istotne przy p ¡ 0.05. Przeprowadzona analiza i uzyskany wynik p niższy niż zakładane 0.05, potwierdziły, że moje podejście jest słuszne, zapewniając, że zmiana wartości F-score jest statystycznie istotna.

3.7 Lokalizacja błędu na podstawie logów

Drugim i docelowym obszarem zastosowania nienadzorowanej segmentacji logów była lokalizacja błędu w logach. Lokalizacja błędów w systemach informatycznych to skomplikowane i czasochłonne zadanie, szczególnie w przypadku dużych aplikacji generujących ogromne ilości logów. Tradycyjne techniki lokalizacji błędów oparte na analizie logów (LBFL [67]) są skuteczne, jednak często mają trudności, gdy liczba podejrzanych linii logów jest przytłaczająca, zwłaszcza gdy różnica między historycznym poprawnym wykonaniem a błędnym jest duża. Nie opisują one dokładnie algorytmu postępowania takiej sytuacji. Intuicyjnym podejściem w momencie otrzymania wielu szablonów o rankingu 1, jest wybranie wszystkich linii z nimi związanych i posortowanie ich po czasie.

Zaproponowana przeze mnie metoda kontekstowego szeregowania idzie krok dalej i umożliwia szeregowane najbardziej podejrzanych linii za pomocą miary obliczanej na podstawie kontekstu. Kontekst otrzymywany jest na dwa sposoby: z okna przesuwnego i VotingExperts.

Drugą propozycją jest stworzenie algorytmu lokalizacji na podstawie istniejącego modelu wykrywania anomalii LogGPT. Jest on niezależny od LBFL. Uczy się poprawnego wzorca na podstawie segmentów logów z poprawnych wykonań, a następnie tworzona jest hierarchia segmentów błędnych używając prawdopodobieństwa predykcji modelu.

Oceny tych dwóch podejść dokonałem w oparciu o kluczowy szablon wskazany dla każdego przykładu przez eksperta. Miarą porównania jest liczba szablonów i podejrzanych linii logów, które każda z metod wskazała do sprawdzenia przez programistę.

Na podstawie przeprowadzonych analiz nienadzorowanej segmentacji dla złotego standardu (Rozdział 4.2), dla algorytmu VotingExperts wybrałem okno o rozmiarze 7, a próg o wartości 4. Rozmiar okna przesuwnego ustaliłem na 0.25 sekundy z krokiem

0.125 sekundy, tak by średnia długość otrzymanych segmentów była zbliżona do otrzymanej z segmentacji VE.

3.7.1 Kontekstowe szeregowanie

W tradycyjnym LBFL wskaźnik podejrzliwości definiowany jest jako:

$$\text{Podejrzliwość}(l) = \frac{\text{Porażka}(l)}{\text{Porażka}(l) + \text{Sukces}(l)}, \tag{3.15}$$

gdzie Porażka(l) to liczba wystąpień szablonu logu l w logach z błędnego wykonania, a Sukces(l) to liczba wystąpień szablonu logu l w logach poprawnego wykonania (Tabela 1). Wartość Podejrzliwosci pozwala uszeregować szablony logów według ich prawdopodobieństwa powiązania z błędem. Szablony logów, które pojawiają się wyłącznie w logach z błędem, otrzymują wskaźnik podejrzliwości równy 1, co wskazuje na wysokie prawdopodobieństwo związku z błędem, podczas gdy te, które pojawiają się tylko w logach z poprawnego wykonania, otrzymują wskaźnik 0.

Metoda LBFL zwraca dużą liczbę podejrzanych logów w sytuacjach, gdy różnica między logiem poprawnym a błędnym jest duża. Taka sytuacja jest częsta w dużych systemach oprogramowania, gdzie pojedynczy problem może spowodować kaskadę awarii i wygenerować tysiące powiązanych logów. Proste zwrócenie wszystkich tych logów może okazać się nieprzydatne.

Tabela 1: Linie logów pochodzące z poprawnego wykonania programu. Każda linia ma przyporządkowany numer szablonu i liczbę jej wystąpień w pliku z logiem.

Nr	Zawartość logu	ID szablonu	Sukces(l)
1	INF, State change notif received [CONNECTED]	1	7
2	INF, State change notif received [CONNECTED]	1	-
3	INF, State change notif received [CONNECTED]	1	-
4	INF, State change notif received [CONNECTED]	1	-
5	INF, State change notif received [CONNECTED]	1	-
6	INF, State change notif received [CONNECTED]	1	-
7	INF, State change notif received [CONNECTED]	1	-
8	INF, SW Version: xxxxx	2	1
9	INF, currentImageType = package.01	3	1
Liczba	9	3	

Aby rozwiązać ten problem, zaproponowałem ranking oparty na kontekście. Intuicyjnie, niewidziane linie logów, które pojawiają się w kontekście dobrze znanych linii logów, są bardziej podejrzane niż niewidziane linie logów występujące razem. Powodem jest to, że zgrupowane niewidziane linie logów często stanowią ślady stosu, zrzuty awarii lub nowe funkcje, podczas gdy pojedyncza anomalia wśród dobrze znanego zachowania może być pierwszym symptomem, że program zaczyna działać nieprawidłowo. Aby obliczyć miarę rankingu kontekstowego, najpierw segmentujemy plik logu, wykorzystując nienadzorowane segmentacja sekwencji logów, a następnie obliczamy średnią wartość podejrzliwości szablonów logów w segmentach, w których występują najbardziej podejrzane szablony logów. Ranking oparty na kontekście jest obliczany za pomocą następującego wzoru:

$$Ranking\_kontekstowy(Segm_k) = 1 - \frac{\sum_{l_i \in Segm_k} \text{Podejrzliwość}(L_i)}{|Segm_k|}, \tag{3.16}$$

Tabela 2: Linie logów pochodzące z błędnego wykonania programu. Każda linia ma przyporządkowany numer szablonu i liczbę jej wystąpień w pliku z logiem.

Nr	Zawartość logu	ID szablonu	Porażka(l)
1	WRN, BlackBoxLoggingService MemoryItem::update	4	1
2	WRN, BlackBoxLoggingService get entry value	5	1
3	WRN, BlackBoxLoggingService get entry value	5	-
4	INF, State change notif received [FAILED]	1	4
5	INF, State change notif received [CONNECTED]	1	-
6	INF, Recovery action requested	6	1
7	INF, State change notif received [CONNECTED]	1	-
8	INF, State change notif received [CONNECTED]	1	-
9	INF, Received reset request because of node2 failure.	7	1
10	INF, SW Version: xxxxx	2	1
11	INF, currentImageType = package.01	3	1
Liczba	11	7	

gdzie $Segm_k$ oznacza k-ty segment pliku logu. Im większa średnia wartość segmentu, tym mniej podejrzane są linie.

W przykładzie z Tabeli 2, błąd znajduje się w linii 6. Pojawia się on w kontekście udanych odpowiedzi na połączenia i wyjaśnia późniejszą awarię. Załóżmy, że pierwszy segment nieudanego logu obejmuje linie od 1 do 3. Odpowiadająca im sekwencja identyfikatorów szablonów to $4, 5, 5$, a wszystkie linie zostały ocenione jako podejrzane. Ranking kontekstu tego segmentu został obliczony jako $Ranking\_kontekstowy(S_1) = 1-1 = 0$. Drugi segment obejmuje linie od 4 do 8 z sekwencją identyfikatorów szablonów $1, 1, 6, 1, 1$, a ranking kontekstu tego segmentu wynosi $Ranking\_kontekstowy(S_2) = 1 - mean(4 \times 0.36 + 1) = 0.5$. Ostatni segment obejmuje linie od 9 do 11, a jego ranking kontekstu wynosi $Ranking\_kontekstowy(S_3) = 1 - mean(0.5 + 0.5 + 1) = 0.3$.

Segment z najwyższym rankingiem kontekstu to S_2, co było spodziewanym wynikiem. Trzeci segment jest niżej w rankingu. Pierwszy segment jest słusznie najniżej w rankingu, ponieważ nie jest związany z rzeczywistym błędem, lecz reprezentuje dodatkowy poziom logowania włączony dla łatwiejszego wykrycia przyczyny awarii.

3.7.2 Kontekstowe szeregowanie z użyciem LogGPT

Druga metoda wykorzystuje algorytm LogGPT. W swoim założeniu służy on do wykrywania anomalii. Jednakże jako model generatywny pozwala na sprawdzenie z jaką pewnością model jest przekonany, że dany następnik jest anomalią. Wartość $Top - K$, która służy do określenia co jest anomalią, a co nie, dobrałem tak, by najważniejszy log (oznaczony przez eksperta dla każdego przypadku) był oznaczony jako anomalia.

Metoda ta uczy się poprawnych wzorców, a następnie ocenia czy otrzymane segmenty są błędem, czy nie używając miary $Top - k$ i zwraca pewność modelu obliczoną jako:

$$Miara\_\text{pewności} = 1 - \min P, \tag{3.17}$$

gdzie P to zbiór prawdopodobieństw wszystkich elementów segmentu.

Tabela 3: Parametry użytych zestawów danych: Nokia 100 - zbiór 100 niesegmentowanych plików logów Nokii; Nokia golden - wybrany plik logów Nokii; oraz pochodzący z CloudStack.

Zestaw danych	Rozmiar (MB)	Zdania/ Wątki	Litery na zdanie	Maks. zdanie	Min. zdanie	Śr. zdanie	Entropia
Nokia 100	228	47171	1979593	6670	1	41	6.15
Nokia golden	2	44	15833	5096	1	359	4.9
CloudStack	186	105	391262	25690	21	3726	1.31

3.8 Zbiór danych

3.8.1 Klasyfikacja sekwencji logów z testów

W eksperymencie dotyczącym HDFS wykorzystano wersję 3.1.1 tego systemu, w której zidentyfikowano problem oznaczony jako HDFS-10453. Zebrano 5101 logów z testów jednostkowych, które odpowiadały 658 różnym zestawom testów. Liczba testów dla każdego zestawu testowego wynosiła od 1 do 78, przy czym najdłuższy test jednostkowy składał się z 3 milionów szablonów logów, a najkrótszy zaledwie z jednego szablonu. Eksperyment miał na celu sprawdzenie, czy zaproponowane podejście może być użyteczne w kontekście identyfikacji zestawu testowego powiązanego z przyczyną problemu.

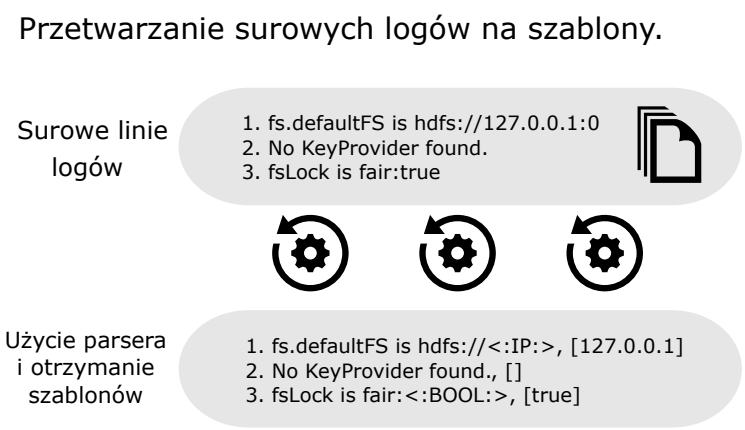
W eksperymencie dotyczącym oprogramowania radiowego firmy Nokia, związanego z rzeczywistym problemem, zebrano dane z 944 testów należących do 180 zestawów testów. Liczba testów w zestawach wahała się od 1 do 27. Wykorzystano balansowanie zbioru danych, zwiększając próbkowanie 10-krotnie i zmniejszając próbkowanie dwukrotnie. Najdłuższy przypadek testowy zawierał 588 902 linie logów, a najkrótszy jedynie trzy linie. Celem eksperymentu było sprawdzenie skuteczności modeli uczenia maszynowego w klasyfikacji linii logów związanych z przyczyną problemu oraz w przypisywaniu ich do odpowiednich zestawów testowych.

3.8.2 Segmentacja sekwencji logów

Eksperymenty przeprowadziłem przy użyciu dwóch zestawów danych: zestawów logów pochodzących z CloudStack i Nokia. Informacje na temat tych zestawów danych przedstawiono w Tabeli 3. Zawiera ona nazwę zestawu danych, liczbę zdań (lub wątków), liczbę szablonów, maksymalną, minimalną i średnią długość zdania oraz entropię znakową korpusu. Tabela jest uporządkowana w kolejności malejącej wartości entropii znakowej (ang. *Entropy*) (szersze wyjaśnienie w Rozdziale 3.1).

Pierwszy zestaw danych składa się z logów CloudStack użytych w [6]. Zawiera on 1 009 280 linii z 279 553 wątków. Usunąłem wszystkie krótkie wątki składające się z mniej niż 20 linii, ponieważ nie wymagają one segmentacji, są często traktowane jako jeden segment przez większość metod segmentacji i nie stanowią wyzwania dla ludzi w zrozumieniu. Dodatkowo wszystkie wątki o liczbie linii mniejszej niż 20 stanowią ponad połowę analizowanych logów. Ich usunięcie przyśpiesza działanie algorytmu, a sekwencje logów o długości 20 są dla człowieka łatwe do zrozumienia. Wartość dodana z segmentowania tak krótkich sekwencji byłaby niewielka. Pozostało 105 wątków dłuższych niż 20 linii, co dało 391 262 linii. Najdłuższy wątek składał się z 25 690 linii

logów, a średnia długość wątku wynosiła 3 726 linii. Te wątki zostały podzielone na 217 297 segmentów. Logi zostały przetworzone w standardowy sposób, jak opisano poniżej. Format logów to: $< timestamp >< level >< location >< thread\_id >< log\_text >$, gdzie $< level >$ to poziom logowania, $< location >$ to nazwa modułu, $< thread\_id >$ to numer wątku, a $< log\_text >$ to tekst zarejestrowanego logu.



Rysunek 6: Proces parsowania logów od tekstu do szablonów logów i zmiennych.

Drugi zestaw danych składał się z logów zebranych podczas 101 poprawnych i błędnych wykonań oprogramowania Nokii. Podzieliłem go na zbiór "Nokia 100" do celów treningowych, oraz złoty standard w którego skład wchodził jeden plik. Log został podzielony według identyfikatora wątku. Całkowita liczba linii logów w zestawie treningowym wynosi 1 979 593 w 47 171 wątkach, a w złotym standardzie jest 15 833 linie w 44 wątkach. Traktowałem zawartość wątku jako zdanie, a identyfikator zdarzenia logu jako literę. W ten sposób zdania są znacznie dłuższe niż zwykle zdania w języku naturalnym. Najdłuższe zdanie w złotym standardzie miało 5096 liter, ze średnią długością 359. Dla Nokia 100 maksymalna długość zdania wynosiła 6670, ale średnio zdania miały 41 liter, co było krótsze niż w złotym standardzie. Moim głównym celem było określenie najlepszego algorytmu do segmentacji logów pod względem wskaźnika F. Półautomatycznie stworzyłem złoty standard dla logów Nokii, używając algorytmu VotingExperts i wiedzy eksperckiej. Implementacja i zestawy danych są dostępne online. Pliki logów mają następujący format: $< component\_id >< timestamp >< thread\_id >< level >< log\_text >$, gdzie $< component\_id >$ to nazwa komponentu, $< thread\_id >$ to numer wątku, $< level >$ to poziom logowania: Info, Debug, Warning lub Error, a $< log\_text >$ to tekst zarejestrowanego logu. Logi zostały przetworzone w następujący sposób (Rysunek 6):

- wybór logów z powyższym formatem logów,
- usunięcie $< component\_id >$ i $< timestamp >$ z każdej linii logu w celu przyspieszenia następnego kroku Drain, ponieważ te kolumny nie wpływają na zawartość wyodrębnionych zdarzeń logów, a jednocześnie znacząco spowalniają proces,
- wyodrębnienie zdarzeń logów za pomocą Drain,
- oddzielenie linii z różnych wątków,
- zastąpienie linii logów odpowiednim identyfikatorem zdarzenia logu,

- połączenie wszystkich wątków z wszystkich plików w jeden zestaw danych.

3.8.3 Przygotowanie złotego standardu segmentacji

Wszystkie przykłady wykorzystane w tym badaniu pochodzą z CloudStack. Dla każdego zestawu logów przeprowadziłem segmentację złotego standardu. Na początku zastosowałem algorytm VotingExperts z domyślnymi ustawieniami, a następnie ręcznie zastosowałem określone reguły w celu dopracowania automatycznej segmentacji.

Identyfikując niepoprawne niepoprawne segmenty, ekspert segmentujący logi użył następujących założeń:

1. Niespójna funkcjonalność: Niepoprawny segment może zawierać linie logów, które nie dotyczą jednej funkcjonalności, np. uruchamiania maszyny wirtualnej.

2. Niezgodność ze słowami kluczowymi: Jeśli segment nie jest spójny lub nie odnosi się do słów kluczowych, np. rozpoczęcia i zakończenia jakiejś funkcji, może być uznany za niepoprawny. Przykładowo, „Loading" może być uznane za początek bloku, a „Loaded" za jego koniec.

3. Błędna interpretacja wzorców: Segmenty powinny dokładnie reprezentować widoczne powtarzające się sekwencje szablonów. Niepoprawne segmenty mogą powstać, jeśli wzorce są błędnie zinterpretowane lub jeśli niepowiązane wzorce są omyłkowo uwzględnione.

4. Niezgodność identyfikatora obiektu/wątku/bloku: Segmentacja musi dokładnie odwzorowywać logi związane z konkretnym obiektem/wątkiem/blokiem. Jeśli logi z niepowiązanych obiektu/wątku/bloku są uwzględnione w segmencie, może to być uznane za niepoprawne.

5. Nieprawidłowe traktowanie wyjątków: Jeśli stos wyjątków jest podzielony na wiele segmentów, mogą one być uznane za niepoprawne.

6. Przekroczenie długości: Segmenty przekraczające zdefiniowane kryterium długości (np. więcej niż 60 linii) mogą zawierać zbyt dużo informacji, co prowadzi do potencjalnego zamieszania i błędnej interpretacji. Zatem jeśli na przykład segment wyznaczony przez "Loading" i "Loaded" jest zbyt długi, należy wyodrębnić z niego mniejsze segmenty. Zasada ta stoi ponad zasadą 2 i 3.

Przestrzegając tych warunków i założeń, można zidentyfikować segmenty w zestawie danych logów, które odbiegają od oczekiwanych kryteriów, wskazując na potencjalne nieścisłości lub błędy. Prawidłową segmentację osiągnięto poprzez wykorzystanie następujących założeń:

- Segmentację oparta na słowach kluczowych.

- Wykorzystanie słów kluczowych oznaczających początek i koniec, takich jak „Loading" i „Loaded".

- Wyodrębnianie wzorców okresowych w celu zapewnienia dokładnego uchwycenia rzeczywistych okresów.

- Segmentacja oparta na identyfikatorze instancji obiektu przetwarzanego.

- Spójne traktowanie wyjątków w obrębie jednego segmentu.

Moim celem było wyodrębnienie segmentów odpowiadających odrębnym funkcjonalnościom — wystarczająco długich, aby ułatwić zrozumienie logów, ale na tyle zwięzłych, aby pozostały łatwe do analizy dla człowieka. Poprzez eksperymenty ustaliłem, że segmenty składające się z maksymalnie 60 linii były najbardziej odpowiednie do ręcznej analizy przez programistę.

3.8.4 Wykrywanie anomalii

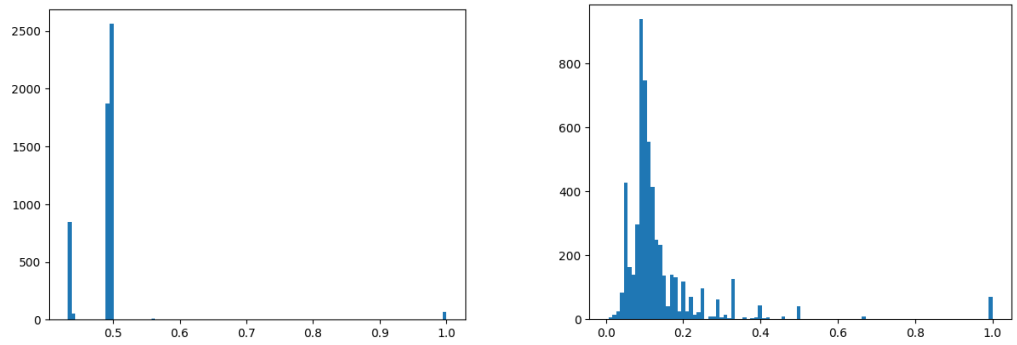
Weryfikację mojego podejścia na danych open source zrealizowałem z użyciem danych BGL i Thunderbird. Zestaw danych BGL [59] składa się z 4 713 494 linii logów, z których 839 024 zostało zidentyfikowanych jako błędne. Podobnie, zestaw danych Thunderbird [59] zawiera 20 000 000 linii, z których 760 167 to linie błędne. Zestaw danych BGL zawiera 396 szablonów logów, natomiast zestaw danych Thunderbird obejmuje 7 703 szablonów (Tabela 4). Wyniki porównałem z oryginalnym algorytmem LogGPT, który opiera się na segmentacji za pomocą okna przesuwnego. Pełna implementacja jest dostępna publicznie [21]. Dla zestawów danych BGL i Thunderbird rozmiar okna został ustawiony na 60 sekund, zgodnie z propozycją w oryginalnym artykule [29]. Na początku użyłem algorytmu Drain do wyodrębnienia identyfikatorów szablonów logów. Zestaw danych BGL został podzielony według identyfikatora wierzchołka, a zestaw danych Thunderbird według PID. Następnie zastosowałem algorytm VotingExperts jako dodatkową metodę segmentacji w celu wyodrębnienia mniejszych segmentów.

Tabela 4: Informacje o zbiorach danych, które wykorzystałem w testach weryfikujących skuteczność wykrywania anomalii.

Zbiór danych	**BGL**	**Thunderbird**	**Nokia**
Liczba Linii	4.713.494	20.000.000	4.685.619
Anomalie	839.024	760.167	38.130
Normalne	3.874.470	19.239.833	4.647.489
Szablony	396	7.703	25.202

Dodatkowo został użyty zbiór Nokii składający się z łącznie 4 685 619 linii, z czego 4 647 489 to linie poprawne, a 38 130 to linie błędne. Liczba unikalnych szablonów to 25 202. Zbiór danych został przygotowany przez zebranie wszystkich plików z fragmentami zapisanymi przez narzędzia raportowania błędów, działających w stacjach bazowych Nokia i dotyczących komponentów radiowych, w okresie od 2021 do 2024 roku. Z tego zbioru wybrane zostały fragmenty, które zostały wskazane przez testerów na etapie ręcznej analizy logów, a następnie wyselekcjonowano te, które jasno wskazywały na błąd (ponieważ niektóre fragmenty logów z raportu o błędzie opisują tylko scenariusz, a nie sam błąd). Warto również zaznaczyć, że zmieniono sposób definiowania okna: z okna czasowego na okno oparte na rozmiarze. Oznacza to, że czas nie jest już brany pod uwagę, a jedynie długość okna.

Podczas etapu parsowania logów użyto Drain. Po wyodrębnieniu identyfikatorów i dodatkowej segmentacji skonstruowałem zestaw danych treningowych zawierający 5 000 poprawnych przykładów. Zbiór testowy zawierał pozostałą część zestawu danych. Model GPT zawierał sześć warstw i sześć głowic. Uczenie ze wzmocnieniem nie było



(a) a (b) b

Rysunek 7: Metryki wystąpień użyte w (a) standardowym LBFL oraz (b) podejściu opartym na kontekście.

stosowane. Metryka $Top-K$ została ustawiona na 50%, co oznacza, że jeśli obserwowany szablon logu nie znajdował się w górnych 50% przewidywanych szablonów logów podczas fazy testowej, cała sekwencja była oznaczana jako błędna. Szczegółowy opis tych parametrów można znaleźć w oryginalnym artykule dotyczącym LogGPT [29].

3.8.5 Lokalizacja błędu na podstawie logów

Przeprowadziłem swoje badanie na rzeczywistych logach przemysłowych z firmy Nokia. Zestaw danych użyty w tym badaniu, który jest publicznie dostępny wraz z kodem [22], zawiera zanonimizowane logi z Nokii, zawierające trzy błędne wykonania. Anonimizacja została wprowadzona poprzez usunięcie zawartości szablonów logów, pozostawiając jedynie identyfikatory szablonów. Nazwy wątków i szczegóły poziomu również zostały zanonimizowane przez zastąpienie ich ciągami "$thread < num >$ " i "$level < num >$ ". Dla pierwszych dwóch scenariuszy błędnego wykonania zebrałem 10 logów z poprawnego wykonania sprzed daty wystąpienia usterki. Dla trzeciego scenariusza dostępny był tylko jeden log poprawnego wykonania i jeden log z błędnego wykonania. Logi zostały znormalizowane przez usunięcie znaczników czasu oraz linii z opisami obiektów interfejsu, podzielone według identyfikatora wątku i przetworzone za pomocą Drain. Następnie użyto VotingExperts do segmentacji sekwencji szablonów logów z każdego wątku.

Realizujac badania szukałem odpowiedzi na dwa pytania badawcze:

Pytanie badawcze 1: Czy ranking według kontekstu zmniejsza liczbę unikalnych szablonów logów do sprawdzenia?

Pytanie badawcze 2: Czy ranking według kontekstu zmniejsza liczbę linii logów do sprawdzenia?

3.8.5.1 Przykład 1

W tym scenariuszu przyczyną błędu był brak komunikacji z jednym z węzłów. Nieudana komunikacja została zarejestrowana w czasie poprawnej komunikacji z innymi węzłami. Na początku logów pojawiło się wiele dodatkowych wpisów z modułu logowania, który został włączony przez testera, aby upewnić się, że wszystkie możliwe logi zostaną zebrane. Te logi, które zwykle nie są włączane, zanieczyściły logi fałszywie pozytywnymi wynikami ocenionymi na 1.

Log z błędnego wykonania zawierał 250 578 linii z 5 557 szablonami, podczas gdy dziesięć logów z poprawnego wykonania logów zawierało 1 172 986 linii z 6 124 szablonami. Tradycyjna metoda LBFL zidentyfikowała 73 podejrzane szablony w logu z błędnego wykonania i wskazała 557 linii logu w tym tą bezpośrednio związaną z błędem. Wskazując 557 linii do oceny przez inżyniera LBFL zmniejszył liczbę linii, którą musi przeanalizować o 78%.

3.8.5.2 Przykład 2

W tym scenariuszu przyczyną błędu przekroczenia limitu czasu na jednym z wątków, ponieważ przy połączeniu ze starszymi jednostkami, zasoby nie były zawsze poprawnie zwalniane. Tradycyjny LBFL ocenił wiele szablonów na 1 w skali podejrzanych wyników, które jednak nie były związane z błędem tylko były wynikiem nowo wprowadzonej funkcjonalności. Pojawienie się błędu nie było związane z wprowadzeniem tej funkcjonalności, dlatego obecność logów z nią związanych była dodatkowym utrudnieniem.

Log z błędnego wykonania zawierał 90 463 linie z 4 624 szablonami, podczas gdy poprawne logi zawierały 676 942 linie z 5 475 szablonami. W logu z błędem było 665 szablonów niespotkanych w logu z poprawnego wykonania, które są związane z 11 081 liniami logów. Tradycyjna metoda LBFL zidentyfikowała 665 podejrzanych szablonów w logu z błędnego wykonania i wskazała 11 081 linii logu w tym tą bezpośrednio związaną z błędem. Wskazując 11 081 linii do oceny przez inżyniera LBFL zmniejszył liczbę linii, którą musi przeanalizować o 98%.

```
1  info coredumps: [Crashed PID:4] File file_name
2  info component2: Child process 10 died
3  info component3: Child process 11 died
4  info coredumps: [Crashed PID:4] coredump stop notification for PID:5 begin.
5  info component2: Child process 12 died
6  info coredumps: [Crashed PID:4] File file
7  info coredumps: [Crashed PID:4] coredump stop notification for PID:4 begin.
8  info coredumps: [Crashed PID:5] coredump stop notification for PID:5 end.
9  info coredumps: [Crashed PID:5] Core dumping completed.
10 err component4: core_dump ERR: -=-= fatal error report. (This is a crash.)
11 wrn component5: Timed out to acquire the mutex
12 err component4: core_dump ERR: description:
   Timed out waiting on mutex
13 err component4: core_dump ERR: parameter: 0
14 err component4: core_dump ERR: id (OS):
15 err component4: core_dump ERR: EU id (OS):
16 err component4: core_dump ERR: EU name: component5
17 err component4: core_dump ERR: EE command:
18 err component4: core_dump ERR: filename:
19 err component4: core_dump ERR: line number: 162
```

Rysunek 8: Fragment logu z Przykładu 2 z najważniejszym błędem w linii 12.

3.8.5.3 Przykład 3

W tym scenariuszu miałem do dyspozycji ograniczoną liczbę linii logów, ponieważ log z poprawnego wykonania zawierał tylko 8 188 linii, a log z błędnego wykonania 6 980 linii. Błąd wynikał z nieprawidłowej konfiguracji niektórych interfejsów. Prawdziwa różnica polegała na tym, że niektóre funkcje nie zostały wykonane; co można wykryć poprzez analizę działań naprawczych podejmowanych przez system, który m.in. próbował je wielokrotnie wykonać. Dlatego najważniejsze były wszystkie logi zawierające ciąg "RecoveryService".

Log z błędnego wykonania zawierał 6 980 linii z 1 268 szablonami, podczas gdy poprawne logi zawierały 8 188 linii z 1 282 szablonami. Tradycyjna metoda LBFL zidentyfikowała 86 podejrzanych szablonów w logu z błędnego wykonania i wskazała 646 linii logu w tym tą bezpośrednio związaną z błędem. Wskazując 646 linii do oceny przez inżyniera LBFL zmniejszył liczbę linii, którą musi przeanalizować o 90%.

Rozdział 4
Wyniki

Przeprowadziłem szereg eksperymentów weryfikujących postawione cele. Zacząłem od sprawdzenia, czy sekwencja logów pochodząca z testu jednostkowego niesie w sobie informację wystarczającą do precyzyjnej klasyfikacji. Zyskując potwierdzenie tej tezy, mogłem przystąpić do segmentacji logów produkcyjnych metodą dokładniejszą niż okno przesuwne (grupującą sekwencje logów według funkcjonalności, a nie czasu pojawienia się). Dokładność tej segmentacji zweryfikowałem według tego co oczekuje ekspert systemu w złotej segmentacji. Drugim poziomem weryfikacji było sprawdzenie wpływu segmentacji na zadanie końcowe, czyli wykrywanie anomalii i lokalizację błędu.

Tabela 5: Modele ML wytrenowane na danych Nokia. Wyniki pokazują, jak dobrze model potrafi rozpoznać zestaw testowy na podstawie sekwencji logów.

Model	Dokładność (%)	Precyzja (%)	Czułość (%)
RFC	96	97	97
GBC	96	96	96
NBC	91	91	91

4.1 Klasyfikacja sekwencji logów z testów

Klasyfikację sekwencji logów oceniłem dla danych z HDFS i Nokia przez obliczenie precyzji, czułości i F-score. Implementacja i przykładowe dane są dostępne w repozytorium [20]. Wyniki na danych Nokia pokazują, że najlepszym modelem jest RFC, który osiąga wysoką dokładność, precyzję i czułość podczas nauki nazw zestawów testowych, odpowiednio 96%, 97% i 97%. Model GBC osiąga nieznacznie gorsze wyniki 96%, 96% oraz 96%. Natomiast NBC znacząco mniejsze 91%, 91% oraz 91% (Tabela 5). Wyniki NBC są jednak wysokie, co może oznaczać, że kolejność zdarzeń w sekwencji nie jest tak ważna jak obecność konkretnych szablonów w zbiorze Nokii by stwierdzić, z którego testu pochodzą. Na logach testów jednostkowych HDFS wygląda to nieco inaczej, gdzie przy wciąż wysokich wynikach RFC 93%, 95% i 93%, oraz GBC 92%, 94% i 92%, NBC osiąga dużo gorsze wyniki 66%, 69% i 66% (Tabela 6). Może to oznaczać, że kolejność w przypadku sekwencji logów HDFS ma kluczowe znaczenie dla rozpoznania, z którego zestawu testów pochodzą. Model RFC osiąga najlepsze wyniki na obu zbiorach logów. Model ten potrafi skutecznie wykryć nieliniowe zależności między danymi, co sugeruje, że takie jest powiązanie między sekwencjami logów a nazwami testów. Podobne, choć nieco niższe wyniki osiąga GBC, a jego charakterystyką jest umiejętność rozróżnia-

nia bardzo podobnych przykładów. To z kolei, może sugerować, że wiele sekwencji pochodzących z różnych zestawów testów jest do siebie podobnych.

Tabela 6: Trzy modele ML wytrenowane na danych HDFS. Wyniki pokazują, jak dobrze model potrafi rozpoznać zestaw testowy na podstawie sekwencji logów.

Model	Dokładność (%)	Precyzja (%)	Czułość (%)
RFC	93	95	93
GBC	92	94	92
NBC	66	69	66

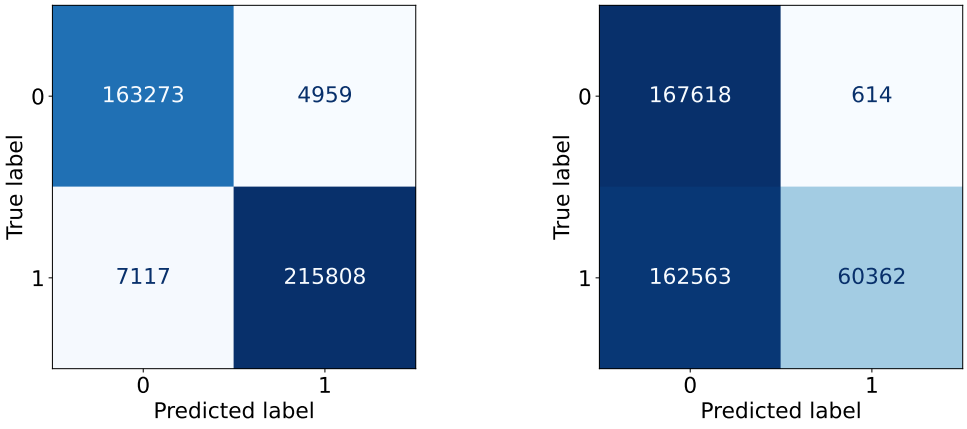
4.2 Segmentacja sekwencji logów

Zweryfikowałem wyniki segmentacji na danych stanowiących złoty standard. Najpierw sprawdziłem jakość segmentacji na logach CloudStack i Nokia. Dla złotej segmentacji CloudStack i Nokii, VotingExperts okazał się lepszym algorytmem.

4.2.1 CloudStack

Na początku oceniłem metody przy użyciu logów open-source. Jakość oceniłem w porównaniu ze złotym standardem, który stworzyłem na podstawie segmentacji VotingExperts. Złoty standard dla CloudStack składa się z 223 030 segmentów. Wątki użyte do segmentacji były dłuższe niż 20 linii. Przeprowadziłem eksperyment dla każdej z metod segmentacji. VotingExperts osiągnął najlepszy wynik z F-score na poziomie 97%.

Wyniki przedstawiłem w Tabeli 7, a macierze pomyłek pokazałem na Rys. 9. Model VotingExperts osiągnął bardzo dobre wyniki w segmentacji logów CloudStack, uzyskując



(a) (b)

Rysunek 9: Macierze pomyłek granic słów wprowadzone przez VotingExperts (a) i NPYLM (b) dla logów CloudStack.

Tabela 7: Porównawcza analiza F-score segmentacji logów CloudStack przy użyciu metod VotingExperts i NPYLM.

Algorytm	F-score (%)	Precyzja (%)	Czułość (%)
VE	**97.3**	97.7	**96.8**
NPYLM	42.5	**98.9**	27.0

F-score na poziomie 97%. Wyniki NPYLM były znacznie gorsze, na poziomie 42%, głównie ze względu na bardzo niską czułość tej metody, wynoszącą 27%. Pod względem precyzji, NPYLM uzyskał nieco wyższy wynik, 98.9%, podczas gdy dla VotingExperts wynosił on 97.3%.

4.2.2 Logi Nokia

Zweryfikowałem wyniki algorytmów w porównaniu ze złotą segmentacją logów Nokia (więcej szczegółów w podrozdziale 3.8.2). Obie sprawdzone metody osiągnęły dużo niższe wyniki niż dla CloudStack. Wynika to z faktu, że logi Nokii są dużo bardziej różnorodne, a oczekiwana przez eksperta segmentacja była dużo bardziej wymagająca, dla tych metod. Patrząc na VotingExperts, który bazuje na silnej wewnętrznej zależności szablonów w segmencie, oznaczałoby to, że ta zależność była dla niego trudna do uchwycenia przy dużej ilości szablonów i różnorodnych sekwencjach. Dla eksperta były to natomiast widoczne zależności.

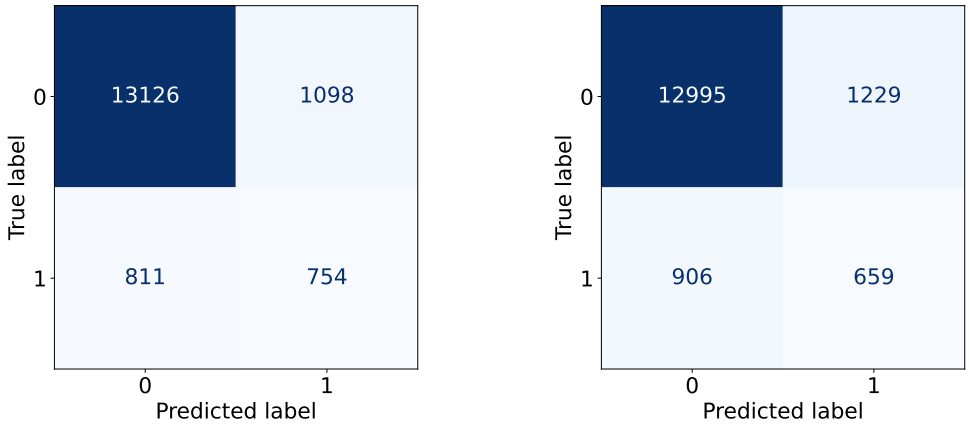
Moje eksperymenty wykazały wyraźną przewagę VotingExperts nad NMLP.

Tabela 8: Porównanie F-score algorytmów segmentacji zastosowanych do logów Nokia w trzech eksperymentach z różnymi danymi treningowymi: złoty standard (exp1), 100 plików historycznych (exp2) i kombinacja złotego standardu oraz 100 plików historycznych (exp3).

Algorytm	F-score (%)	Precyzja (%)	Czułość (%)
VotingExperts exp1	**44.1**	**40.7**	**48.1**
VotingExperts exp2	n/d	n/d	n/d
VotingExperts exp3	38.1	34.9	42.1
NPYLM exp1	22.5	**16.3**	36.0
NPYLM exp2	22.5	14.6	**45.8**
NPYLM exp3	**23.6**	16.1	44.2

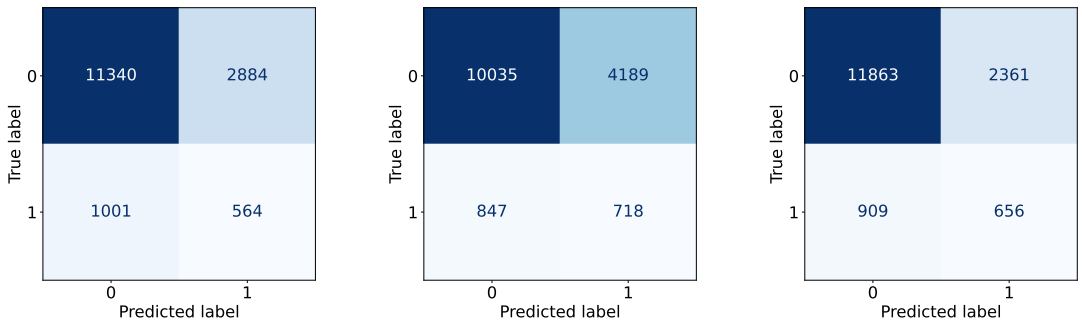
Przeprowadziłem trzy eksperymenty. W pierwszym, plik ze złotym standardem był użyty jako zbiór treningowy i testowy, w drugim 100 plików historycznych było użytych jako zbiór treningowy, a złoty standard jako zbiór testowy. W drugim eksperymencie VotingExpert nie mógł przeprowadzić żadnej segmentacji z powodu braku testowych n-gramów w zbiorze treningowym. W trzecim eksperymencie było 100 plików historycznych i złoty standard jako zbiór treningowy, a złoty standard jako zbiór testowy. W pierwszym eksperymencie VotingExpert osiągnął najlepszą segmentację z F-score 44.1% (Tabela 8) dla maksymalnej długości okna 7 i progu 4. Średnia długość segmentu wynosiła 8. W drugim eksperymencie NPYLM osiągnął drugą najlepszą segmentację z 22.5% (Tabela 8) po 100 epokach i średnią długością segmentu 4. Czułość dla VotingExperts wynisła 48%, a dla NPYLM 36%. NPYLM osiągnął F-score 22%, czułość 45% i średnią

długość segmentu 3. W trzecim eksperymencie VotingExperts osiągnął F-score 38.1% (okno 17, próg 4), średnią długość segmentu 8, a NPYLM 23% i średnią długość segmentu 3. W tym przypadku czułość wynosiła 42.1% dla VotingExperts i 44.2% dla NPYLM.



(a) (b)

Rysunek 10: Macierze pomyłek granic słów wprowadzone przez VotingExperts dla logów Nokia po treningu na złotym standardzie (a) oraz 100 plikach plus złoty standard (b).



(a) (b) (c)

Rysunek 11: Macierze pomyłek granic słów wprowadzone przez NPYLM dla logów Nokia po treningu na złotym standardzie (a), 100 plikach (b) i 100 plikach plus złoty standard (c).

Wyniki przedstawiłem w Tabeli 8, macierze pomyłek dla VotingExperts pokazałem na Rys. 10, a dla NPYLM na Rys. 11. VotingExperts osiągnął lepsze wyniki w segmentacji logów Nokia. Pod względem precyzji, VotingExperts zawsze był lepszy od NPYLM. W przypadku czułości, z wyjątkiem eksperymentu 3, gdzie wyniki są nieznacznie lepsze na rzecz NPYLM, VotingExperts również zanotował lepsze rezultaty. Widać znaczną różnicę między precyzją a czułością dla NPYLM. Dla NPYLM, rosnąca liczba danych

Tabela 9: Porównanie wyników wykrywania anomalii różnych metod na zestawach danych BGL, Thunderbird.

Metoda	BGL			Thunderbird		
	Precyzja	Czułość	F-1 wynik	Precyzja	Czułość	F-1 wynik
LogGPT + SW	0.940±0.010	0.977±0.018	0.958±0.011	0.973±0.004	1.000±0.000	0.986±0.002
LogGPT + VE	**0.986**±0.001	**0.978**±0.006	**0.982**±0.001	**0.983**±0.000	**1.000**±0.000	**0.991**±0.000

do nauki poprawiała czułość, ale miała niewielki wpływ na precyzję. Dla VotingExperts, zwiększenie danych treningowych zmniejszało ogólny F-score. Ze średniej długości segmentu wynika, że NPYLM wprowadził mniejsze segmenty niż VotingExperts.

4.3 Wykrywanie anomalii

Tabela 9 przedstawia wyniki oraz odchylenia standardowe LogGPT i różnych metod segmentacji z 10 eksperymentów na zestawach danych BGL i Thunderbird. Pogrubienie w tabeli oznacza, że LogGPT znacznie przewyższa najlepszy model bazowy dla każdego zestawu danych na poziomie istotności 0.05, co zostało ustalone za pomocą testu t Studenta.

Zaobserwowałem, że na zestawach danych BGL i Thunderbird wyniki uzyskane przez LogGPT z VotingExperts były lepsze niż w przypadku modeli bazowych. Wyniki osiągnięte przez LogGPT i tak były już wysokie, nie pozostawiając wiele miejsca na poprawę. Mimo to na zestawie danych BGL, VotingExperts z oknem 37 i progiem 7 dał poprawę F-score o 2.4%. Dla Thunderbird najlepsze wyniki uzyskałem przy oknie 4 i progu 6 (Tabela 10) i oznaczało to poprawę F-score o 0.5%. W obu eksperymentach poprawiła się również czułość i precyzja. W kontekście anomalii, ważne jest by nie otrzymywać powiadomień o anomalii, gdy jej nie ma, a z drugiej strony, by anomalie nie pozostały niezauważone przez model. Dlatego, choć preferuje się zazwyczaj czułość nad precyzją (lepiej sprawdzić dodatkowe logi, zamiast nie zauważyć istotnego błędu), to przyrost w obu miarach zapewnia, że ważne anomalie nie zostaną pominięte i nie będzie nadmiarowych logów do sprawdzenia.

Tabela 10: F-score wykrywania anomalii dla algorytmu VotingExperts na różnych zbiorach.

Zbiór	**Okno**	**Próg**	**F-score**
BGL	37	7	0.982
Thunderbird	4	6	0.991

Wyniki uzyskane na zestawach danych BGL i Thunderbird pokazują, że zastosowanie segmentacji z algorytmu VE poprawia wydajność wybranych metod, w tym przypadku LogGPT, do wykrywania anomalii w logach open source.

Tabela 11: Porównanie wyników wykrywania anomalii różnych metod na zestawie danych Nokia.

Metoda	Precyzja	Czułość	F-1 wynik
LogGPT + SW	0.222	0.708	0.334
LogGPT + VE	0.288	0.961	0.443

W przypadku Nokii (Tabela 11), wykrywanie anomalii bazujące na nienadzorowanej segmentacji również dało lepszy rezultat niż w przypadku okna przesuwnego. F-score wyniósł 44.34% dla nienadzorowanej segmentacji w stosunku do 33.83% dla okna przesuwnego. Zwraca uwagę niski poziom obu wyników. Sugeruje to, że zadanie wykrywania anomalii jest dużo trudniejsze na zbiorze logów Nokii, którego przyczyny opiszę dokładniej w dyskusji. Mimo to segmentacja VE wpłynęła korzystnie na wynik.

4.4 Lokalizacja błędu na podstawie logów

Celem kolejnego badania było sprawdzenie czy ranking według kontekstu zmniejsza liczbę unikalnych szablonów logów do sprawdzenia. Tradycyjna metoda LBFL nadaje najwyższy poziom podejrzliwości wszystkim szablonom nie obecnym w logu z poprawnego wykonania. W sytuacji, gdy takich linii jest wiele, nie ma jasno określonego algorytmu postępowania. Intuicyjnym podejściem jest sortowanie linii logu według czasu i ich kolejne analizowanie, co jest zadaniem czasochłonnym. Moje rozwiązanie używa bardziej zaawansowanej metody i szereguje te linie po wskaźniku obliczonym na podstawie kontekstu. Działanie rozwiązania zostanie zaprezentowane na kolejnych przykładach. W celu lepszego opisania każdego przykładu zaczynam od liczby szablonów ze wskaźnikiem podejrzliwości 1 zwróconych przez LBFL. Następnie korzystając z kluczowego szablonu wskazanego przez eksperta pokazuję jakie wyniki dają kolejne metody szeregowania: po czasie i według kontekstu. W metodach opartych o kontekst wskazuję, który segment w szeregu zawierał kluczowy szablon i ile programista musiał przeanalizować podejrzanych szablonów, aby go napotkać.

W Przykładzie 1 tradycyjna metoda LBFL odszukała 73 szablony logów z wskaźnikiem podejrzliwości równym 1. Szeregując odpowiadające im linie po czasie najistotniejsza linia logu znajdowała się na pozycji 445 wymagając od programisty przeanalizowania 33 szablonów. Pierwsza z kontekstowych metod, szeregująca linie według kontekstu opartego na oknie przesuwnym (LBFL + SW) o wielkości 0.25 sekundy, umieściła najistotniejszy szablon w segmencie 102. Sumarycznie wymagało to analizy 50 podejrzanych szablonów zanim programista napotkał kluczowy szablon. W porównaniu do pierwszej metody liczba podejrzanych szablonów logów do analizy została zwiększona o 51.51% (Tabela 13). Druga metoda szeregująca według kontekstu z VE (LBFL + VE) umieściła najważniejszy szablon w segmencie 29. Sumarycznie wymagało to analizy 18 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Liczba szablonów logów do analizy została zmniejszona o 45.45% (Tabela 13).

Tabela 12: Liczba szablonów do sprawdzenia dla różnych metod w kolejnych przykładach.

Nr	LBFL + timestamp	LBFL + SW	LBFL + VE	LogGPT + SW	LogGPT + VE
1	33	50	**18**	93	95
2	454	148	93	219	**22**
3	22	8	15	24	**7**

Trzecia metoda kontekstowa, wykorzystująca model LogGPT oraz kontekst z okna przesuwnego (LogGPT + SW) o wielkości 0.25 sec umieściła najważniejszy szablon w segmencie na pozycji 411. Sumarycznie wymagało to analizy 93 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Metoda ta nie

zredukowała liczby szablonów do analizy w porównaniu z LBFL + timestamp. Wręcz przeciwnie wzrost wyniósł 181.82%.

Czwarta metoda kontekstowa, wykorzystująca model LogGPT oraz kontekst z algorytmu VE o parametrach 7 i 4 (LogGPT + VE) umieściła najważniejszy szablon w segmencie na pozycji 530. Sumarycznie wymagało to analizy 95 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Metoda ta również nie zredukowała liczby szablonów do analizy w porównaniu z LBFL + timestamp, a wzrost wyniósł 187.88%.

Widać zatem, że w Przykładzie 1 najlepiej sprawdziła się metoda LBFL + VE, która jako jedyna zredukowała liczbę szablonów do analizy. Natomiast zastosowanie bardziej zaawansowanych metod opartych na sieciach Transformer nie przyniosło oczekiwanego zysku.

Wielkość okna przesuwnego została zmieniona w Przykładzie 2 tak, aby najważniejsze błędy nie zostały odrzucone wskutek zbyt małej wielkości okna (ograniczenie LogGPT). Kluczowy szablon jest bowiem oddalony od najbliższego szablonu w wątku o więcej jak 1 sekundę. W Przykładzie 2 tradycyjna metoda LBFL odszukała 668 szablonów logów z wskaźnikiem podejrzliwości równym 1. Szeregując odpowiadające im linie po czasie najistotniejsza linia logu znajdowała się w linii 3576 wymagając od programisty przeanalizowania 454 szablonów. Pierwsza z kontekstowych metod, szeregująca linie według kontekstu opartego na oknie przesuwnym (LBFL + SW) o wielkości 1.5 sekundy, umieściła najistotniejszy szablon w segmencie 277. Sumarycznie wymagało to analizy 148 podejrzanych szablonów zanim programista napotkał kluczowy szablon. W porównaniu do pierwszej metody liczba podejrzanych szablonów logów do analizy została zmniejszona o 67.40% (Tabela 13). Druga metoda szeregująca według kontekstu z VE (LBFL + VE) umieściła najważniejszy szablon w segmencie 285. Sumarycznie wymagało to analizy 93 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Liczba szablonów logów do analizy została zmniejszona o 79.51% (Tabela 13).

Tabela 13: Redukcja liczby szablonów do sprawdzenia dla różnych metod (%) względem metody opartej o znaczniki czasu.

Nr	LBFL + SW (%)	LBFL + VE (%)	LogGPT + Sw (%)	LogGPT + VE (%)
1	-51.515151	**45.454545**	-181.818182	-187.878788
2	67.400881	79.515418	51.762114	**95.154185**
3	63.636363	31.818182	-9.090909	**68.181818**

Trzecia metoda kontekstowa, wykorzystująca model LogGPT oraz kontekst z okna przesuwnego (LogGPT + SW) o wielkości 1.5 sec umieściła najważniejszy szablon w segmencie na pozycji 493. Sumarycznie wymagało to analizy 219 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Metoda ta zredukowała liczbę szablonów do analizy w porównaniu z LBFL + timestamp o 51.76%.

Czwarta metoda kontekstowa, wykorzystująca model LogGPT oraz kontekst z algorytmu VE o parametrach 7 i 4 (LogGPT + VE) umieściła najważniejszy szablon w segmencie na pozycji 62. Sumarycznie wymagało to analizy 22 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Metoda ta również zredukowała liczbę szablonów do analizy w porównaniu z LBFL + timestamp. Redukcja wyniosła 95.15% i była to największa redukcja spośród wszystkich metod.

W tym przykładzie zastosowanie kombinacji LogGPT oraz algorytmu VE przyniosło największe efekty. Na drugim miejscu była druga metoda oparta na kontekście VE, LBFL +VE notując 79%. Kontekst otrzymany z tej metody po raz kolejny okazał się przydatny do osiągnięcia dobrych wyników.

W Przykładzie 3 tradycyjna metoda LBFL odszukała 86 szablonów logów z wskaźnikiem podejrzliwości równym 1. Szeregując odpowiadające im linie po czasie najistotniejsza linia logu znajdowała się w linii 47 wymagając przeanalizowania 22 szablonów. Pierwsza z kontekstowych metod, szeregująca linie według kontekstu opartego na oknie przesuwnym (LBFL + SW) o wielkości 0.25 sekundy, umieściła najistotniejszy szablon w segmencie 11. Sumarycznie wymagało to analizy 8 podejrzanych szablonów zanim programista napotkał kluczowy szablon. W porównaniu do pierwszej metody liczba podejrzanych szablonów logów do analizy została zmniejszona o 63.63% (Tabela 13). Druga metoda szeregująca według kontekstu z VE (LBFL + VE) umieściła najważniejszy szablon w segmencie 32. Sumarycznie wymagało to analizy 15 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Liczba szablonów logów do analizy została zmniejszona o 31.82% (Tabela 13).

Trzecia metoda kontekstowa, wykorzystująca model LogGPT oraz kontekst z okna przesuwnego (LogGPT + SW) o wielkości 0.25 sec umieściła najważniejszy szablon w segmencie na pozycji 21. Sumarycznie wymagało to analizy 24 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Metoda ta nie zredukowała liczby szablonów do analizy w porównaniu z LBFL + timestamp, a wzrost wyniósł 9.09%.

Czwarta metoda kontekstowa, wykorzystująca model LogGPT oraz kontekst z algorytmu VE o parametrach 7 i 4 (LogGPT + VE) umieściła najważniejszy szablon w segmencie na pozycji 13. Sumarycznie wymagało to analizy 7 podejrzanych szablonów zanim programista napotkał kluczowy szablon (Tabela 12). Metoda po raz kolejny zredukowała liczbę szablonów do analizy w porównaniu z LBFL + timestamp. Redukcja wyniosła 68.18% i była to znów największa redukcja spośród wszystkich metod.

Zauważalny jest pozytywny wpływ segmentów otrzymanych z algorytmu VE na wyniki zastosowanych metod lokalizacji. Metody te uzyskują najwyższe wyniki, często mocno dystansując metody oparte na znaczniku czasowym i kontekście z okna przesuwnego.

Drugim celem badań było sprawdzenie czy ranking według kontekstu zmniejsza liczbę linii logów do sprawdzenia. Liczba linii do sprawdzenia jest ściśle powiązana z liczbą szablonów. Programista w toku sprawdzania uszeregowanych podejrzanych linii ma łatwiejsze zadanie jeśli linie te są powiązane z mniejszą liczbą szablonów, gdyż widząc, że linie są powiązane z tym samym szablonem może pominąć od razu całą grupę. Nie zawsze jednak jest to możliwe, i dlatego liczba linii do zweryfikowanie jest ważną wartością służącą do oceny metody.

Tabela 14: Liczba linii logów do sprawdzenia dla różnych metod w kolejnych przykładach.

Nr	LBFL+timestamp	LBFL+SW	LBFL+VE	LogGPT+Sw	LogGPT+VE
1	455	711	**35**	2762	691
2	3576	1217	320	1680	**103**
3	47	**14**	34	84	15

W Przykładzie 1, po zastosowaniu szeregowania po czasie do najbardziej podejrza-

nych linii z LBFL (LBFl + timestamp), kluczowa linia była na pozycji 455 (Tabela 14). Pierwsza z kontekstowych metod, szeregująca linie według kontekstu opartego na oknie przesuwnym (LBFL + SW) o wielkości 0.25 sekundy, umieściła najistotniejszą linię na pozycji 711 (Tabela 14). W porównaniu do LBFL + timestamp liczba podejrzanych szablonów logów do analizy została zwiększona o 56.26% (Tabela 15). Druga metoda szeregująca według kontekstu z VE (LBFL + VE) umieściła najważniejszą linię na pozycji 35 (Tabela 14). Oznaczało to zmniejszenie liczby linii do analizy o 92.31% (Tabela 15). Trzecia metoda kontekstowa LogGPT + SW zwiększyła liczbę linii do analizy do 2762, co stanowi wzrost o 507.03% (Tabela 15). Czwarta metoda LogGPT + VE również zwiększyła liczbę linii do analizy do 691 (Tabela 14), co oznacza wzrost o 51.87% (Tabela 15).

Tabela 15: Redukcja linii logów do sprawdzenia dla różnych metod (%) względem metody opartej o znaczniki czasu.

Nr	LBFL + SW (%)	LBFL + VE (%)	LogGPT + SW (%)	LogGPT + VE (%)
1	56.26	**92.31**	-507.03	-51.87
2	65.97	91.05	53.02	**97.12**
3	**70.21**	27.66	-78.72	68.09

W Przykładzie 2, po zastosowaniu szeregowania po czasie do najbardziej podejrzanych linii z LBFL (LBFl + timestamp), kluczowa linia była na pozycji 3576 (Tabela 14). Pierwsza z kontekstowych metod, szeregująca linie według kontekstu opartego na oknie przesuwnym (LBFL + SW) o wielkości 1.5 sekundy, umieściła najistotniejszą linię na pozycji 1217 (Tabela 14). W porównaniu do LBFL + timestamp liczba podejrzanych szablonów logów do analizy została zmniejszona o 65.96% (Tabela 15). Druga metoda szeregująca według kontekstu z VE (LBFL + VE) umieściła najważniejszą linię na pozycji 320 (Tabela 14). Oznaczało to zmniejszenie liczby linii do analizy o 91.05% (Tabela 15). Trzecia metoda kontekstowa LogGPT + SW zmniejszyła liczbę linii do analizy do 1680, co stanowi redukcję o 53.02% (Tabela 15). Czwarta metoda LogGPT + VE również zmniejszyła liczbę linii do analizy do 103 (Tabela 14), co oznacza redukcję o 97.12% (Tabela 15).

W Przykładzie 3, po zastosowaniu szeregowania po czasie do najbardziej podejrzanych linii z LBFL (LBFl + timestamp), kluczowa linia była na pozycji 47 (Tabela 14). Pierwsza z kontekstowych metod, szeregująca linie według kontekstu opartego na oknie przesuwnym (LBFL + SW) o wielkości 0.25 sekundy, umieściła najistotniejszą linię na pozycji 14 (Tabela 14). W porównaniu do LBFL + timestamp liczba podejrzanych szablonów logów do analizy została zmniejszona o 70.21% (Tabela 15). Druga metoda szeregująca według kontekstu z VE (LBFL + VE) umieściła najważniejszą linię na pozycji 34 (Tabela 14). Oznaczało to zmniejszenie liczby linii do analizy o 27.65% (Tabela 15). Trzecia metoda kontekstowa LogGPT + SW zwiększyła liczbę linii do analizy do 84, co stanowi wzrost o 78.72% (Tabela 15). Natomiast czwarta metoda LogGPT + VE zmniejszyła liczbę linii do analizy do 15 (Tabela 14), co oznacza redukcję o 68.08% (Tabela 15).

W tym przykładzie wyjątkowo metoda oparta o okno przesuwne (LBFL + SW) zanotowała największą redukcję, jest to jednak odosobniony przykład dotyczący najkrótszego logu oraz najmniejszej ilości danych historycznych. Drugi najlepszy wynik należał do metody LogGPT + VE, która po raz kolejny osiąga jeden z najlepszych wyników.

Rozdział 5
Dyskusja

5.1 Klasyfikacja sekwencji logów z testów

Wysokie wyniki klasyfikacji sekwencji logów nazwą zestawu testów zwracają uwagę na dwie kwestie: sekwencje logów opisujące daną funkcjonalność niosą ze sobą wystarczającą informację by je rozróżniać, ale muszą być precyzyjnie segmentowane. Takie precyzyjne wyodrębnienie z logu produkcyjnego jest trudnym zadaniem. Zastosowanie okna przesuwnego do wyodrębnienia tych sekwencji dało dosyć dobre rezultaty (predykcje z prawdopodobieństwo modelu powyżej 0.5) jednak wymagało nakładu pracy w celu wybrania odpowiedniej długości okna, oznacza to, że każde logi wymagają zaangażowania eksperta w celu dostrojenia wielkości okna przesuwnego. Rezultatem działania modelu na logach produkcyjnych była sekwencja nazw zestawu testów odpowiadających za funkcjonalność w danym fragmencie logów produkcyjnych.

5.2 Segmentacja sekwencji logów

Przygotowałem złoty standard dla segmentacji sekwencji logów za pomocą półautomatycznego algorytmu. Najpierw użyłem algorytmu VotingExperts z hiperparametrami uzyskanymi z oryginalnego badania [15]. Następnie, korzystając ze swojej najlepszej wiedzy, skorygowałem segmenty, które uznałem za nieprawidłowe. Problem ten przypomina segmentację słów w języku chińskim, gdzie dla różnych wytycznych segmentacji różne algorytmy uzyskują najlepsze wyniki [74]. Na chwilę obecną opracowana przeze mnie złota segmentacja jest jedyną, która jest publicznie dostępna, może być wykorzystana w badaniach i do wiarygodnego porównania różnych metod.

VotingExperts jest algorytmem o niskiej złożoności obliczeniowej - liniowej w funkcji liczby linii w logu. Jednakże, nie posiada on mechanizmu wygładzania (ang. *smoothing*), co czyni go podatnym na niewidziane wcześniej n-gramy. Z tego powodu nie mogłem dokonać segmentacji logów Nokii przy użyciu VotingExperts, mimo treningu na stosunkowo dużym zbiorze treningowym. NPYLM nie osiągnął wysokiego F-score na logach CloudStack i Nokii; jednak im większy zbiór danych treningowych, tym lepsza była wartość czułości. W eksperymencie z logami Nokii, przy największym zbiorze treningowym, wartość czułości dla NPYLM była lepsza niż dla VotingExperts. Przy niskiej precyzji na logach Nokii, NPYLM podzielił log na więcej segmentów, niż oczekiwał ekspert. Potwierdza to średnia długość segmentów, która wynosiła 3, podczas gdy dla VotingExperts było to 8. W przypadku logów CloudStack, NPYLM osiągnął lepszą precyzję niż VotingExperts, ale miał bardzo niską czułość. To pokazuje, że segmenty proponowane przez NPYLM są bardzo dobre, ale pomijają wiele z proponowanych

złotych segmentów.

Przeprowadzone badania pozwoliły również zaobserwować zależność pomiędzy entropią a wynikami segmentacji. Zależność ta wskazuje, że niższa entropia jest związana z lepszymi wynikami segmentacji. Jest to intuicyjne, ponieważ można oczekiwać, że sekwencje z bogatym alfabetem i znaczną niepewnością będą stanowić wyzwanie dla algorytmu segmentacji. W przypadku logów CloudStack, duża część sekwencji posiadała prosty powtarzalny wzorzec związany z częstymi logami typu żądanie-odpowiedź. Miało to niewątpliwie wpływ na wyższy wynik segmentacji.

5.3 Wykrywanie anomalii

Zastosowanie nienadzorowanej segmentacji do wykrywania anomalii w logach wpłynęło na znaczącą poprawę wyników w porównaniu z poprzednimi metodami. Dla oceny wpływu segmentacji na poprawę wyniku LogGPT, zebrałem informację o poprawie wyników jakie wprowadził oryginalny LogGPT w stosunku do poprzednich metod. Na przykład LogGPT osiągnął wynik F-score 0.958 na zbiorze BGL, podczas gdy poprzednia najlepsza metoda, LogBERT, uzyskała 0.905, co oznacza przyrost o 5.86%. W przypadku Thunderbird, LogGPT osiągnął wynik 0.986, a poprzedni najlepszy wynik wynosił 0.966. Daje to przyrost o 2.07%. W przypadku LogGPT z nienadzorowaną segmentacją (LogGPT+ULS), wyniki 0.982 na zbiorze BGL oraz 0.991 na zbiorze Thunderbird dały odpowiednio zysk 2.82% oraz 0.51% w stosunku do wyników uzyskanych przez LogGPT z oknem przesuwnym (Tabela 16).

W przypadku Nokii zwraca uwagę niski wynik osiągnięty bez względu na metodę segmentacji (Tabela 17). W Tabeli tej dla okna przesuwnego nie ma wpisanego zysku, gdyż jest to metoda referencyjna. Warto zauważyć, że liczba szablonów i różnorodność sekwencji logów Nokii jest dużo większa niż w przypadku logów BGL i Thunderbird i sięga 25 tys szablonów. Powoduje to, że proces uczenia LogGPT jest dużo dłuższy. By zdiagnozować przyczynę słabszego zachowania LogGPT przeprowadziłem szczegółową analizę zbioru danych. W toku eksperymentów odkryłem, że zbiór przygotowany w sposób opisany wcześniej nie może posłużyć do prawidłowego nauczenia modelu poprawnego wzorca. Sposób przygotowania danych pozwala tylko na stwierdzenie co na pewno jest błędnym wzorcem, natomiast nie pozwala na określenie co z całą pewnością jest wzorcem poprawnym. Przez logi poprawne w tym zbiorze, rozumiem wszystkie logi, które nie znalazły się w analizie i opisie błędu. Jest to jednak informacja niekompletna, gdyż z natury rzeczy logi wymienione w opisie błędu są tylko podzbiorem wszystkich linii logu (które mogłyby się pojawić w pliku logu) wskazujących na błąd. Dodatkowo utrudnieniem jest fakt, że czasami opis błędu zawiera fałszywie pozytywne fragmenty logów, czyli logi, które pojawiają się zarówno w poprawnych jak i niepoprawnych wykonaniach. Powoduje to, że zbioru logów Nokii nie mogłem użyć do wytrenowania wzorca poprawnego wykonania.

5.4 Lokalizacja błędu na podstawie logów

Celem badań nad lokalizacją z użyciem kontekstu było opracowanie metody dokładniej szeregującej podejrzane linie logów, niż tradycyjna metoda LBFL zwracająca wiele równie podejrzanych linii logów. Do oceny użyłem kluczowych logów oznaczonych przez ekspertów, które ujawniają rzeczywistą przyczynę problemu. Porównywałem liczbę linii

Tabela 16: Porównanie wyników wykrywania anomalii metodami LogGPT i LogGPT+ULS z wynikami wcześniej publikowanymi w literaturze na zbiorach BGL i Thunderbird.

Zbiór	Metoda	Wynik	Poprzednia metoda	Poprzedni wynik	Zysk procentowy (%)
BGL	LogGPT	0.958	LogBERT	0.905	5.86
Thndrbrd	LogGPT	0.986	PCA	0.966	2.07
BGL	LogGPT+ULS	0.982	LogGPT	0.955	2.82
Thndrbrd	LogGPT+ULS	0.991	LogGPT	0.986	0.51
BGL	LogGPT+ULS	0.982	LogGPT	0.955	2.82
Thndrbrd	LogGPT+ULS	0.991	LogGPT	0.986	0.51

Tabela 17: Porównanie wyników wykrywania anomalii metodami nienadzorowanej segmentacji oraz okna przesuwnego na danych Nokii.

Metoda	F-score (%)	Zysk procentowy (%)
Nienadzorowana segmentacja	44.34	+10.51
Okno przesuwne	33.83	-

i liczbę szablonów potrzebną do analizy zanim osiągnięta zostanie pozycja z kluczowym logiem. Aby zmierzyć zysk otrzymany przez daną metodę, używałem procentowego wskaźnika redukcji danych do analizy, obliczonego według równania:

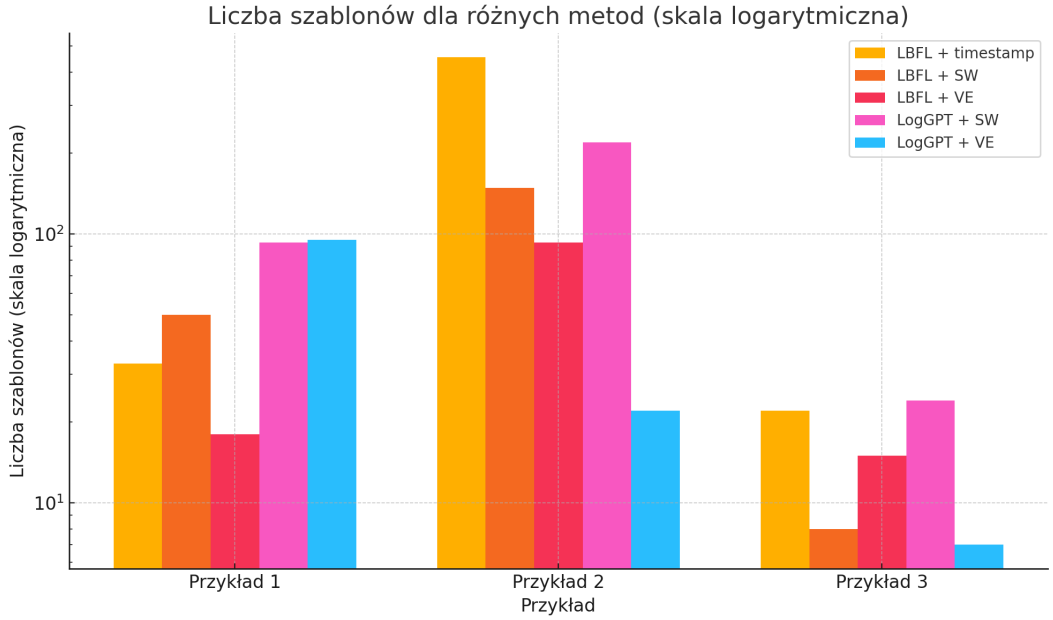
$$Red = \left(\frac{X_1 - X_2}{X_1}\right) \times 100 \tag{5.1}$$

gdzie X_1 reprezentuje wartość bazową, w tym przypadku liczbę szablonów i linii logów pochodzącą z metody LBFL + timestamp, a X_2 reprezentuje uzyskaną wartość przez nowe metody LBFL + SW, LBFL + VE, LogGPT + SW, oraz LogGPT + VE.

Analiza wyników porównania metod LBFL + SW, LBFL + VE, LogGPT + SW, oraz LogGPT + VE pozwala na wyciągnięcie kilku kluczowych wniosków dotyczących liczby szablonów i liczby linii logów do analizy i procentowej ich redukcji. Każda z metod ma inne mocne i słabe strony, które wpływają na jej efektywność w różnych przypadkach.

5.4.1 Liczba szablonów do analizy

Metoda **LBFL + VE** dostarcza spójne i stosunkowo niskie wyniki dla liczby szablonów. W **Przykładzie 1** liczba szablonów wyniosła 18, co jest znaczną redukcją w porównaniu do wartości bazowej 33 szablonów. W **Przykładzie 2** liczba szablonów wynosiła 93, co jest również dużo mniejsze niż wartość bazowa (454) (Rysunek 12), ale jest to znacznie więcej niż dla metody LogGPT+VE (22). W **Przykładzie 3** LBFL + VE dostarczył 15 szablonów do analizy, co jest mniejsze od bazowej wartości 22, ale i tu LogGPT+VE okazał się lepszy (7). Gdy dla metody LBFL użyjemy kontekstu pochodzącego z okna przesuwnego otrzymujemy znacząco gorsze wyniki. I tak, dla **Przykładu 1**, LBFL + SW zwraca 50 szablonów, dla **Przykładu 2** jest to 148. Jedynie dla **Przykładu 3** zwraca mniejszą liczbę szablonów logów do analizy, 8, i jest to jeden z najlepszych wyników spośród wszystkich metod. Lepszy jest tylko **LogGPT + VE**, zwracając 7 szablonów. LBFl + VE jest metodą, która wydaje się być najbardziej obiecująca. Poza pierwszym przykładem, gdzie liczba szablonów jest znacząco większa od bazowej metody (95), w Przykładzie 2 i 3 metoda ta notuje najlepsze wyniki, zdecydowanie wyróżniając się na tle pozostałych metod (22 - Przykład 2, i wspomniane



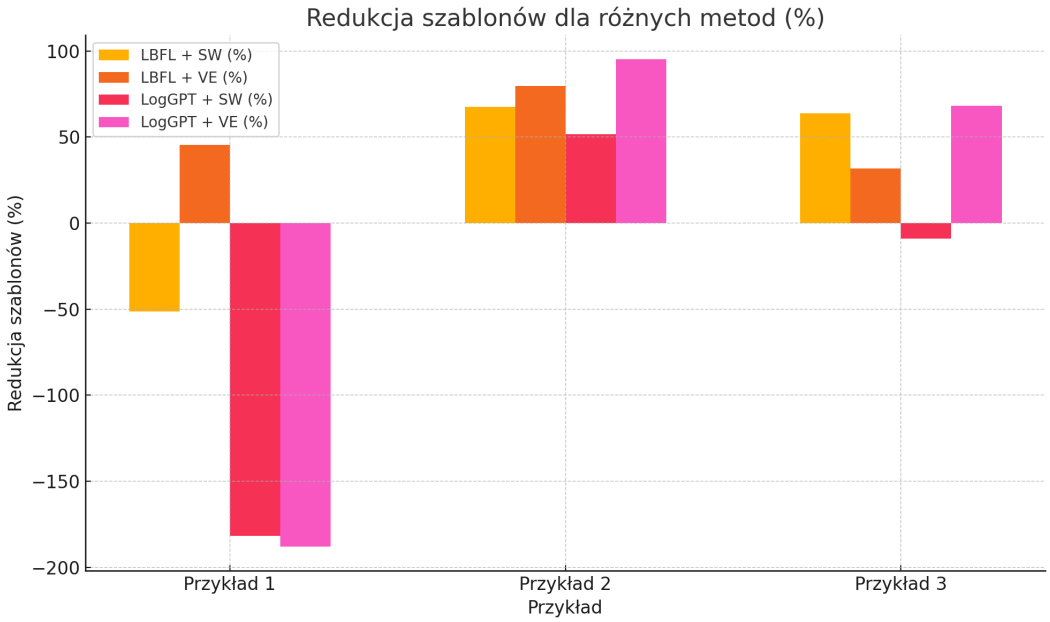
Rysunek 12: Porównanie liczby szablonów logów dla LBFL + (SW/VE) i LogGPT +(SW/VE).

już 7 - Przykład 3). Przykład 1 jest dowodem na to jak stosowanie metod parsowania opartych na wyrażeniach regularnych może wprowadzić fałszywie pozytywne wyniki. W tym przypadku w logach błędnych pojawiły dwie kluczowe zmiany względem logów z poprawnych wykonań, z punktu widzenia wyrażeń regularnych metody Drain. Zmiany te choć nie zmieniają semantycznie wydźwięku dotkniętych linii logów, dla metody Drain oznaczają wprowadzenie nowych szablonów. Konsekwencją jest duża liczba nieznanych wcześniej sekwencji (np. zamiast 12, pojawia się 2034, gdzie różnica między 12 a 2034 wynika z niewielkiej zmiany formatu, nieistotnej dla człowieka, ale istotnej dla algorytmu Drain - Rozdział 5.4.5). Z kolei **LogGPT + SW** charakteryzuje się większą liczbą szablonów w porównaniu do kontekstowych wersji LBFL oraz w stosunku do siostrzanej metody używającej precyzyjnej segmentacji. W **Przykładzie 1** metoda ta dostarczyła 93 szablony do analizy, co oznacza znaczny wzrost w porównaniu do wartości bazowej 33 szablonów. W **Przykładzie 2** liczba szablonów wyniosła 219, i tylko w tym przykładzie jest to wartość niższa niż bazowe 454 szablonów. W **Przykładzie 3** liczba szablonów wyniosła 24, i znów jest to gorszy wynik niż bazowe 22 szablony.

5.4.2 Procentowa redukcja szablonów

LBFL + VE osiągnął bardzo dobrą redukcję liczby szablonów w **Przykładzie 1** i **Przykładzie 2**, gdzie redukcja wyniosła odpowiednio 45.45% i 79.51% (Rysunek 13). Oznacza to, że metoda ta skutecznie zmniejsza liczbę szablonów, co czyni ją stabilnym wyborem. W **Przykładzie 3** redukcja wyniosła 31.82%, co nadal jest dobrym wynikiem, choć niższym niż w pozostałych przypadkach.

LBFL + SW osiągnął dużo gorsze wyniki w dwóch pierwszych przykładach w porównaniu do LBFL + VE, najpierw zwiększając liczbę szablonów o 51% (w stosunku do zmniejszenia o 45%), a następnie zmniejszając o 67% (w stosunku do 79%). W Przykładzie 3 widoczna jest dość zaskakująca jedna z wyższych redukcji 63.63% na tle wyników innych metod w tym przykładzie.



Rysunek 13: Porównanie redukcji szablonów logów dla LBFL + (SW/VE) i LogGPT + (SW/VE) względem meotdy opartej na znacznikach czasowych.

LogGPT + SW jest mniej efektywny w zmniejszeniu liczby szablonów. W **Przykładzie 1** i **Przykładzie 3** nastąpił wzrost liczby szablonów o odpowiednio 181.82% i 9.09%. Jedynie w **Przykładzie 2** redukcja liczby szablonów była na poziomie 51.76%. W ogólnym ujęciu ta metoda jest mniej skuteczna niż poprzednie.

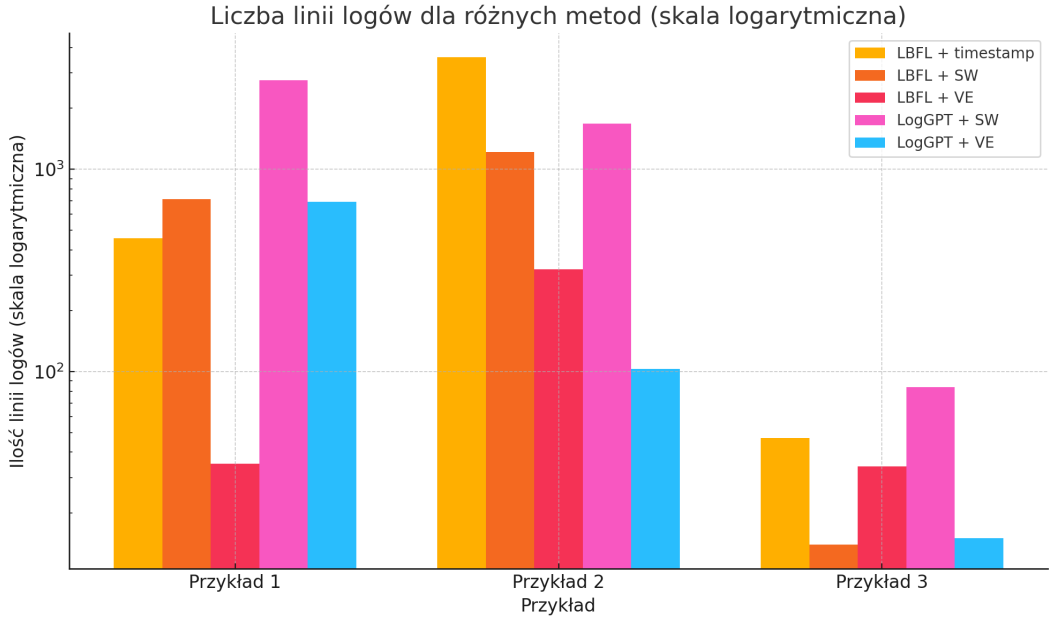
Najlepsze wyniki w zakresie redukcji liczby szablonów osiągnęła metoda **LogGPT + VE**, zwłaszcza w **Przykładzie 2**, gdzie redukcja wyniosła aż 95.15%. W **Przykładzie 3** redukcja również była bardzo wysoka, wynosząc 68.18%. Jednakże w **Przykładzie 1** metoda ta wykazała wzrost liczby szablonów o 187.88%, co jest związane z ograniczeniami metody wyznaczającej szablony, Drain (Rysunek 13), a nie z samą metodą opartą na LogGPT.

5.4.3 Liczba linii logów do analizy

Metoda **LBFL + VE** konsekwentnie dostarcza mniejszą liczbę linii do analizy w porównaniu z modelami LBFL + timestamp i LogGPT + SW. W Przykładzie 1 liczba linii wynosiła zaledwie 35, co jest znacząco mniejszą liczbą w porównaniu do 2762 linii generowanych przez **LogGPT + SW** oraz 691 linii dla **LogGPT + VE**. Również w Przykładzie 2, LBFL + VE oferował znacznie mniejszą liczbę linii (32) niż LogGPT + SW (1680) i LogGPT + VE (103). Podobnie w Przykładzie 3, LBFL + VE dał jedynie 34 linie do analizy, co jest gorsze niż 15 linii dla LogGPT + VE, ale znacznie lepsze od 84 linii w LogGPT + SW (Rysunek 14).

LogGPT + SW, w każdym z przypadków, generuje znacznie więcej linii do analizy, co wpływa negatywnie na efektywność tej metody. Wskazuje to, że segmentacja z oknem przesuwnym w tej metodzie utrudnia wyliczenia miary i zwiększa pracę potrzebną do analizy.

Z kolei **LogGPT + VE** zachowuje się lepiej niż LogGPT + SW, oferując mniejszą liczbę linii do analizy. Choć w Przykładzie 1 wynik tej metody nie jest tak imponujący (691 linii), to w Przykładzie 2 (103 linie) i Przykładzie 3 (15 linii) ta metoda oferuje



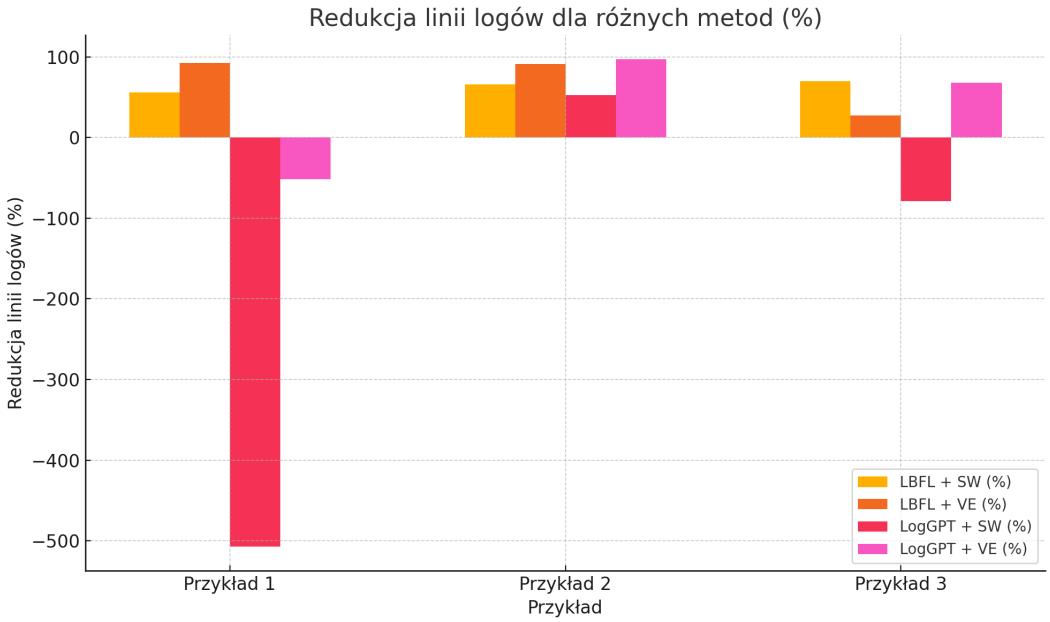
Rysunek 14: Porównanie liczby linii logów dla LBFL + (SW/VE) i LogGPT + (SW/VE).

znacznie bardziej efektywne wyniki niż inne podejścia (Rysunek 14).

5.4.4 Procentowa redukcja liczby linii logów

Jeśli chodzi o procentową redukcję liczby linii logów względem metody opartej na znacznikach czasowych, **LBFL + VE** osiąga bardzo wysoką efektywność w Przykładzie 1 i Przykładzie 2, gdzie redukcja wynosi odpowiednio 92.31% i 91.05% (Rysunek 15). W Przykładzie 3 redukcja wyniosła 27.65%, co nadal jest wynikiem godnym uwagi, jednak nieco niższym w porównaniu do innych metod. Mimo to, LBFL + VE zapewnia spójne i przewidywalne wyniki redukcji liczby linii logów, co czyni go solidnym rozwiązaniem w analizie logów. Odpowiednik oparty o okno przesuwne miał gorsze wyniki dla Przykładu 1 i 2, a lepszy dla przykładu 3. **Potwierdza to osiągnięciu celu C3, jakim było opracowanie algorytmu lokalizacji błędów działającego w czasie rzeczywistym bazującego na kontekście otrzymanym z segmentacji.**

LogGPT + SW osiąga wyjątkowo słabe wyniki pod względem redukcji liczby linii logów w stosunku do **LogGPT + VE**. W Przykładzie 1 oraz Przykładzie 3 liczba linii wzrosła o odpowiednio 507.03% i 78.72%. Tylko w Przykładzie 2 osiągnięto umiarkowaną redukcję na poziomie 53.02%, jednak to wciąż znacznie mniej niż w przypadku innych metod. **LogGPT + VE** natomiast prezentuje dobre wyniki w Przykładzie 2 i Przykładzie 3, osiągając redukcję odpowiednio 97.12% i 68.09%. Te wyniki sugerują, że oparcie się o segmenty pochodzące z metody VE pozytywnie wpłynęło na zmniejszenie liczby linii do analizy w tych przypadkach **co potwierdza osiągnięcie celu C3, jakim było opracowanie algorytmu lokalizacji błędów działającego w czasie rzeczywistym bazującego na kontekście otrzymanym z segmentacji**. Jednakże, w Przykładzie 1, metoda ta generowała wzrost liczby linii o 51.87%, co wskazuje na zależność tej techniki od metody parsowania logów (Rozdział 5.4.5).



Rysunek 15: Porównanie redukcji linii logów dla LBFL + (SW/VE) i LogGPT + (SW/VE) względem bazowej metody opartej na znacznikach czasowych.

5.4.5 Zależność od jakości szablonów

W Przykładzie 1 obie metody oparte o LogGPT zanotowały drastyczne pogorszenie wyników. Po dokładniejszym sprawdzeniu okazało się, że przyczyną jest sposób tworzenia szablonów przez Drain. Sama istota działania tego algorytmu powoduje, że taka sytuacja jest trudna do wykluczenia. We wspomnianym przykładzie w logach z błędem w znanych liniach logu pojawiło się nowe ich zakończenie. W przypadku znanego z treningu szablonu:

Message received from :addr: - service = :service_name:, flag = :value:

Pojawiło się nowe zakończenie *suspended = :value:*, co spowodowało dodanie nowego szablonu:

Message received from :addr: - service = :service_name:, flag = :value:, suspended = :value:

Należy ocenić, że algorytm Drain zadziałał poprawnie. Słowo *suspended* jest nową składową szablonu, po którym następuje nowa część zmienna. Problem leży w samej idei Drain, który nie zachowuje znaczenia semantycznego. I tak, nowy identyfikator szablonu nie przekazuje do metody LogGPT informacji o podobieństwie do starego szablonu. Stąd też LogGPT zwraca informację o błędzie. Wskazuje to na rosnącą potrzebę metod opartych na semantyce logów, a jednocześnie zachowujących potencjał korzystania z precyzyjnego kontekstu i sieci Transformer. Nie jest bowiem trywialnym zadaniem segmentacja sekwencji logów bez zamieniania ich na identyfikatory szablonów, ani nie jest możliwe korzystanie z sieci Transformer nie bazując na zamkniętym słowniku. Pytanie o odpowiednik zaproponowanej metody niezależny od Drain, ale wykorzystujący inne rozwiązanie niż Drain i pozbawiony jego wad, pozostaje zatem otwarte.

5.4.6 Podsumowanie

Przeprowadzone prace badawcze pokazały, że algorytm **LogGPT + VE** jest najbardziej obiecującą metodą w zakresie redukcji liczby linii logów do analizy. Z drugiej strony, **LBFL + VE** oferuje bardziej stabilne i przewidywalne wyniki redukcji, co czyni go solidnym wyborem w analizie logów, mniej zależnym od jakości wyrażeń regularnych i algorytmu Drain. Metody oparte na oknie przesuwnym nie są rekomendowane, ponieważ w większości przypadków liczba szablonów wzrastała. Przekłada się to na liczbę linii logów do analizy, z jednym drobnym odstępstwem, gdy LogGPT + VE zwraca więcej linii do analizy (15) mimo mniejszej liczby podejrzanych szablonów (7) w stosunku do LBFL + SW (8 - 14).

W związku z tym, choć **LogGPT + VE** może być bardziej efektywny w wybranych przypadkach, **LBFL + VE** wydaje się być najlepszym, stabilnym rozwiązaniem dla większości analiz jako mniej zależny od metody parsowania.

Rozdział 6
Wnioski

Automatyczne metody analizy logów są w obecnym momencie rozwoju ściśle zależne od metod parsowania logów. Przygotowanie algorytmu Drain wymagało wiele wysiłku i wielokrotnego testowania. Fazie tej poświęciłem wiele czasu gdyż wiadomo, że szum w parsowaniu logów ma wpływ na końcowe zadania [26]. Przygotowanie wyrażeń regularnych nie wymaga wiedzy eksperckiej na temat systemu, co jest niewątpliwą przewagą nad wyrażeniami regularnymi stosowanymi zazwyczaj przez doświadczonych programistów, a służących do rozpoznania typowych błędów. Wiedza na temat wyrażeń regularnych jest dużo łatwiejsza do przeniesienia do nowego zestawu logów niż wiedza, jakich dokładnie wyrażeń szukać w konkretnym systemie. Z drugiej strony, powstaje zestaw reguł, który należy utrzymywać i aktualizować w momencie pojawienia się nowych problematycznych linii logów. Potrzeba zatem nowych metod do parsowania logów, w pełni nienadzorowanych, aby umożliwić łatwiejszą automatyczną analizę logów, która mogłaby być skutecznie stosowana dla dużych systemów.

6.1 Klasyfikacja sekwencji logów z testów

Zapewnienie niezawodności dużych systemów oprogramowania jest procesem iteracyjnym. Pewne awarie są nieuniknione, dlatego szybki czas reakcji i dostarczenie poprawek poprawiają niezawodność. Metoda klasyfikacji fragmentów logów nazwami zestawów testów pomaga odtworzyć scenariusz z logu produkcyjnego, co z kolei pomaga programiście zrozumieć, co się wydarzyło, a tym samym szybciej znaleźć przyczynę błędu. Dostarcza cennych wskazówek dotyczących segmentów zdarzeń w logu na podstawie wiedzy eksperckiej wyciągniętej z testów jednostkowych. Jest również łatwo dostosowywany do wymaganej wersji oprogramowania, co wynika z zastosowanie testów jednostkowych tej wersji, w której wykryto błąd. Proces parsowania i uczenia się może być przeprowadzony automatycznie i dodany jako element zadania ciągłej integracji (ang. *continous integration*). Rozwiązanie nie wymaga również przechowywania dużej liczby danych historycznych.

Moje badania są podstawą do ulepszeń w dziedzinie samych testów. Zestawy testów powinny być bardziej szczegółowe, z sensownymi nazwami i wieloma przypadkami testowymi. Uważam, że świadomość wykorzystania testów jednostkowych do analizy prawdziwych błędów pomoże w motywowaniu programistów do pisania wysokiej jakości testów jednostkowych. Nie chodzi tylko o pokrycie kodu, ale także o zrozumienie jego zachowania i dokładniejsze oddzielanie funkcjonalności.

6.2 Segmentacja sekwencji logów

Analizowanie fragmentów logów produkcyjnych otrzymanych za pomocą okna przesuwnego ma swoje wady. Po pierwsze, tak otrzymane fragmenty kodu mogą łączyć w sobie wiele funkcjonalności, po drugie, są trudniejsze do analizy przez człowieka. Porównałem zatem nienadzorowane probabilistyczne metody segmentacji sekwencji logów. Użyłem dwóch metod: VotingExperts, stosowanej do segmentacji tekstu i logów, oraz NPYLM, metody segmentacji języka naturalnego. Dostosowałem NPYLM do dziedziny segmentacji logów. Pokazałem, że VotingExperts osiągnął najlepsze wyniki pod względem F-score w segmentacji logów (97% i 43%). Zwiększenie rozmiaru zbioru treningowego nieznacznie pogorszyło wyniki dla VotingExperts, a dla NPYLM wprowadziło istotną poprawę czułości (Tabela 18).

Drugim wnioskiem jest to, że segmentacja logów nie jest jednorodnym problemem, ale różni się znacząco między platformami. W przypadku CloudStack, duża liczba krótkich segmentów w złotym standardzie (o długościach 1 i 2) z niską entropią pozwoliła metodom segmentacji osiągnąć F-score powyżej 97%. Złoty standard Nokii zawiera dłuższe segmenty i wysoką entropię. W konsekwencji segmentacja jest znacznie bardziej wymagająca, a wyniki są gorsze. **Powyższe wyniki oznaczają osiągnięcie celu C1 polegającego na rozszerzeniu koncepcji sekwencji logów jako języka naturalnego oraz zastosowaniu metod nienadzorowanej segmentacji słów do segmentowania sekwencji logów.**

Wyniki osiągane przez metody segmentacji pokazały również zależność od preferencji eksperta systemu dotyczącymi segmentacji. To ekspert decyduje jak bardzo rozdrobnione segmenty preferuje. Łączy się to z obserwacją, że NPYLM wprowadził dużo więcej krótkich segmentów niż VE. W przypadku logów Nokii było to zachowanie nieporządne przez eksperta, i tam metoda NPYLM była mniej przydatna niż VE. Pokazuje to również, że złoty standard nie musi wyznaczać górnej granicy przydatności danej metody. Wykrywanie anomalii, czy lokalizacja błędu są pod tym względem bardziej obiektywnymi metodami oceny algorytmów segmentacji. Robią to w sposób pośredni, ale jednocześnie bardziej obiektywny. Jeśli segmenty dostarczone przez algorytm pomogą w wykryciu anomalii, to znaczy, że metoda segmentacji okazała się przydatna. Rozumowanie to stanowiło ważną część motywacji do rozpoczęcia badań nad metodami wykrywania anomalii i lokalizacji błędu. Konwencjonalne metody detekcji skupiają się głównie na analizie na poziomie wątków lub technice okna przesuwnego. W wyniku przeprowadzonych badań pokazałem jednak, że metody oparte na segmentach mogą przewyższać te, które opierają się na oknach przesuwnych.

Tabela 18: F-score, Precyzja i Czułość w wydajności segmentacji dla zbiorów danych CloudStack, Nokia.

Zbiór danych	Algorytm	F-score (%)	Precyzja (%)	Czułość (%)	Entropia
CloudStack	VE	**97.3**	97.7	**96.8**	2.1
CloudStack	NPYLM	42.5	**98.9**	27.0	
Nokia	NPYLM exp3	23.6	16.1	44.2	10.22
Nokia	VE exp1	**44.1**	**40.7**	**48.1**	

6.3 Wykrywanie anomalii

Przystąpienie do wykrywania anomalii miało swoją podstawę w pozytywnych wynikach segmentacji w odniesieniu do złotego standardu. Głównym problemem wykrywania anomalii jest to, że bazują na oknie przesuwnym, które może przypadkowo podzielić linie logów związane z błędem na odrębne segmenty. To utrudnia modelom wykrywania anomalii identyfikację błędnych sekwencji. Aby ułatwić zrozumienie przyczyny błędu, metody segmentacji logów powinny utrzymywać powiązane linie logów razem w jednym segmencie. Często nie jest to prawdą dla najnowocześniejszych metod [97]. Wykazałem, że zastosowanie algorytmu VotingExperts, znacznie poprawia wydajność wykrywania anomalii w stosunku do okna przesuwnego, zwiększając miarę F o 3% i 1% odpowiednio dla zestawów danych BGL i Thunderbird. Test t Studenta potwierdził, że zyski we wskaźniku F były istotne statystycznie. Osiągnąłem zatem znaczące poprawy na obu zestawach danych poprzez wyodrębnienie bardziej precyzyjnej struktury logów. Pokazuje to, że segmenty, które utrzymują powiązane logi razem, pozwalają metodom wykrywania anomalii dokładniej uczyć się, a w konsekwencji dokładniej znajdować anomalie co stanowi oczekiwane zachowanie.

Proponowane podejście poprawia wyniki wykrywania anomalii przy użyciu dobrze znanej metody VotingExperts. **Potwierdza to osiągnięcie celu C2, jakim było wykorzystanie uzyskanych segmentów do poprawy wykrywania anomalii w logach oraz celu C4, jakim była eksperymentalna weryfikacja zaproponowanej metody.** Jestem przekonany, że podobnej lub większej poprawy można się spodziewać po zastosowaniu dokładniejszych metod nienadzorowanej segmentacji zaproponowanych niedawno w literaturze, takich jak SLM i MSLM. Jest to interesujący obszar do przyszłych badań, który jednak wymaga najpierw rozwiązania problemu ograniczeń pamięciowych tych metod. Weryfikacja zaproponowanych metod z inżynierami Nokia, choć przeprowadzona w małej skali, potwierdziła że opracowane rozwiązanie ułatwia i przyspiesza ich pracę. Badania te będą kontynuowane, aby otrzymać dokładne obliczenia o ile został przyspieszony czas rozwiązania błędu, badania te jednak wymagają większej ilości czasu.

Pozytywne opinie inżynierów, obiecujące ale wciąż niskie wartości miar dla logów Nokia powodują ze konieczne są dalsze prace. Przeprowadzone już badania pokazały, że zbiór logów oznaczonych na podstawie opisów jest niewystarczający do użycia w tych eksperymentach. Rozwiązaniem tego problemu było wykorzystanie logów z testów integracyjnych, są bowiem dostępne w dosyć dużej liczbie. Niestety uniemożliwia to zastosowanie tak wytrenowanego modelu do ogólnych problemów. Innym, bardziej ogólnym rozwiązaniem jest rezygnacja z modeli głębokich na rzecz modeli statystycznych, które nie wymagają dużej ilości danych, i mogłyby operować na przypadku jednego testu poprawnego i jednego błędnego.

6.4 Lokalizacja błędu na podstawie logów

Głównym celem było znalezienie lepszej metody lokalizacji błędu. Wcześniejsze badania pokazały, że nienadzorowana segmentacja logów poprawiła dokładność metod wykrywania anomalii. Jako podstawę wykorzystałem znaną metodą lokalizacji błędu bazującą na logach (LBFL). Ma ona poważne ograniczenie wynikające z faktu, że może zwracać wiele linii podejrzanych, które mogą być dalej szeregowane tylko według znacz-

nika czasu. Stworzyłem nowy algorytm rozwiązujący ten problem i umożliwiający dalsze szeregowanie wykorzystujący nową miarę o nazwie Ranking Kontekstowy. Rozwiązanie to bazuje na segmentach pochodzących z nienadzorowanej segmentacji sekwencji logów. Wykorzystuje ono intuicję, że najważniejsze błędy pojawiają się w znanym wcześniej kontekście, natomiast zgrupowane zmiany (wyglądające jak błędy) to najczęściej już konsekwencja wcześniejszego błędu, lub logi pochodzące z nowej funkcjonalności.

By rozwiązać problem działania w czasie rzeczywistym dostosowałem istniejącą metodę LogGPT do stworzenia hierarchii podejrzanych segmentów, dzięki możliwościom jakie mają modele generatywne. Metoda przyniosła pozytywne rezultaty. W przypadku, gdy LogGPT nie okazał się lepszy, winą były ograniczenia użytej metody parsowania logów. Wykazałem też, że skuteczność użycia LogGPT zależy od jakości segmentacji. Dla segmentacji z użyciem okna przesuwnego, LogGPT osiągnął dużo gorsze wyniki niż LBFL korzystający tylko ze znaczników czasowych. Jednakże korzystając z segmentów pochodzących z nienadzorowanej segmentacji algorytmem VE, LogGPT okazał się dużo lepszy niż metoda bazowa i metoda LBFL z kontekstem. Oznacza to, że segmentacja algorytmem VE daje podstawy do dokładniejszej lokalizacji błędu w przypadku logów z firmy Nokia. **Wyniki te również potwierdzają osiągnięcie celów C3 i C4, czyli opracowanie algorytmu lokalizacji błędów działającego w czasie rzeczywistym bazującego na kontekście otrzymanym z segmentacji oraz eksperymentalna weryfikacja zaproponowanego modelu.**

Metodę LBFL z użyciem kontekstu można uznać za bardziej stabilną, jednak LogGPT bazujący na kontekście z VE niesie ze sobą potencjał dużo większych zysków, jeśli zminimalizowana zostanie zbytnia wrażliwość na metodę parsowania logów, lub przy wykorzystaniu innej metody parsowania logów.

Rozdział 7
Wdrożenie

7.1 Zapotrzebowanie firmy

Jednym z kluczowych wyzwań, z którym boryka się firma, jest potrzeba porównania wykonań poprawnych z wykonaniami błędnymi. Jest to problem, który jest zgłaszany na wielu różnych poziomach organizacyjnych: od testerów, którzy napotykają na błąd i chcą przypisać go do początkowej grupy odpowiedzialnej za analizę, przez specjalistów odpowiedzialnych za wstępne poszukiwanie przyczyny, którzy przyjmują zgłoszenia o błędach i kierują je do odpowiednich zespołów, aż po inżynierów oprogramowania, których zadaniem jest odnalezienie i naprawa błędu. Logi z wielu poprawnych wykonań historycznych są często dostępne, zwłaszcza w przypadku scenariuszy dobrze znanych i wielokrotnie testowanych. Na pierwszy rzut oka mogłoby się wydawać, że liczba dostępnych historycznych wykonań powinna umożliwić szybką manualną analizę i diagnozę problemu. Jednakże, mimo dostępności logów, istnieje kilka poważnych problemów, które utrudniają manualne porównywanie tych zapisów. Po pierwsze, długość sekwencji logów bywa bardzo duża, co sprawia, że manualne przeglądanie i analiza są czasochłonne i podatne na błędy. Po drugie, tradycyjne podejście oparte na wyszukiwaniu po słowach kluczowych takich jak "fail" czy "error", często może prowadzić do fałszywie pozytywnych wskazań. Problem ten jest szczególnie dotkliwy, gdyż wiele wpisów w logach zawiera błędne komunikaty, które jednak nie oznaczają błędnego działania programu. Po trzecie, porównanie poszczególnych linii logów jest trudne, ponieważ zmienne części w szablonach logów uniemożliwiają bezpośrednie porównanie. Te zmienne elementy często maskują rzeczywiste problemy, czyniąc analizę manualną bardzo skomplikowaną.

Oczekiwano zatem, że powstanie rozwiązanie automatyczne, które pozwoli na porównanie wykonania poprawnego i błędnego, zredukuje liczbę logów do analizy, a także wskaże miejsca najbardziej podejrzane. Miało to być rozwiązanie ogólnodostępne oraz niezależne od komponentu z którego pochodzą logi oraz słów kluczowych, działające w czasie rzeczywistym lub bliskim do rzeczywistego.

7.2 Proces wdrożenia

7.2.1 Zakres wdrożenia

Zostały wdrożone dwa rozwiązania: jedno służące do segmentacji logów, oraz drugie do lokalizacji błędu. Zasięgiem swoim obejmować ono ma wszystkie komponenty radiowe. Posiadają one ten sam format logów, co ułatwia działanie programu. Obecnie działający pilot programu jest dostosowany do podzbioru wszystkich komponentów, jednak jego

rozszerzenie jest łatwe i wymaga tylko dostosowania do formatu logów nie dotyczy zmian w działaniu modelu. Rozwiązanie jest obecnie w formie pilotażu dla komponentów radiowych.

7.2.2 Przygotowanie wdrożenia

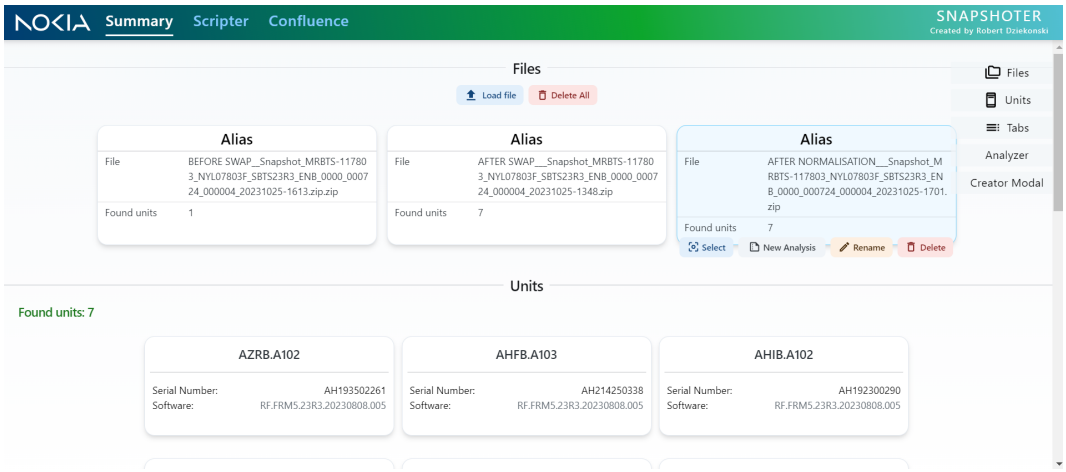
Aby uzyskać odpowiednie zasoby obliczeniowe został zarezerwowany serwer z 12 procesorami Intel Core CPU 2394.454 MHz, 8GB RAM, i 100GB pamięci dyskowej. Wybór technologii do implementacji rozwiązania sprowadzał się do wyboru między dwoma istniejącymi w firmie aplikacjami webowymi. Obie aplikacje miały na celu analizę logów ze skompresowanych paczek o charakterystycznej strukturze typowej dla Nokii. Pierwsza aplikacja, działająca w firmie od dłuższego czasu, oferowała usystematyzowany interfejs, do którego można było podłączyć własne rozwiązanie. Proces integracji był jednak bardzo sformalizowany, wymagający wyboru konkretnej technologii aplikacji klienckiej i podążania ściśle wytyczoną ścieżką, która często prowadziła do zakończenia przeglądu rozwiązania i integracji po kilku miesiącach. Drugim rozwiązaniem była nowo utworzona aplikacja (Rysunek 16), oferująca większą elastyczność w doborze technologii i niewymagająca tak ścisłego procesu przeglądu. Wybrałem drugie rozwiązanie, ponieważ jako osoba niezwiązana wcześniej z aplikacjami webowymi, nie wiedziałem, jakich technologii będę używał. W toku wdrożenia okazało się to trafnym wyborem, ponieważ mogłem skupić się na prostocie i funkcjonalności, wybierając technologię Flask, zamiast skupiać się na złożonych wymaganiach architektonicznych starszej aplikacji. Możliwość szybkiego wypuszczenia pierwszych wersji aplikacji pozwoliła mi również na uzyskanie szybkiej informacji zwrotnej od użytkowników. Nowy interfejs aplikacji był również lepiej oceniany przez użytkowników, co przyspieszyło proces akceptacji rozwiązania. Dzięki temu, mając do dyspozycji niewielki zespół (obecnie składający się z trzech osób, w tym mnie), została stworzona aplikacja, która spełniała oczekiwania użytkowników w wystarczającym stopniu.

7.2.3 Harmonogram wdrożenia

Po uzyskaniu zasobów i wyborze technologi przystąpiłem do realizacji wdrożenia. Z pomocą wcześniej wyszkolonego zespołu 4 osób (a od czerwca 2024 dwóch) realizowałem następujące kroki:

- Wybór technologii komunikacji z serwerem rozpakowującym logi (wrzesień 2023).
- Wybór i implementacja sposobu dostarczenia wyników analizy do użytkownika (październik 2024).
- Wdrożenie segmentacji logów (styczeń 2024) (Rysunek 17).
- Wdrożenie lokalizacji błędu (czerwiec 2024).

Do zarządzania wdrożeniem skorzystałem z metodologii Zwinnego Wytwarzania Oprogramowania [66]. Wybór tej metodologii kierowany był faktem, że byłem z tą metodologią dobrze zaznajomiony przez wcześniejsze doświadczenie pracy w zespole, a także tym, że rozwiązanie oferuje krótkie pętle implementacji i komunikacji z inżynierami, którzy z tego rozwiązania korzystają. Zależało mi na tym, aby dostarczone rozwiązanie, spełniało oczekiwania przyszłych użytkowników w jak największym stopniu.



Rysunek 16: Wygląd ogólny aplikacji, do której dodano funkcjonalność lokalizacji błędu.

Każdy etap implementacji był zatem zakończony weryfikacją użytkowników, którzy rekrutowali się spośród pracowników firmy.

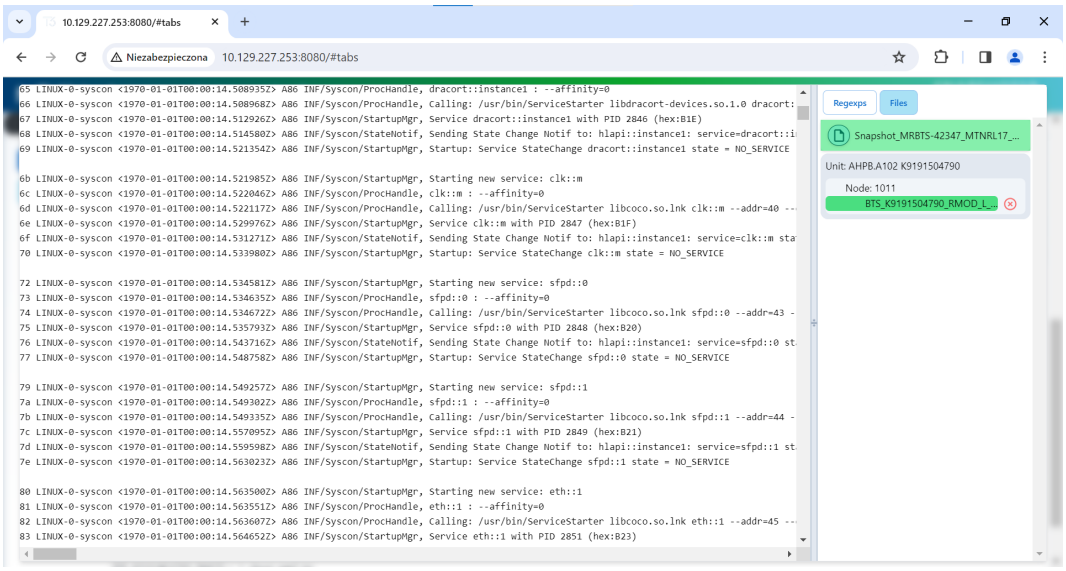
7.2.4 Problemy i wyzwania

Największym problemem było wdrożenie się i zespołu w nowe technologie internetowe, a także zaangażowanie się w promocje rozwiązania. Przejście od opracowywania nowych algorytmów, publikowania ich w literaturze, planowania badań do prowadzenia akcji marketingowej wśród pracowników i kadry zarządzającej było dla mnie największym wyzwaniem. Do zrozumienia zasad działania takiej akcji, zbudowania odpowiedniego zestawu kompetencji, a także odnalezienia odpowiedniej motywacji pomogły mi dodatkowe szkolenia organizowane na uczelni.

7.2.5 Wdrożenie

Wdrożenie rozwiązania zostało podzielone na dwie fazy. W pierwszej fazie, w 2023 roku, wdrożyłem segmentację logów na znaczące elementy. Druga faza, polegała na wdrożeniu lokalizacji błędów na podstawie poprawnych i błędnych logów, a jej wdrożenie rozpoczęło się w II kwartale 2024 a w całości została uruchomiona w czerwcu.

W czasie wdrażania lokalizacji błędu, w stosunku do pierwszej fazy, zrezygnowałem z wyświetlania analizy w aplikacji na rzecz ściągania pliku z analizą w formie txt na komputer użytkownika. Ma to o tyle przewagę, że inżynierowie są przyzwyczajeni do swoich narzędzi, w zaawansowany sposób skonfigurowanych i dostrojonych do indywidualnych potrzeb. Tworzenie nowego interfejsu użytkownika do przeglądania analizy wymagałoby dużych zasobów i pracy nie związanej bezpośrednio z analizą i jej jakością. Dodatkowo rezultat takiej pracy i tak mógł być niewystarczający dla użytkowników. Odszedłem zatem od prób stworzenia nowego narzędzia, na rzecz zwrócenia wysokiej jakości analizy i pozostawieniu użytkownikowi decyzji co do sposobu jego wyświetlania i analizowania.



Rysunek 17: Wygląd posegmentowanego logu przez algorytm VotingExperts w aplikacji webowej.

7.2.6 Wnioski i rekomendacje

Pierwszy etap wdrożenia, czyli segmentacja logów z błędnego wykonania był wprowadzony w pod koniec 2023 roku. Jego celem było ułatwienie manualnej analizy logów przez podział na znaczące segmenty. W etapie tym niewątpliwie zabrakło szerszej akcji promocyjnej i informacyjnej. Nie pomógł na pewno fakt reorganizacji firmy, w toku którego wiele osób już poinformowanych i przekonanych do rozwiązania odeszło z firmy. Drugim istotnym czynnikiem było nie dostosowanie rozwiązania do wszystkich komponentów i dostępnych architektur, a użytkownicy nie byli poinformowani dokładnie o tym jaki rezultat otrzymają. Często spodziewali się wykrywania anomalii, lub lokalizacji błędu, a otrzymywali segmentację na znaczące segmenty. Z rozmów z użytkownikami dowiedziałem się, że nie wiedzieli w jaki sposób to narzędzie może im pomóc w pracy. Dopiero po dłuższych rozmowach nabierali przekonania, że rozwiązanie to może być przydatne. Dodatkowo osiągnięcie przez używany algorytm F1-score równego 45% na zbiorze Nokii oznaczało również, że metoda ta wprowadza duży szum do segmentów względem oczekiwań człowieka, co mogło dodatkowo zniechęcać. Wnioskiem z tego etapu jest niewątpliwie potrzeba przeprowadzania odpowiednio wczesnej akcji informacyjnej, a także bardzo precyzyjne przygotowanie wczesnych wersji programu, oraz szybkie reagowanie na błędy by utrzymać zainteresowanie pierwszych użytkowników, którzy są kluczowi dla dalszego propagowania rozwiązania.

Drugim etapem było wdrożenie lokalizacji na podstawie poprawnych i błędnych logów. Tutaj punktem wyjścia było zapotrzebowanie konkretnych ludzi i wyrażane zainteresowanie takim rozwiązaniem zanim ono powstało. Jest to zasadnicza różnica w stosunku do segmentacji logów, gdzie rozwiązanie to było raczej ”nieuświadomioną potrzebą”. Lokalizacja błędów, jest w trakcie pilotażu. Wstępne informacje są pozytywne i wskazują, że to rozwiązanie odpowiada konkretnym potrzebom wielu użytkowników. W przypadku segmentacji logów konieczne byłoby najpierw przekonanie użytkowników, że segmentacja jest potrzebą, której wcześniej nie dostrzegali. Lokalizacja błędów w logach poprawnych i błędnych trafia w istniejące oczekiwania i już na początku spotkała się z dużym zainteresowaniem. W planach są cykliczne spotkania, na któ-

rych to rozwiązanie będzie prezentowane i rozwijane. Dzięki bliskiemu kontaktowi z użytkownikami nie muszę zdobywać ich zainteresowania, co znacząco ułatwia proces wdrożenia. Wyciągając wnioski z pierwszego etapu wdrożenia, kładę teraz nacisk na promocję tego rozwiązania i ścisłą współpracę z użytkownikami w celu usunięcia błędów i wdrożenia oczekiwanych funkcjonalności. **Oznacza to, że osiągnięto cel C5 polegający na opracowaniu i implementacji systemu opartego na wcześniejszych badaniach, umożliwiający pracownikom Nokii szybką analizę logów pochodzących z błędnego wykonania programu, zwracającego segmenty błędne oraz sugestie lokalizacji błędu**.

Dorobek

Mój dorobek naukowy obejmuje 4 opublikowane prace, w tym 3 w czasopismach, oraz 1 publikację konferencyjną:

1. **Dobrowolski**, W., Nikodem, M., Zawistowski, M., Unold, O. (2022). *Improved Software Reliability Through Failure Diagnosis Based on Clues from Test and Production Logs*. In *International Conference on Dependability and Complex Systems*, pp. 42-49. DOI: [10.1007/978-3-031-06746-4_5] Springer.

    - **ImpactFactor:** -
    - **Punkty MNiSW:** 40
    - **Koncepcja:** W. Dobrowolski
    - **Metodologia:** O. Unold, W. Dobrowolski
    - **Badania:** W. Dobrowolski
    - **Pisanie - oryginalny szkic:** W. Dobrowolski
    - **Pisanie - korekta** M. Nikodem, W. Dobrowolski, O. Unold
    - **Oprogramowanie:** W.Dobrowolski
    - **Nadzór:** M. Zawistowski, M. Nikodem, O. Unold
    - **Zaprezentowana na konferencji przez:** W.Dobrowolski

2. **Dobrowolski, W.**, Nikodem, M., Unold, O. (2023). *Software Failure Log Analysis for Engineers-Review. Electronics*, 12(10), 2260. DOI: [10.3390/electronics12102260](https://doi.org/10.3390/electronics12102260) MDPI.

    - **ImpactFactor:** 2.6
    - **Punkty MNiSW:** 100
    - **Koncepcja:** O. Unold, W. Dobrowolski
    - **Zebranie materiałów:** W. Dobrowolski
    - **Pisanie - oryginalny szkic:** W. Dobrowolski
    - **Pisanie - korekta** M. Nikodem, W. Dobrowolski, O. Unold
    - **Przegląd literatury:** W. Dobrowolski
    - **Nadzór:** M. Nikodem, O. Unold

3. **Dobrowolski, W.**, Libura, M., Nikodem, M., Unold, O. (2024). *Unsupervised Log Sequence Segmentation. IEEE Access*, pp. 1-1. DOI: [10.1109/ACCESS.2024.3409425].

- **ImpactFactor:** 3.557
- **Punkty MNiSW:** 100
- **Koncepcja:** O. Unold, W. Dobrowolski
- **Metodologia:** O. Unold, W. Dobrowolski
- **Badania:** W. Dobrowolski, M. Libura
- **Pisanie - oryginalny szkic:** W. Dobrowolski
- **Pisanie - korekta** M. Nikodem, W. Dobrowolski, O. Unold
- **Oprogramowanie:** W. Dobrowolski, M. Libura
- **Nadzór:** M. Nikodem, O. Unold

4. **Dobrowolski, W.**, Iwach-Kowalski, K., Nikodem, M., Unold, O. (2024). *Log-Based Fault Localization with Unsupervised Log Segmentation. Applied Sciences*, vol. 14, no. 18, article 8421. DOI: 10.3390/app14188421.

   - **ImpactFactor:** 2.5
   - **Punkty MNiSW:** 100
   - **Koncepcja:** W. Dobrowolski
   - **Metodologia:** W. Dobrowolski, O. Unold
   - **Badania:** W. Dobrowolski, K. Iwach-Kowalski
   - **Pisanie - oryginalny szkic:** W. Dobrowolski
   - **Pisanie - korekta:** M. Nikodem, W. Dobrowolski, O. Unold
   - **Oprogramowanie:** W. Dobrowolski
   - **Nadzór:** M. Nikodem, O. Unold

Zakończenie

Oprogramowanie odgrywa kluczową rolę w naszym codziennym życiu i jest obecne niemal w każdym aspekcie naszej rzeczywistości. Zawiera ono nie tylko aplikacje, z których korzystamy na co dzień, ale także złożone systemy, które zarządzają infrastrukturą, zdrowiem, finansami, czy bezpieczeństwem. Błędy w oprogramowaniu mogą mieć zatem ogromny wpływ na nasze życie, prowadząc do poważnych konsekwencji, zarówno w skali jednostkowej, jak i globalnej. Niezawodność oprogramowania staje się kwestią o fundamentalnym znaczeniu.

Oprogramowanie stacji bazowych BTS, jako oprogramowanie odpowiedzialne za jakość komunikacji bezprzewodowej, które kontroluje wymianę informacji między systemami, a także między ludźmi, jest szczególnie istotne. Mimo że jego funkcjonowanie często pozostaje niezauważone, awarie lub błędy w tym obszarze mogą prowadzić do poważnych zakłóceń, które mają dalekosiężne skutki. Z perspektywy przedsiębiorstw, niezawodność takiego oprogramowania jest nie tylko kwestią technologiczną, ale również ekonomiczną, ponieważ błędy mogą generować ogromne koszty.

Z tego powodu dbałość o niezawodność oprogramowania jest priorytetem na każdym etapie jego życia – od fazy wytwarzania, poprzez utrzymanie, aż po aktualizacje. Jednym z najbardziej czasochłonnych zadań w tym procesie jest lokalizacja błędów, czyli identyfikacja i zrozumienie miejsc w kodzie, gdzie te błędy występują. W swojej pracy skoncentrowałem się na rozwiązaniu tego problemu w oparciu o logi, które są najbardziej powszechnym i dostępnym źródłem informacji o błędach. Wprowadzają minimalny narzutu na czas wykonania programu, a interwencja w system również jest niewielka, i dokonywana w dobrze znany sposób.

Poprzez dogłębną analizę struktury logów, które odzwierciedlają działanie systemu, zostały zidentyfikowane interesujące zależności rządzące tymi danymi. Dzięki zastosowaniu nienadzorowanej segmentacji logów byłem w stanie opracować metody, które nie tylko ułatwiają manualną analizę poprzez podział logów na segmenty, ale także poprawiają jakość wykrywania anomalii i lokalizacji problemów w dużych systemach informatycznych. Dotyczy to również systemu BTS firmy Nokia, na którym zweryfikowałem i oceniłem opracowane rozwiązanie.

Realizacja pracy pozwolila mi osiagnąc następujące cele:

- Rozszerzenie koncepcji sekwencji logów jako języka naturalnego oraz zastosowanie metod nienadzorowanej segmentacji słów do segmentowania sekwencji logów, pozwoliło uzyskać zadowalającą segmentację w stosunku do złotego wzorca (Rozdział 4.2).

- Wykorzystanie uzyskanych segmentów z nienadzorowanej segmentacji poprawiło wykrywanie anomalii w logach (Rozdział 4.3).

- Opracowałem dwa algorytmy sztucznej inteligencji do lokalizacji błędów. Jeden działający w czasie bliskim do rzeczywistego, bazujący na LogGPT, drugi

statystyczny o większej złożoności, oba bazujące na kontekście otrzymanym z nienadzorowanej segmentacji (Rozdział 4.4).

- Eksperymenty przeprowadzone w ramach poprzednich badań pokazują, że opracowane i przebadane metody są lepsze niż metody znane z literatury.

- Opracowałem i zaimplementowałem system oparty na przeprowadzonych badaniach, umożliwiający pracownikom Nokii szybką analizę logów pochodzących z błędnego wykonania programu, zwracający segmenty błędne oraz sugestie lokalizacji błędu i jest on obecne w fazie pilotażu (Rozdział 7).

Uzyskane wyniki potwierdzają tezę, że zastosowanie metod uczenia maszynowego działających w czasie rzeczywistym pozwala na diagnostykę i lokalizację błędów w stacji przekaźnikowej o jakości nie gorszej od metod referencyjnych. Opracowane przeze mnie rozwiązanie zostało zaimplementowane i wdrożone w formie aplikacji internetowej, która na podstawie porównania logów z poprawnych i błędnych wykonań potrafi wskazać oraz uszeregować najbardziej podejrzane segmenty. Przeprowadzone eksperymenty wykazały, że moje podejście jest skuteczniejsze od dotychczasowych metod opisywanych w literaturze, nie wymagając ingerencji i narzutu na system.

Moje rozwiązanie ma również tę zaletę, że nie wymaga dużej ilości danych historycznych, co jest istotne w kontekście dynamicznie zmieniających się systemów informatycznych, gdzie zawartość logów może ulegać częstym zmianom. Jest natomiast taka możliwość, przy wykorzystaniu modelu opartego na LogGPT, by z takich danych skorzystać. Trzeba jednak wziąć pod uwagę obecną jego zależność od jakości dostarczonych szablonów. Systemy oparte na danych historycznych mogą także nie być adekwatne do aktualnych wymagań, a częste zmiany w oprogramowaniu wymagają kosztownych procesów douczania modeli.

Bibliografia

[1] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89--98. IEEE.

[2] Aggarwal, P., Gupta, A., Mohapatra, P., Nagar, S., Mandal, A., Wang, Q., and Paradkar, A. (2020). Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In *International Conference on Service-Oriented Computing*, pages 137--149. Springer.

[3] Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246--256.

[4] Alves, E., Gligoric, M., Jagannath, V., and d'Amorim, M. (2011). Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 520--523. IEEE.

[5] Amar, A. and Rigby, P. C. (2019). Mining historical test logs to predict bugs and localize faults in the test logs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 140--151.

[6] Bao, L., Li, Q., Lu, P., Lu, J., Ruan, T., and Zhang, K. (2018). Execution anomaly detection in large-scale systems through console log analysis. *Journal of Systems and Software*, 143:172--186.

[7] Binkley, D., Gold, N., and Harman, M. (2007). An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8--es.

[8] Chaparro, O., Florez, J. M., and Marcus, A. (2019). Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *Empirical Software Engineering*, 24:2947--3007.

[9] Chen, A. R. (2019). An empirical study on leveraging logs for debugging production failures. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 126--128. IEEE.

[10] Chen, A. R., Chen, T.-H., and Wang, S. (2022). Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, 48(8):2905--2919.

[11] Chen, M., Zheng, A. X., Lloyd, J., Jordan, M. I., and Brewer, E. (2004). Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36--43. IEEE.

[12] Chen, Z., Kang, Y., Li, L., Zhang, X., Zhang, H., Xu, H., Zhou, Y., Yang, L., Sun, J., Xu, Z., et al. (2020). Towards intelligent incident management: why we need it and how we make it. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1487--1497.

[13] Chen, Z., Liu, J., Gu, W., Su, Y., and Lyu, M. R. (2021). Experience report: deep learning-based system log analysis for anomaly detection. *arXiv preprint arXiv:2107.05908*.

[14] Cho, K., Van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724--1734.

[15] Cohen, P., Heeringa, B., and Adams, N. M. (2002). An unsupervised algorithm for segmenting categorical timeseries into episodes. In *Pattern Detection and Discovery: ESF Exploratory Workshop London, UK, September 16--19, 2002 Proceedings*, pages 49--62. Springer.

[16] Dang, Y., Lin, Q., and Huang, P. (2019). Aiops: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4--5. IEEE.

[17] de Castro Silva, J. M. R. et al. (2022). Spectrum-based fault localization for microservices via log analysis.

[18] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107--113.

[19] Debnath, B., Solaimani, M., Gulzar, M. A. G., Arora, N., Lumezanu, C., Xu, J., Zong, B., Zhang, H., Jiang, G., and Khan, L. (2018). Loglens: A real-time log analysis system. In *2018 IEEE 38th international conference on distributed computing systems (ICDCS)*, pages 1052--1062. IEEE.

[20] Dobrowolski, W. (2022). Defects4All. Version [version number], [Branch name].

[21] Dobrowolski, W. (2024a). Anomaly detection with unsupervised word segmentation. `https://github.com/dobrowol/anomaly_detection_with_uws`. Accessed: 03-Jul-2024.

[22] Dobrowolski, W. (2024b). Context based fault locaclization. `https://github.com/dobrowol/log_based_fault_localization`. Accessed: 12-Aug-2024.

[23] Dobrowolski, W., Libura, M., Nikodem, M., and Unold, O. (2024). Unsupervised log sequence segmentation. *IEEE Access*, pages 1--1.

[24] Dobrowolski, W., Nikodem, M., Zawistowski, M., and Unold, O. (2022). Improved software reliability through failure diagnosis based on clues from test and production logs. In *International Conference on Dependability and Complex Systems*, pages 42--49. Springer.

[25] Du, M., Li, F., Zheng, G., and Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285--1298.

[26] Fu, Y., Yan, M., Xu, Z., Xia, X., Zhang, X., and Yang, D. (2023). An empirical study of the impact of log parsers on the performance of log-based anomaly detection. *Empirical Software Engineering*, 28(1):6.

[27] Geman, S. and Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, PAMI-6(6):721--741.

[28] Granger, C. W. (1969). Investigating causal relations by econometric models and cross-spectral methods. *Econometrica: journal of the Econometric Society*, pages 424--438.

[29] Han, X., Yuan, S., and Trabelsi, M. (2023). Loggpt: Log anomaly detection via gpt. In *2023 IEEE International Conference on Big Data (BigData)*, pages 1117--1122. IEEE.

[30] He, P., Zhu, J., Zheng, Z., and Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*, pages 33--40. IEEE.

[31] He, Y. (1988). Extended viterbi algorithm for second order hidden markov process. In *9th International conference on pattern recognition*, pages 718--719. IEEE Computer Society.

[32] Ikram, A., Chakraborty, S., Mitra, S., Saini, S., Bagchi, S., and Kocaoglu, M. (2022). Root cause analysis of failures in microservices through causal discovery. *Advances in Neural Information Processing Systems*, 35:31158--31170.

[33] Jia, T., Chen, P., Yang, L., Li, Y., Meng, F., and Xu, J. (2017). An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *2017 IEEE international conference on web services (ICWS)*, pages 25--32. IEEE.

[34] Jiang, X., Pan, Y., Ma, M., and Wang, P. (2023). Look deep into the microservice system anomaly through very sparse logs. In *Proceedings of the ACM Web Conference 2023*, pages 2970--2978.

[35] John, G. H. and Langley, P. (1995). Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338--345. Morgan Kaufmann Publishers Inc.

[36] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467--477.

[37] Kam, H. T. et al. (1995). Random decision forest. In *Proceedings of the 3rd international conference on document analysis and recognition*, volume 1416, page 278282. Montreal, Canada, August.

[38] Kaufman, L. and Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons.

[39] Kaupp, L., Beez, U., Hülsmann, J., and Humm, B. G. (2019). Outlier detection in temporal spatial log data using autoencoder for industry 4.0. In *Engineering Applications of Neural Networks: 20th International Conference, EANN 2019, Xersonisos, Crete, Greece, May 24-26, 2019, Proceedings 20*, pages 55--65. Springer.

[40] Khatiwada, S., Tushev, M., and Mahmoud, A. (2018). Just enough semantics: An information theoretic approach for ir-based software bug localization. *Information and Software Technology*, 93:45--57.

[41] Kong, L., Dyer, C., and Smith, N. A. (2015). Segmental recurrent neural networks. *arXiv preprint arXiv:1511.06018*.

[42] Levin, A., Garion, S., Kolodner, E. K., Lorenz, D. H., Barabash, K., Kugler, M., and McShane, N. (2019). Aiops for a cloud object storage service. In *2019 IEEE International Congress on Big Data (BigDataCongress)*, pages 165--169. IEEE.

[43] Li, X., Chen, P., Jing, L., He, Z., and Yu, G. (2022). Swisslog: Robust anomaly detection and localization for interleaved unstructured logs. *IEEE Transactions on Dependable and Secure Computing*, 20(4):2762--2780.

[44] Li, X., Li, W., Zhang, Y., and Zhang, L. (2019). Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. ISSTA 2019, page 169–180, New York, NY, USA. Association for Computing Machinery.

[45] Li, Y., Jiang, Z. M., Li, H., Hassan, A. E., He, C., Huang, R., Zeng, Z., Wang, M., and Chen, P. (2020). Predicting node failures in an ultra-large-scale cloud computing platform: an aiops solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1--24.

[46] Li, Y., Wang, S., and Nguyen, T. (2021). Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661--673. IEEE.

[47] Liang, Y., Zhang, Y., Xiong, H., and Sahoo, R. (2007). Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 583--588. IEEE.

[48] Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., and Chen, X. (2016). Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102--111. IEEE.

[49] Lu, S., Wei, X., Li, Y., and Wang, L. (2018). Detecting anomaly in big data system logs using convolutional neural network. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and*

Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), pages 151--158.

[50] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

[51] Mao, X., Lei, Y., Dai, Z., Qi, Y., and Wang, C. (2014). Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51--62.

[52] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[53] Marwede, N., Rohr, M., van Hoorn, A., and Hasselbring, W. (2009). Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 47--58. IEEE.

[54] Mason, L., Baxter, J., Bartlett, P., and Frean, M. (1999). Boosting algorithms as gradient descent. *Advances in neural information processing systems*, 12.

[55] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26.

[56] Mills, C., Parra, E., Pantiuchina, J., Bavota, G., and Haiduc, S. (2020). On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering*, 25:3086--3127.

[57] Mochihashi, D., Yamada, T., and Ueda, N. (2009). Bayesian unsupervised word segmentation with nested pitman-yor language modeling. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 100--108.

[58] Nolle, T., Seeliger, A., and Mühlhäuser, M. (2016). Unsupervised anomaly detection in noisy business process event logs using denoising autoencoders. In *International conference on discovery science*, pages 442--456. Springer.

[59] Oliner, A. and Stearley, J. (2007). What supercomputers say: A study of five system logs. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 575--584. IEEE.

[60] OpenTelemetry Contributors (2024). Opentelemetry. `https://opentelemetry.io`. Accessed: 2024-07-12.

[61] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532--1543.

[62] Probst, P. and Boulesteix, A.-L. (2018). To tune or not to tune the number of trees in random forest. *Journal of Machine Learning Research*, 18(181):1--18.

[63] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9.

[64] Remil, Y., Bendimerad, A., Mathonat, R., and Kaytoue, M. (2024). Aiops solutions for incident management: Technical guidelines and a comprehensive literature review. *arXiv preprint arXiv:2404.01363*.

[65] Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513--523.

[66] Schwaber, K. and Sutherland, J. (2011). The scrum guide. *Scrum Alliance*, 21(1):1--38.

[67] Sha, Y., Nagura, M., and Takada, S. (2022). Fault localization in server-side applications using spectrum-based fault localization. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1139--1146. IEEE.

[68] Shani, G., Meek, C., and Gunawardana, A. (2009). Hierarchical probabilistic segmentation of discrete events. In *2009 Ninth IEEE International Conference on Data Mining*, pages 974--979. IEEE.

[69] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, 27(3):379--423.

[70] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25.

[71] Soremekun, E., Kirschner, L., Böhme, M., and Zeller, A. (2021). Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26:1--45.

[72] Studiawan, H., Sohel, F., and Payne, C. (2020). Anomaly detection in operating system logs with deep learning-based sentiment analysis. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2136--2148.

[73] Sun, Z. and Deng, Z.-H. (2018a). Unsupervised neural word segmentation for chinese via segmental language modeling. *arXiv preprint arXiv:1810.03167*.

[74] Sun, Z. and Deng, Z.-H. (2018b). Unsupervised neural word segmentation for chinese via segmental language modeling. *arXiv preprint arXiv:1810.03167*.

[75] Tan, J., Pan, X., Marinelli, E., Kavulya, S., Gandhi, R., and Narasimhan, P. (2010). Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. In *2010 IEEE Network Operations and Management Symposium-NOMS 2010*, pages 112--119. IEEE.

[76] Tantithamthavorn, C., Abebe, S. L., Hassan, A. E., Ihara, A., and Matsumoto, K. (2018). The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology*, 102:160--174.

[77] Teh, Y., Jordan, M., Beal, M., and Blei, D. (2004). Sharing clusters among related groups: Hierarchical dirichlet processes. *Advances in neural information processing systems*, 17.

[78] Tomek, I. (1976). Two modifications of cnn.

[79] Vaswani, A. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

[80] Wang, H., Wu, Z., Jiang, H., Huang, Y., Wang, J., Kopru, S., and Xie, T. (2021). Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419--429. IEEE.

[81] Wang, J., Tang, Y., He, S., Zhao, C., Sharma, P. K., Alfarraj, O., and Tolba, A. (2020a). Logevent2vec: Logevent-to-vector based anomaly detection for large-scale logs in internet of things. *Sensors*, 20(9):2451.

[82] Wang, L., Zhao, N., Chen, J., Li, P., Zhang, W., and Sui, K. (2020b). Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE international conference on web services (ICWS)*, pages 142--150. IEEE.

[83] Wang, Q., Parnin, C., and Orso, A. (2015). Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 1--11.

[84] Wardat, M., Le, W., and Rajan, H. (2021). Deeplocalize: Fault localization for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 251--262. IEEE.

[85] Weber, I., Li, C., Bass, L., Xu, X., and Zhu, L. (2015). Discovering and visualizing operations processes with pod-discovery and pod-viz. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 537--544. IEEE.

[86] Weiser, M. D. (1979). *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. University of Michigan.

[87] Wong, W. E., Debroy, V., Gao, R., and Li, Y. (2013). The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290--308.

[88] Wong, W. E., Gao, R., Li, Y., Abreu, R., Wotawa, F., and Li, D. (2023). Software fault localization: An overview of research, techniques, and tools. *Handbook of Software Fault Localization: Foundations and Advances*, pages 1--117.

[89] Wong, W. E. and Qi, Y. (2009). Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573--597.

[90] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. (2011). Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15--25.

[91] Xiao, Y., Keung, J., Bennin, K. E., and Mi, Q. (2018). Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology*, 99:58--61.

[92] Xiao, Y., Keung, J., Bennin, K. E., and Mi, Q. (2019). Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 105:17--29.

[93] Xu, J., Chen, P., Yang, L., Meng, F., and Wang, P. (2017). Logdc: Problem diagnosis for declartively-deployed cloud applications with log. In *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, pages 282--287. IEEE.

[94] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117--132.

[95] Yang, B., He, Y., Liu, H., Chen, Y., and Jin, Z. (2020). A lightweight fault localization approach based on xgboost. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 168--179. IEEE.

[96] Yang, L., Chen, J., Wang, Z., Wang, W., Jiang, J., Dong, X., and Zhang, W. (2021a). Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1448--1460.

[97] Yang, N., Cuijpers, P., Schiffelers, R., Lukkien, J., and Serebrenik, A. (2021b). An interview study of how developers use execution logs in embedded software engineering. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 61--70. IEEE.

[98] Yen, S., Moh, M., and Moh, T.-S. (2019). Causalconvlstm: Semi-supervised log anomaly detection through sequence modeling. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1334--1341. IEEE.

[99] Yu, J., Lei, Y., Xie, H., Fu, L., and Liu, C. (2022). Context-based cluster fault localization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 482--493.

[100] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. (2010). Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 143--154.

[101] Zaidman, A. and Demeyer, S. (2004). Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 329--338. IEEE.

[102] Zhang, Q., Jia, T., Wu, Z., Wu, Q., Jia, L., Li, D., Tao, Y., and Xiao, Y. (2022). Fault localization for microservice applications with system logs and monitoring metrics. In *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 149--154. IEEE.

[103] Zhang, S., Xia, S., Fan, W., Shi, B., Xiong, X., Zhong, Z., Ma, M., Sun, Y., and Pei, D. (2024). Failure diagnosis in microservice systems: A comprehensive survey and analysis. *arXiv preprint arXiv:2407.01710*.

[104] Zhang, X., Gupta, N., and Gupta, R. (2007). A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12:143--160.

[105] Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z., et al. (2019a). Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 807--817.

[106] Zhang, Y., Makarov, S., Ren, X., Lion, D., and Yuan, D. (2017a). Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 19--33.

[107] Zhang, Z., Lei, Y., Mao, X., and Li, P. (2019b). Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445--455. IEEE.

[108] Zhang, Z., Lei, Y., Tan, Q., Mao, X., Zeng, P., and Chang, X. (2017b). Deep learning-based fault localization with contextual information. *IEICE TRANSACTIONS on Information and Systems*, 100(12):3027--3031.

[109] Zhong, H. and Mei, H. (2020). Learning a graph-based classifier for fault localization. *Science china information sciences*, 63:1--22.

[110] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., and He, C. (2019). Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 683--694.