http://www.moderntechno.de/index.php/meit/article/view/meit41-01-022
DOI: 10.30890/2567-5273.2025-41-01-022

УДК 004.415.3

INTELLIGENT ANALYSIS OF AUTOMATED WEB APPLICATION TESTING LOGS: THE LIPSI METHOD

ІНТЕЛЕКТУАЛЬНИЙ АНАЛІЗ ЛОГІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ: МЕТОД LIPSI

Lipskyi D.O. / Ліпський Д.О.
PhD Student /Аспірант
ORCID: 0009-0000-4068-9453
Taras Shevchenko National University of Kyiv,
Kyiv, 4d Akademika Glushkova Avenue, 02000
Київський національний університет імені Тараса Шевченка,
м. Київ, проспект Академіка Глушкова, 4д, 02000

Анотація. *Автоматизація тестування веб-застосунків є важливим аспектом сучасних процесів гарантування якості програмного забезпечення, особливо у великих і складних проєктах, де виникає потреба у перевірці сотень або навіть тисяч сценаріїв у стислі терміни. Одним із критичних завдань під час виконання автоматизован__их тестів є збір, обробка та аналіз технічних повідомлень (логів). Лог у цьому контексті — це текстовий рядок, який розробник вставляє у програмний код для фіксації виняткових чи помилок при виконанні. Такі повідомлення містять ключову інформацію про хід виконання тестів, виявлені помилки, а також поведінку системи й окремих її компонентів.*

Вони є важливим джерелом для визначення пріоритетності знайдених у продукті дефектів. Для визначення черговості подальшого виправлення виявлених дефектів дуже важливо знати порядок їх виправлення (пріоритетність). Проте ручний аналіз пріоритетності логів для сучасних систем є надзвичайно трудомістким, суб'єктивним і схильним до пропусків важливих деталей, що може затримувати виявлення критичних дефектів та впливати на загальну надійність продукту.

Зі зростанням обсягів даних зростає і потреби в оперативному усуненні можливих недоліків. Це визначає необхідність для аналізу логів створювати інтелектуальні системи, здатних як автоматично визначати пріоритетність повідомлень, так і виділяти найважливіші з них. Класичні ШІ-підходи, попри очевидні переваги, потребують попереднього навчання на великих наборах даних або використовують складні нейронні архітектури. Це виявляється малоефективними в умовах обмеженої кількості даних, високої варіативності логів і потреби в прозорості обґрунтування рішень.

У статті запропоновано авторський метод LIPSI (Log-based inference of priority in software issues - визначення пріоритетності програмних помилок на основі аналізу логів) інтелектуального аналізу логів, що базується на багатоетапній оцінці ознак без використання навчальних вибірок. Метод використовує ключові слова, статистичні характеристики логів і динамічне моделювання ваг, адаптоване до поточного контексту. При тестовому використанні метод демонструє високу точність визначення пріоритетності програмних помилок навіть у складних умовах із нестабільними сценаріями та мінімальними ресурсами. Запропонований метод поєднує простоту впровадження, адаптивність до змін середовища й ефективність у виявленні критичних помилок.

Ключові слова: *інтелектуальний аналіз логів, автоматизоване тестування, класифікація помилок, логістична регресія, динамічне вагове моделювання, лог-файли, LIPSI, пріоритезація дефектів, штучний інтелект у тестуванні, аналіз тестових результатів.*

Вступ.

Зростання складності програмних систем, кількості сценаріїв їх використання та швидкості розгортання релізів у рамках DevOps-практик суттєво збільшує інтенсивність тестування. Відповідно зростають і обсяги логів, що генеруються в процесі автоматизованих перевірок. Ці логи вказують на потенційні дефекти, містять ключову інформацію про хід виконання тестів, про внутрішні помилки системи, помилки з'єднання, виняткові ситуації та інші сигнали [2]. Проте обробка таких даних вручну — навіть за наявності досвідчених інженерів — є надзвичайно ресурсозатратною, повільною, містить ризик пропуску помилок людиною.

У відповідь на ці виклики дедалі актуальнішим стає впровадження інтелектуальних засобів аналізу логів, здатних автоматично визначати критичність проблем, групувати подібні помилки [3], виділяти основні шаблони несправностей та генерувати рекомендації. Попри численні досягнення у сфері штучного інтелекту, більшість класичних моделей машинного навчання демонструють обмежену ефективність у контексті логів автотестування. Причини цього — нестабільність середовища (часта зміна лог-структури через оновлення функціоналу), обмежений обсяг якісно анотованих даних, залежність від контексту виконання тестів, а також потреба в інтерпретованості результатів.

У цій роботі для вирішення зазначених проблем запропоновано метод LIPSI, що поєднує просту багатоетапну логіку з динамічним зважуванням та обліком контексту, без використання навчальних вибірок. Його ключова ідея полягає в комбінуванні евристичних правил, ознак повідомлень та статистики виконання для пріоритизації логів за ступенем критичності. Такий метод забезпечує швидке реагування, прозорість обґрунтувань та гнучкість при зміні умов середовища, що робить його практично придатним для реального використання в автоматизованих платформах контролю якості програмного забезпечення [6].

Обмеження класичних ШІ-підходів у контексті логів автотестування.

Оскільки логи є текстовими рядками природною мовою, що містять слова, словосполучення та характеристики які відображають стани системи й

зафіксовані розробником у коді, цілком логічно розглядати їх аналіз пріоритетності у контексті методів обробки природної мови (NLP). Для автоматизації класифікації логів і помилок у промислових проєктах активно використовувалися класичні моделі машинного навчання, що належать до сфери обробки природної мови (NLP). Наприклад, Google у ранніх дослідженнях застосовував наївний баєсівський класифікатор для автоматичної категоризації логів Android-систем, Microsoft використовувала Logistic Regression і Linear SVM для аналізу повідомлень у системах моніторингу Azure [1], а компанії Atlassian та Facebook експериментували з деревоподібними моделями й градієнтним бустингом (LightGBM) для виявлення закономірностей у логах своїх CI/CD-процесів. Практика показала життєздатність цих підходів, проте ефективне використання методів штучного інтелекту (ШІ) для аналізу логів автотестів стикнулося з низкою критичних обмежень [5], які роблять класичні підходи непридатними або малоефективними у реальних умовах.

Таким проблемним обмеженням є, наприклад, висока варіативність і контекстна специфіка тестового середовища. Логи генеруються динамічно, на основі поточного функціоналу та умов виконання тестів. Навіть незначна зміна у веб-застосунку (оновлення модуля, нова мікросервісна інтеграція, удосконалення інтерфейсу прикладного програмування) може радикально змінити структуру повідомлень у логах. У швидкозмінному середовищі заздалегідь навчені ШІ-моделі швидко втрачають актуальність, оскільки не встигають адаптуватися до нових шаблонів поведінки системи.

Друга проблема полягає в обмеженості навчальних даних. Повноцінне навчання класичних моделей потребує великої кількості анотованих прикладів (логів з відповідними класами критичності). На практиці для контролю якості програмного забезпечення таке маркування здійснюється вручну й точково, оскільки не всі повідомлення є однозначно трактованими. Як наслідок, моделі часто перенавчаються на вузькоспеціалізованих вибірках і демонструють низьку здатність до генералізації на нові дані.

Третє обмеження — контекстна неоднозначність логів. Одне й те саме

ключове слово логів, наприклад timeout, може означати як критичну помилку в одному сценарії (наприклад, втрату зв'язку з основним сервером), так і нормальну поведінку в іншому (наприклад, очікуване завершення сесії). Автоматична генерація правил класифікації в таких умовах є ненадійною, оскільки загальні правила ведуть до хибнопозитивних результатів, а надто специфічні — до пропуску критичних ситуацій.

Четверта проблема — висока обчислювальна складність моделей. Навіть якщо модель тренується заздалегідь, її застосування в реальному CI/CD-процесі може вимагати додаткових ресурсів (GPU, пам'ять), що є критично неприйнятним в умовах швидких релізів. Більше того, складні ШІ-системи часто є «чорними скриньками», а отже, важко інтерпретуються та не дають при контролю якості розуміння причин ухвалення певного рішення.

У результаті, класичні підходи на основі штучного інтелекту виявляються непрактичними в умовах реального автоматизованого тестування, де домінують обмеження за часом, ресурсами й доступністю якісних даних. Це створює потребу у нових підходах — прозорих і адаптивних — здатних працювати з малими вибірками, швидко реагувати на зміни й інтегруватися в існуючі платформи без ускладнення архітектури. Одним із таких методів і може стати метод LIPSI.

Архітектура та компоненти системи.

Запропонована система інтелектуального аналізу логів побудована на модульному принципі та включає дві основні підсистеми: компонент збору логів під час виконання автоматизованих тестів і компонент інтелектуального аналізу, що реалізує метод LIPSI. На вході система отримує потоки логів, які генеруються як тестовими сценаріями, так і інфраструктурними модулями платформи. у компонент збору ці логи структуруються, зберігаються й фільтруються за базовими ознаками (час виконання, джерело, рівень повідомлення) [2]. Далі дані передаються до компоненти інтелектуального аналізу, де за методом LIPSI здійснюється поетапне автоматичне ранжування повідомлень за рівнем пріоритетності. На виході користувач отримує впорядковані результати у

вигляді проаналізованого набору логів, де кожному запису призначено ступінь ризику та категорію пріоритету.

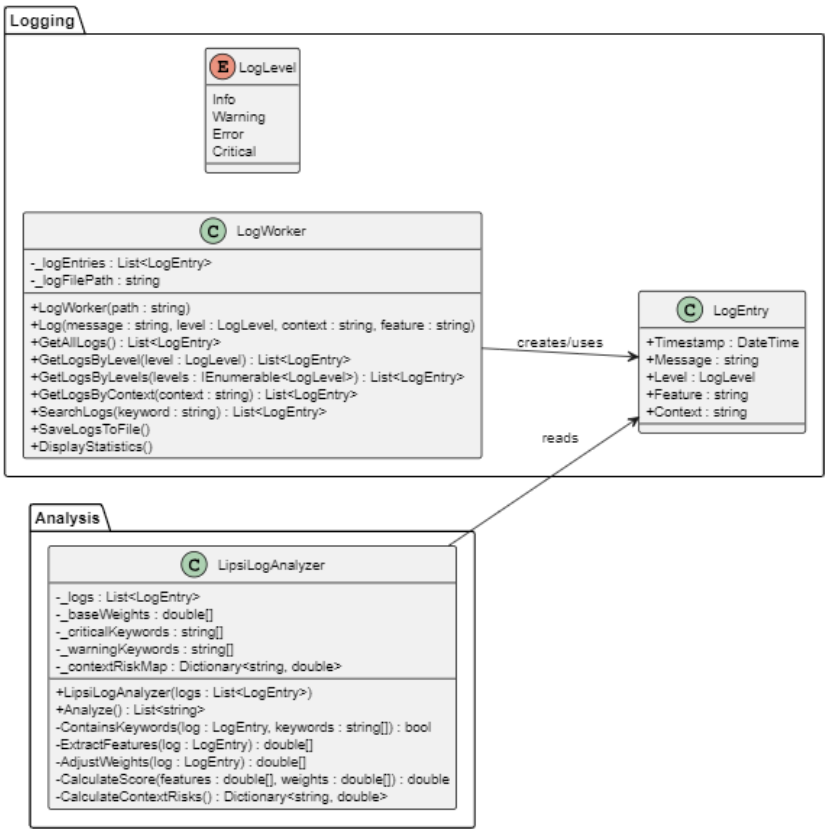

Рисунок 1 - Модель взаємодії LogWorker, LogEntry та LIPSI Log Analyzer

Ключовим елементом підсистеми збору логів є клас LogWorker, який реалізує механізм структурованого логування в процесі тестування [4]. Логи формуються з використанням об’єктної моделі LogEntry, яка містить такі поля, як час події, рівень повідомлення, назва контексту, функціональний блок і текст повідомлення. Це дозволяє зафіксувати не лише факт події, а й її позицію у контексті конкретного тесту чи підсценарію [4].

У процесі тестування, LogWorker викликається на кожному кроці, де необхідно зафіксувати дію, подію або помилку. Наприклад, при запуску тесту авторизації можуть генеруватися такі логи:

```
var logger = new LogWorker("test_log.log");
logger.Log("Start login process", LogLevel.Info, "LoginTest", "Authentication");
logger.Log("Timeout occurred while connecting to server", LogLevel.Error, "LoginTest", "Authentication");
logger.Log("User redirected to dashboard", LogLevel.Info, "LoginTest", "Navigation");
logger.SaveLogsToFile();
```

Рисунок 2 - Фрагмент коду логування подій із застосуванням LogWorker

Усі логи зберігаються у вигляді списку об’єктів LogEntry та можуть бути передані на обробку в аналітичний компонент LipsiLogAnalyzer. Цей клас реалізує багатоетапну стратегію аналізу логів, визначаючи їх пріоритетність відповідно до методу LIPSI. Використання аналітичного модуля відбувається у два кроки: спочатку передається список логів, після чого викликається метод аналізу.

```
var logs = logger.GetAllLogs();
var priorities = LipsiLogAnalyzer.Analyze(logs);
foreach (var (log, priority) in priorities)
{
    Console.WriteLine($"[{log.Level}] {log.Message} => Priority: {priority}");
}
```

Рисунок 3 - Визначення пріоритетності повідомлень логів

Така стратегія дозволяє відразу отримати карту пріоритетів для всіх логів тестування, що є основою для швидкої діагностики проблем у CI/CD процесі [10].

Таким чином, архітектура системи базується на принципах структурованого логування, розділення відповідальностей та адаптивного інтелектуального аналізу, що робить її ефективною як для інтерактивного, так і для автоматизованого режиму тестування [7].

Метод LIPSI: багатоетапний аналіз логів.

В процесі автоматизованого тестування Метод LIPSI виконує багатоетапний аналіз згенерованих логів. Мета алгоритму аналізу полягає в тому, щоб на основі самих лише логів, без додаткової розмітки чи попереднього навчання, забезпечити швидке, адаптивне й достовірне ранжування повідомлень за рівнем пріоритетності виявлених проблем. Архітектура методу побудована у вигляді п’яти послідовних етапів обробки. Ключову роль у функціонуванні відіграють алгоритми, які реалізують ці етапи та забезпечують взаємодію між ними. Саме алгоритмічні механізми роблять можливим перетворення даних у кількісні оцінки критичності, що дозволяє отримати інтегровану картину ризиків.

Перший етап виконує евристичну фільтрацію логів за ключовими словами.

Якщо повідомлення містить терміни, що зазвичай сигналізують про серйозні проблеми, наприклад "exception", "fatal", "timeout", "crash" або "stacktrace", воно негайно класифікується як "Critical". Якщо виявлено слова, що натякають на погіршення продуктивності або потенційні ризики, такі як "retry", "slow", "deprecated", "delay", запису надається статус "High Priority". Це дозволяє системі швидко обробляти очевидні випадки без залучення складніших розрахунків. Реалізація цього етапу в коді виглядає так:

```
private bool ContainsKeywords(LogEntry log, string[] keywords)
{
    return keywords.Any(k => log.Message.Contains(k, StringComparison.OrdinalIgnoreCase));
}
```

Рисунок 4 - Реалізація евристичної фільтрації логів за ключовими словами

Якщо ж повідомлення не містить жодного з зазначених ключових слів, активується алгоритм другого етапу — векторизація ознак. Метод ExtractFeatures формує числовий опис повідомлення, який включає значення наявності рівнів Info, Warning, Error та довжину повідомлення в символах. Така векторизація переводить лог у формат, придатний для подальших обчислень [9].

```
private double[] ExtractFeatures(LogEntry log)
{
    return new double[]
    {
        log.Level == LogLevel.Info ? 1 : 0,
        log.Level == LogLevel.Warning ? 1 : 0,
        log.Level == LogLevel.Error ? 1 : 0,
        log.Message.Length
    };
}
```

Рисунок 5 - Метод векторизації ознак логів

На третьому етапі виконується динамічне вагове моделювання. Початкові ваги ознак задані у масиві _baseWeights = [2.0, 1.5, 3.0, 0.01], що відповідає рівням Info, Warning, Error та довжині повідомлення. Ваги є числовими коефіцієнтами, які відображають відносну важливість ознак під час обчислення сумарного балу й дозволяють посилювати або послаблювати вплив характеристик повідомлення на його пріоритетність. Значення обрані емпірично: Error отримує найбільшу вагу (3.0), оскільки найчастіше сигналізує про критичні

збої; Warning має середнє значення (1.5), Info — найменше (2.0); довжина повідомлення враховується мінімально (0.01) як допоміжний індикатор нетипових логів [9].

Метод AdjustWeights масштабує ці коефіцієнти з урахуванням контексту виконання. Часовий множник підвищує ваги на 20%, якщо протягом останніх п'яти хвилин зафіксовано понад п'ять повідомлень рівня Error, що відображає сплески нестабільності. Значення «5 хвилин» та «5 повідомлень» вибрані експериментально на основі емпіричного аналізу виконання тестів: саме ці порогові параметри продемонстрували оптимальний баланс між чутливістю до короткочасних збоїв та стійкістю до випадкових поодиноких помилок. Контекстний множник визначається картою ризиків _contextRiskMap і для модулів із більшою історичною часткою помилок наближається до 2.0. У результаті базові ваги динамічно адаптуються до поточних умов виконання тестів і накопиченої статистики помилок, забезпечуючи баланс між стабільністю та гнучкістю моделі. Код цього етапу виглядає наступним чином:

```
private double[] AdjustWeights(LogEntry log)
{
    int recentErrors = _logs.Count(l => l.Level == LogLevel.Error && l.Timestamp > DateTime.Now.AddMinutes(-5));
    double timeFactor = (recentErrors > 5 ? 2.0 : 1.0);
    double contextWeight = _contextRiskMap.TryGetValue(log.Context, out double risk) ? risk : 1.0;
    return _baseWeights.Select(w => w * timeFactor * contextWeight).ToArray();
}
```

Рисунок 6 - Метод динамічного вагового моделювання логів

Четвертий етап відповідає за обчислення оцінки пріоритетності — числового значення, яке відображає важливість лог-запису. Для цього використовується проста лінійна формула, в якій значення кожної ознаки множиться на її вагу, а потім підсумовується. Такий підхід є аналогом логістичної регресії, що не потребує навчання. Метод CalculateScore реалізує це як:

```
private double CalculateScore(double[] features, double[] weights)
{
    return features.Zip(weights, (f, w) => f * w).Sum();
}
```

Рисунок 7 - Метод обчислення оцінки пріоритетності логів

На завершальному, п’ятому етапі виконується порогова класифікація. Якщо обчислений бал перевищує 8.0, лог позначається як "Critical"; якщо перевищує 6.0 — "High Priority"; якщо перевищує 4.0 — "Normal"; інакше — "Low Priority". Класифікація здійснюється в діапазоні від 0 до 10, оскільки така шкала є інтуїтивно зрозумілою, дозволяє легко інтерпретувати результати та забезпечує достатню градацію для розмежування рівнів важливості. Вибір саме цих порогових значень ґрунтується на принципі рівноваги між чутливістю й специфічністю: нижні пороги (4.0 та 6.0) відповідають переходу від малозначущих до помітних повідомлень, тоді як верхній поріг (8.0) виділяє винятково критичні ситуації, які потребують негайного реагування. Увесь процес обробки логів, що охоплює виклики всіх попередніх етапів, реалізовано в методі Analyze:

```
public List<string> Analyze()
{
    return _logs.Select(log =>
    {
        if (ContainsKeywords(log, _criticalKeywords)) return "Critical";
        if (ContainsKeywords(log, _warningKeywords)) return "High Priority";

        var features = ExtractFeatures(log);
        var dynamicWeights = AdjustWeights(log);
        double score = CalculateScore(features, dynamicWeights);

        if (score > 8.0) return "Critical";
        if (score > 6.0) return "High Priority";
        if (score > 4.0) return "Normal";
        return "Low Priority";
    }).ToList();
}
```

Рисунок 8 - Метод порогової класифікації пріоритетності логів

Формування карти ризиків контекстів відіграє ключову роль у ваговій моделі. Метод CalculateContextRisks обчислює для кожного контексту співвідношення кількості критичних повідомлень до загальної кількості записів, отримуючи таким чином коефіцієнт ризику, який потім масштабує ваги під час аналізу. Якщо, наприклад, для контексту "PaymentFlow" серед 10 логів п’ять є помилковими, коефіцієнт ризику буде 1.5. Це дозволяє системі навчатися на основі історичних даних без потреби в попередньому маркуванні чи використанні зовнішніх наборів даних.

```
private Dictionary<string, double> CalculateContextRisks()
{
    var contextGroups = _logs.GroupBy(l => l.Context);
    var result = new Dictionary<string, double>();

    foreach (var group in contextGroups)
    {
        int total = group.Count();
        int errors = group.Count(g => g.Level == LogLevel.Error || g.Level == LogLevel.Critical);
        double factor = 1 + (double)errors / Math.Max(1, total);
        result[group.Key] = factor;
    }

    return result;
}
```

Рисунок 9 - Метод формування карти ризиків контекстів логів

Таким чином, метод LIPSI поєднує простоту логіки, адаптивність вагового моделювання та інтерпретованість результатів [7]. Він дозволяє досягати ефективної класифікації логів навіть у нестабільних тестових середовищах, без потреби у великих обсягах тренувальних даних. Його модульна структура робить його гнучким для подальшого розвитку, інтеграції з CI/CD-процесами або розширення новими евристичними правилами.

Експериментальне порівняння з NLP-моделями.

У межах проведеного дослідження було здійснено експериментальне порівняння ефективності методу LIPSI з рядом традиційних моделей машинного навчання, які зазвичай застосовуються в задачах обробки природної мови. Такий аналіз був необхідний для оцінки доцільності використання класичних NLP-рішень у контексті інтелектуального аналізу логів автоматизованого тестування та обґрунтування переваг запропонованого методу.

Серед обраних для порівняння алгоритмів було протестовано логістичну регресію, класифікатор на основі наївного баєсівського підходу, деревоподібні моделі FastTree і LightGBM, а також інші поширені підходи — Linear SVM, Averaged Perceptron, L-BFGS Maximum Entropy і SDCA Non-Calibrated [8]. Ці моделі демонструють широкий спектр популярних класифікаційних методів, кожен з яких має свої переваги залежно від особливостей вхідних даних [8].

Логістична регресія є одним із найпростіших лінійних методів, що оцінює ймовірність належності до певного класу. Вона ефективна в умовах добре

роздільних даних, проте обмежена в здатності моделювати складні залежності. Наївний байєсівський підхід працює швидко, однак базується на припущенні незалежності ознак, що рідко відповідає реальним умовам логів. Моделі FastTree і LightGBM є алгоритмами градієнтного підсилення на основі дерев рішень, які зазвичай добре справляються зі складною структурою даних, але вимагають великої кількості прикладів для досягнення стабільної продуктивності. Лінійний метод опорних векторів (Linear SVM) відомий своєю високою точністю на лінійно роздільних вибірках, хоча в умовах обмеженої кількості ознак і варіативності повідомлень його ефективність знижується. Averaged Perceptron є базовим варіантом одношарового персептрону, що демонструє стабільну, але невисоку точність. Методи L-BFGS Maximum Entropy та SDCA Non-Calibrated застосовують оптимізаційні стратегії, однак без індивідуального налаштування і контекстної інформації часто не дають значного покращення результатів.

Усі моделі були протестовані на одному й тому самому наборі логів, отриманих під час виконання автоматизованих тестів. Для оцінки точності застосовувалися дві метрики: MicroAccuracy та MacroAccuracy. Перша метрика враховує загальну кількість правильних класифікацій незалежно від класу, що дозволяє оцінити загальну ефективність моделі. Друга — MacroAccuracy — розраховується як середнє арифметичне точності по кожному класу, що дає змогу оцінити, наскільки модель збалансовано працює з усіма типами повідомлень, включаючи рідкісні випадки. Отримані результати свідчать про значну перевагу підходу LIPSI.

Як видно з таблиці 1, жодна з традиційних моделей не досягла рівня точності, співставного з результатами методу LIPSI. Найближче за MicroAccuracy підійшла модель Linear SVM (50.00%), однак її MacroAccuracy залишилася нижчою (38.89%), що свідчить про нерівномірність класифікації між класами. Модель Naive Bayes показала відносно високу MacroAccuracy (50.00%), але її MicroAccuracy залишилася на рівні 33.33%. Решта моделей не перевищили поріг 33.33% за жодною з метрик.

Таблиця 1 - Порівняння точності методу LIPSI та традиційних NLP-моделей за метриками MicroAccuracy та MacroAccuracy

Модель	MicroAccuracy	MacroAccuracy
LIPSI	**60.00%**	**61.45%**
Logistic Regression	16.67%	16.67%
Naive Bayes	33.33%	50.00%
FastTree	16.67%	33.33%
LightGBM	16.67%	11.11%
Averaged Perceptron	16.67%	16.67%
Linear SVM	50.00%	38.89%
L-BFGS Maximum Entropy	16.67%	33.33%
SDCA Non-Calibrated	16.67%	16.67%

Таким чином, можна зробити висновок, що метод LIPSI демонструє вищу ефективність порівняно з класичними підходами. Завдяки використанню контекстно-залежного вагового моделювання, частотного аналізу, логістичної регресії на ознаках та семантичного розпізнавання ключових слів, LIPSI краще адаптується до особливостей логів, зберігаючи високу точність навіть за умов обмеженого обсягу даних.

Переваги та обмеження підходу LIPSI.

Метод LIPSI має низку переваг, які зумовлюють його ефективність у контексті аналізу логів, отриманих під час автоматизованого тестування веб-застосунків. По-перше, LIPSI не потребує попереднього навчання на великих наборах даних, що дозволяє застосовувати його в середовищах з обмеженим обсягом розмічених даних або в умовах, коли структура логів динамічно змінюється. Це забезпечується завдяки багатоетапній евристично-математичній природі методу: ключові слова, динамічне вагове моделювання, векторизація ознак і статистична оцінка ризику контекстів функціонують у рамках єдиної логіки, яка не потребує зовнішніх залежностей або складного машинного навчання [6].

По-друге, система повністю інтерпретована. Кожне рішення можна пояснити через вирахувану оцінку, набір ознак і контекстну вагу. Це дозволяє інженерам і аналітикам не лише отримати підсумкову класифікацію, а й

зрозуміти, чому саме певний лог був оцінений як критичний або неважливий. На відміну від "чорних скринь" типових для багатьох сучасних моделей машинного навчання, LIPSI надає повну прозорість розрахунків.

Третьою перевагою є гнучкість і адаптивність. Завдяки механізму AdjustWeights система може реагувати на зміну поведінки тестованої системи у реальному часі. Наприклад, у разі сплеску помилок в одному з модулів (скажімо, через нестабільність зовнішнього API), метод автоматично підвищує вагу відповідного контексту, що дозволяє швидше виявляти нові критичні області.

Однією з сильних сторін є швидкість виконання. Завдяки простим лінійним обчисленням, навіть великі обсяги логів можуть бути оброблені за лічені секунди. Це робить LIPSI придатним для інтеграції безпосередньо у процеси CI/CD, де час виконання тестів критично важливий [10].

Водночас метод має і свої обмеження. Головним з них є відсутність здатності до узагальнення на основі нових шаблонів, які не були враховані у початковій логіці. Наприклад, якщо з’явиться новий тип помилки з повідомленням, яке не містить ключових слів, і має незвичну структуру, система може не надати йому високого пріоритету, навіть якщо він є критичним для бізнесу. Іншими словами, LIPSI обмежений жорстко заданими правилами, які хоч і адаптивні в рамках заданих ваг і ризиків, але не в змозі автоматично вивчати нові закономірності.

Також варто зазначити, що метод не враховує повний контекст виконання тесту в термінах кроків, сценаріїв або тривалості взаємодії. Для цього необхідно поєднувати LIPSI з іншими джерелами даних або розширювати модель новими параметрами, наприклад, часу між логами, кількості логів у потоці тощо.

Нарешті, хоча система інтерпретована, її ефективність залежить від якості логів: якщо лог-файли мають бідну або непослідовну структуру, то точність класифікації може знижуватися. В таких випадках бажано доповнити метод попередньою нормалізацією логів або генерацією стандартизованих повідомлень у рамках тестових фреймворків.

Отже, LIPSI — це легкий, ефективний, адаптивний та інтерпретований

метод, який добре працює в умовах обмежених ресурсів, але потребує регулярного перегляду ключових термінів, адаптації до нових шаблонів помилок і, за потреби, доповнення моделі новими ознаками з метою підвищення точності контекстного аналізу.

Висновки.

У роботі описується метод LIPSI - теоретично обґрунтований та реалізований новий метод до аналізу логів (технічних повідомлень), що виникають під час автоматизованого тестування веб-застосунків. Основною метою цього підходу є автоматизоване визначення пріоритетності помилок, виявлених у логах, без потреби у використанні складних глибоких нейронних мереж чи великих навчальних вибірок. Це робить метод LIPSI надзвичайно зручним для застосування у середовищах з обмеженими обчислювальними ресурсами, нестабільними форматами логів або обмеженим історичним обсягом даних.

У роботі описується метод LIPSI - теоретично обґрунтований та реалізований новий метод до аналізу логів (технічних повідомлень), що виникають під час автоматизованого тестування веб-застосунків. Основною метою цього підходу є автоматизоване визначення пріоритетності помилок, виявлених у логах, без потреби у використанні складних глибоких нейронних мереж чи великих навчальних вибірок. Це робить метод LIPSI надзвичайно зручним для застосування у середовищах з обмеженими обчислювальними ресурсами, нестабільними форматами логів або обмеженим історичним обсягом даних.

Такі результати підтверджують ефективність методу LIPSI у задачах пріоритизації помилок у логах автоматизованого тестування. Важливо зазначити, що ця модель не потребує навчання на великих обсягах даних і не вимагає складного налаштування — усі параметри є прозорими й легко адаптуються до особливостей конкретного середовища. Серед переваг можна відзначити високу інтерпретованість, можливість роботи у ресурсно-обмежених середовищах, швидку обробку та зручність інтеграції в існуючі тестові

фреймворки. До обмежень слід віднести чутливість до якості логів, неможливість виявлення нових типів аномалій поза вбудованими евристиками та залежність від контекстної інформації, яка не завжди буває доступною або структурованою.

Метод LIPSI передбачає у перспективі можливість підтримки часових взаємозв'язків між подіями, контекстної пам'яті та інтеграції з CI/CD-середовищем [10]. Таким чином, метод LIPSI є практичним підґрунтям для створення інтелектуальних, адаптивних та легких у впровадженні систем аналізу логів, що відповідають реальним потребам сучасного автоматизованого тестування.

Література:

1. Landauer M., Wüchner T., Reuter C., et al. Deep Learning for Anomaly Detection in Log Data: A Survey // Computers & Security. – 2023. – Vol. 131. – DOI: https://doi.org/10.1016/j.cose.2023.103263

2. Khan Z.A., et al. Impact of Log Parsing on Deep Learning-based Anomaly Detection // Empirical Software Engineering. – 2024. – Vol. 29. – DOI: https://doi.org/10.1007/s10664-024-10533-w

3. Duan Y., Zhang S., Wang Q., Zhao H. Log Anomaly Detection via Evidential Deep Learning (LogEDL) // Applied Sciences. – 2024. – Vol. 14, No. 16. – DOI: https://doi.org/10.3390/app14167055

4. Mäntylä M., Wang Y., Nyyssölä J. LogLead: Fast and Integrated Log Loader, Enhancer, and Anomaly Detector // arXiv preprint. – 2023. – URL: https://arxiv.org/abs/2311.11809

5. Liu J., Huang J., Huo Y., Jiang Z., Gu J., Chen Z., Feng C., Lyu M. Log-based Anomaly Detection Based on EVT Theory with Feedback // arXiv preprint. – 2023. – URL: https://arxiv.org/abs/2306.05032

6. Biehl M. Fundamentals of Machine Learning. – Cham: Springer Nature, 2023. – 316 c. – ISBN: 978-3-031-35770-2

7. Arora S., Gupta A. Hands-On Exploratory Data Analysis with Python. –

Birmingham: Packt Publishing, 2023. – 402 c. – ISBN: 978-1-80461-619-2

8. Zhang S., Duan Y. Modern Log Analytics and Anomaly Detection. – Singapore: Springer, 2024. – 228 c. – ISBN: 978-981-99-6147-2

9. Sawhney R. Practical MLOps for DevOps Engineers. – Birmingham: Packt Publishing, 2023. – 342 c. – ISBN: 978-1-80324-845-5.

10. Kotsiantis S., Kanellopoulos D., Pintelas P. Fundamentals of Machine Learning: A Practical Approach. – London: Springer, 2023. – 289 c. – ISBN: 978-3-031-34123-7

Abstract. *Automated testing of web applications is a crucial aspect of modern software quality assurance processes, especially in large and complex projects where hundreds or even thousands of scenarios must be verified within tight timeframes. One of the critical tasks during automated test execution is the collection, processing, and analysis of technical messages (logs). A log in this context is a text string inserted by a developer into the code to record exceptions or errors during execution. Such messages contain key information about the test execution flow, detected errors, and the behavior of the system and its individual components.*

They serve as an important source for determining the priority of defects found in the product. Establishing the order in which identified defects should be addressed is vital for efficient remediation. However, manual log prioritization analysis in modern systems is highly labor-intensive, subjective, and prone to overlooking important details, which may delay the identification of critical issues and impact overall product reliability.

As data volumes increase, so does the need for timely remediation of potential flaws. This highlights the necessity of developing intelligent systems for log analysis that can automatically determine message priority and identify the most significant entries. Despite their advantages, classical AI approaches require large training datasets or rely on complex neural architectures. These approaches often prove inefficient in environments with limited data, high log variability, and a need for transparent decision rationale.

This paper introduces the author's method LIPSI (Log-based inference of priority in software issues), an intelligent log analysis approach based on multi-stage feature evaluation without training datasets. The method employs keyword heuristics, statistical log characteristics, and dynamic weight modeling adapted to the current execution context. In testing, the method demonstrates high accuracy in prioritizing software issues even under unstable test scenarios and with minimal resources. The proposed approach combines ease of implementation, adaptability to environmental changes, and effectiveness in identifying critical errors.

Key words: *intelligent log analysis, automated testing, bug classification, logistic regression, dynamic weight modeling, log files, LIPSI, defect prioritization, artificial intelligence in testing, test result analysis.*

Науковий керівник: к.ф.-м.н., доц. Шевченко В.П.

Статтю надіслано: 12.10.2025