Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «Дослідження методів прогнозування появи помилки в програмному коді»
за освітньою програмою **12 Інженерія програмного забезпечення**
зі спеціальності: **121 Інженерія програмного забезпечення**

Виконав: студент групи ПЗ2226: /Тетяна БЕРДНИК/

Керівник: /Олександра ГОРБОВА/

Нормоконтролер: /Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis
master

on the topic: « A study of methods for predicting the appearance of an error in the program code»

according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2226: ___ ____ /Tetiana BERDYK/

Scientific Supervisor: _____/ Oleksandra HORBOVA/

Normative controller: _____ / Svitlana VOLKOVA/

Dnipro – 2024

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем

Кафедра: Комп’ютерні інформаційні технології

Рівень вищої освіти: магістр

Освітня програма: **12** Інженерія програмного забезпечення

Спеціальність: **121** Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри КІТ
_____ Вадим ГОРЯЧКІН
_____ грудня 2024 р.

ЗАВДАННЯ

На кваліфікаційну роботу Магістр

студенту Бердник Тетяні Володимирівні

1. Тема дипломної роботи: «Дослідження методів прогнозування появи помилки в програмному коді».
   Керівник роботи: Горбова Олександра Вікторовна
   затверджені наказом 224 ст від 23.03.23 року
2. Строк подання студентом роботи ___.___. 2023 року
3. Вихідні дані до дипломної роботи: _____.
4. Зміст пояснювальної записки (перелік питань до розробки):
   4.1. КРИТИЧНИЙ АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖЕННЯ ПРОБЛЕМАТИКИ ПОМИЛОК У ПРОГРАМНОМУ КОДІ;
   4.2. МЕТОДИ І ТЕХНІКИ ВИЯВЛЕННЯ ТА ПРОГНОЗУВАННЯ ПОМИЛОК У ПРОГРАМНОМУ КОДІ;

4.3. ПРОЕКТУВАННЯ Й РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТОХАСТИЧНИХ АЛГОРИТМІВ;

4.4. АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ПРОГНОЗУВАННЯ ПОМИЛОК У ПРОГРАМНОМУ КОДІ

5. Перелік демонстраційного матеріалу:

5.1. Презентація «Дослідження методів прогнозування появи помилки в програмному коді»

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1	Вступ	01.11.23 – 03.11.23	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	06.11.23 – 10.11.23	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	13.11.23 – 24.11.23	
4	Постановка задачі, технічне завдання	27.11.23 – 01.12.23	30%
5	Техніко-економічні показники	04.12.23 – 08.12.23	
6	Розробка інструментальних засобів дослідження	11.12.23 – 15.12.23	
7	Виконання досліджень	18.12.23 – 22.12.23	60%
8	Оформлення тез доповідей	25.12.23 – 26.12.23	
9	Оформлення статті у фаховий журнал	27.12.23 – 29.12.23	
10	Оформлення пояснювальної записки	01.01.24– 08.01.24	
11	Розробка демонстраційних матеріалів	09.01.24 – 10.01.24	100%
12	Подання кваліфікаційної роботи до кафедри	09.01.24 – 10.01.24	

13	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії		

Студент _____ /Тетяна БЕРДНИК/

Керівник роботи _____ /Олександра ГОБОВА/

РЕФЕРАТ

На сторінки реферат не треба ставити номер сторінки. Треба видалити

Об’єктом цього дослідження є процес розробки програмного забезпечення, зокрема, ті аспекти процесу, які пов'язані з виникненням та виявленням помилок у програмному коді.

Предметом дослідження є методи та техніки, що використовуються для прогнозування та виявлення помилок в програмному коді. Це включає в себе аналіз різних підходів та алгоритмів, використання інструментів статичного та динамічного аналізу, а також застосування методів штучного інтелекту та машинного навчання для прогнозування потенційних дефектів і вразливостей у програмному коді.

Метою роботи є розробка ефективного методу для прогнозування появи помилок у програмному коді, що сприятиме підвищенню якості та надійності ПЗ.

Методи дослідження: математичне моделювання, штучний інтелект та машинне навчання.

Результати та їх новизна: удосконалено методику прогнозування помилок у програмному коді за допомогою машинного навчання, використовуючи алгоритм логістичної регресії із стохастичним спуском. Результати дослідження можуть бути використані у навчальних закладах для підготовки фахівців у сфері ІТ.

Пояснювальна записка складається зі вступу, 4 розділів, висновків, бібліографічного списку та 2 додатків:

– у вступі описується сутність розробки, її актуальність. Складається із 4 сторінок;

– у першому розділі проведено аналіз сучасного стану дослідження проблеми на основі наукових літературних джерел та оцінці сучасного стану програмно-апаратного забезпечення. Складається з 21 сторінки;

– у другому розділі обґрунтовано вибір експериментального методу дослідження. Складається з 21 сторінки;

– у третьому розділі представлено проектування та розробку інструментального забезпечення для дослідження. Складається з 32 сторінок;

– у четвертому розділі проведені дослідження та проаналізовано їх результати. Складається з 14 сторінок;

– додатки містять скрипт створення бази даних та лістинги розробленого програмного забезпечення.

Таблиць – 5, рисунків – 33, бібліографія – 42 джерела.

Ключові слова: прогнозування помилок , код, програмне забезпечення, машинне навчання, тестування, логістична регресія із стохастичним спуском. Виправити розмір абзацу у всій роботі. Абзац=0,7

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ..... 7
ВСТУП 8
1 КРИТИЧНИЙ АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖЕННЯ ПРОБЛЕМАТИКИ ПОМИЛОК У ПРОГРАМНОМУ КОДІ..... 13
1.1 Комплексний аналіз літературних джерел 13
1.1.1 Історія виникнення та еволюція проблеми помилок у програмному коді..... 13
1.1.2 Сучасні дослідження в області виявлення та прогнозування помилок 14
1.1.3 Аналіз невирішених питань у сфері прогнозування помилок 17
1.1.4 Критичний огляд існуючих методів та інструментів..... 20
1.1.5 Обґрунтування вибору напряму дослідження та формулювання завдань..... 28
1.2 Аналіз сучасного стану програмно-апаратного забезпечення 30
1.2.1 Використання програмно-апаратних рішень у виявленні помилок 30
1.2.2 Оцінка потенціалу сучасних інструментів прогнозування помилок.... 31
1.2.3 Переваги та недоліки існуючих програмно-апаратних комплексів 33
1.2.4 Визначення прогалин у функціоналі та ефективності існуючих систем 35
Висновки до розділу 1 36
2 МЕТОДИ І ТЕХНІКИ ВИЯВЛЕННЯ ТА ПРОГНОЗУВАННЯ ПОМИЛОК У ПРОГРАМНОМУ КОДІ 38
2.1 Теоретичний аналіз методів прогнозування 38
2.1.1 Класифікація методів прогнозування помилок 38
2.1.2 Критичний аналіз методів прогнозування 43
2.2 Аналіз програмно-апаратних засобів..... 45
2.2.1 Інструменти статичного аналізу коду..... 45
2.2.2 Інструменти динамічного аналізу коду 46

2.2.3 Оцінка можливостей сучасних інтегрованих середовищ розробки (IDE) 47
2.3 Методи оцінки якості та надійності програмного коду 48
2.3.1 Метрики якості програмного коду 49
2.3.2 Моделі надійності програмного забезпечення 51
2.3.3 Верифікація та валідація програмного коду 52
2.4 Вибір та обґрунтування методу дослідження для розробки ПЗ 54
2.4.1 Вибір методу на основі аналізу вимог до проекту 54
2.4.2 Адаптація вибраного методу для специфіки проекту 56
Висновки до розділу 2 59
3 ПРОЕКТУВАННЯ Й РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТОХАСТИЧНИХ АЛГОРИТМІВ. 61
3.1 Формалізація задачі 61
3.2 Базова архітектура системи 66
3.3 Внутрішнє проектування 68
3.3.1 Вибір мови програмування 68
3.3.2 Технологічна платформа 71
3.3.3 Ієрархія та взаємодія класів системи 72
3.3.4 Використані принципи проектування 79
3.3.5 Використані шаблони проектування 79
3.4 Розробка інтерфейсу користувача 81
3.4.1 Розробка структури екранів системи 81
3.4.2 Реалізація інтерфейсу користувача 84
3.5 Тестування та налагодження програми 90
3.5.1 Аналіз методів тестування та відлагодження 90
3.5.2 Тестування системи методом покриття операторів 91
3.5.3 Тестування системи методом припущення про похибку 93
Висновки до розділу 3 95
4 АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ПРОГНОЗУВАННЯ ПОМИЛОК У ПРОГРАМНОМУ КОДІ 97

4.1 Підготовка до експерименту 97
4.1.1 Опис використаного програмно-апаратного середовища 97
4.1.2 Опис підходів для визначення точності та повноти методів прогнозування 98
4.1.3 Вплив налаштувань середовища виконання на результати прогнозування 99
4.1.4 Вплив зовнішніх факторів на результати прогнозування 100
4.2 Проведення експерименту 101
4.3 Аналіз результатів експерименту 109
Висновки до розділу 4 112
ВИСНОВКИ 114
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 118
ДОДАТКИ ..... **Помилка! Закладку не визначено.**
Додаток А. Скрипти формування бази даних ..... **Помилка! Закладку не визначено.**
Додаток Б. Лістинги програмного коду ..... **Помилка! Закладку не визначено.**

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ - програмне забезпечення.

IDE - інтегроване середовище розробки (Integrated Development Environment).

Статичний аналіз - процес виявлення помилок та вразливостей у програмному коді без його виконання.

Динамічний аналіз - процес тестування та аналізу програмного коду в процесі його виконання.

Метрики якості коду - кількісні показники, що відображають рівень якості програмного коду.

Моделі надійності ПЗ - математичні моделі, що оцінюють і прогнозують надійність програмного забезпечення.

Верифікація коду - процес перевірки відповідності програмного коду встановленим вимогам та специфікаціям.

Валідація коду - процес підтвердження, що програмний продукт відповідає потребам користувачів.

Тестування покриття операторів - методика тестування, яка оцінює, яка частина коду виконується під час тестування.

Метод припущення про похибку - тестування з метою виявлення потенційних місць виникнення помилок у програмному коді.

ВСТУП

У сучасному світі програмування відіграє ключову роль у розвитку інформаційних технологій, впливаючи на всі аспекти нашого життя. Однак, як і будь-яка інша людська діяльність, програмування схильне до помилок, які можуть призвести до серйозних наслідків, включаючи фінансові втрати, порушення роботи критичних систем та навіть загрози безпеці людей. У цьому контексті, розробка і впровадження ефективних методів прогнозування помилок у програмному коді стає особливо важливою задачею.

Актуальність теми. Збільшення складності програмного забезпечення, розширення його функціоналу та зростання вимог до його надійності актуалізують проблему виявлення та усунення помилок на ранніх етапах розробки. Традиційні методи тестування вже не здатні ефективно впоратися з цим завданням у сучасних умовах [1]. Це призводить до зростаючої потреби у вдосконаленні методів прогнозування помилок, зокрема за допомогою штучного інтелекту, машинного навчання, аналізу великих даних та інших передових технологій. Розв'язання цієї задачі матиме велике значення не тільки для індустрії програмного забезпечення, але й для виробництва в цілому, особливо в Україні, де IT-сектор активно розвивається та займає одну з провідних позицій у економіці країни.

На основі аналізу існуючих підходів до виявлення та усунення помилок у програмному коді, а також з огляду на швидкий розвиток сучасних технологій, виникає потреба у розробці нових, більш ефективних методів прогнозування помилок. Це дослідження має на меті розробку таких методів, зокрема з використанням алгоритмів машинного навчання та штучного інтелекту, які дозволять знизити ризики та витрати, пов'язані з помилками в програмному коді, та підвищити ефективність процесу розробки ПЗ.

Розробка ефективних методів прогнозування помилок у програмному коді має велике значення для підвищення якості програмного забезпечення та зменшення витрат, пов'язаних з його розробкою та обслуговуванням. Враховуючи швидкий розвиток IT-сектору в Україні, такі дослідження можуть

суттєво вплинути на подальший розвиток цієї галузі, забезпечуючи її конкурентоспроможність на світовому ринку.

Об’єкт дослідження - це процес розробки програмного забезпечення, зокрема, ті аспекти процесу, які пов'язані з виникненням та виявленням помилок у програмному коді. Це включає в себе різні стадії розробки програмного забезпечення, від концепції та дизайну до реалізації, тестування та обслуговування, а також інструменти та методики, що використовуються на цих стадіях для забезпечення якості та надійності коду.

Предметом дослідження є методи та техніки, що використовуються для прогнозування та виявлення помилок в програмному коді. Це включає в себе аналіз різних підходів та алгоритмів, використання інструментів статичного та динамічного аналізу, а також застосування методів штучного інтелекту та машинного навчання для прогнозування потенційних дефектів і вразливостей у програмному коді.

Мета дослідження полягає у розробці ефективного методу для прогнозування появи помилок у програмному коді, що сприятиме підвищенню якості та надійності програмного забезпечення. Поставлена мета передбачає зниження ризиків та витрат, пов'язаних з помилками в програмному коді, та підвищення ефективності процесу розробки ПЗ.

Для досягнення цієї мети визначені наступні основні задачі:

– критичний аналіз сучасного стану дослідження проблематики помилок у програмному коді. Включає у себе комплексний огляд історії виникнення та еволюції проблеми помилок у програмному коді, аналіз сучасних досліджень, оцінку невирішених питань, критичний огляд існуючих методів та інструментів, а також обґрунтування вибору напряму дослідження;

– розробка методу виявлення та прогнозування помилок у програмному коді. Включає теоретичний аналіз класифікації та критичний аналіз методів прогнозування помилок, аналіз програмно-апаратних засобів, включаючи інструменти статичного та динамічного аналізу коду, а також оцінку якості та надійності програмного коду;

– проектування та розробка інструментального забезпечення для дослідження ефективності методів. Ця задача передбачає формалізацію задачі, розробку базової архітектури системи, внутрішнє проектування, вибір мови програмування, технологічної платформи, шаблонів проектування та розробку інтерфейсу користувача;

– аналіз ефективності методів прогнозування помилок у програмному коді. Включає підготовку та проведення експерименту, аналіз використаного програмно-апаратного середовища, визначення точності та повноти методів прогнозування, впливу зовнішніх факторів на результати прогнозування та аналіз результатів експерименту.

Для досягнення мети дослідження у роботі на тему «Дослідження методів прогнозування появи помилки в програмному коді» були використані наступні методи:

– методи математичного моделювання. Використовувались для створення теоретичних моделей, що дозволяють прогнозувати появу помилок у програмному коді. Ці моделі базуються на статистичних даних та теорії ймовірностей і дозволяють здійснити кількісну оцінку ризиків появи помилок;

– методи штучного інтелекту та машинного навчання. Використовувались для розробки і вдосконалення алгоритмів прогнозування помилок. Включає в себе використання нейронних мереж, алгоритмів класифікації та інших технік машинного навчання для аналізу програмного коду та ідентифікації потенційних вразливостей.

Кожен з цих методів мав свою роль у дослідженні та сприяв всебічному вивченню проблеми прогнозування помилок у програмному коді, дозволяючи ефективно досягти поставленої мети.

У ході дослідження були отримані наступні результати, які мають значний науковий внесок та інноваційний потенціал:

– удосконалено методику прогнозування помилок у програмному коді за допомогою машинного навчання, використовуючи алгоритм логістичної

регресії із стохастичним спуском (SdcaLogisticRegression) в середовищі ML.NET [2];

– розширено можливості бінарної класифікації в програмному коді, застосовуючи передові методи обробки даних. Це включає виявлення наявності помилок у коді, що є значним вдосконаленням порівняно зі стандартними методами статичного аналізу;

– вперше реалізовано практичне застосування машинного навчання для оцінки ймовірності помилок на основі навченої моделі. Це дозволяє переходити від теоретичного аналізу до практичного використання методів прогнозування, забезпечуючи високу точність та оперативність у виявленні потенційних помилок.

Отримані результати дослідження мають велике практичне значення для галузі розробки програмного забезпечення, зокрема у контексті підвищення якості та надійності програмних продуктів. Інноваційні методики та інструменти, розроблені у ході дослідження, можуть бути впроваджені на підприємствах та в організаціях, що займаються розробкою ПЗ, для виявлення та усунення помилок на ранніх стадіях розробки, що істотно знижує витрати на підтримку та виправлення помилок після запуску продукту.

Застосування розроблених алгоритмів та інструментів дозволить значно покращити процеси тестування програмного забезпечення, зробивши їх більш автоматизованими та точними. Це, у свою чергу, підвищує ефективність роботи тестувальників та розробників, зменшуючи час, необхідний на виявлення та виправлення помилок.

Апробація результатів дослідження. Основні положення магістерської роботи доповідалися та були схвалені на одинадцятій щорічній міжнародній науково-практичній конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті» (Дніпро, УДУНТ, 13-14 грудня 2023 року) та семінарах кафедри КІТ (протягом жовтня -листопада 2023 р.).

Публікації за темою роботи. За результатами роботи опубліковано 1 наукова праця: тези доповідей на міжнародній науковій конференції] [слід вказати посилання на свої праці в бібліографічному списку].

Впровадження розроблених методів та інструментів на підприємствах та в навчальних процесах не тільки підвищить якість програмного забезпечення, але й сприятиме підготовці більш кваліфікованих фахівців у сфері програмування, що має велике значення для подальшого розвитку IT-галузі.

1 КРИТИЧНИЙ АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖЕННЯ ПРОБЛЕМАТИКИ ПОМИЛОК У ПРОГРАМНОМУ КОДІ

1.1 Комплексний аналіз літературних джерел

1.1.1 Історія виникнення та еволюція проблеми помилок у програмному коді

Історія виникнення та еволюція проблеми помилок у програмному коді тісно пов'язана з розвитком комп'ютерних технологій. У 1940-60-х роках, на зорі комп'ютерної ери, програмування здійснювалося переважно на машинному коді або асемблері. Обмеження апаратного забезпечення та відсутність розвинутих мов програмування призводили до частих помилок. Історичною віхою став випадок з Грейс Гоппер, яка виявила «помилку» у вигляді молі у ранньому комп'ютері, що призвело до появи терміну «debugging» [3].

З появою вищих мов програмування, таких як FORTRAN, COBOL, а пізніше C у 1960-80-х роках, програмісти отримали більше можливостей для реалізації складних проектів. Однак, зі зростанням складності програм, зросла і кількість помилок, вказуючи на необхідність більш уважного підходу до написання та тестування коду [4].

1990-ті роки принесли об'єктно-орієнтоване програмування з мовами, такими як Java та C++, а також широке розповсюдження Інтернету, що ще більше підвищило доступність та популярність програмування. В цей період помилки у програмному коді почали мати значно серйозніші наслідки, зокрема у фінансовому та соціальному аспектах [5-6].

З настанням 2000-х років та розвитком таких технологій, як хмарні обчислення, мобільні технології та Інтернет речей, програмне забезпечення стало ще більш складним. Це породило нові виклики у сфері виявлення та управління помилками у коді, адже помилки стали мати ще більш широкий вплив, включаючи питання безпеки та надійності систем.

Відповідно, індустрія розробки програмного забезпечення змушена була адаптуватися, розвиваючи більш складні та ефективні методи виявлення та

виправлення помилок, від статичного аналізу коду до використання передових технологій машинного навчання. Це призвело до виникнення нових професійних ролей, таких як тестувальники програмного забезпечення та інженери якості, а також до зростання значення автоматизації та неперервного інтеграційного тестування в процесах розробки.

1.1.2 Сучасні дослідження в області виявлення та прогнозування помилок

Сучасні дослідження в області виявлення та прогнозування помилок у програмному коді зосереджуються на розробці більш ефективних і авангардних методів та інструментів. Це напрямок активно розвивається у відповідь на зростаючу складність програмних систем і високі вимоги до якості та безпеки програмного забезпечення.

Одним із ключових напрямків є використання методів штучного інтелекту та машинного навчання. Дослідники розробляють алгоритми, які можуть аналізувати великі обсяги коду, виявляючи в ньому потенційні помилки та вразливі місця, засновані на попередньо накопичених даних про помилки і їх контекст. Ці методи дозволяють значно збільшити швидкість і точність виявлення помилок порівняно з традиційними підходами.

Дослідження [7] розглядає складність виявлення частин програмного коду, які найбільш схильні до помилок, перш ніж вони стануть критичними. Основним завданням дослідження є розробка та оптимізація методів, які можуть ефективно визначати потенційні дефекти в коді. Особливу увагу приділяється різноманітним технікам глибинного навчання, які використовуються для аналізу та прогнозування дефектів у програмному коді. Ці методи включають в себе складні алгоритми, які здатні аналізувати великі масиви даних та ідентифікувати закономірності та аномалії, які можуть свідчити про наявність дефектів.

Однією з ключових переваг використання глибинного навчання у цій області є його здатність до самонавчання та адаптації. Моделі глибинного

навчання можуть постійно вдосконалюватися, аналізуючи нові дані, тим самим підвищуючи точність та надійність прогнозування дефектів.

У дослідженні також розглядаються різні підходи до використання глибокого навчання, включаючи різні типи нейронних мереж та стратегії навчання. Важливою частиною є аналіз та порівняння різних методів, щоб визначити, які з них найбільш ефективні у виявленні дефектів коду. Завдяки цьому дослідженню, можливо визначити кращі практики та найефективніші методи для прогнозування дефектів у програмному забезпеченні, що забезпечить більш надійну та якісну розробку програмних продуктів.

Дослідження [8], яке використовує метрики коду з репозиторію Promise для прогнозування дефектів у програмному забезпеченні, є інноваційним підходом у сфері виявлення та управління якістю програмного забезпечення. Центральною частиною цього дослідження є застосування штучних імунних систем (Artificial Immune Systems, AIS) для визначення та прогнозування дефектів.

Штучні імунні системи (AIS) - це клас обчислювальних систем, натхненних біологічними імунними системами, які використовуються для рішення складних завдань, таких як класифікація та виявлення аномалій. У контексті цього дослідження, AIS застосовуються для ідентифікації потенційно дефектних частин коду в різних версіях та проектах програмного забезпечення.

Дослідники випробували вісім варіантів AIS, включаючи CSCA та CLONALG, на різних релізах одинадцяти проектів. Це дозволило оцінити ефективність різних AIS моделей у контексті різних проектів та версій програмного забезпечення. Використання AIS дозволяє моделювати процес виявлення дефектів, схожий на той, як імунна система виявляє та реагує на патогени. Аналіз різних варіантів AIS та їх застосування у різних умовах надає цінну інформацію щодо того, які системи найбільш ефективні для певних типів проектів або версій. Особливо цінним є порівняльний аналіз різних AIS, оскільки він дає змогу визначити оптимальні стратегії та налаштування для конкретних сценаріїв використання.

Це дослідження відіграє важливу роль у розвитку методів прогнозування дефектів, надаючи можливості для покращення якості та надійності програмного забезпечення. Воно також демонструє важливість інноваційних підходів, таких як AIS, у вирішенні складних завдань у сфері розробки програмного забезпечення.

Емпіричне дослідження [9], що зосереджується на використанні CodeBERT для прогнозування дефектів у програмному забезпеченні, представляє собою значний крок у розвитку інструментів аналізу програмного коду. CodeBERT — це нейронна мовна модель, спеціально адаптована для роботи з кодом, що поєднує в собі переваги глибокого навчання зі специфікою обробки програмного коду. Це дослідження виконує емпіричні експерименти з використанням CodeBERT для прогнозування дефектів як у межах однієї версії програмного забезпечення, так і між різними проектами. Головне питання, на яке шукає відповідь дослідження, — чи може використання такої моделі, як CodeBERT, покращити точність та ефективність виявлення потенційних дефектів у програмному коді.

Важливим аспектом є аналіз різних шаблонів прогнозування. Це дозволяє визначити, які підходи є найбільш ефективними для різних типів проектів та як різні методи прогнозування впливають на загальну продуктивність моделі. Особлива увага приділяється вивченню того, як модель обробляє контекстуальну інформацію, вбудовану в код, та як це впливає на здатність моделі точно ідентифікувати потенційні проблеми.

Використання CodeBERT у цьому контексті відкриває нові можливості для підвищення якості програмного забезпечення. Воно демонструє, як сучасні техніки машинного навчання та обробки природної мови можуть бути адаптовані для специфічних потреб розробки програмного забезпечення. Це дослідження також надає інсайти щодо того, як різні фактори, такі як контекст коду та типи дефектів, можуть впливати на ефективність прогнозування, та як це можна використовувати для розробки більш точних і надійних інструментів виявлення помилок.

Враховуючи ці напрямки, сучасні дослідження в області виявлення та прогнозування помилок у програмному коді є багатогранними та включають широкий спектр методів і технологій, спрямованих на поліпшення якості, надійності та безпеки програмного забезпечення.

Використання глибинного навчання, штучних імунних систем, та нейромережевих мовних моделей, таких як CodeBERT, у прогнозуванні дефектів програмного забезпечення відкриває нові можливості для підвищення точності та ефективності виявлення помилок. Ці методи дозволяють обробляти великі обсяги даних та ідентифікувати складні залежності, що забезпечує більш точне та швидке виявлення потенційних дефектів. Використання цих передових технологій сприяє розвитку більш надійного та якісного програмного забезпечення, зменшуючи час та витрати на виявлення та виправлення помилок.

1.1.3 Аналіз невирішених питань у сфері прогнозування помилок

Аналіз невирішених питань у сфері прогнозування помилок у програмному коді виявляє декілька ключових викликів, які залишаються актуальними у цій галузі.

Висока складність сучасного програмного коду та програмних систем є однією з найбільших викликів у сфері виявлення та прогнозування помилок. У міру розвитку технологій, програмні продукти еволюціонували до високоінтегрованих та складних систем, які включають множину взаємодій між різними компонентами, мовами програмування та технологічними стеками.

Сучасні програмні системи часто включають в себе різноманітні модулі, бібліотеки та залежності, які можуть бути розроблені різними командами, іноді з використанням різних мов програмування та підходів до розробки. Це робить код більш складним для аналізу, адже помилки можуть виникати не тільки через проблеми в окремих лінійках коду, але й через складні взаємодії між різними частинами системи.

Окрім того, інтеграція з зовнішніми системами та сервісами, такими як бази даних, веб-сервіси та хмарні платформи, вносить додаткову складність. Взаємодія з такими зовнішніми компонентами може призвести до непередбачуваних помилок, особливо якщо зовнішні системи оновлюються або змінюються без попередження.

Конкурентність та паралелізм у програмному коді додають важливий шар складності у процес виявлення та управління помилками. Цей аспект стає особливо важливим у сучасних багатопотокових та асинхронних системах, де програми або компоненти працюють одночасно та незалежно один від одного.

Однією з ключових проблем у таких системах є умови гонки (race conditions), які виникають, коли декілька процесів або потоків одночасно намагаються доступитися або змінити спільні дані. Якщо ці процеси не належним чином синхронізовані, результати виконання можуть бути непередбачуваними та вести до некоректної поведінки системи. Умови гонки можуть бути важко виявити та діагностувати, оскільки вони можуть не проявлятися у всіх випадках виконання програми, а лише за певних умов чи конкретного порядку операцій.

Іншою проблемою є проблеми синхронізації. У багатопотокових додатках необхідно забезпечити правильну координацію між потоками для уникнення взаємоблокувань (deadlocks), коли два або більше потоків взаємно блокують один одного, очікуючи на ресурси, захоплені іншими потоками. Також важливо уникнути надмірної синхронізації, яка може знижувати ефективність та продуктивність додатку [10].

Крім того, асинхронне програмування, що стало широко використовуватися для підвищення реактивності та ефективності програм, вносить додаткові виклики. Асинхронні операції ускладнюють потік виконання програми, роблячи її поведінку менш передбачуваною та ускладнюючи відстеження виникнення помилок.

Обмеження існуючих інструментів та методів для виявлення помилок є значною проблемою у сфері програмування. Інструменти, які зараз

використовуються у промисловості, часто спеціалізуються на певних типах помилок, але не завжди ефективно справляються з широким спектром потенційних проблем, особливо у складних системах. Це може включати обмежену здатність виявлення логічних помилок, помилок, пов'язаних з паралелізмом та конкурентністю, а також помилок, що виникають у специфічних сценаріях використання системи.

Інструменти статичного аналізу, наприклад, ефективні у виявленні синтаксичних помилок та деяких типів помилок використання ресурсів, але вони можуть не виявляти складніші помилки, що виникають тільки під час виконання програми. Це означає, що для повного покриття потенційних проблем потрібно використовувати комбінацію різних інструментів і методик.

Що стосується методів, заснованих на машинному навчанні та штучному інтелекті, то їх ефективність часто залежить від обсягу та якості доступних даних. Великі набори даних із прикладами помилок необхідні для того, щоб системи машинного навчання могли ефективно «навчитися» виявляти подібні помилки у майбутньому. Проте, збір достатнього обсягу релевантних та різноманітних даних може бути складним завданням.

Крім того, існує ризик, що системи, навчені на обмеженому або застарілому наборі даних, можуть не виявляти нові типи помилок або помилки, що виникають у нових технологічних контекстах. Це вимагає постійного оновлення та адаптації навчальних наборів даних, а також розробки більш гнучких алгоритмів машинного навчання.

Враховуючи ці обмеження, стає очевидним, що потрібні подальші дослідження та розробки у сфері виявлення та прогнозування помилок. Це включає в себе не тільки розробку нових інструментів та підходів, але й постійне оновлення та адаптацію існуючих методів до змінюваних умов та нових викликів.

1.1.4 Критичний огляд існуючих методів та інструментів

Критичний огляд існуючих методів та інструментів для виявлення та прогнозування помилок у програмному коді виявляє як сильні, так і слабкі сторони сучасних підходів.

Інструменти статичного аналізу коду є одними з найбільш поширених засобів для виявлення помилок. Вони дозволяють швидко виявити синтаксичні та деякі семантичні помилки без необхідності виконання програми. Однак, обмеження цих інструментів полягає в тому, що вони не завжди ефективні у виявленні складніших помилок логіки або взаємодії між різними частинами програми. Також статичний аналіз може призводити до значної кількості хибнопозитивних результатів, що вимагає додаткової перевірки з боку розробників.

Інструменти статичного аналізу коду мають ряд переваг та недоліків, які важливо розглянути для їх ефективного використання в процесі розробки програмного забезпечення (рис. 1.1).

Рисунок 1.1 – Переваги та недоліки інструментів статичного аналізу

Переваги інструментів статичного аналізу:

– раннє виявлення помилок. Статичний аналіз дозволяє виявити помилки на ранніх стадіях розробки, що може заощадити час та витрати на їх виправлення у майбутньому;

– економія часу та ресурсів. Автоматизація процесу пошуку помилок допомагає зменшити необхідність ручного перегляду коду та збільшує продуктивність розробників;

– покращення якості коду. Статичний аналіз сприяє підвищенню стандартів якості коду, вказуючи на потенційні проблемні місця, такі як використання застарілих практик або неправильне використання мовних конструкцій;

– масштабованість. Інструменти статичного аналізу ефективно справляються з великими обсягами коду, що робить їх ідеальними для великих проектів та команд;

– підтримка безперервної інтеграції. Ці інструменти легко інтегруються з системами безперервної інтеграції, що дозволяє автоматично перевіряти код на наявність помилок під час його розробки.

Недоліки інструментів статичного аналізу:

– обмеження у виявленні складних помилок. Статичний аналіз може не виявляти складніші помилки, пов'язані з логікою або взаємодією між різними компонентами системи;

– хибнопозитивні результати. Інструменти можуть іноді визначати коректний код як помилковий, що вимагає додаткової перевірки з боку розробників;

– обмежена здатність до контекстного аналізу. Статичний аналіз не завжди може правильно інтерпретувати контекст, у якому використовується код, особливо у більш складних системах;

– необхідність налаштування. Інструменти статичного аналізу часто вимагають налаштування під специфіку проекту, що може бути трудомістким;

– не виявляє помилок, які виникають тільки під час виконання. Статичний аналіз не може виявити помилки, що проявляються тільки під час виконання програми, такі як помилки використання пам'яті або проблеми з конкурентним доступом.

Враховуючи ці переваги та недоліки, статичний аналіз краще використовувати у поєднанні з іншими методами перевірки якості коду, такими як динамічний аналіз, ручне тестування та рецензування коду [11].

Динамічні методи аналізу, які включають тестування програми під час її виконання, можуть виявити помилки, які статичний аналіз пропускає. Однак, ефективність таких методів значною мірою залежить від якості тестових сценаріїв та вхідних даних, а також від обсягу покриття коду тестами.

Динамічні методи аналізу, які включають тестування програмного забезпечення в реальному часі під час його виконання, також мають свої переваги та недоліки (рис. 1.2).

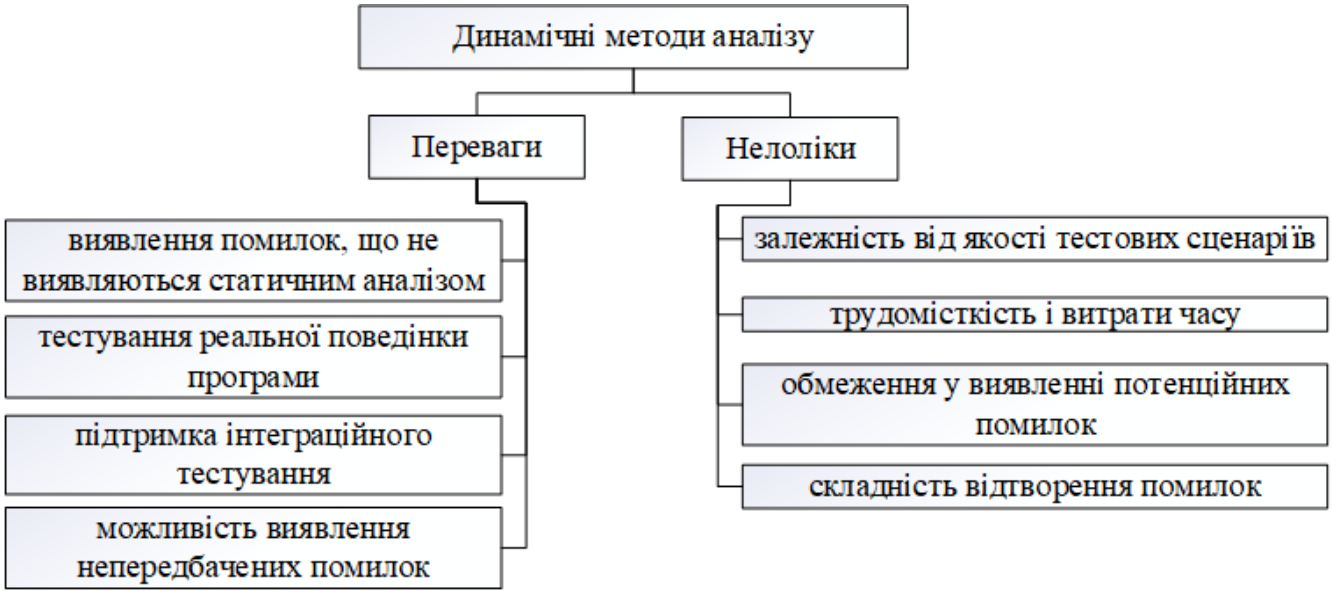

Рисунок 1.2 – Переваги та недоліки інструментів динамічного аналізу

Переваги динамічних методів аналізу:

– виявлення помилок, що не виявляються статичним аналізом. Динамічний аналіз може виявити помилки, які можливо тільки під час виконання програми, такі як помилки доступу до пам'яті, проблеми синхронізації у

багатопотокових програмах та інші випадки, які статичний аналіз може пропустити;

– тестування реальної поведінки програми. Динамічний аналіз дозволяє перевірити, як програма буде вести себе в різних сценаріях, що надає реалістичніше уявлення про її поведінку та надійність;

– підтримка інтеграційного тестування. Ці методи особливо ефективні для тестування взаємодії між різними модулями або компонентами системи, дозволяючи виявляти помилки інтеграції;

– можливість виявлення непередбачених помилок. Динамічний аналіз часто включає тестування на основі реальних даних користувача або реалістичних сценаріїв використання, що може виявити помилки, на які не звертали увагу під час розробки.

Недоліки динамічних методів аналізу:

– залежність від якості тестових сценаріїв. Ефективність динамічного аналізу сильно залежить від охоплення та якості тестових випадків. Неповне або неякісне тестування може призвести до пропуску серйозних помилок;

– трудомісткість і витрати часу. На відміну від статичного аналізу, динамічний аналіз часто вимагає значних зусиль і часу на підготовку та виконання тестів, особливо для великих і складних систем;

– обмеження у виявленні потенційних помилок. Динамічний аналіз може виявляти лише помилки, що проявляються під час виконання тестів. Помилки, які виникають у не тестованих сценаріях або унікальних умовах, можуть залишатися непоміченими;

– складність відтворення помилок. Іноді помилки, виявлені під час динамічного аналізу, можуть бути складними для відтворення та діагностики, особливо якщо вони залежать від специфічних умов або зовнішніх факторів.

Враховуючи ці переваги та недоліки, динамічний аналіз є важливим компонентом загальної стратегії забезпечення якості програмного забезпечення, але він повинен використовуватися у комбінації з іншими методами, включаючи статичний аналіз та ручне тестування, для досягнення найкращих результатів.

Інструменти, засновані на машинному навчанні, відкривають нові горизонти у виявленні помилок у програмному коді, пропонуючи суттєві інновації порівняно з традиційними підходами. Основна сила цих інструментів полягає в їх здатності обробляти великі обсяги коду і виявляти складні, часто неочевидні шаблони та аномалії, які можуть вказувати на наявність помилок. Ця можливість значно підвищує шанси виявлення складних помилок, які могли б залишитися непоміченими під час ручного перегляду або за допомогою традиційних інструментів аналізу [12].

Однак, ефективність таких інструментів на машинному навчанні значною мірою залежить від обсягу, різноманітності та якості даних, на яких вони навчаються. Інструменти повинні мати доступ до обширних наборів даних з прикладами коду, включаючи як коректно написаний код, так і код з помилками, щоб навчитися ефективно ідентифікувати потенційні проблемні місця. Недостатність або невідповідність даних для навчання може призвести до того, що інструменти будуть виявляти помилки з меншою точністю або упускати їх.

Крім того, здатність цих інструментів адаптуватися до нових типів помилок або змін у парадигмах програмування є критично важливою. Світ програмування постійно розвивається, і методи, які були ефективними вчора, можуть виявитися недостатніми завтра. Тому інструменти машинного навчання мають бути здатні до швидкої адаптації, навчаючись на оновлених даних і враховуючи новітні тенденції та практики в програмуванні.

Інструменти, засновані на машинному навчанні, для виявлення та прогнозування помилок у програмному коді стають дедалі популярнішими завдяки своїм унікальним можливостям. Однак, як і будь-яка технологія, вони мають свої переваги та недоліки (рис. 1.3).

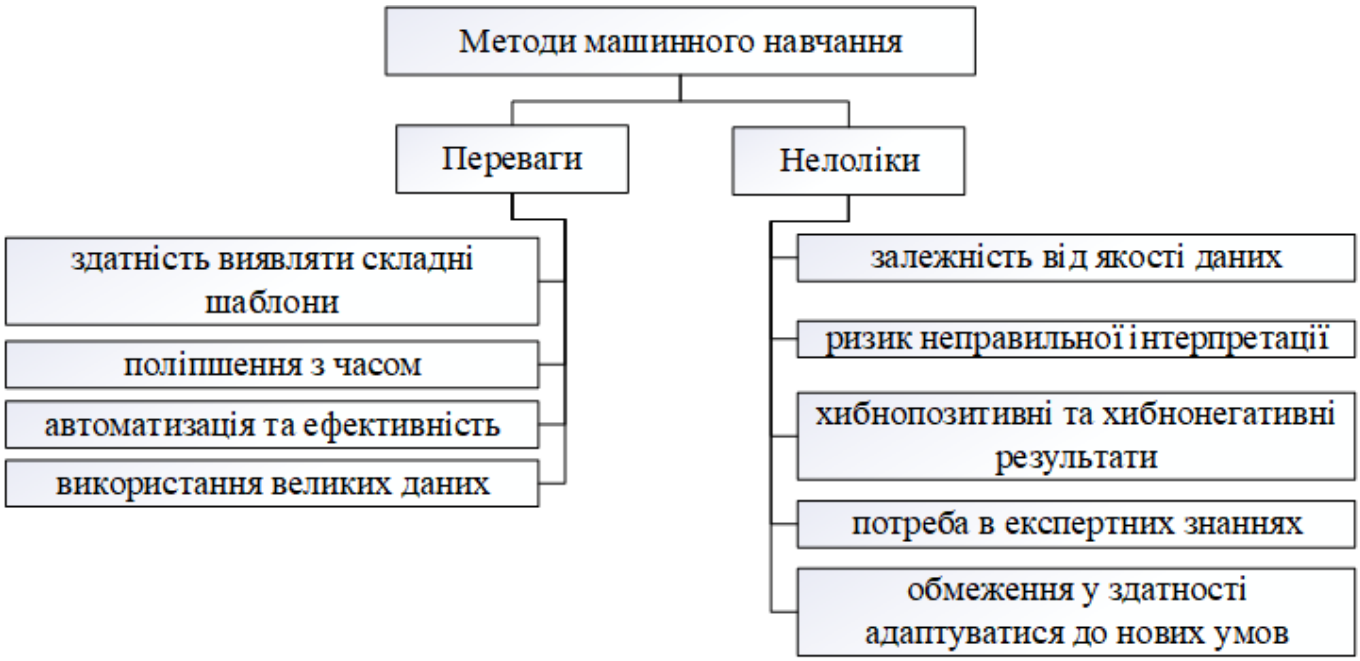

Рисунок 1.3 – Переваги та недоліки інструментів машинного навчання

Переваги інструментів, заснованих на машинному навчанні:

– здатність виявляти складні шаблони. Ці інструменти можуть виявляти складні та неочевидні шаблони помилок, які можуть бути не доступні для традиційних методів аналізу;

– поліпшення з часом. Інструменти на основі машинного навчання мають потенціал до самовдосконалення, оскільки вони навчаються на зростаючому обсязі даних і здатні адаптуватися до нових типів помилок;

– автоматизація та ефективність. Ці інструменти забезпечують високий рівень автоматизації процесу виявлення помилок, що може значно підвищити продуктивність розробників;

– використання великих даних. Машинне навчання ідеально підходить для аналізу великих обсягів коду та історичних даних, що дозволяє виявляти тенденції та закономірності, які можуть бути неочевидними при ручному аналізі.

Недоліки інструментів, заснованих на машинному навчанні:

– залежність від якості даних. Ефективність цих інструментів сильно залежить від якості та обсягу тренувальних даних. Некоректні або неповні дані можуть призвести до неточностей у виявленні помилок;

– ризик неправильної інтерпретації. Результати, отримані за допомогою машинного навчання, можуть бути складними для інтерпретації, що вимагає додаткових знань та зусиль для правильного розуміння та використання;

– хибнопозитивні та хибнонегативні результати. Як і у випадку з іншими методами аналізу, існує ризик отримання хибнопозитивних (помилково ідентифікованих як помилкові) або хибнонегативних (пропущених) результатів;

– потреба в експертних знаннях. Розробка та налаштування інструментів машинного навчання вимагають глибоких знань у галузі статистики, машинного навчання та програмування;

– обмеження у здатності адаптуватися до нових умов. Хоча інструменти машинного навчання можуть адаптуватися до змін, вони можуть не виявляти нові типи помилок, якщо не були навчені на відповідних даних.

Зважаючи на ці переваги та недоліки, інструменти на основі машинного навчання можуть бути ефективним доповненням до традиційних методів виявлення помилок, але їх слід використовувати з обережністю та у поєднанні з іншими підходами для забезпечення найкращого охоплення потенційних помилок [13].

Розробка і використання інтегрованих середовищ розробки (IDE) також зробила значний внесок у зменшення кількості помилок на ранніх етапах розробки. Сучасні IDE надають багато функцій для виявлення помилок, таких як підсвічування синтаксису, автоматичне доповнення коду, інтерактивне налагодження та інші корисні інструменти. Однак, вони можуть не повністю вирішити проблему виявлення більш складних помилок, особливо тих, які виникають у складних системах або в результаті неправильного використання фреймворків та бібліотек [14].

Отже, інтегровані середовища розробки відіграють ключову роль у процесі програмування, пропонуючи розробникам ряд інструментів та функцій для полегшення процесу розробки. Однак, як і будь-який інструмент, IDE мають свої переваги та недоліки (рис. 1.4).

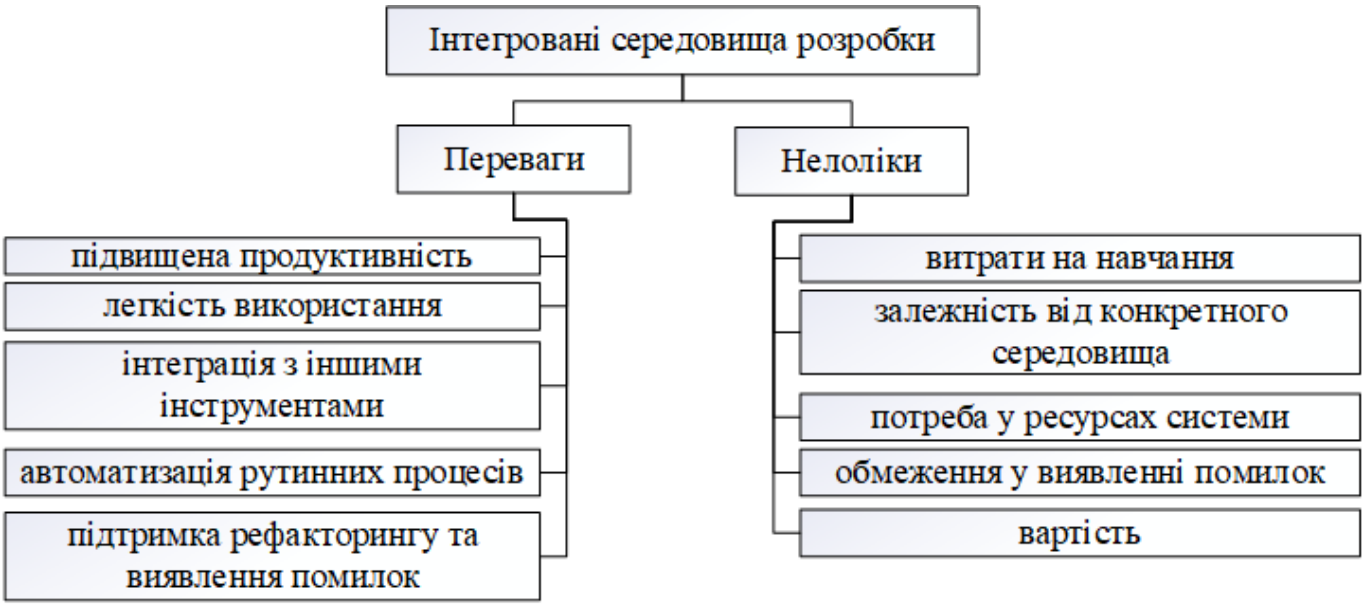

Рисунок 1.4 – Переваги та недоліки використання інтегрованих середовищ розробки

Переваги інтегрованих середовищ розробки:

– підвищена продуктивність. IDE забезпечують розробникам широкий спектр інструментів (редактор коду, компілятор, дебагер тощо) в одному середовищі, що сприяє підвищенню продуктивності;

– легкість використання. Зазвичай IDE мають користувальницький інтерфейс, який дозволяє легко навігувати по коду, управляти проектами та використовувати різні функції;

– інтеграція з іншими інструментами. Багато IDE можуть інтегруватися з іншими інструментами та сервісами, такими як системи контролю версій, бази даних та різноманітні фреймворки;

– автоматизація рутинних процесів. IDE можуть автоматизувати багато рутинних процесів, таких як компіляція, налагодження, деплоймент тощо;

– підтримка рефакторингу та виявлення помилок. Більшість сучасних IDE мають вбудовані інструменти для рефакторингу коду та виявлення помилок, що може значно полегшити відладку та покращення якості коду.

Недоліки інтегрованих середовищ розробки (IDE) [15]:

– витрати на навчання. Нові користувачі можуть витратити значний час на вивчення усіх функцій та особливостей конкретного IDE;

– залежність від конкретного середовища. Розробники можуть стати залежними від функцій та інтерфейсу певного IDE, що може ускладнити перехід на інші інструменти або середовища;

– потреба у ресурсах системи. Деякі IDE можуть бути вимогливими до системних ресурсів, особливо якщо вони включають велику кількість функцій та інструментів;

– обмеження у виявленні помилок. Хоча IDE можуть допомогти виявити деякі помилки, вони не завжди можуть виявити більш складні проблеми, які вимагають глибшого аналізу або використання спеціалізованих інструментів;

– вартість. Деякі потужні комерційні IDE можуть бути досить дорогими, особливо для невеликих команд або індивідуальних розробників.

Отже, використання IDE може значно полегшити та прискорити процес розробки, але важливо враховувати їх обмеження та вибирати інструменти, які найкраще відповідають конкретним потребам проекту та команди.

Хоча існуючі методи та інструменти забезпечують значну підтримку у виявленні та прогнозуванні помилок, вони мають певні обмеження та не можуть гарантувати повного виявлення всіх потенційних помилок. Це створює потребу в подальших дослідженнях та розвитку більш вдосконалених та інтегрованих підходів.

1.1.5 Обґрунтування вибору напряму дослідження та формулювання завдань

У сучасному світі програмування, визначення та прогнозування помилок у програмному коді стає все більш важливим завданням, що впливає на якість та ефективність програмних продуктів. В контексті цього завдання, необхідно розглянути кілька ключових аспектів, які підкреслюють важливість цього дослідження:

– зростаюча складність ПЗ. Сучасне програмне забезпечення характеризується високою складністю та інтеграцією, що збільшує ймовірність помилок у коді. Дослідження методів прогнозування цих помилок є критично важливим для підвищення якості та надійності програмних продуктів;

– важливість надійності та безпеки ПЗ. Помилки в програмному коді можуть призвести не тільки до втрати даних або зниження продуктивності, але й становити серйозні ризики для безпеки. Тому дослідження способів їх прогнозування є актуальним та важливим;

– потреба в ефективних методах раннього виявлення помилок. Чим раніше виявляється помилка у процесі розробки, тим менше витрат потрібно для її усунення. Розробка методів прогнозування може допомогти зменшити час та витрати на відладку і тестування.

Для забезпечення високої якості та надійності ПЗ ключовим аспектом є здатність ефективно прогнозувати та запобігати помилкам у програмному коді. Це дослідження має на меті вивчити та оцінити різні підходи та стратегії, які можуть бути використані у цьому процесі:

– аналіз існуючих методів прогнозування помилок у програмному коді. Вивчити та оцінити сучасні методи та інструменти, які використовуються для прогнозування помилок, їх ефективність та обмеження;

– дослідження впливу різних факторів на появу помилок у коді. Визначити, які фактори (наприклад, складність коду, використання певних мов програмування, стиль кодування) найбільш суттєво впливають на появу помилок;

– розробка та тестування нового методу прогнозування помилок. На основі аналізу існуючих методів та виявлених факторів розробити нові підходи або вдосконалити існуючі методи для більш ефективного прогнозування помилок;

– оцінка ефективності розроблених методів. Провести тестування та аналіз ефективності нових або вдосконалених методів на різних видах програмного забезпечення та в різних умовах.

Цей дослідницький напрямок має велике значення для подальшого розвитку сфери програмування, оскільки він сприяє покращенню якості програмного забезпечення, зниженню витрат на його розробку та підвищенню загальної надійності та безпеки програмних продуктів.

1.2 Аналіз сучасного стану програмно-апаратного забезпечення

1.2.1 Використання програмно-апаратних рішень у виявленні помилок

Використання програмно-апаратних рішень у процесі виявлення помилок стає необхідністю в контексті сучасної розробки програмного забезпечення. Цей комплексний підхід поєднує можливості програмного забезпечення та фізичного обладнання, створюючи більш потужні, точні та ефективні системи для виявлення та аналізу помилок. Центральним елементом такого підходу є використання високопродуктивного обладнання, здатного обробляти величезні масиви даних. Наприклад, сервери високої продуктивності та оптимізовані системи зберігання даних дозволяють швидко обробляти запити та аналізувати великі обсяги інформації, що є критично важливим для роботи з об'ємними базами даних або розгорнутими програмними системами.

Ці інструменти дозволяють виконувати складні операції обробки та аналізу, які були б неможливими або надто ресурсомісткими для стандартного програмного забезпечення. Використання спеціалізованого обладнання також дозволяє розробникам виконувати більш глибокі та всебічні аналізи помилок, що може включати складні алгоритми пошуку, симуляції різних сценаріїв та використання передових методів машинного навчання .

На програмному рівні використовуються різноманітні інструменти та платформи. Це може включати інструменти статичного аналізу коду, які автоматично перевіряють програмний код на наявність помилок синтаксису, стилістики кодування та інших потенційних проблем. Також застосовуються системи динамічного аналізу, які виконують програмне забезпечення в

контрольованому середовищі для виявлення помилок, що виникають під час виконання.

Крім того, важливу роль відіграють інструменти автоматизованого тестування. Вони дозволяють створювати, виконувати та аналізувати результати великої кількості тестів, значно збільшуючи охоплення тестування та зменшуючи ризик пропуску помилок. Автоматизовані тести особливо корисні для виявлення помилок у багатократно використовуваних компонентах та для проведення регресійного тестування.

Використання хмарних технологій також стає дедалі популярнішим у контексті виявлення помилок, оскільки хмарні платформи надають гнучкість та масштабованість, необхідні для ефективного тестування великих систем. Хмарні сервіси можуть надавати не тільки обчислювальні ресурси, але й спеціалізовані інструменти та сервіси для тестування та аналізу коду.

Отже, використання програмно-апаратних рішень для виявлення помилок дозволяє створити більш потужні, гнучкі та ефективні системи, здатні впоратися з викликами сучасної розробки програмного забезпечення. Це включає в себе не тільки підвищення точності виявлення помилок, але й зниження витрат часу та ресурсів, необхідних для їх усунення [16].

1.2.2 Оцінка потенціалу сучасних інструментів прогнозування помилок

Сучасний світ програмування вимагає різноманітних підходів та інструментів для ефективного виявлення та прогнозування помилок у програмному коді. Від статичного аналізу до складних систем на основі машинного навчання, кожен інструмент пропонує унікальні переваги та має свої обмеження. З огляду на складність та різноманітність програмних продуктів, важливо обрати найбільш відповідні методи для конкретного проекту. Водночас, використання комбінованого підходу, що включає різні інструменти, часто є ключем до максимально ефективного виявлення та запобігання помилок. Нижче

наведено порівняльну таблицю (табл. 1.1), яка відображає оцінку потенціалу сучасних інструментів прогнозування помилок, демонструючи їх сильні та слабкі сторони.

Таблиця 1.1 – Оцінка потенціалу інструментів прогнозування помилок

Критерій	Статичний аналіз	Динамічний аналіз	Машинне навчання	Автоматизоване тестування
Область застосування	Перевірка коду без виконання	Тестування коду під час виконання	Аналіз шаблонів і аномалій у великих наборах даних	Автоматичне виконання тестових сценаріїв
Типи виявлених помилок	Синтаксичні, стилістичні	Помилки часу виконання, контекстні помилки	Складні шаблони помилок	Помилки у функціональності і логіці
Швидкість обробки	Висока	Залежить від складності тестів	Помірна, залежить від обсягу даних	Залежить від складності тестів
Автоматизація	Висока	Помірна	Висока	Висока
Необхідність у ресурсах	Мінімальна	Помірна до високої	Висока	Помірна до високої
Потенціал адаптації	Обмежений	Обмежений	Високий	Обмежений
Ризик хибнопозитивних результатів	Високий	Низький	Залежить від якості даних	Низький

Кожен тип інструменту має свої сильні та слабкі сторони. Статичний аналіз є швидким і ефективним для виявлення стандартних помилок, але має обмеження у виявленні складніших проблем. Динамічний аналіз дає реальну картину поведінки програми, але може бути часомістким і вимагати більше ресурсів. Інструменти на основі машинного навчання пропонують значний потенціал для виявлення складних помилок, але їх ефективність залежить від

якості та обсягу даних. Автоматизовані інструменти тестування забезпечують високий рівень автоматизації і ефективні при перевірці функціональності, але можуть не виявити складніші помилки, які залежать від специфічних сценаріїв використання.

Таким чином, для отримання найкращих результатів можна використовувати комбінований підхід, включаючи різні типи інструментів в залежності від контексту проекту та специфічних вимог [17].

1.2.3 Переваги та недоліки існуючих програмно-апаратних комплексів

У сучасному програмуванні велику увагу приділяється виявленню та запобіганню помилок у програмному коді, особливо у контексті все зростаючої складності та інтегрованості програмних систем. В цьому контексті, програмно-апаратні комплекси відіграють ключову роль, пропонуючи інтегровані та ефективні рішення для виявлення та аналізу помилок. Однак, як і будь-яке технологічне рішення, вони мають свої переваги та обмеження, які важливо розглянути перед їх впровадженням у робочий процес. Ефективність, швидкість обробки, інтеграція з іншими системами та вартість впровадження - все це критичні аспекти, які впливають на вибір програмно-апаратних комплексів. На рис. 1.5 представлені основні переваги та недоліки цих комплексів.

Рисунок 1.5 – Переваги та недоліки програмно-апаратних комплексів

Переваги програмно-апаратних комплексів:

– підвищена обчислювальна потужність. Сучасні програмно-апаратні комплекси часто включають високопродуктивне обладнання, яке забезпечує швидку обробку великих обсягів даних, що є критично важливим для складних програмних систем;

– інтегровані рішення. Інтеграція апаратного та програмного забезпечення в один комплекс забезпечує кращу сумісність та оптимізацію роботи, знижуючи можливість виникнення помилок, пов'язаних з несумісністю компонентів;

– спеціалізоване обладнання. Використання спеціалізованого обладнання, такого як сервери, забезпечені специфічними функціями для тестування та аналізу, може значно підвищити ефективність виявлення та усунення помилок;

– ефективність тестування та відладки. Програмно-апаратні комплекси часто включають вбудовані інструменти для тестування та відладки, що може забезпечити більш ефективне та систематизоване виявлення помилок.

Недоліки програмно-апаратних комплексів:

– висока вартість. Комплекси, які включають спеціалізоване обладнання та ліцензійне програмне забезпечення, можуть бути дорогими, особливо для невеликих компаній або індивідуальних розробників;

– складність налаштування та утримання. Інтегровані системи можуть бути складними у налаштуванні та утриманні, особливо якщо вони включають спеціалізоване обладнання або складні програмні конфігурації;

– залежність від одного виробника. Інтегровані рішення часто залежать від продуктів одного виробника або невеликої групи постачальників, що може обмежити гнучкість та можливості вибору для користувачів;

– потенційні обмеження у функціональності. Деякі програмно-апаратні комплекси можуть мати обмежену функціональність або не бути повністю адаптованими до специфічних потреб користувачів, особливо у випадку швидких змін у технологіях [18].

На основі проведеного аналізу переваг та недоліків сучасних програмно-апаратних комплексів для виявлення помилок у програмному коді, можна зробити висновок, що ці комплекси пропонують значні переваги у підвищенні обчислювальної потужності, інтегрованих рішень, спеціалізованого обладнання та ефективності тестування та відладки. Це робить їх цінним інструментом для великих проектів та комплексних систем, де потрібна висока продуктивність та точність обробки великих обсягів даних. Водночас, висока вартість, складність налаштування та утримання, залежність від виробників та потенційні обмеження функціональності можуть бути значними бар'єрами, особливо для невеликих організацій або проектів з обмеженим бюджетом.

Вибір між використанням програмно-апаратних комплексів та іншими методами виявлення помилок повинен базуватися на конкретних потребах та ресурсах проекту, а також на здатності організації адаптуватися до швидко змінних технологічних умов [19]. Важливо зважувати вартість інвестицій у відповідні технології з потенційними перевагами, які вони можуть принести в довгостроковій перспективі.

1.2.4 Визначення прогалин у функціоналі та ефективності існуючих систем

Детальний аналіз прогалин у функціоналі та ефективності існуючих систем, зокрема тих, що використовуються для виявлення та прогнозування помилок у програмному коді, виявляє критичні аспекти, які потребують удосконалення або інноваційного підходу.

Перш за все, багато існуючих систем, хоча і ефективні у виявленні певних типів помилок, часто мають обмеження у виявленні складніших помилок логіки або помилок, що виникають у специфічних сценаріях використання. Наприклад, інструменти статичного аналізу можуть швидко ідентифікувати синтаксичні помилки, але вони можуть не виявляти помилки, пов'язані з паралелізмом або некоректним використанням системних ресурсів.

Крім того, багато систем не здатні адекватно враховувати контекст реального середовища виконання програми. Це може призвести до хибних позитивних або хибних негативних результатів, коли система виявляє помилки, які насправді не впливають на роботу програми, або, навпаки, пропускає критичні помилки.

Іншим важливим аспектом є висока залежність від якості та обсягу вхідних даних, особливо у системах, заснованих на машинному навчанні. Такі системи потребують великих та різноманітних наборів даних для ефективного навчання, і якість прогнозування помилок прямо залежить від якості цих даних.

Також, важливим фактором є здатність систем швидко адаптуватися до нових технологій та змін у парадигмах програмування. Зі зростанням популярності нових мов програмування та технологій, існуючі системи можуть швидко стати застарілими, якщо вони не оновлюються та не адаптуються до цих змін.

Отже, хоча сучасні системи пропонують потужні інструменти для виявлення помилок, існують значні прогалини у їх функціоналі та ефективності, що вимагають постійного оновлення, удосконалення та інтеграції з новими технологічними розвитками [20].

Висновки до розділу 1

У рамках критичного аналізу сучасного стану дослідження проблематики помилок у програмному коді було виконано глибокий аналіз літературних джерел, який охоплює історію виникнення та еволюцію цієї проблеми, а також сучасні дослідження в області їх виявлення та прогнозування. Цей аналіз виявив, що з розвитком технологій та збільшенням складності програмного забезпечення зростає необхідність у розробці ефективних методів для раннього виявлення помилок, забезпечуючи надійність та безпеку програмних продуктів.

Було проведено критичний огляд існуючих методів та інструментів, включаючи інструменти статичного аналізу, динамічні методи аналізу, інструменти на основі машинного навчання, та інструменти інтегрованих

середовищ розробки. Оцінка їх потенціалу, проведена у вигляді таблиці, підкреслила, що кожен метод має свої унікальні переваги та обмеження, що підтверджує необхідність використання комбінованого підходу у виявленні помилок.

Аналіз програмно-апаратного забезпечення, зокрема використання програмно-апаратних рішень у виявленні помилок, виявив їх значні переваги у підвищенні продуктивності та ефективності. Водночас було ідентифіковано ряд прогалин у функціоналі та ефективності існуючих систем, особливо у випадку складної обробки даних та адаптації до нових вимог.

На основі цього аналізу було сформульовано ряд завдань, спрямованих на розробку удосконалених методів та інструментів для виявлення та прогнозування помилок. Ці завдання охоплюють розробку нових підходів, які могли б ефективно враховувати зростаючу складність програмного забезпечення, забезпечувати високу надійність та безпеку, а також виявляти помилки на ранніх етапах їх виникнення. Виконання цих завдань дозволить значно підвищити якість розробки програмного забезпечення, знизивши ризики та витрати, пов'язані з виявленням та усуненням помилок.

2 МЕТОДИ І ТЕХНІКИ ВИЯВЛЕННЯ ТА ПРОГНОЗУВАННЯ ПОМИЛОК У ПРОГРАМНОМУ КОДІ

2.1 Теоретичний аналіз методів прогнозування

У даному підрозділі розглядаються основні підходи та теоретичні аспекти, які лежать в основі прогнозування помилок у програмному коді. Цей аналіз охоплює широкий спектр методів, починаючи від традиційних статичних та динамічних технік аналізу, до більш сучасних підходів, заснованих на машинному навчанні та штучному інтелекті. Важливою частиною цього аналізу є розуміння, як різні методи взаємодіють із специфікою програмного коду, зокрема, як вони адаптуються до різних мов програмування та стилів написання коду.

2.1.1 Класифікація методів прогнозування помилок

Методи прогнозування помилок включають різноманітні підходи, які варіюються від статичного аналізу, що не вимагає виконання коду для виявлення помилок, до динамічного аналізу, який залучає виконання програми для їх виявлення. Окрім цього, вони включають застосування алгоритмів машинного навчання, які використовують історичні дані та метрики коду для прогнозування потенційних помилок. Гібридні методи, що комбінують елементи різних підходів, також грають важливу роль у забезпеченні комплексного аналізу. Ефективність кожного з цих методів залежить від різних факторів, таких як мова програмування, розмір та складність проекту, а також специфіка самого коду. Також деякі з цих методів є більш вимогливими з точки зору ресурсів та потребують особливих навичок для їх ефективного застосування.

Статичні методи аналізу

Статичні методи аналізу програмного коду для прогнозування помилок є важливим інструментом у процесі розробки програмного забезпечення, дозволяючи виявляти помилки на ранніх етапах, ще до того, як програма буде

виконана або протестована. Ці методи базуються на аналізі коду без його виконання, використовуючи різні техніки для виявлення потенційних проблем.

Одним з ключових аспектів статичного аналізу є синтаксичний аналіз. Ця процедура включає перевірку коду на відповідність синтаксичним правилам мови програмування. Це означає, що аналізатор може виявити такі проблеми, як неправильне використання ключових слів, невідповідність у використанні дужок, відсутність необхідних символів завершення рядка тощо. Такий аналіз є фундаментальним, оскільки багато помилок у програмуванні виникають через прості синтаксичні помилки.

На наступному рівні статичного аналізу знаходиться семантичний аналіз. Він виявляє помилки, які стосуються значення використовуваних елементів коду, такі як некоректне використання типів даних, використання невизначених змінних, неправильне використання функцій тощо. Семантичний аналіз допомагає забезпечити, що код не тільки виглядає правильно, але й логічно вірний.

Інша важлива область статичного аналізу – це аналіз потоків даних. Цей метод виявляє проблеми пов'язані з управлінням даними в коді, такі як використання змінних до їх ініціалізації, недосяжні блоки коду, витік ресурсів тощо. Аналіз потоків даних є критично важливим для забезпечення надійності та безпеки програми.

Окрім цього, статичний аналіз також може включати аналіз залежностей, який допомагає виявити залежності між різними частинами коду. Це особливо важливо для великих систем, де взаємодія між різними модулями або компонентами може бути складною та неочевидною. Аналіз залежностей допомагає забезпечити, що зміни в одній частині коду не призведуть до непередбачених проблем в інших частинах.

Загалом, статичні методи аналізу є важливим інструментом для забезпечення якості та надійності програмного коду, дозволяючи виявляти та виправляти помилки на ранніх стадіях розробки, що значно спрощує процес виправлення та знижує вартість подальшої розробки [21].

Динамічні методи аналізу

На відміну від статичних методів, динамічний аналіз вимагає виконання програми, що дозволяє виявляти помилки, які можуть проявлятися лише під час її роботи. Це включає в себе широкий спектр технік, які дозволяють глибше зрозуміти поведінку програми в реальних умовах.

Одним з ключових елементів динамічного аналізу є тестування на основі використання. Це включає в себе створення та виконання тестових сценаріїв, які імітують реальні умови використання програми. Такий підхід дозволяє не лише виявити помилки у коді, але й перевірити, наскільки ефективно програма справляється з задачами, для яких вона була розроблена.

Інший важливий аспект динамічного аналізу – профілювання. Ця процедура полягає у моніторингу програми під час її виконання з метою визначення ділянок коду, що споживають надмірну кількість ресурсів, таких як час процесора або пам'ять. Профілювання допомагає виявити не тільки помилки в коді, але й оптимізувати його для підвищення загальної продуктивності програми.

Також варто згадати про моніторинг виключень який під час виконання програми відстежує виняткові ситуації, які можуть вказувати на проблеми в коді. Аналіз цих винятків може допомогти виявити не лише очевидні помилки, але й ті, що виникають у специфічних умовах або при певних вхідних даних.

Особливо важливим є динамічний аналіз у контексті великих та складних програмних систем, де взаємодія між різними модулями та компонентами може призводити до неочікуваних проблем. Тут він дозволяє не тільки виявляти помилки, але й глибше розуміти взаємодію між різними частинами системи, а також оцінювати вплив змін у одній частині на інші.

Загалом, динамічні методи аналізу є невід'ємною частиною процесу розробки та тестування ПЗ. Вони допомагають не лише виявити помилки, але й забезпечити, що програма відповідає всім вимогам ефективності та надійності, які ставляться перед нею [22].

Методи машинного навчання

Методи машинного навчання у прогнозуванні помилок пропонують сучасний підхід до ідентифікації потенційних проблем, які можуть не бути виявлені за допомогою традиційних методів аналізу. Ці методи базуються на використанні алгоритмів, які навчаються на історичних даних та зразках коду, щоб виявляти закономірності, які можуть вказувати на наявність помилок.

Основою для машинного навчання у цьому контексті є збір великої кількості даних, які можуть включати історії змін коду, повідомлення про помилки, коментарі розробників, та інші метадані. Ці дані використовуються для тренування моделей машинного навчання, які потім можуть аналізувати нові набори коду, виявляючи потенційні помилки на основі навчених патернів.

Один з підходів полягає у використанні класифікації для визначення, чи містить певний фрагмент коду помилки. Моделі машинного навчання можуть бути навчені визначати характеристики коду, які часто асоціюються з помилками, і використовувати ці знання для ідентифікації проблем у новому коді.

Іншим підходом є аналіз тексту програмного коду за допомогою технік обробки природної мови. Цей метод дозволяє моделям машинного навчання «розуміти» код на більш глибокому рівні, виявляючи неочевидні зв'язки та аномалії, які можуть вказувати на проблемні місця.

Окрім того, машинне навчання може застосовуватися для аналізу метрик коду, таких як його складність, частота змін, та інші фактори, які можуть вказувати на підвищений ризик помилок. Цей підхід дозволяє виявляти потенційно проблемні області коду, які можуть потребувати додаткової уваги або перевірки.

Однією з основних переваг методів МН є їх здатність адаптуватися та вдосконалюватися з часом. Чим більше даних надходить до моделі, тим точнішими стають її прогнози. Також вони здатні виявляти складні та нетривіальні помилки, які можуть не бути очевидними для традиційних методів аналізу.

Проте, важливо розуміти, що методи машинного навчання не є панацеєю і мають свої обмеження. Їх ефективність залежить від якості та обсягу навчальних даних, а також від складності і різноманітності сценаріїв, з якими може зіткнутися програмний код. Вони вимагають значних ресурсів для розробки, тренування та підтримки, а також спеціалізованих знань у галузі машинного навчання та програмування [23].

Гібридні методи

Гібридні методи прогнозування помилок у програмному коді представляють собою інноваційний підхід, що поєднує різні техніки та методології з метою підвищення ефективності та точності виявлення помилок. Ці методи інтегрують елементи статичного аналізу, динамічного аналізу, а також застосування машинного навчання для створення більш комплексної та всебічної системи перевірки якості програмного коду.

Основна ідея гібридних методів полягає у використанні сильних сторін різних технік для взаємного покращення. Наприклад, статичний аналіз може швидко виявити синтаксичні та деякі семантичні помилки без необхідності виконання коду, тоді як динамічний аналіз дозволяє виявляти помилки, які проявляються лише під час виконання програми. Інтеграція цих методів може забезпечити більш глибоке та всебічне виявлення помилок.

Машинне навчання, в свою чергу, вносить унікальний вклад у гібридні системи, аналізуючи великі обсяги історичних даних та виявляючи складні закономірності, які можуть вказувати на потенційні помилки. Це дозволяє системі не тільки виявляти відомі типи помилок, але й прогнозувати ймовірність виникнення нових, раніше не виявлених проблем.

Гібридні методи також можуть включати інструменти, які аналізують залежності у коді, виконують глибокий семантичний аналіз, та використовують розширені техніки верифікації для оцінки якості коду. Це дозволяє виявляти помилки, пов'язані зі складними взаємодіями між різними частинами коду, які можуть бути неочевидними при застосуванні лише одного методу аналізу.

Однак, гібридні системи стикаються з викликами, пов'язаними з інтеграцією різних технік та інструментів. Це може включати технічні складнощі, пов'язані з об'єднанням різних форматів даних та інтерфейсів, а також необхідність балансувати між швидкістю виявлення помилок та глибиною аналізу. Крім того, управління такими комплексними системами вимагає значних ресурсів та фахових знань у різних областях.

В цілому, гібридні методи прогнозування помилок відкривають нові можливості для розробників програмного забезпечення, забезпечуючи більш точне та всебічне виявлення помилок у коді. Це, у свою чергу, сприяє підвищенню якості та надійності розроблюваних програмних продуктів [24].

2.1.2 Критичний аналіз методів прогнозування

У рамках дослідження методів прогнозування помилок у програмному коді важливо здійснити глибокий аналіз різних підходів, які використовуються в сучасній розробці програмного забезпечення. Особлива увага приділяється порівняльному аналізу методів, щоб зрозуміти їхні сильні та слабкі сторони, а також визначити найбільш ефективні стратегії для їх застосування. Нижче наведено табл. 2.1 порівняльного аналізу, яка допоможе виявити унікальні переваги та потенційні обмеження кожного з них.

Таблиця 2.1 – Порівняльний аналіз методів

Метод	Статичний аналіз	Динамічний аналіз	Методи машинного навчання	Гібридні методи
Алгоритм	Інструменти лінтингу (ESLint)	Модульне тестування (JUnit)	Логістична регресія зі стохастичним спуском	Комбінація статичного аналізу та ШІ
Суть методу	Аналіз синтаксису та стилю кодування	Тестування окремих частин програми	Прогнозування помилок на основі історичних даних	Інтеграція різних технік

Переваги	Швидке виявлення синтаксичних помилок	Виявлення помилок у логіці програми	Висока точність, адаптивність	Комплексний аналіз
Недоліки	Обмежений аналіз	Час на написання тестів	Потреба у великому обсязі даних	Складність інтеграції
Здатність до адаптації	Низька	Низька	Висока	Середня
Точність прогнозування	Низька	Середня	Висока	Висока
Вимоги до ресурсів	Низькі	Середні	Високі	Дуже високі

Виходячи з проведеного аналізу в таблиці, можна стверджувати, що використання алгоритму логістичної регресії зі стохастичним спуском для розробки системи прогнозування помилок у програмному коді є дуже доцільним з кількох причин:

– висока точність прогнозування. Логістична регресія ефективно використовує історичні дані для виявлення закономірностей та аномалій, які можуть вказувати на помилки в коді. Це робить метод особливо цінним для розробників, які прагнуть мінімізувати кількість помилок у своїх продуктах. Виходячи з великої кількості даних, алгоритм може точно прогнозувати потенційні проблемні місця;

– адаптивність. Логістична регресія добре адаптується до змін у патернах кодування та еволюції мов програмування;

– використання у широкому спектрі сценаріїв. Логістична регресія може бути застосована до різних мов програмування та проектів різної складності. Це робить її універсальним інструментом, який може бути інтегрований у різні системи та процеси розробки;

– автоматизація процесу прогнозування. Застосування машинного навчання дозволяє автоматизувати процес прогнозування помилок, звільняючи розробників від необхідності ручного аналізу великих обсягів коду.

Отже, використання логістичної регресії зі стохастичним спуском у системі прогнозування помилок є високоефективним і доцільним, особливо для середовищ, де доступна велика кількість даних і де є можливість залучення спеціалізованих знань для розробки та підтримки системи. Вона пропонує високу точність та адаптивність, роблячи цей підхід цінним доповненням до інших методів прогнозування помилок.

2.2 Аналіз програмно-апаратних засобів

Аналіз програмно-апаратних засобів включає в себе розгляд інструментів статичного та динамічного аналізу коду, які кожен по-своєму внесли значний вклад у підвищення якості та надійності програмних продуктів. Також особливе місце в цьому процесі займають інтегровані середовища розробки, які надають розробникам потужні інструменти та можливості для ефективної роботи. Важливо проаналізувати, як ці інструменти та середовища можуть бути використані для попередження та виявлення помилок, а також яким чином вони впливають на загальний процес розробки.

2.2.1 Інструменти статичного аналізу коду

Статичний аналіз коду, що виконується без його фактичного запуску, є невід'ємною частиною сучасного процесу розробки ПЗ. Цей метод орієнтований на виявлення різноманітних помилок, починаючи від синтаксичних та закінчуючи логічними. Важливість інструментів статичного аналізу, як-от SonarQube, ESLint, або PMD, в сучасній розробці програмного забезпечення не можна недооцінювати.

SonarQube, наприклад, є високоефективним інструментом для не тільки виявлення помилок, але й для відстеження якості коду на протязі всього життєвого циклу розробки. Він дозволяє аналізувати та визначати недоліки в коді, забезпечуючи зворотний зв'язок і рекомендації щодо їх виправлення. Це

значно спрощує процес рефакторингу та допомагає підтримувати високі стандарти якості коду [25].

ESLint, зі свого боку, є інструментом, який зосереджений на ідентифікації і виправленні проблем, що стосуються синтаксису і стилю кодування, особливо у JavaScript. Це допомагає забезпечити консистентність коду, що є важливим для великих команд розробників і сприяє підвищенню читабельності та підтримуваності коду [26-27].

PMD є ще одним інструментом статичного аналізу, який знаходить загальні помилки програмування, такі як неефективний код, невикористані змінні, непотрібні об'єкти тощо. Це дозволяє розробникам швидко виявляти та усувати потенційні проблеми, перш ніж вони перетворяться на серйозні помилки.

Використання цих інструментів статичного аналізу в розробці ПЗ значно зменшує ризик помилок, покращує загальну якість коду та спрощує процес розробки. Вони дозволяють розробникам вчасно виявляти та виправляти помилки, забезпечуючи ефективний та продуктивний розвиток програмних продуктів.

2.2.2 Інструменти динамічного аналізу коду

На відміну від статичного аналізу, який зосереджений на перевірці коду без його виконання, динамічний аналіз вимагає запуску програми та спрямований на виявлення помилок, які проявляються саме під час її роботи. Цей вид аналізу забезпечує виявлення різних видів помилок виконання, включаючи витоки пам'яті, проблеми з синхронізацією і конкурентним доступом, некоректну обробку виключних ситуацій та інші помилки, які не можуть бути виявлені під час статичного аналізу.

Інструменти динамічного аналізу, такі як JUnit, Selenium та LoadRunner, забезпечують різні види тестування для оцінки якості ПЗ. JUnit, який широко використовується у розробці на Java, є інструментом для модульного тестування,

що дозволяє розробникам створювати та запускати автоматизовані тести для окремих компонентів програми. Це сприяє швидкому виявленню та виправленню помилок на ранніх стадіях розробки.

Selenium є потужним інструментом для автоматизації тестування веб-додатків, дозволяючи імітувати дії користувача у веб-браузері. Це допомагає перевірити, як програма поводить себе під час реального використання, та виявити помилки, пов'язані з взаємодією користувача з додатком [28].

LoadRunner використовується для навантажувального та стресового тестування, дозволяючи оцінити продуктивність програми під високим навантаженням та визначити, як вона реагує на критичні умови експлуатації. Таке тестування є критично важливим для визначення меж витривалості програми та її здатності правильно функціонувати під час пікових навантажень [29].

Комбінування різних інструментів динамічного аналізу дозволяє розробникам отримати повне уявлення про якість та надійність їх програмного продукту. Це допомагає забезпечити, що програма не тільки вільна від помилок у коді, але й ефективно функціонує в реальних умовах її використання.

2.2.3 Оцінка можливостей сучасних інтегрованих середовищ розробки (IDE)

Сучасні інтегровані середовища розробки, такі як Visual Studio, IntelliJ IDEA, та Eclipse, є незамінними інструментами в арсеналі будь-якого розробника програмного забезпечення. Ці середовища не тільки полегшують процес написання коду, але й забезпечують широкий спектр інструментів для його аналізу, відладки та оптимізації.

Visual Studio відоме своєю здатністю інтегрувати численні інструменти статичного аналізу та динамічного тестування, що дозволяє розробникам ефективно виявляти та виправляти помилки в коді. Це IDE також пропонує потужні можливості для управління проектами, включаючи підтримку різних

мов програмування, розширені функції відладки та інтеграцію з системами контролю версій [30].

IntelliJ IDEA відрізняється своїм інтуїтивним інтерфейсом та розширеними можливостями для роботи з Java, Kotlin та іншими мовами. Вона надає потужні інструменти для рефакторингу коду, аналізу залежностей та виявлення проблем у коді. IntelliJ IDEA особливо цінується за свої розумні підказки та автоматичне виправлення помилок, що значно прискорює процес розробки [31].

Eclipse, з іншого боку, є відкритим середовищем, яке підтримує широкий спектр мов програмування та технологій. Його гнучкість та модульна структура дозволяють розробникам налаштовувати середовище під свої специфічні потреби, інтегруючи численні плагіни та розширення. Eclipse відоме своєю здатністю до глибокого аналізу коду та ефективного управління проектами [32].

Всі ці інтегровані середовища розробки надають важливі інструменти для підвищення продуктивності розробників. Від автоматизації рутинних завдань до надання складних інструментів для аналізу коду, IDE сприяють підвищенню якості та надійності розроблюваного програмного забезпечення. Їх роль у виявленні та виправленні помилок є ключовою, оскільки вони дозволяють розробникам швидко ідентифікувати та вирішувати проблеми, що зустрічаються під час кодування, тим самим зменшуючи ризики, пов'язані з розробкою програмного забезпечення.

2.3 Методи оцінки якості та надійності програмного коду

У сфері програмної інженерії, оцінка якості та надійності програмного коду займає важливе місце, вимагаючи ретельного аналізу та використання спеціалізованих підходів. Розглядаючи різноманітні методи, зосереджуємося на виявленні та оцінці ключових параметрів, які впливають на якість та надійність програмного продукту. Від використання метрик якості коду до застосування моделей надійності програмного забезпечення, важливо підібрати відповідні інструменти та методики. Це дозволяє не лише досягти високих стандартів

якості, але й забезпечити надійну роботу програмного продукту, задовольняючи потреби та вимоги кінцевих користувачів.

2.3.1 Метрики якості програмного коду

У сфері розробки програмного забезпечення, глибоке розуміння та оцінка якості коду є критично важливими для створення надійних та ефективних програмних продуктів. В цьому контексті, метрики якості коду відіграють ключову роль, дозволяючи аналізувати та оцінювати різні аспекти програмного коду об'єктивно та систематично. Ці метрики надають цінну інформацію про різні характеристики коду, такі як його складність, згуртованість, зв'язність та інші важливі параметри. Вони допомагають розробникам ідентифікувати потенційні проблемні області та напрямки для покращення, а також підтримувати високі стандарти якості та надійності розроблюваного програмного продукту.

Для детальнішого розуміння, як метрики якості коду можуть бути застосовані у процесі оцінки та покращення програмного продукту необхідно розглянути найбільш поширені та важливі з них. Ці метрики включають кількість методів у класі, глибину дерева наслідування, зв'язність з іншими класами, внутрішню згуртованість класу, середню цикломатичну складність та багато інших. У табл. 2.3 детально описано кожну з цих метрик:

Таблиця 2.2 – Аналіз метрик якості коду

№	Назва метрики	Опис
1	Кількість методів у класі	Вимірює загальну кількість методів, визначених у класі.
2	Глибина дерева наслідування	Показує, наскільки глибоко клас розташований у ієрархії наслідування.
3	Кількість дочірніх класів	Вказує на кількість класів, які успадковуються від даного класу.
4	Зв'язність з іншими класами	Міряє кількість класів, з якими взаємодіє даний клас.

5	Кількість викликаних функцій	Обраховує загальну кількість унікальних викликів методів у класі.
6	Внутрішня згуртованість класу	Оцінює, наскільки поля класу використовуються його методами.
7	Зв'язки з іншими класами	Кількість зв'язків класу з іншими класами в його пакеті.
8	Зв'язки з зовнішніми класами	Кількість зв'язків класу з класами поза його пакетом.
9	Кількість публічних методів	Визначає кількість публічних методів у класі.
10	Згуртованість класу	Альтернативний показник згуртованості класу.
11	Кількість рядків коду	Вимірює загальну кількість рядків коду в класі.
12	Доступність методів класу	Визначає відсоток приватних та захищених методів класу.
13	Використання інших об'єктів	Кількість об'єктів, що створюються в класі.
14	Частка успадкованого коду	Відсоток методів класу, успадкованих від базових класів.
15	Когезія серед методів класу	Вимірює рівень когезії серед методів класу.
16	Кількість реалізованих інтерфейсів	Кількість інтерфейсів, які реалізує клас.
17	Число базових методів	Кількість методів, які клас використовує з базових класів.
18	Середня складність методів	Середня складність усіх методів класу.
19	Максимальна цикломатична складність	Найвища цикломатична складність серед усіх методів класу.
20	Середня цикломатична складність	Середнє значення цикломатичної складності для методів класу.

Розглянуті метрики дають можливість глибокого аналізу програмного коду, виявлення його слабких місць та потенційних поліпшень, а також допомагають розробникам підтримувати високі стандарти якості та надійності в розробці програмного продукту.

2.3.2 Моделі надійності програмного забезпечення

Моделі надійності програмного забезпечення застосовуються для прогнозування та аналізу ризиків, пов'язаних із відмовами програмного забезпечення. Ці моделі можуть базуватися на історичних даних про помилки та їх вплив на роботу системи. Вони включають різні математичні та статистичні підходи, такі як моделі на основі часу між відмовами, моделі на основі кількості виявлених помилок тощо. Використання цих моделей допомагає розробникам оцінити потенційні ризики та планувати необхідні заходи для забезпечення високої надійності продукту.

На рис 2.1 зображено модель, яка зосереджена на візуалізації та аналізі часових інтервалів між послідовними відмовами програмного продукту. Вона включає хронологічну лінію відмов, яка показує моменти в часі, коли відбувалися відмови, де кожна точка на лінії представляє одну відмову. Це дозволяє швидко оцінити частоту та розподіл відмов у часі.

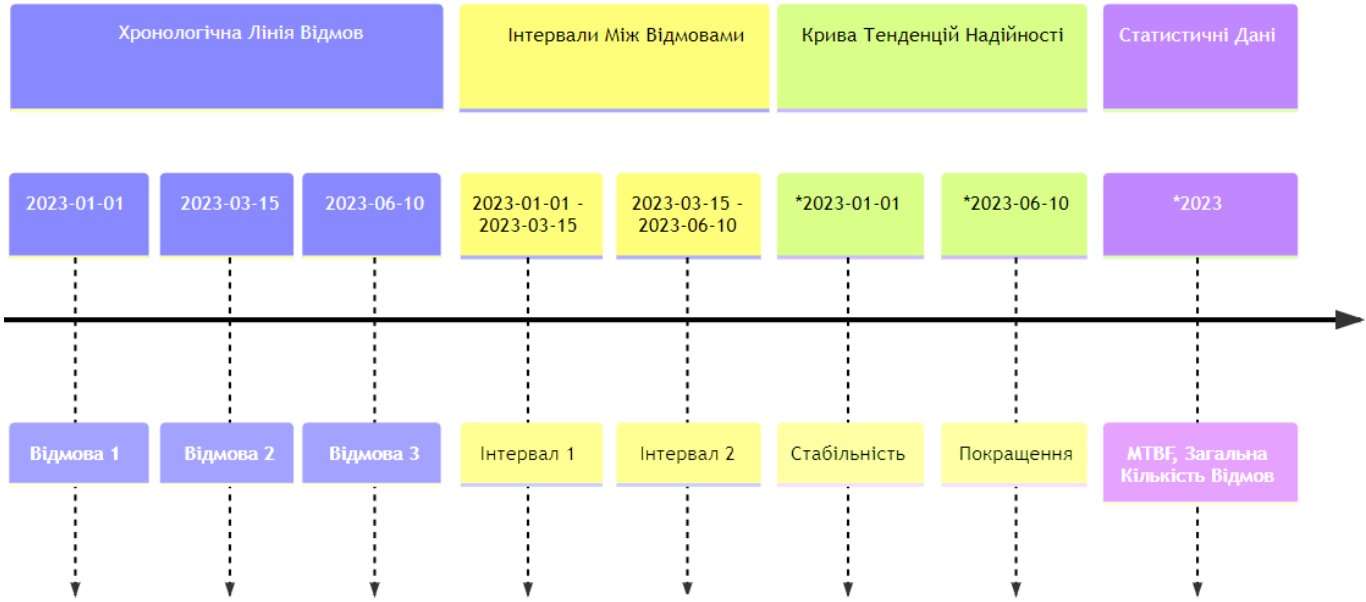

Рисунок 2.1 – Модель аналізу часових інтервалів між відмовами

Інтервали між відмовами, відстань між послідовними точками на хронологічній лінії відмов, вимірюють час між відмовами. Ці інтервали допомагають визначити, чи стає програмне забезпечення більш або менш надійним з часом.

Крива тенденцій надійності, лінія або крива, яка з'єднує точки відмов, відображає загальну тенденцію у надійності програмного продукту. Висхідна крива вказує на покращення надійності (інтервали між відмовами збільшуються), тоді як низхідна крива показує погіршення.

Статистичні дані, такі як середній час між відмовами (MTBF) та загальна кількість відмов за певний період, включаються у модель для кількісного аналізу надійності.

Ця модель допомагає не тільки візуально представити дані про надійність, але й забезпечує основу для аналізу причин відмов та планування поліпшень у програмному забезпеченні. Вона дозволяє аналізувати часові інтервали та частоту відмов, виявляти патерни, що можуть вказувати на систематичні проблеми у ПЗ, а також оцінювати ефективність виправлень та внесених змін у програмне забезпечення. Використання цієї моделі може значно підвищити якість та надійність програмного забезпечення.

Застосування цих моделей не тільки допомагає виявити та усунути існуючі недоліки у ПЗ, але й надає цінну інформацію для планування майбутніх дій з метою попередження потенційних помилок. Це включає в себе встановлення пріоритетів у тестуванні, вибір стратегій відладки та розробку планів підтримки та оновлення програмного забезпечення.

Крім того, моделі надійності ПЗ сприяють розумінню того, як зміни в коді або архітектурі ПЗ можуть впливати на його загальну надійність. Використання моделей надійності ПЗ дозволяє розробникам більш точно прогнозувати поведінку продукту в реальних умовах експлуатації та відповідно адаптувати процеси розробки та тестування, щоб забезпечити високий рівень якості та задоволення вимог кінцевих користувачів.

2.3.3 Верифікація та валідація програмного коду

Верифікація та валідація в програмній інженерії призначені для забезпечення якості та надійності програмного продукту. Верифікація,

зосереджена на аналізі точності кожного етапу розробки, гарантує, що програмний продукт розроблено відповідно до визначених вимог та специфікацій. Вона включає в себе ряд процедур та перевірок, спрямованих на визначення того, чи були дотримані всі технічні та проектні стандарти на кожному кроці розробки. Процеси верифікації включають ретельне тестування коду, перевірку відповідності документації, статичний аналіз коду, а також використання інших методів, які дозволяють виявити технічні помилки, недоліки в дизайні або невідповідності вимогам проекту.

Верифікація забезпечує, що кожен компонент програмного продукту був створений правильно і функціонує належним чином, відповідаючи на питання «Чи робимо ми продукт правильно?». Цей процес допомагає ідентифікувати та виправити проблеми на ранніх стадіях розробки, перш ніж вони перетворяться на більш серйозні помилки, що можуть вплинути на загальну якість та надійність продукту.

З іншого боку, валідація зосереджена на визначенні того, чи відповідає розроблений продукт очікуванням та потребам кінцевих користувачів. Вона відповідає на питання: «Чи робимо ми правильний продукт?» Процес валідації передбачає аналіз вимог користувачів, тестування програмного продукту в реальних умовах експлуатації, а також перевірку його функціональності та зручності використання. Це може включати бета-тестування, користувацьке тестування та інші види досліджень, які допомагають зрозуміти, чи задовольняє продукт потреби та очікування його цільової аудиторії.

Обидва ці процеси є важливими для забезпечення високого рівня якості та надійності програмного забезпечення. Вони дозволяють розробникам не лише забезпечити технічну точність продукту, але й гарантувати, що він відповідає реальним потребам користувачів, що є ключовим для успіху будь-якого програмного продукту. Верифікація та валідація взаємодоповнюють одна одну, формуючи всебічний підхід до забезпечення якості програмного забезпечення.

2.4 Вибір та обґрунтування методу дослідження для розробки ПЗ

З огляду на зростаючу складність ПЗ та важливість його надійності та безпеки, існує гостра потреба у розробці ефективних методів для раннього виявлення та прогнозування помилок. Це стає особливо важливим, враховуючи, що помилки в програмному коді можуть призвести до серйозних наслідків, включаючи втрату даних, зниження продуктивності, а у деяких випадках - становити загрозу безпеці.

2.4.1 Вибір методу на основі аналізу вимог до проекту

Логістична регресія зі стохастичним спуском була обрана як метод для прогнозування помилок у програмному коді, оскільки вона ідеально підходить для вирішення задач, що виникають в умовах зростаючої складності сучасного програмного забезпечення. Цей метод відрізняється здатністю ефективно обробляти великі обсяги даних, що є критично важливим у контексті розробки складних програмних систем. Завдяки стохастичному спуску, логістична регресія дозволяє швидко адаптуватися до змін у даних, що є важливим для динамічного середовища програмування.

При детальному розгляді вимог до проекту, особливо в контексті прогнозування помилок у програмному коді, важливо врахувати ряд ключових аспектів, які впливають на вибір методології. Серед них:

– складність та інтеграція в ПЗ. Зі зростанням складності програмних систем та їх інтеграції з іншими компонентами, ризики пов'язані з помилками у коді збільшуються. У цьому контексті логістична регресія зі стохастичним спуском виступає як ефективний інструмент для аналізу великих обсягів коду та ідентифікації потенційних слабких місць;

– забезпечення надійності та безпеки ПЗ. Усвідомлення важливості надійності та безпеки програмного забезпечення ставить підвищені вимоги до точності та своєчасності виявлення помилок. Логістична регресія зі

стохастичним спуском дозволяє прогнозувати потенційні помилки, що значно знижує ризики пов'язані з безпекою та надійністю;

– раннє виявлення помилок. Ефективність раннього виявлення помилок у програмному коді має безпосередній вплив на зниження витрат та часу на відлагодження. Застосування логістичної регресії в ранніх стадіях розробки дозволяє вчасно виявляти помилки, перш ніж вони переростуть у більш серйозні проблеми.

Перелічені аспекти формують основу для вибору логістичної регресії зі стохастичним спуском як методу дослідження. Цей підхід дозволяє не тільки виявляти ймовірні помилки, але й забезпечує глибше розуміння динаміки їх виникнення, що є критично важливим для підвищення ефективності та якості розробки програмного забезпечення.

Обґрунтування вибору методу логістичної регресії зі стохастичним спуском для розробки системи прогнозування помилок у програмному коді базується на декількох ключових факторах:

– висока адаптивність до різних даних. Логістична регресія є ефективним методом для обробки даних різної природи. Це особливо важливо у контексті програмного коду, де можуть виникати різні типи помилок в залежності від конкретних умов розробки та використання програмного забезпечення;

– ефективне виявлення шаблонів помилок. Логістична регресія дозволяє ідентифікувати ймовірність наявності помилок на основі шаблонів та закономірностей, виявлених у історичних даних. Цей метод є корисним для визначення потенційних помилок в програмному коді, оскільки він дозволяє моделювати взаємозв'язки між різними характеристиками коду та їх впливом на виникнення помилок;

– обробка великих обсягів даних. Однією з переваг логістичної регресії зі стохастичним спуском є її здатність ефективно обробляти великі набори даних. Це особливо важливо у сучасних проектах програмування, де обсяги коду та даних про помилки можуть бути дуже великими.

– гнучкість та налаштування. Метод може бути налаштований та адаптований під конкретні потреби проекту, що робить його вельми гнучким у використанні. Логістична регресія може бути налаштована для роботи з різними типами функцій та врахування специфічних характеристик програмного коду;

– висока Інтерпретабельність: На відміну від багатьох інших алгоритмів машинного навчання, логістична регресія забезпечує високий рівень інтерпретабельності результатів. Це дозволяє розробникам не тільки ідентифікувати потенційні помилки, але й розуміти, чому певні фрагменти коду є схильними до помилок.

Отже, вибір логістичної регресії зі стохастичним спуском для прогнозування помилок у програмному коді обумовлений її здатністю до точної класифікації, обробки великих даних, високою адаптивністю, налаштовуваністю та інтерпретабельністю. Ці якості роблять її ідеальною для використання в різних проектах програмного забезпечення для забезпечення їх якості та надійності.

2.4.2 Адаптація вибраного методу для специфіки проекту

Для адаптації методу логістичної регресії зі стохастичним спуском до специфіки проекту з прогнозування помилок у програмному коді, було проведено ряд важливих дій, спрямованих на підготовку та налаштування даних. Спершу було обрано датасет «Software Defect Prediction Dataset» [33] із авторитетного сайту Kaggle, який відповідає потребам проекту за своєю структурою та вмістом. Цей датасет містить різноманітні метрики коду, що є ідеальними для нашої задачі прогнозування.

Основна робота полягала у ретельній очистці даних, включаючи видалення дублікатів та коригування невідповідностей, а також у заповненні пропущених значень, щоб забезпечити цілісність та якість даних для аналізу. Після цього було здійснено нормалізацію та стандартизацію характеристик, що допомогло уніфікувати масштаби та спростити обробку даних моделлю.

Далі відбувся аналіз кореляцій між різними характеристиками в датасеті, щоб визначити, які з них мають найбільшу вагу для моделі. Виявлені ключові функції були використані для налаштування параметрів моделі логістичної регресії, забезпечуючи її максимальну ефективність.

На завершення, датасет було розділено на тренувальний та тестовий набори, що дозволило не тільки адекватно навчити модель, але й точно оцінити її здатність до прогнозування на нових даних. Таким чином, кожен крок у підготовці даних був націлений на те, щоб зробити модель логістичної регресії зі стохастичним спуском максимально ефективною та адаптованою до специфіки нашого проекту з прогнозування помилок у програмному коді.

Логістична регресія є методом машинного навчання, що використовується для розв'язання задач класифікації. Вона моделює ймовірність того, що дане спостереження належить до одного з двох класів. Цей метод часто використовується для випадків, де виходом є бінарний результат (наприклад, так/ні, успіх/невдача).

Формула логістичної регресії виглядає наступним чином:

$$P(Y = 1 \mid X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_n X_n)}} \qquad (2.1)$$

де:

$P(Y = 1 \mid X)$ – ймовірність того, що спостереження X належить до класу 1 (наприклад, є дефектним у контексті дефектів програмного коду);

$\beta_0, \beta_1, \ldots \beta_n$ – коефіцієнти моделі, які визначають вагу кожного атрибуту X_i у моделі.

$X_1, X_2, \ldots, X_n$– характеристики або атрибути спостереження, наприклад, метрики програмного коду.

Для знаходження оптимальних значень коефіцієнтів β буде використана стратегія оптимізації, зокрема стохастичний градієнтний спуск (СГС). СГС є методом оптимізації, використовуваним для знаходження оптимальних значень коефіцієнтів β у моделі логістичної регресії. Він особливо корисний, коли

працює з великими обсягами даних, оскільки оновлення ваг моделі відбувається поступово з кожним окремим спостереженням.

Суть СГС полягає в ітеративному оновленні кожного коефіцієнта моделі, використовуючи набір даних, де кожен крок оновлення базується на одному або невеликій групі спостережень (міні-пакетах). Це відрізняється від традиційного градієнтного спуску, де оновлення ваг відбувається після обробки всього набору даних.

Формула оновлення коефіцієнта в СГС виглядає наступним чином:

$$\beta_j := \beta_j - \alpha \cdot \frac{\partial}{\partial \beta_j} J(\beta; x^{(i)}, y^{(i)}) \qquad (2.2)$$

номер формули –по лівому боці (у всій роботі)

де, β_j – поточне значення j-го коефіцієнта;

α – швидкість навчання, яка контролює розмір кроку в процесі оновлення;

$\frac{\partial}{\partial \beta_j} J\left(\beta; x^{(i)}, y^{(i)}\right)$– часткова похідна функції втрат J по коефіцієнту β_j, обчислена для i-го спостереження з тренувального набору даних.

У контексті логістичної регресії, функція втрат J часто визначається як бінарна крос-ентропія, що вимірює розбіжність між фактичними класами $y^{(i)}$ та передбаченими ймовірностями $P(Y = 1|X^{(i)}; \beta)$.

Часткова похідна функції втрат по коефіцієнту β_j може бути обчислена як:

$$\frac{\partial}{\partial \beta_j} J\left(\beta; x^{(i)}, y^{(i)}\right) = \left(P\left(Y = 1 \middle| X^{(i)}; \beta\right) - y^{(i)}\right) \cdot x_j^{(i)} \qquad (2.3)$$

де: $x_j^{(i)}$ – значення j-го атрибуту i-го спостереження.

$P\left(Y = 1 \middle| X^{(i)}; \beta\right)$ – передбачена моделлю ймовірність того, що і-те спостереження належить до класу 1.

В процесі тренування моделі кожен крок СГС використовує випадково вибране спостереження або міні-пакет спостережень для оновлення вагів. Це дозволяє моделі поступово знаходити оптимальні значення коефіцієнтів, мінімізуючи функцію втрат на тренувальних даних.

Адаптація математичної моделі логістичної регресії до коду зосереджена на використанні вхідних ознак, таких як Wmc, Dit, Noc, тощо для тренування моделі та прогнозування результатів.

Вхідні дані (Wmc, Dit, Noc, тощо) формують вектор ознак $X = [X_1, X_2, \dots X_n]$, де кожна X_i відповідає окремій характеристиці коду. Ці ознаки служать вхідними змінними для моделі.

Під час тренування моделі, використовуючи метод Fit, відбувається оптимізація коефіцієнтів β за допомогою СГС. Це включає обчислення градієнта функції втрат для кожного спостереження та оновлення коефіцієнтів відповідно.

Після тренування, модель використовує навчені коефіцієнти для прогнозування ймовірності наявності помилки в нових даних. Оцінка моделі здійснюється за допомогою тестового набору даних, де порівнюються передбачення моделі з фактичними результатами. Метрики, такі як точність (Accuracy), площа під ROC-кривою (Area Under the Receiver Operating Characteristic Curve), логарифмічні втрати (Log Loss) та гармонійне середнє між точністю та повнотою (F1 Score), дозволяють оцінити загальну ефективність моделі у прогнозуванні помилок у програмному коді.

Висновки до розділу 2

У даному розділі було проведено дослідження та аналіз різних підходів до виявлення та прогнозування помилок у програмному коді. В рамках теоретичного аналізу методів прогнозування була виконана класифікація методів, включаючи статичні та динамічні методи аналізу, методи машинного навчання та гібридні методи. Це дозволило визначити переваги та недоліки кожного підходу.

Критичний аналіз методів прогнозування, включаючи порівняльний аналіз таких інструментів, як інструменти лінтингу (ESLint), модульне тестування (JUnit), та логістичну регресію зі стохастичним спуском, показав переваги останнього у контексті задачі прогнозування. Вибір логістичної регресії був

обгрунтований її ефективністю та гнучкістю у виявленні ймовірності помилок в програмному коді.

Аналіз програмно-апаратних засобів зосередився на інструментах статичного аналізу, таких як SonarQube, ESLint, PMD, інструментах динамічного аналізу, включаючи JUnit, Selenium, LoadRunner, а також на оцінці можливостей сучасних IDE як Visual Studio, IntelliJ IDEA, та Eclipse. Це дозволило оцінити інструменти, які можуть бути інтегровані для покращення процесу розробки та забезпечення якості коду.

Також були розглянуті метрики якості програмного коду та моделі надійності ПЗ, що дозволило визначити ключові показники для оцінки якості коду та його потенційної надійності. Особлива увага була приділена верифікації та валідації програмного коду, що є важливими аспектами забезпечення високої якості програмних продуктів.

На етапі вибору методу на основі аналізу вимог до проекту було зроблено акцент на адаптації вибраного методу логістичної регресії для специфіки проекту. Це включало вибір відповідного датасету із авторитетного сайту Kaggle, адаптацію методу до конкретних метрик датасету, а також розробку математичного апарату для прогнозування помилок.

3 ПРОЕКТУВАННЯ Й РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТОХАСТИЧНИХ АЛГОРИТМІВ

3.1 Формалізація задачі

Формалізація задачі дослідження методів прогнозування появи помилок у програмному коді з використанням штучного інтелекту передбачає створення комплексної моделі машинного навчання. Ця модель аналізує різноманітні аспекти програмного коду та визначає потенційні помилки. Процес формалізації включає наступні етапи:

Вхідні дані

Вхідні дані для цієї моделі включають в себе метрики, які відображають різні характеристики програмного коду. Кожна метрика описує певний аспект, такий як кількість методів у класі, зв'язність класів, глибина дерева наслідування тощо. Для кожної метрики визначається вектор вхідних даних $X = \{X_{1,}X_{2,\dots}X_{n,}\}$, де $X_{i,}$ представляє кожну метрику, яка має своє значення та відображає певний аспект коду, як описано нижче:

– кількість методів у класі (WMC). Ця метрика вимірює загальну кількість методів, які визначені в класі. Висока кількість методів може вказувати на високу складність класу, що збільшує ризик появи помилок;

– глибина дерева наслідування (DIT). Показує, наскільки глибоко клас розташований у ієрархії наслідування. Глибші рівні наслідування можуть призводити до більш складного коду та потенційних помилок;

– кількість дочірніх класів (NOC). Вказує на кількість класів, які успадковуються від даного класу. Велика кількість дочірніх класів може свідчити про важливість базового класу в архітектурі програми;

– зв'язність з іншими класами (CBO). Міряє кількість класів, з якими взаємодіє даний клас. Висока зв'язність може призводити до більшої залежності між класами та складності управління змінами;

– кількість викликаних функцій (RFC). Обраховує загальну кількість унікальних викликів методів у класі. Високе значення RFC може вказувати на високу складність та взаємозалежність у коді;

– внутрішня згуртованість класу (LCOM). Оцінює, наскільки поля класу використовуються його методами. Низька згуртованість може бути ознакою слабкої структури класу;

– зв'язки з іншими класами (CA) та зв'язки з зовнішніми класами (CE). Ці метрики визначають кількість зв'язків класу в межах його пакету та з класами поза пакетом відповідно. Висока кількість зв'язків може підвищувати складність та ризик помилок;

– кількість публічних методів (NPM). Визначає кількість публічних методів у класі. Більше публічних методів може свідчити про більшу відповідальність та складність класу;

– кількість рядків коду (LOC). Вимірює загальну кількість рядків коду в класі. Велика кількість рядків коду часто вказує на високу складність;

– доступність методів класу (DAM). Визначає відсоток приватних та захищених методів класу. Вищий рівень інкапсуляції може сприяти кращому управлінню кодом;

– використання інших об'єктів (MOA), частка успадкованого коду (MFA), когезія серед методів класу (CAM), кількість реалізованих інтерфейсів (IC), число базових методів (CBM), Середня складність методів (AMC), Максимальна та середня цикломатична складність (Max CC, Avg CC). Ці метрики далі розкривають аспекти, пов'язані з взаємодією між класами, спадкуванням, внутрішньою логікою класів, та загальною складністю методів. Вони допомагають визначити потенційні джерела помилок та точки, які потребують уваги під час перевірки коду.

Кожна з цих метрик вносить важливий вклад у здатність моделі точно прогнозувати помилки. Вони дозволяють моделі оцінювати програмний код з різних боків, враховуючи його структуру, складність, взаємодію між

компонентами, та інші ключові фактори, які впливають на ймовірність появи помилок.

Нормалізація числових змінних

Нормалізація числових змінних - це процес приведення всіх числових даних до єдиного масштабу, який допомагає уникнути переваги великих значень та сприяє більш ефективному навчанню моделей машинного навчання.

У контексті задачі є різні метрики програмного коду (наприклад, кількість методів у класі, кількість рядків коду тощо), ці метрики можуть мати різні масштаби. Деякі метрики можуть мати значення у діапазоні від 0 до 10, в той час як інші - від 0 до 1000. Така розбіжність може призвести до ситуації, де великі числа непропорційно впливають на навчання моделі, роблячи її менш чутливою до маленьких, але важливих змін у менших метриках.

Процес нормалізації вирівнює ці масштаби. Один із популярних методів нормалізації - Min-Max Scaling. Він приводить всі значення до діапазону між 0 та 1. Це досягається шляхом віднімання мінімального значення метрики від кожного значення, а потім поділу результату на різницю між максимальним та мінімальним значеннями цієї метрики.

Формула для нормалізації виглядає наступним чином:

$$N_x = \frac{x - x_{min}}{x_{max} - x_{min}}, \qquad (3.1)$$

де:

N_x – це нормалізоване значення змінної;

x – це поточне значення змінної, яке потребує нормалізації;

x_{min} – це мінімальне значення змінної у датасеті;

x_{max} – це максимальне значення змінної у датасеті.

Ця формула працює шляхом віднімання мінімального значення змінної від її поточного значення, щоб визначити її відносне положення у діапазоні між мінімумом та максимумом. Потім цей результат ділиться на загальний діапазон значень змінної, щоб перетворити його на пропорційне значення у діапазоні від

0 до 1. Це дозволяє стандартизувати різні метрики на однаковий масштаб, сприяючи більш ефективному та точному аналізу.

Вибір моделі

Для задачі прогнозування появи помилок у програмному коді було обрано модель логістичної регресії зі стохастичним спуском. Логістична регресія - це популярний статистичний метод, що використовується для прогнозування ймовірності наявності або відсутності певної характеристики (у цьому випадку - помилки в програмному коді) на основі однієї або декількох незалежних змінних (метрик програмного коду).

Основною перевагою логістичної регресії є її здатність оцінювати ймовірності. Відмінно від лінійної регресії, яка прогнозує безпосередньо значення змінної, логістична регресія оцінює ймовірність того, що зразок належить до певного класу. В контексті вашої задачі, це означає визначення ймовірності того, чи містить програмний код помилку чи ні.

Стосовно стохастичного спуску, це метод оптимізації, використовуваний у процесі навчання логістичної регресії. Він допомагає знайти оптимальні параметри моделі, мінімізуючи функцію втрат. Стохастичний градієнтний спуск працює шляхом оновлення ваг моделі на кожному кроці навчання, використовуючи лише невеликий вибірок (або навіть одиничний приклад) з даних, що робить його ефективним і швидшим у порівнянні з традиційним градієнтним спуском, який використовує весь датасет для кожного оновлення.

Завдяки цьому поєднанню - логістична регресія для оцінювання ймовірностей та стохастичний спуск для оптимізації - модель здатна ефективно прогнозувати помилки в програмному коді, враховуючи набір вхідних метрик, які характеризують код. Цей підхід дозволяє швидко адаптуватися до нових даних і ефективно масштабуватися навіть при великих обсягах даних.

Оцінка моделі

Оцінка моделі є важливою частиною процесу розробки моделі машинного навчання. Для моделі логістичної регресії, яка прогнозує наявність помилок у

програмному коді, використовуються декілька стандартних метрик оцінки, кожна з яких надає унікальне розуміння ефективності моделі, а саме:

– точність (Accuracy) – це відсоток випадків, коли модель правильно класифікує дані. Вона розраховується як співвідношення кількості правильних прогнозів до загальної кількості прогнозів. Наприклад, якщо модель правильно ідентифікувала наявність або відсутність помилки у 95 випадках з 100, її точність складе 95%;

– площа під кривою ROC (AUC-ROC) - це показник, який вимірює здатність моделі розрізняти між класами. ROC (Receiver Operating Characteristic) - це крива, яка показує відношення між чутливістю моделі та її специфічністю при різних порогових значеннях. AUC (Area Under the Curve) - це площа під кривою ROC, яка вимірює здатність моделі класифікувати зразки. Чим більша ця площа, тим краще модель розрізняє між класами;

– логарифмічні втрати (Log Loss) - це метрика, яка вимірює точність прогнозів моделі. Вона карає модель за впевненість у неправильних прогнозах. Низьке значення логарифмічних втрат свідчить про кращу точність прогнозів моделі;

– F1-скор - це гармонічне середнє між точністю та повнотою. Він враховує як помилки першого типу (коли модель помилково класифікує відсутність помилки як її наявність), так і помилки другого типу (коли модель не виявляє наявність помилки). Високий F1-скор свідчить про збалансовану точність та повноту моделі.

Кожна з цих метрик надає важливу інформацію про якість моделі. Точність показує загальну здатність моделі правильно класифікувати випадки, AUC-ROC оцінює її спроможність розрізняти між класами, логарифмічні втрати вказують на впевненість моделі в її прогнозах, а F1-скор забезпечує баланс між точністю та повнотою. Використання цих метрик дозволяє отримати всебічну оцінку моделі перед її впровадженням у реальні умови.

Вихідні дані моделі

Вихідні дані моделі логістичної регресії, яка використовується для прогнозування наявності помилок у програмному коді, містять кілька ключових елементів, які дають змогу інтерпретувати результати моделі та приймати рішення про подальші дії:

– прогнозована мітка (Prediction). Це булеве значення (так/ні або істина/неправда), яке вказує, чи прогнозує модель наявність помилки у відповідному випадку програмного коду. Якщо прогнозована мітка є «істиною» (або «так»), це означає, що модель передбачає наявність помилки. Навпаки, «неправда» (або «ні») вказує на відсутність помилки за прогнозами моделі;

– ймовірність наявності помилки (Probability). Це числове значення, яке виражає ймовірність того, що програмний код містить помилку. Ймовірність виражається у форматі від 0 до 1, де значення близькі до 1 вказують на високу ймовірність наявності помилки, а значення, близькі до 0, свідчать про малоймовірність її наявності;

– оцінка кожного класу (Score). Це масив чисел, який відображає ймовірність для кожного класу. Наприклад, масив [0.2, 0.8] означає, що модель оцінює ймовірність того, що програмний код не містить помилку (клас 0), як 20%, і ймовірність того, що він містить помилку (клас 1), як 80%. Це дає змогу оцінити рівень впевненості моделі в своїх прогнозах для кожного можливого результату.

Ці вихідні дані відіграють важливу роль у визначенні подальших дій. Наприклад, якщо модель вказує високу ймовірність наявності помилки у коді, це може бути сигналом для розробника перевірити цей фрагмент коду більш ретельно. Навпаки, низька ймовірність може свідчити про те, що код, швидше за все, не містить помилок, що дозволяє розробнику бути більш впевненим у його надійності.

3.2 Базова архітектура системи

Архітектура системи для прогнозування помилок у програмному коді розроблена таким чином, щоб забезпечити ефективний збір, обробку та аналіз

даних. Вона починається з компоненту збору даних, де збираються історія змін коду, метрики коду, коментарі до коду та інші відомості, необхідні для прогнозування помилок. Після збору, ці дані передаються для передобробки, де вони очищаються, нормалізуються, кодуються та структуруються, готуючись до подальшого аналізу.

Оброблені дані надходять у модуль машинного навчання, де вони проходять крізь процеси тренування, валідації, тестування та оновлення моделі. Цей модуль використовує алгоритми машинного навчання для аналізу даних та робить прогнози щодо можливих помилок.

База даних служить як центральне сховище для всієї інформації, включаючи вхідні дані, результати аналізу та дані, пов'язані з ефективністю моделі. Вона забезпечує управління та зберігання даних, необхідних для системи.

Інтерфейс користувача дозволяє розробникам взаємодіяти з системою, вносити нові дані, переглядати результати прогнозів та отримувати звіти. Цей інтерфейс забезпечує зручний спосіб введення та отримання даних із бази даних та перегляду звітів, створених модулем звітності.

Модуль звітності генерує звіти про точність моделі, F1-скор, AUC-ROC та інші ключові метрики, використовуючи дані з бази даних. Це дозволяє користувачам системи оцінити ефективність моделі та зрозуміти її поточний стан.

На рис. 3.1 представлено базову архітектуру системи у вигляді діаграми компонентів, що відображає основні модулі системи та їх взаємодію.

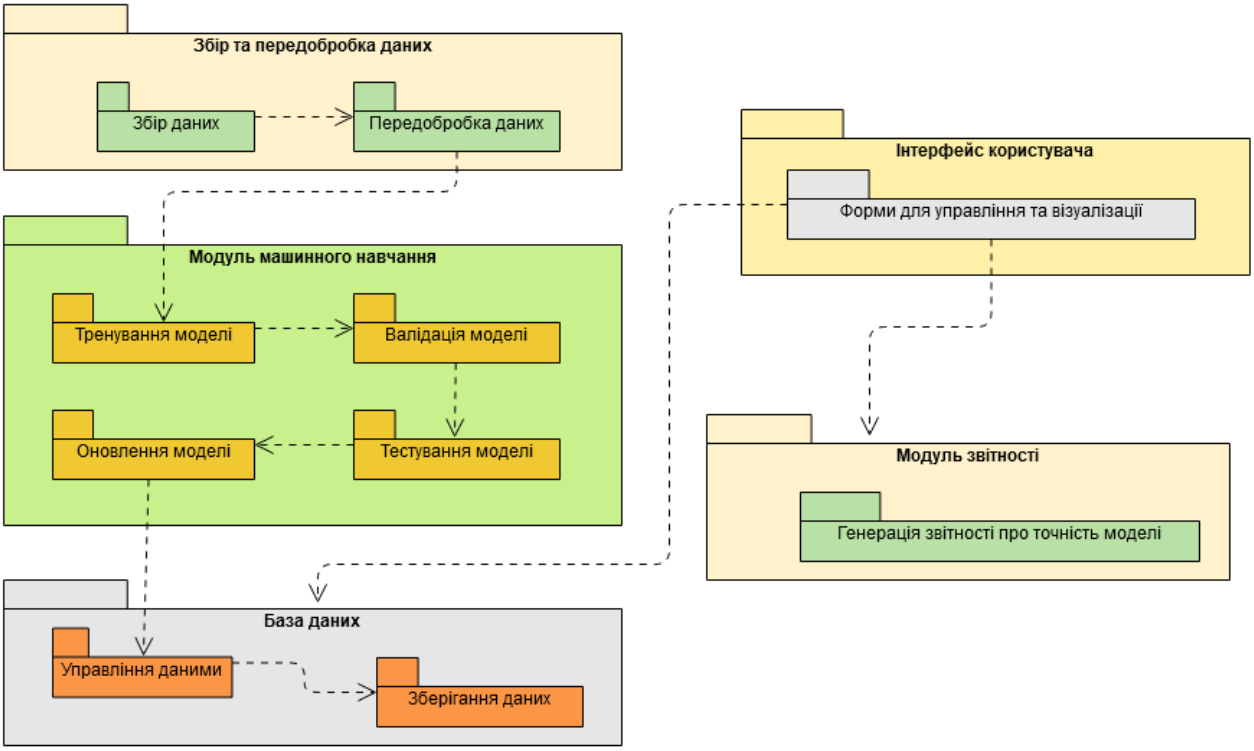

Рисунок 3.1 – Базова архітектура системи

Кожен компонент системи взаємодіє з іншими, утворюючи інтегровану структуру, яка забезпечує ефективне прогнозування помилок у програмному коді, дозволяючи користувачу попереджати помилки та підвищувати якість своїх продуктів.

3.3 Внутрішнє проектування

3.3.1 Вибір мови програмування

Підбір відповідної мови програмування є вирішальним кроком у процесі створення програмного забезпечення, оскільки від цього залежать можливості розробки, легкість підтримки та масштабування проекту.

Python, розроблена Гвідо ван Россумом на початку 90-х, є однією з найбільш гнучких та доступних мов програмування [34]. Завдяки своєму чіткому та лаконічному синтаксису, Python стала відмінним вибором для новачків, а також для ветеранів програмування. Її величезний екосистема із багатою бібліотекою забезпечує підтримку у сферах, таких як веб-розробка, наукові

дослідження, аналіз даних та розробка у галузях штучного інтелекту та машинного навчання.

Java, розроблена компанією Sun Microsystems у середині 90-х, відома своєю універсальністю та надійністю [35]. Використання Java віртуальної машини (JVM) дозволяє розробникам писати код, який може виконуватися на будь-якій платформі без змін, що відповідає принципу «Напиши одного разу, запускай будь-де». Java є популярною мовою в корпоративному секторі, для розробки веб-додатків, мобільних додатків на Android, а також у системах для вбудованих пристроїв та Інтернету речей.

C#, створена Microsoft на початку 2000-х років, є мовою, що поєднує в собі продуктивність C++ та простоту Java [36]. Ця мова є ключовою частиною платформи .NET, що забезпечує потужний набір інструментів для розробки програмного забезпечення. C# широко використовується для створення різноманітних додатків, включаючи веб-додатки, мобільні застосунки, додатки для Windows, ігри (зокрема, з використанням ігрового движка Unity) та корпоративні програми. Екосистема .NET надає розробникам потужні фреймворки та бібліотеки для швидкої та ефективної розробки.

У табл. 3.1 проведено порівняльний аналіз можливостей кожної мови програмування.

Таблиця 3.1 – Порівняльний аналіз мов програмування

Характеристика	Python	Java	C#
Типізація	Динамічна	Статична	Статична
Інтеграція з .NET	Обмежена (через IronPython)	Ні	Відмінна
Кросплатформенність	Висока	Висока (через JVM)	Висока (через .NET Core та Mono)

Використання у корпоративному секторі	Широке, але з обмеженнями	Дуже високе, особливо в корпоративних системах	Високе, інтеграція з корпоративними інструментами Microsoft
Підтримка багатопотоковості	Обмежена через Global Interpreter Lock (GIL) [37]	Відмінна	Відмінна з Task Parallel Library (TPL)
Підтримка шаблонів	Ні	Часткова (Generics)	Відмінна (Generics та LINQ)
Вбудована підтримка властивостей	Ні	Ні	Так
Швидкість виконання	Помірна	Висока	Висока
Спільнота та підтримка	Велика	Велика	Велика, з сильною підтримкою від Microsoft
Розробка інтерфейсу користувача	Обмежені можливості	Свінг, JavaFX	WPF, WinForms

Вибір C# для розробки застосунку прогнозування появи помилки в програмному коді можна обґрунтовується кількома ключовими факторами. По-перше, C# має сильну інтеграцію з .NET Framework та .NET Core, що надає великий набір бібліотек та інструментів, корисних для обробки даних та машинного навчання. Це включає підтримку для комплексних математичних операцій, статистичного аналізу та ефективної обробки великих наборів даних. По-друге, C# забезпечує високу продуктивність та оптимізацію, що є важливим для аналітичних застосунків, які вимагають обробки великих обсягів даних та

складних обчислень. Мова також підтримує багатопотоковість, що дозволяє ефективно використовувати ресурси обчислювальної системи.

Третім фактором є міцна підтримка безпеки та надійності в C#, що є критично важливим для будь-якого програмного забезпечення, особливо того, що обробляє чутливі або важливі дані. C# пропонує розширені можливості управління пам'яттю, обробки винятків та безпечної роботи з ресурсами.

В сукупності ці переваги роблять C# відмінним вибором для розробки комплексних та високоефективних систем прогнозування помилок в програмному коді.

3.3.2 Технологічна платформа

Вибір платформи є критичним, адже від цього залежать можливості майбутньої системи, її ефективність, гнучкість та масштабування. Серед доступних технологій, .NET від Microsoft виокремлюється своїми унікальними характеристиками, які роблять її особливо підходящою для реалізації проекту [38]:

– широкі можливості для машинного навчання. .NET підтримує ML.NET, бібліотеку машинного навчання, яка дозволяє інтегрувати моделі машинного навчання безпосередньо у .NET додатки. Це важливо для створення моделей, які можуть прогнозувати помилки в коді, базуючись на історичних даних та різних метриках коду;

– сумісність та інтеграція. .NET пропонує відмінну інтеграцію з різними інструментами розробки, базами даних та іншими технологічними рішеннями. Це забезпечує легкість в інтеграції системи прогнозування помилок із існуючими проектами та розробкою;

– мовна гнучкість. Підтримує кілька мов програмування, включаючи C#, F# та Visual Basic, надаючи розробникам можливість вибрати найбільш відповідну мову для конкретного випадку використання;

– продуктивність та надійність. .NET відомий своєю високою продуктивністю та надійністю, що є ключовими для розробки надійних систем прогнозування. Особливо це актуально при обробці великих обсягів даних та виконанні складних обчислень;

– безпека та управління ресурсами. Пропонує розширені можливості управління пам'яттю та безпеки, що є важливим при обробці чутливих даних та виконанні критично важливих операцій;

– підтримка спільноти та ресурси. .NET має велику спільноту розробників та багатий набір ресурсів для навчання та підтримки, що може сприяти швидкому розв'язанню проблем та обміну знаннями;

– кросплатформенність. З появою .NET Core, .NET став кросплатформенним, що дозволяє запускати додатки на Windows, Linux та macOS, розширюючи можливості використання системи прогнозування помилок на різних платформах.

У цілому, вибір .NET як технологічної платформи для розробки системи прогнозування помилок у програмному коді забезпечує потужні інструменти, гнучкість та надійність, що є необхідними для створення ефективного та надійного рішення.

3.3.3 Ієрархія та взаємодія класів системи

У рамках розробки системи для прогнозування появи помилок у програмному коді була використана трьохрівнева архітектура, що забезпечує чітке розділення функціональності та спрощує процес розробки та масштабування [39]. Ця архітектура включає наступні рівні:

– рівень Даних (Data Access Layer - DAL). Цей рівень забезпечує інтерфейс для доступу до даних, що зберігаються в базі даних. Він включає класи, які відповідають за взаємодію з базою даних та за управління даними. Кожен клас на цьому рівні представляє певну таблицю або набір даних і забезпечує методи для їх читання, додавання, оновлення та видалення. Це

дозволяє ізолювати логіку роботи з даними від інших частин системи, спрощуючи подальшу розробку та підтримку;

– рівень Бізнес-Логіки (Business Logic Layer). На цьому рівні реалізується основна логіка програми. Цей рівень відповідає за обробку вхідних даних, виконання необхідних обчислень та алгоритмів прогнозування. Класи бізнес-логіки обробляють дані, отримані від рівня DAL, і готують результати для їх подальшої візуалізації або використання. Важливим є те, що цей рівень повністю відокремлений від рівня представлення, що дозволяє змінювати внутрішню логіку без впливу на користувацький інтерфейс;

– рівень Представлення (Presentation Layer). Цей рівень відповідає за відображення інформації користувачеві та взаємодію з ним. Він включає інтерфейси користувача (графічні, веб-інтерфейси тощо), через які відбувається взаємодія з системою. Рівень представлення отримує оброблені дані з рівня бізнес-логіки і відображає їх у зручному для користувача форматі. Це може включати таблиці результатів, графіки, повідомлення про статус аналізу та інші елементи інтерфейсу.

На самому початку був розроблений рівень доступу до даних, який забезпечує інтерактивний доступ до бази даних і є фундаментальним для всієї системи (рис. 3.2). У рамках DAL, для кожного типу даних, який використовується в системі, створено відповідний клас. Ці класи виконують функції доступу до конкретних таблиць бази даних та забезпечують взаємодію з ними. Вони дозволяють виконувати стандартні операції CRUD (створення, читання, оновлення та видалення даних) для кожної таблиці. Наприклад, може бути клас для зберігання метрик коду, історії змін, результатів аналізу тощо.

Кожен з класів DAL має відповідний інтерфейс, який визначає методи доступу до даних. Ця практика дозволяє легко модифікувати чи замінювати реалізацію доступу до даних без впливу на інші частини системи, що забезпечує високу гнучкість та масштабованість архітектури.

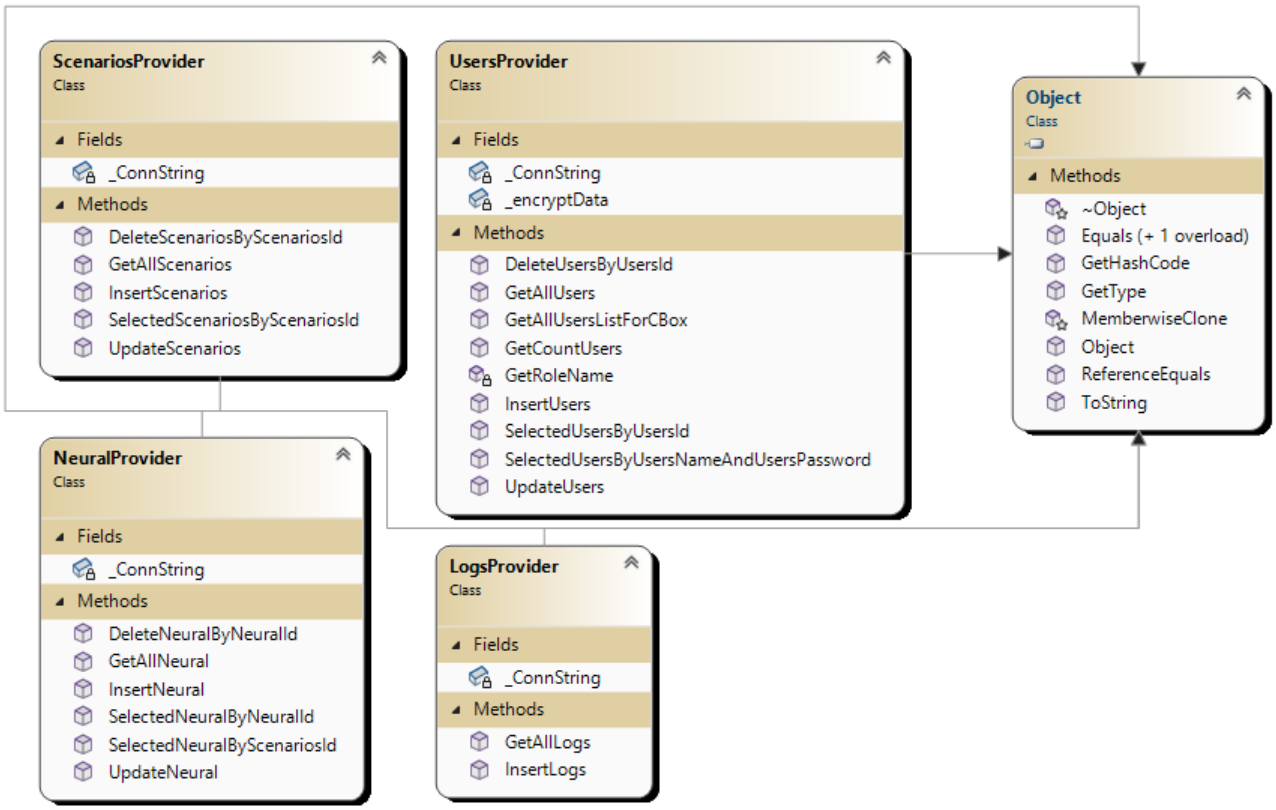

Рисунок 3.2 – Діаграма класів рівня даних

Діаграма рівня даних складається із наступних класів:

– клас LogsProvider забезпечує зберігання та обробку журналів подій або логів, які відображають різні дії або зміни в системі. Ці дані включають записи про внесені зміни в код, повідомлення про помилки, аудиторські записи та інші відомості, які допомагають зрозуміти контекст та історію роботи системи. LogsProvider забезпечує методи для збору, зберігання та витягу цих логів, дозволяючи системі ефективно використовувати ці дані для аналізу та прогнозування;

– клас NeuralProvider відповідає за взаємодію з моделями машинного навчання, використовуваними для прогнозування помилок. Цей клас управляє тренуванням, оцінюванням та застосуванням нейронних мереж, забезпечуючи обробку вхідних даних та інтерпретацію вихідних результатів;

– клас ScenariosProvider відповідає за зберігання та управління сценаріями використання, які система використовує для тестування та аналізу програмного коду. Ці сценарії включають специфікації тестових випадків, умови

використання та інші відомості, які допомагають системі імітувати реальні ситуації або виклики, з якими може зустрітися програмний код;

– клас UsersProvider призначений для управління даними користувачів системи. Він включає інформацію про користувачів, їх ролі, доступ та інші налаштування. Це включає дані для аутентифікації, права доступу до різних модулів системи, персоналізацію налаштувань тощо. UsersProvider гарантує, що система має надійний механізм управління користувацькими даними, що є критично важливим для забезпечення безпеки та персоналізованого досвіду користувачів.

Кожен з цих класів є важливою частиною архітектури системи та відіграє свою роль у забезпеченні її функціональності та ефективності. Взаємодія класів дозволяє створити комплексну та інтегровану систему, яка здатна ефективно вирішувати задачі прогнозування помилок у програмному коді.

Після цього розроблено бізнес логіку додатку, діаграма класів якого зображена на рис. 3.3.

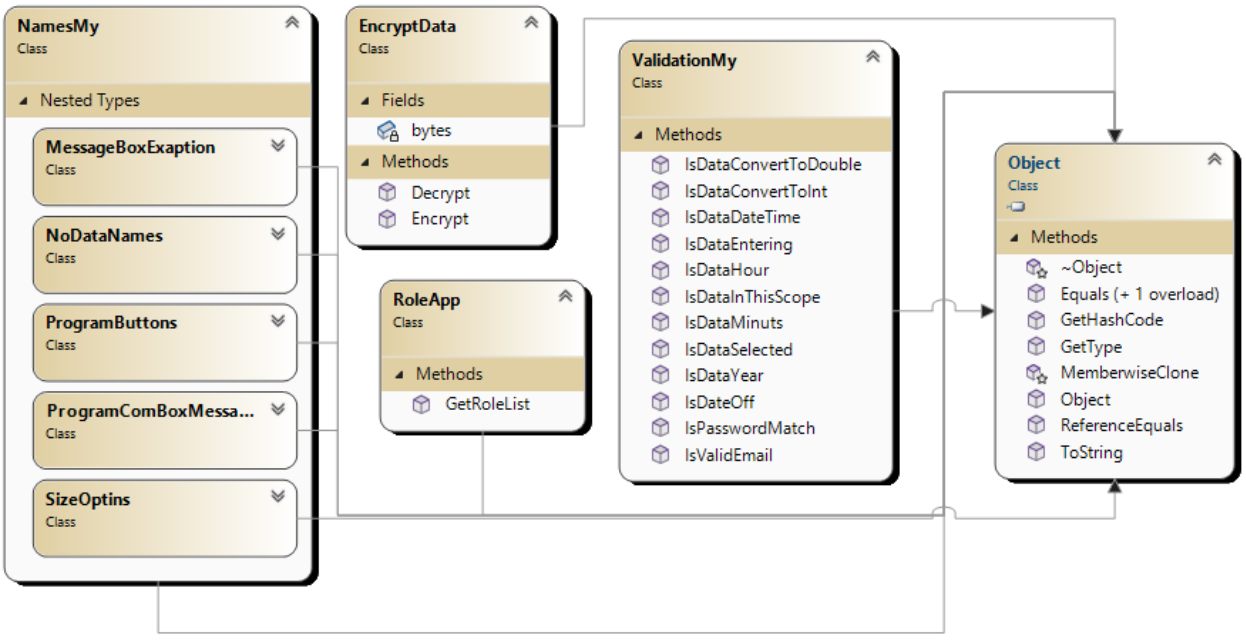

Рисунок 3.3 – Діаграма бізнес-логіки додатку

Дана діаграма складається із таких класів:

– клас EncryptData забезпечує безпеку системи. Він відповідає за шифрування та дешифрування даних, що обробляються або зберігаються в

системі. Це включає шифрування користувацьких даних, конфіденційної інформації, а також результатів аналізу. Використання EncryptData гарантує, що чутлива інформація залишається захищеною від несанкціонованого доступу або витоків;

– клас NamesMy функціонує як управлінський компонент для обробки та зберігання номенклатур і іменувань, які використовуються в системі. Цей клас включає логіку для створення, зберігання та управління різними назвами або категоріями, які використовуються в різних модулях системи;

– клас RoleApp призначений для управління ролями користувачів у системі. Цей клас відповідає за визначення різних ролей та їх дозволів у контексті системи. Він управляє тим, які дії можуть виконувати користувачі з різними ролями, наприклад, доступ до певних даних, можливість зміни параметрів системи, запуску аналізу тощо. Використання RoleApp сприяє ефективному управлінню доступом та безпекою системи;

– клас ValidationMy забезпечує точність та надійність даних, що використовуються в системі. Він відповідає за валідацію даних, що вводяться користувачами або отримуються з зовнішніх джерел. ValidationMy перевіряє коректність форматів, діапазони значень, відповідність встановленим правилам та інші критерії, що гарантують, що дані є правильними та надійними для подальшої обробки.

Розробка користувацького інтерфейсу є вирішальним та завершальним етапом у процесі створення системи для дослідження методів прогнозування появи помилки в програмному коді. Цей етап має велике значення, оскільки користувацький інтерфейс виступає в ролі моста між складними алгоритмами системи та її кінцевими користувачами.

На рис. 3.4 представлена діаграма, що демонструє структуру та взаємодію різних компонентів користувацького інтерфейсу. Ця діаграма відіграє важливу роль у візуалізації взаємозв'язків між різними елементами інтерфейсу та допомагає розробникам зрозуміти, як користувачі будуть взаємодіяти з системою.

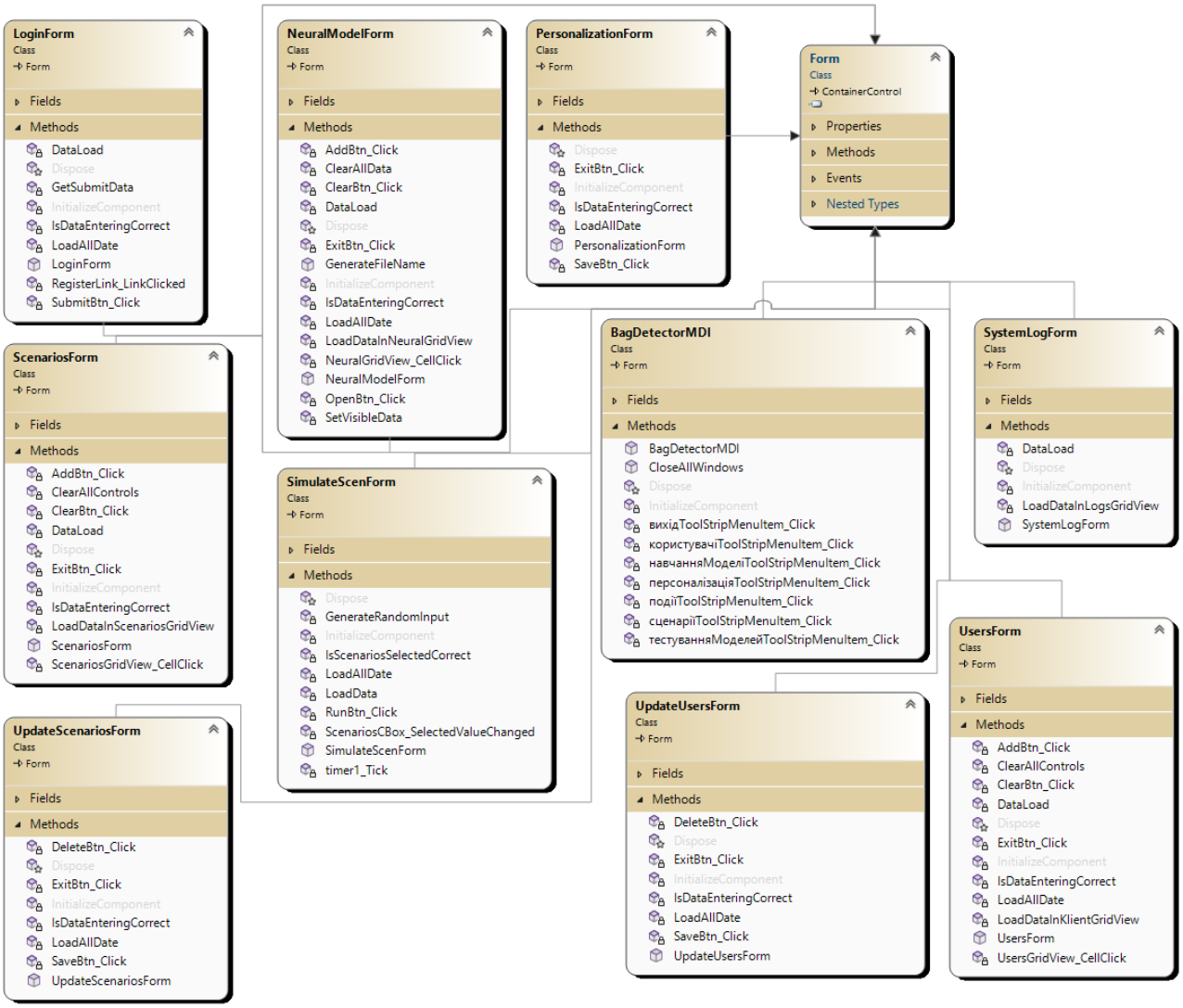

Рисунок 3.4 – Діаграма класів презентаційного рівня застосунку

Діаграма класів даного рівня складається із:

– BagDetectorMDI є головним інтерфейсним вікном (MDI - Multiple Document Interface) системи, яке використовується для відображення та управління різними вікнами чи формами всередині одного основного інтерфейсу. Цей клас включає навігаційні елементи для доступу до різних функцій системи, таких як симуляція сценаріїв, аналіз даних, перегляд логів тощо;

– SimulateScenForm є формою, призначеною для симуляції різних сценаріїв, що дозволяє користувачам тестувати та аналізувати різні умови та випадки використання. Цей надає інтерфейс для вибору сценарію та запуску симуляцій;

– UpdateScenariosForm використовується для оновлення та редагування існуючих сценаріїв. Ця форма дозволяє користувачам змінювати деталі сценаріїв, вносити нові дані або коригувати параметри для точнішого моделювання;

– ScenariosForm є інтерфейсом для перегляду та управління списком сценаріїв. Ця форма забезпечує огляд доступних сценаріїв, їх статусів та іншої важливої інформації;

– NeuralModelForm призначена для взаємодії з нейронними моделями системи. Цей клас надає інструменти для вибору файлів із тренувальними наборами, запуску процесу тренування та оцінки її ефективності;

– LoginForm використовується для аутентифікації користувачів у системі. Ця форма забезпечує введення імені користувача та пароля для доступу до функцій системи, залежно від ролі та прав користувача;

– PersonalizationForm дозволяє користувачам налаштовувати персональні параметри користувача;

– SystemLogForm призначений для перегляду системних логів. Цей клас дозволяє користувачам переглядати записи про події в системі, такі як помилки, операції користувачів або системні повідомлення, що важливо для моніторингу та аудиту;

– UpdateUsersForm використовується для управління користувацькими обліковими записами. Ця форма дозволяє адміністраторам додавати, видаляти або змінювати облікові записи користувачів, їх ролі та права доступу;

– UsersForm забезпечує перегляд та управління списком користувачів системи. Цей клас дозволяє переглядати інформацію про користувачів, їх статуси та ролі в системі, сприяючи легкому управлінню доступом та безпекою.

Кожен з цих класів відіграє важливу роль у забезпеченні зручного та інтуїтивно зрозумілого інтерфейсу користувача, що дозволяє ефективно взаємодіяти з системою та використовувати її можливості.

3.3.4 Використані принципи проектування

У процесі розробки системи для прогнозування появи помилок у програмному коді були використані наступні ключові принципи об'єктно-орієнтованого проектування:

– принцип єдиного обов’язку. Кожен клас у системі відповідає за виконання лише однієї функції. Це допомагає уникнути змішування різної функціональності в одному класі, що спрощує розуміння, тестування та підтримку коду. Наприклад, в системі може бути окремий клас для обробки даних, інший для аналізу даних та третій для відображення результатів;

– принцип відкритості/закритості. Система розроблена таким чином, що її компоненти відкриті для розширення, але закриті для змін. Це означає, що можна додавати нові функції, не змінюючи існуючий код. Наприклад, модуль прогнозування може бути розширений новими алгоритмами без необхідності зміни існуючих модулів;

– розділення інтерфейсу. Система використовує спеціалізовані інтерфейси замість універсальних, щоб забезпечити більшу гнучкість та ясність у взаємодії між різними компонентами системи. Це означає, що кожен модуль має визначений, чіткий інтерфейс для взаємодії з іншими частинами системи.

Застосування цих принципів об'єктно-орієнтованого проектування забезпечує створення системи, яка є гнучкою, масштабованою та легкою в підтримці. Це створює надійну основу для ефективного прогнозування помилок в програмному коді.

3.3.5 Використані шаблони проектування

Для розробки системи для прогнозування появи помилок у програмному коді, використання шаблону проектування MVP (Model-View-Presenter) виявляється особливо доцільним з кількох причин [40]:

– чітке розділення відповідальностей. MVP розділяє програму на три основні компоненти: Model (модель), View (представлення) та Presenter

(презентер). Модель відповідає за бізнес-логіку та обробку даних, View - за інтерфейс користувача, а Presenter - за зв'язок між Model та View. Таке розділення сприяє легшій підтримці та розширенню системи;

– гнучкість у тестуванні. MVP полегшує написання модульних тестів, оскільки бізнес-логіка відокремлена від інтерфейсу користувача. Presenter можна легко тестувати незалежно від користувацького інтерфейсу, що підвищує якість коду та забезпечує надійність системи;

– легкість заміни компонентів. Використання MVP дозволяє легко замінювати окремі компоненти системи. Наприклад, можна змінити інтерфейс користувача (View), не впливаючи на бізнес-логіку (Model) чи презентаційну логіку (Presenter). Це особливо корисно для розробки в умовах, де вимоги до інтерфейсу користувача часто змінюються;

– підтримка чистоти коду та легкість. MVP сприяє написанню чистого, організованого коду. Кожен компонент має своє чітке призначення, що робить код більш читабельним та зрозумілим для нових розробників або при переході на нові версії системи;

– вдосконалення користувацького досвіду. MVP забезпечує кращий користувацький досвід, оскільки Presenter може активно управляти тим, як дані відображаються в View. Це дозволяє реагувати на зміни у моделі або вимогах користувачів швидше та ефективніше.

При реалізації проекту з прогнозування помилок у програмному коді, MVP може бути застосований для розробки користувацького інтерфейсу, який дозволяє користувачам взаємодіяти з системою, наприклад, вводити дані для аналізу або переглядати результати прогнозу. Модель може обробляти вхідні дані та виконувати обчислення, Presenter може обробляти логіку відображення даних та інтерактивність, а View відображати результати користувачу.

Отже, MVP є оптимальним вибором для розробки системи прогнозування помилок, оскільки він забезпечує ефективну організацію коду, полегшує тестування та підтримку, та дозволяє швидко реагувати на зміни вимог чи інтерфейсу.

3.4 Розробка інтерфейсу користувача

3.4.1 Розробка структури екранів системи

У системі для дослідження методів прогнозування появи помилок у програмному коді, першим критичним модулем, з яким зіткнеться будь-який користувач, стане модуль авторизації та реєстрації. Важливість цього модуля полягає у забезпеченні контролю доступу до системи, що є ключовим для збереження безпеки та конфіденційності даних.

У рамках цього модуля передбачені два окремих інтерфейси: один для авторизації та інший для реєстрації нових користувачів. У формі авторизації користувачам необхідно буде ввести свої дані для входу у систему, зазвичай логін та пароль (рис. 3.5). Після підтвердження даних система перевіряє їх коректність і, у разі успішної авторизації, надає доступ до основного інтерфейсу системи. У випадку помилок авторизації користувачеві буде надано відповідне повідомлення. Також у вікні авторизації передбачена можливість переходу до форми реєстрації, яка необхідна для нових користувачів.

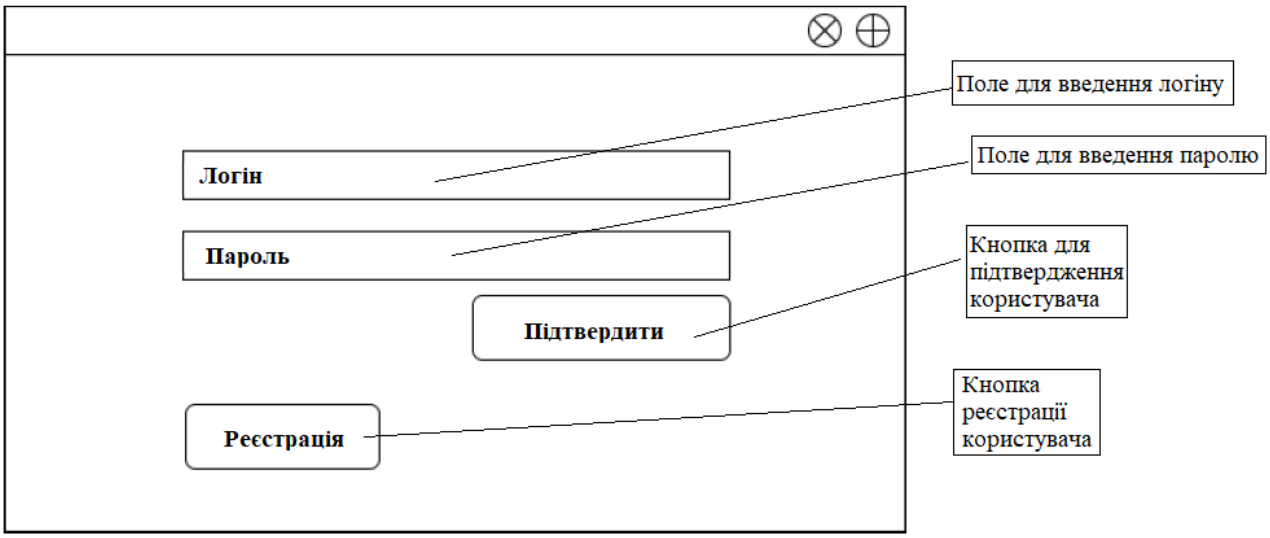

Рисунок 3.5 – Форма авторизації користувача

Форма реєстрації містить поля для введення основних реєстраційних даних, включаючи персональну інформацію, таку як прізвище, ім'я, електронну пошту, а також обрані логін та пароль (рис. 3.6). Після заповнення цієї інформації

користувач має можливість підтвердити введені дані, натиснувши відповідну кнопку. Це ініціює процес реєстрації, при якому дані заносяться у систему.

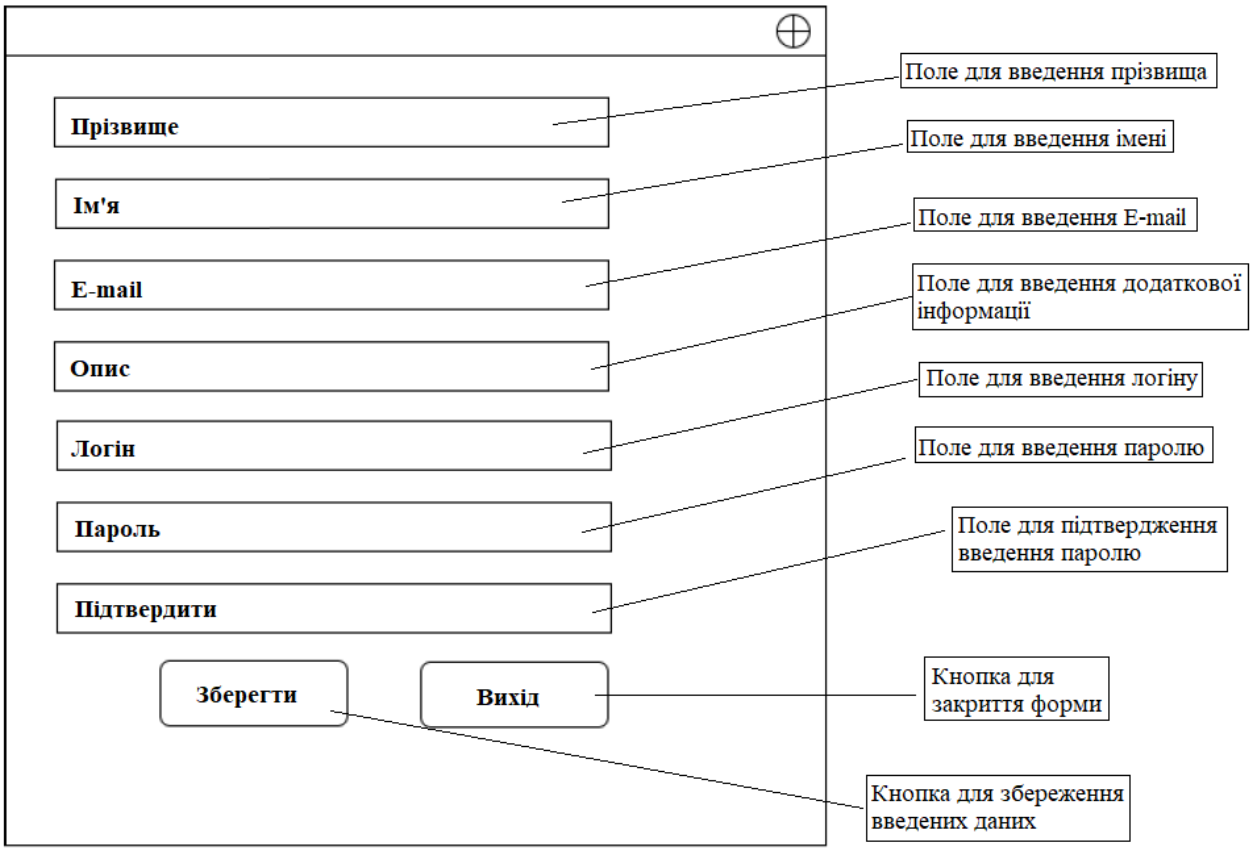

Рисунок 3.6 – Форма реєстрації користувача

Обидва інтерфейси, авторизації та реєстрації, розроблені з урахуванням зручності користувачів та безпеки системи. Вони є першим етапом взаємодії користувачів із системою прогнозування помилок у програмному коді, і їх якісна реалізація важлива для створення позитивного першого враження та забезпечення безпечного використання системи.

Форма навчання моделей є ключовим інструментом, який дозволяє користувачам тренувати та оцінювати алгоритми машинного навчання (рис. 3.7). Ця форма забезпечує інтерфейс для вибору даних, їх підготовки та запуску процесу навчання. Користувачі можуть використовувати цей інструмент для завантаження наборів даних, встановлення параметрів моделі та відстеження процесу навчання, що включає моніторинг точності, AUC, логарифмічного втрати та інших важливих метрик ефективності. Завершивши процес навчання, користувачі мають можливість зберегти навчену модель для подальшого

використання в системі. Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

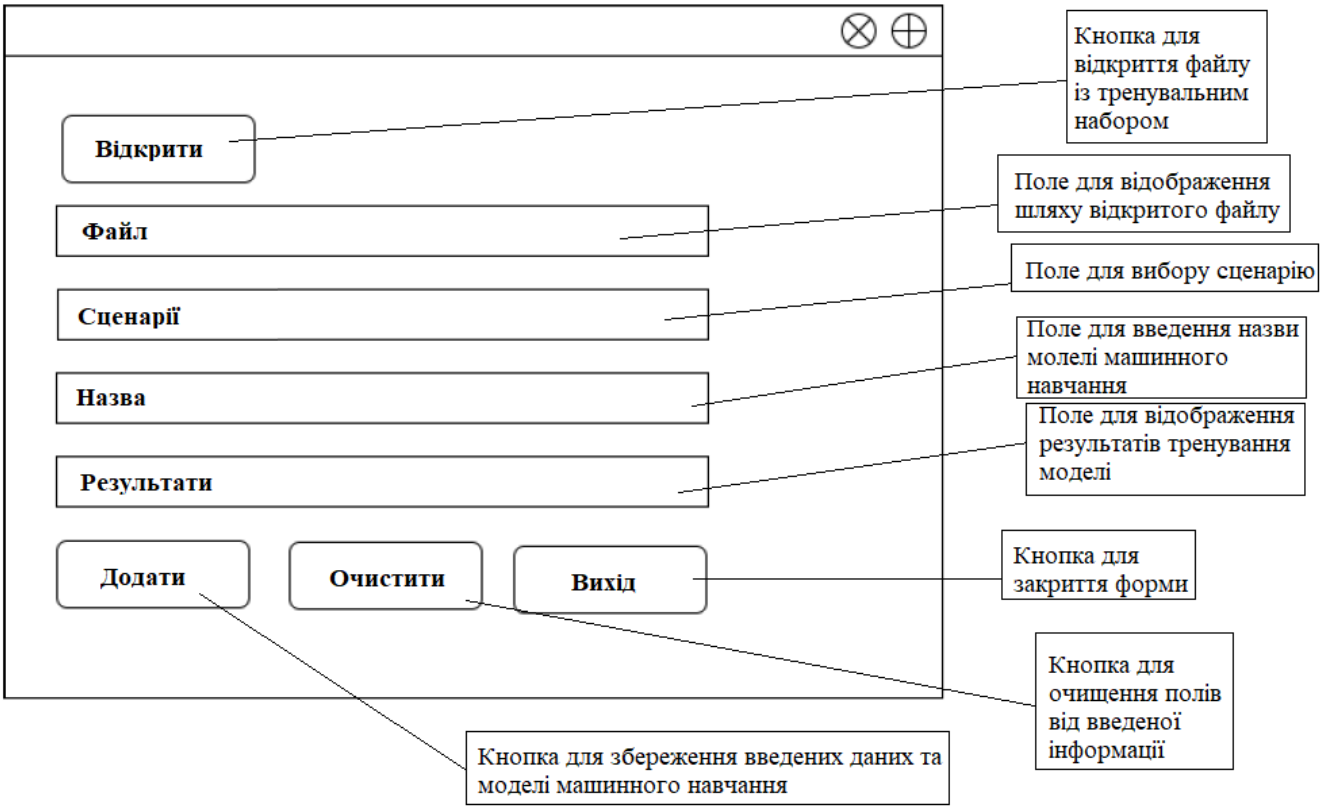

Рисунок 3.7 – Форма тренування моделей машинного навчання

Також важливою є форма симуляції сценаріїв у системі, що призначена для аналізу та тестування моделей (рис. 3.8). . Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

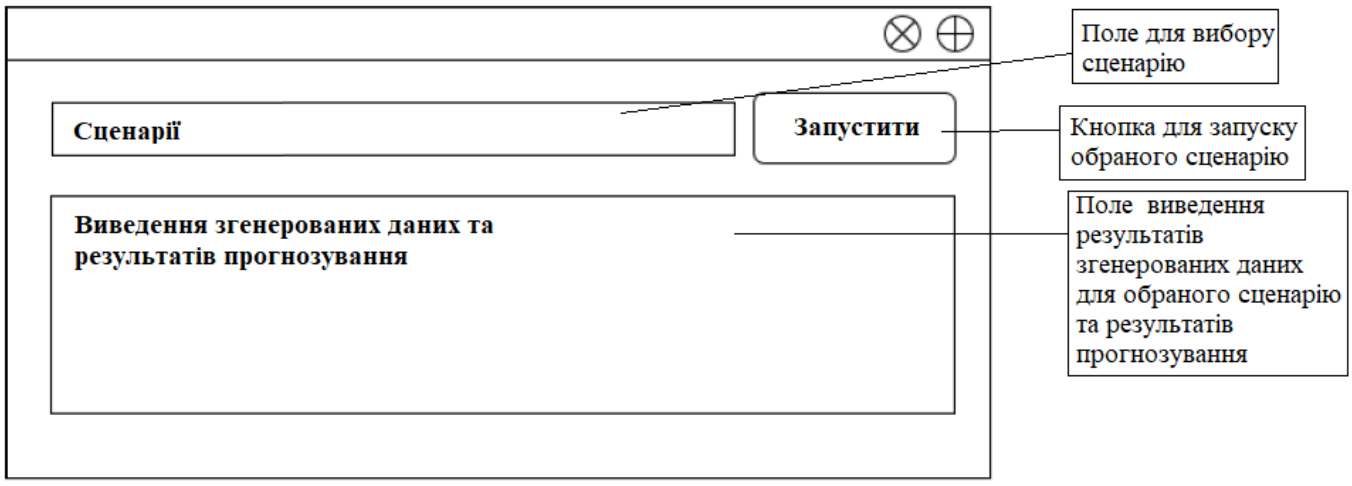

Рисунок 3.8 – Форма аналізу та тестування моделі

Ця форма дозволяє користувачам вибирати різні сценарії та запускати моделі машинного навчання на згенерованих або вибраних даних. У ній реалізовано функціонал для вибору конкретного сценарію з доступного переліку, а також забезпечено можливість завантаження необхідних даних для симуляції. Після запуску симуляції користувачі можуть спостерігати за процесом та аналізувати результати прогнозування, що відображаються у формі в реальному часі. Таким чином, форма симуляції сценаріїв є ключовим компонентом системи, який надає інструменти для глибокого аналізу та перевірки ефективності моделей прогнозування.

3.4.2 Реалізація інтерфейсу користувача

На наступному етапі було створено користувацький інтерфейс, який служить мостом між користувачами та програмним забезпеченням. Цей інтерфейс, ілюстрований на рисунку 3.9, забезпечує зручне та інтуїтивно зрозуміле середовище для взаємодії з різними функціями системи. . Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

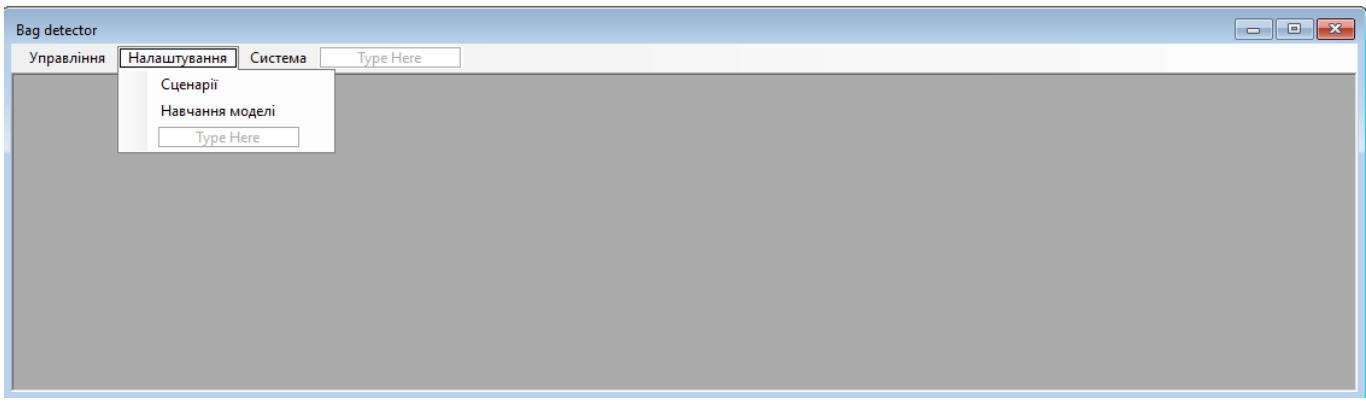

Рисунок 3.9 – Головна форма

Для кожної опції в меню було розроблено специфічну реакцію системи, яка активізується при її виборі. Наприклад, як показано на рисунку 3.10, при виборі опції "Тестування моделей" користувачем система відповідає відповідним чином, відкриваючи інструментарій для тестування та оцінки моделей.

```
private void тестуванняМоделейToolStripMenuItem_Click(object sender, EventArgs e) {
  CloseAllWindows();
  SimulateScenForm simulateScenForm = new SimulateScenForm();
  simulateScenForm.MdiParent = this;
  simulateScenForm.WindowState = FormWindowState.Maximized;
  simulateScenForm.Show();
}
```

Рисунок 3.10 – Події виклику форми «Тестування моделей»

Важливою є розробка форми для проведення тренування моделей машинного навчання, зовнішній вигляд якої показано на рис. 3.11. . Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

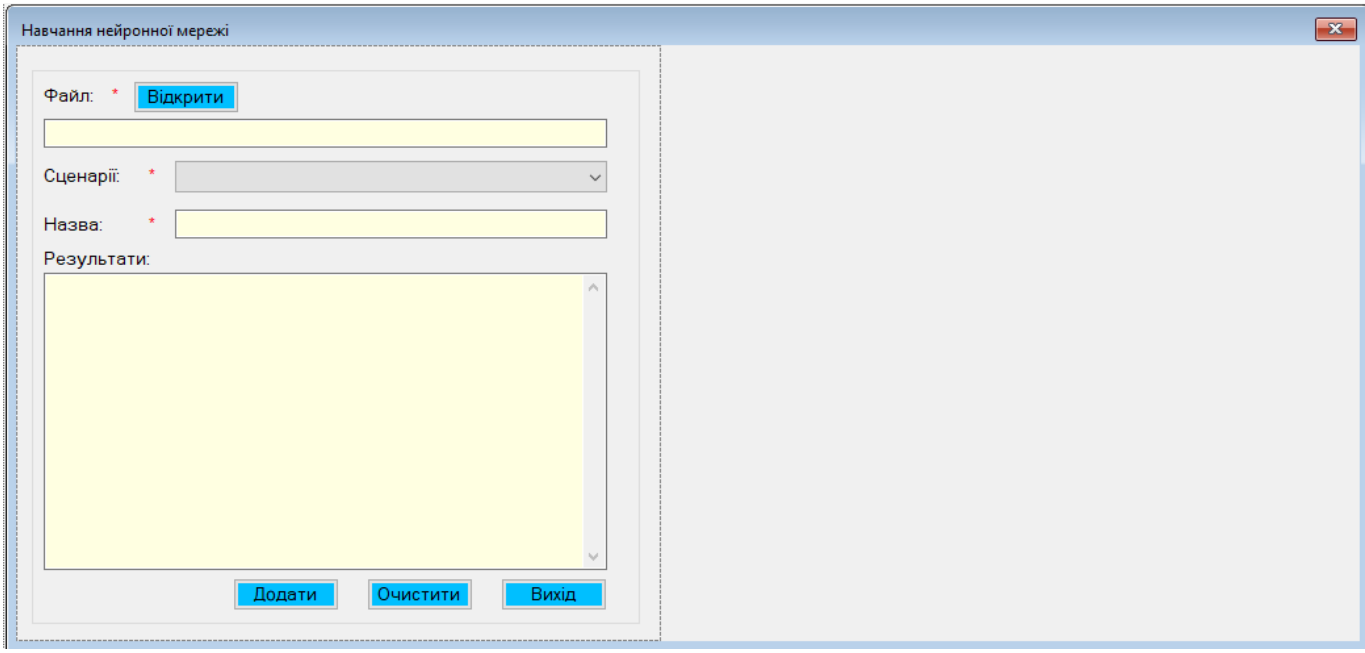

Рисунок 3.11 – Форма для навчання моделей

На рис 3.12 наведено фрагмент коду, який використовується для того, щоб користувач міг вибрати файл зі своєї системи та щоб програма могла отримати шлях до цього файлу для подальшої роботи з ним.

```
// Відображення діалогового вікна та обробка результату
if (openFileDialog.ShowDialog() == DialogResult.OK) {

  _Path = openFileDialog.FileName;
  FileNameTBox.Text = openFileDialog.FileName;
```

Рисунок 3.12 – Код для відкриття файлу із тренувальним набором

Після відкриття файлу проходить ініціалізація контексту машинного навчання в програмі (рис. 3.13).

```
// Створення контексту ML
mlContext = new MLContext(seed: 0);
```

Рисунок 3.12 – Ініціалізація контексту

Команда створює новий екземпляр основного об'єкта для роботи з машинним навчанням, задаючи початкове значення для генерації випадкових чисел. Це початкове значення дозволяє забезпечити повторюваність результатів при кожному запуску програми, що є важливим для точності та надійності виконання алгоритмів машинного навчання.

Далі проходить завантаження даних із текстового файлу для подальшого використання у машинному навчанні (рис. 3.13).

```
// Завантаження даних
dataView = mlContext.Data.LoadFromTextFile<ModelInput>(_Path,
  hasHeader: true,
  separatorChar: ',');
```

Рисунок 3.13 – Завантаження даних

Використовується функція, яка бере шлях до файлу та вказує, що цей файл має заголовок та дані в ньому розділені комою. Результатом цієї операції є структура даних, яка готова до обробки алгоритмами машинного навчання. Це важливий крок у процесі підготовки даних, оскільки правильне завантаження та форматування даних є ключовим для успішного аналізу та прогнозування.

Завантажені дані необхідно розділити на дві частини: одну для тренування моделі, іншу для її тестування (рис. 3.14). Встановлено, що 20% даних будуть використані для тестування, а решта 80% - для тренування моделі. Це дозволяє оцінити, наскільки добре модель працює з новими даними, які не використовувалися під час її тренування, що є важливим для перевірки точності та надійності прогнозів моделі.

```
// Розділення даних на тренувальний та тестовий набори
var splitData = mlContext.Data.TrainTestSplit(dataView, testFraction: 0.2);
var trainData = splitData.TrainSet;
var testData = splitData.TestSet;
```

Рисунок 3.14 – Розподіл даних із тренувального набору

Після цього потрібно підготувати дані для машинного навчання (рис. 3.15). Створюється послідовність обробки даних, де об'єднуються різні

характеристики, які використовуються для тренування моделі. Ці характеристики включають різноманітні параметри, такі як кількість методів у класі, глибина дерева наслідування, зв'язність класів, кількість викликаних функцій та інші. Після об'єднання цих характеристик, вони нормалізуються для забезпечення однорідності даних, що поліпшує якість та ефективність тренування моделі.

```
// Визначення потоку обробки даних
var dataProcessPipeline = mlContext.Transforms.Concatenate("Features",
    nameof(ModelInput.Wmc),  nameof(ModelInput.Dit), nameof(ModelInput.Noc),
    nameof(ModelInput.Cbo),  nameof(ModelInput.Rfc), nameof(ModelInput.Lcom),
    nameof(ModelInput.Ca),  nameof(ModelInput.Ce), nameof(ModelInput.Npm),
    nameof(ModelInput.Lcom3), nameof(ModelInput.Loc), nameof(ModelInput.Dam),
    nameof(ModelInput.Moa), nameof(ModelInput.Mfa),  nameof(ModelInput.Cam),
    nameof(ModelInput.Ic), nameof(ModelInput.Cbm), nameof(ModelInput.Amc),
    nameof(ModelInput.MaxCc), nameof(ModelInput.AvgCc))
.Append(mlContext.Transforms.NormalizeMinMax("Features"));
```

Рисунок 3.15 – Підготовка конвеєра даних

Далі необхідно обрати алгоритм машинного навчання. Задіяно метод логістичної регресії (рис. 3.16). Він використовується для аналізу залежності між вибраними характеристиками та цільовою змінною. Після вибору алгоритму, він додається до попередньо створеного потоку обробки даних. Це дозволяє системі виконувати тренування моделі на основі підготовлених даних, використовуючи вибраний алгоритм.

```
// Вибір алгоритму навчання
var trainer =
mlContext.BinaryClassification.Trainers.SdcaLogisticRegression(labelColumnName: "Label",
featureColumnName: "Features");
var trainingPipeline = dataProcessPipeline.Append(trainer);
```

Рисунок 3.16 – Вибір алгоритму

На наступному етапі відбувається процес тренування моделі (рис. 3.17) при якому застосовується попередньо визначений потік обробки даних та обраний алгоритм навчання до набору даних, призначеного для тренування. Цей крок є вирішальним для формування моделі, яка зможе виконувати задачі прогнозування або класифікації на основі вхідних даних. Модель "вчиться" на основі наданих тренувальних даних, адаптуючи свої параметри для досягнення найкращої продуктивності.

```
//Навчання моделі
model = trainingPipeline.Fit(splitData.TrainSet);
```

Рисунок 3.17 – Навчання моделі

Після цього відбувається процес оцінки результатів навчання моделі та виведення її метрик у відповідне поле (рис. 3.18). Спочатку модель використовується для виконання прогнозів на основі тестового набору даних. Потім виконується оцінка цих прогнозів, щоб визначити ефективність моделі, зокрема за такими метриками, як точність, площа під кривою характеристики роботи приймача, логарифмічні втрати та F1-оцінка. Результати оцінки додаються до текстового поля для інформування користувача про ефективність моделі. Це дозволяє користувачам отримати уявлення про надійність та точність моделі у рішеннях прогнозування.

```
//Оцінка моделі
var predictions = model.Transform(splitData.TestSet);
var metrics = mlContext.BinaryClassification.Evaluate(predictions);

RaportTBox.Text += ($"Accuracy: {metrics.Accuracy:P2}") + "\r\n";
RaportTBox.Text += ($"AUC: {metrics.AreaUnderRocCurve:P2}") + "\r\n";
RaportTBox.Text += ($"Log Loss: {metrics.LogLossReduction:P2}") + "\r\n";
RaportTBox.Text += ($"F1Score: {metrics.F1Score:P2}") + "\r\n";
```

Рисунок 3.18 – Оцінка моделі та виведення її метрик

Для збереження даних навчання моделі реалізовано подію «AddBtn_Click», код якої представлено на рис. 3.19. . Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

```
1 reference
private void AddBtn_Click(object sender, EventArgs e) {
  if (IsDataEnteringCorrect()) {
    //Save model
    string pathName = @"\teach\" + GenerateFileName() + ".zip";
    string localProj =
      System.IO.Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().Location);
    _NeuralProvider.InsertNeural(NeuralNamesTBox.Text,
      Convert.ToInt32(ScenariosCBox.SelectedValue), pathName);
    mlContext.Model.Save(model, dataView.Schema, localProj + pathName);
    ClearAllData();
    _LogsProvider.InsertLogs(LoginForm.CurrentUser.UsersId,
      "Було навчено нейронну мережу " +
      NeuralNamesTBox.Text, DateTime.Now);
    MessageBox.Show("Дані успішно збережено!");
  }
}
```

Рисунок 3.19 – Метод збереження даних моделі

Метод активується після натискання користувачем на кнопку «Додати». Спочатку перевіряється, чи правильно введені дані. Після підтвердження правильності даних, виконується процедура збереження моделі. Формується унікальне ім'я файлу та шлях для збереження моделі. Модель зберігається у вказаному місці. Після збереження моделі, очищаються введені дані та вноситься запис у журнал подій про навчання моделі. На завершення користувач отримує повідомлення про успішне збереження даних. Це дозволяє користувачам зберегти свої навчені моделі для подальшого використання та аналізу.

Для тестування навчених моделей було реалізовано форму «Тестування моделей», зовнішній вигляд якої представлено на рис. 3.20. . Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

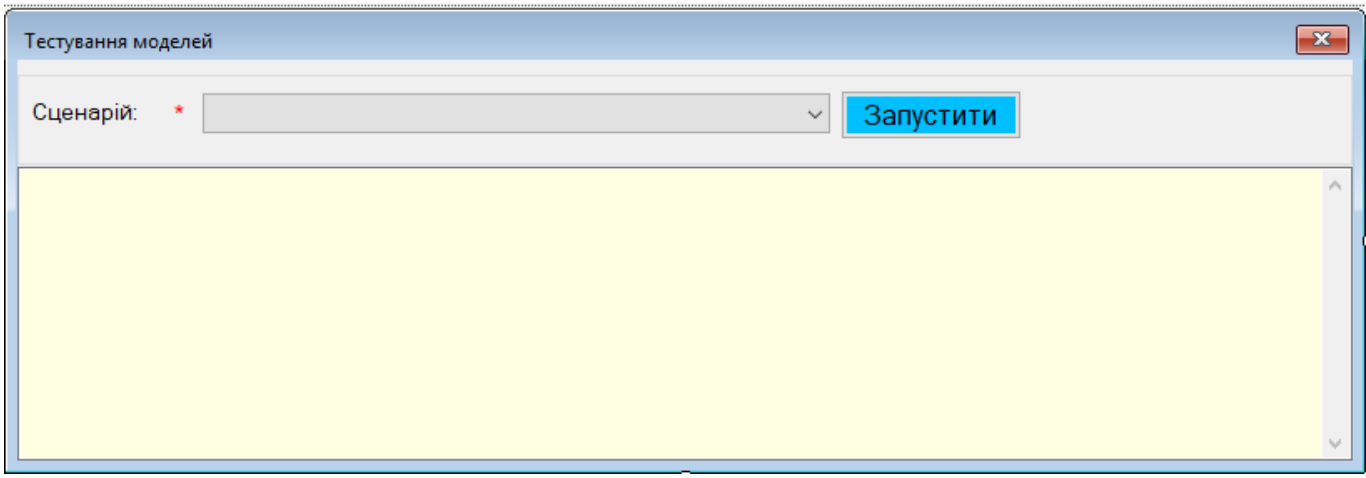

Рисунок 3.20 – Вигляд форми «Тестування моделей»

Для зміни моделі яку необхідно тестувати у випадаючому списку, реалізовано подію «ScenariosCBox_SelectedValueChanged» (рис. 3.21).

```
private void ScenariosCBox_SelectedValueChanged(object sender, EventArgs e) {
  if (_IsThemesLoad && _ScenariosList[0].Message != NamesMy.NoDataNames.NoDataInScenarios) {
    _SelectedNeural = _NeuralProvider.SelectedNeuralByScenariosId(
      Convert.ToInt32(ScenariosCBox.SelectedValue));
    LoadData(_SelectedNeural.NeuralFileModel);
  }
}
```

Рисунок 3.21 – Код події подію «ScenariosCBox_SelectedValueChanged»

Коли користувач вибирає сценарій, програма перевіряє, чи вже завантажені сценарії та чи існують вони взагалі. Якщо умови виконані, то

програма визначає відповідну нейронну модель, асоційовану з обраним сценарієм, та виконує завантаження даних цієї моделі. Це дозволяє користувачам працювати з конкретною моделлю, яка відповідає обраному сценарію, забезпечуючи більш цілеспрямований та ефективний аналіз.

Для реалізації процесу генерації даних тестування моделі або зупинки в залежності від її поточного стану розроблено подію "RunBtn_Click" (рис. 3.22). Спочатку виконується перевірка, чи правильно вибрано сценарій. Якщо сценарій вибрано коректно, то при натисканні на кнопку відбувається перевірка, чи активовано таймер. Якщо таймер активовано, він вимикається, і текст кнопки змінюється на "Генерувати". У протилежному випадку таймер активується, а текст кнопки змінюється на "Зупинити". Це дозволяє користувачам керувати процесом генерації даних або іншими діями, які виконуються через певні інтервали часу, використовуючи таймер.

```
private void RunBtn_Click(object sender, EventArgs e) {
  if (IsScenariosSelectedCorrect()) {
    if (timer1.Enabled) {
      timer1.Enabled = false;
      RunBtn.Text = "Генерувати";
    } else {
      timer1.Enabled = true;
      RunBtn.Text = "Зупинити";
    }
  }
}
```

Рисунок 3.22 – Код події подію «RunBtn_Click»

У даному підрозділі коротко описано основні моменти розробки форм для взаємодії користувача, а також важливого функціоналу системи.

3.5 Тестування та налагодження програми

3.5.1 Аналіз методів тестування та відлагодження

У рамках дослідження методів прогнозування появи помилок у програмному коді, особливу увагу заслуговують два методи тестування: тестування системи методом покриття операторів та тестування системи методом припущення про похибку.

Тестування системи методом покриття операторів зосереджується на аналізі того, які частини коду виконуються під час тестування. Цей метод вимірює відсоток операторів або рядків коду, які були виконані, щоб забезпечити, що тести охоплюють значну частину програмного забезпечення. Цей підхід ефективний для виявлення непокритих частин коду, які можуть містити невиявлені помилки, але він не гарантує виявлення всіх помилок, оскільки покриття коду не завжди відображає його логічну коректність.

З іншого боку, тестування системи методом припущення про похибку базується на ідеї, що помилки в програмному коді часто виникають через типові неправильні припущення або поширені помилки, які роблять розробники. У цьому методі розробники або тестувальники створюють тести, які спеціально спрямовані на виявлення таких типових помилок. Цей підхід дозволяє більш цілеспрямовано знаходити потенційно вразливі місця у коді, але вимагає глибшого розуміння загальних помилок у програмуванні та специфіки конкретного проекту.

Обидва методи відіграють важливу роль у процесі тестування, але їх ефективність залежить від контексту та специфіки програмного забезпечення. Тому оптимальний підхід часто включає комбінацію цих та інших методів тестування для всебічного аналізу та виявлення помилок у програмному коді.

3.5.2 Тестування системи методом покриття операторів

Тестування ПЗ виступає ключовим елементом у процесі розробки, надаючи гарантію, що код працює так, як очікується, і є безпечним для подальшої експлуатації. В контексті цього проекту, особлива увага приділялась юніт-тестуванню, яке вважається одним з найбільш витончених методів перевірки індивідуальних компонентів програми. Використовуючи бібліотеку MSTestv [41], були розроблені та виконані тести, які охоплюють критичні аспекти системи. Починаючи з тестування бізнес-логіки, були створені тестові випадки, що перевіряють точність алгоритмів обробки даних, включаючи

розрахунок оптимальних маршрутів, оцінку часу, необхідного для паркування, та вартість наданих послуг. Це забезпечує, що основні функції системи працюють без помилок і згідно з бізнес-вимогами.

Для забезпечення надійності збереження та обробки даних, було виконано тестування доступу до даних. Тут тести були зосереджені на перевірці взаємодії з базою даних, а саме правильності SQL-запитів, операций вставки нових записів, оновлення існуючих даних та видалення застарілих або невідповідних записів.

З огляду на важливість безпеки програмного забезпечення, було проведено тестування безпеки. Ці тести фокусувалися на методах шифрування даних, процесах автентифікації користувачів та механізмах авторизації доступу до різних частин системи. Таке тестування важливе для запобігання несанкціонованого доступу та забезпечення конфіденційності інформації.

На рис. 3.23 представлено результати проведення модульного тестування.

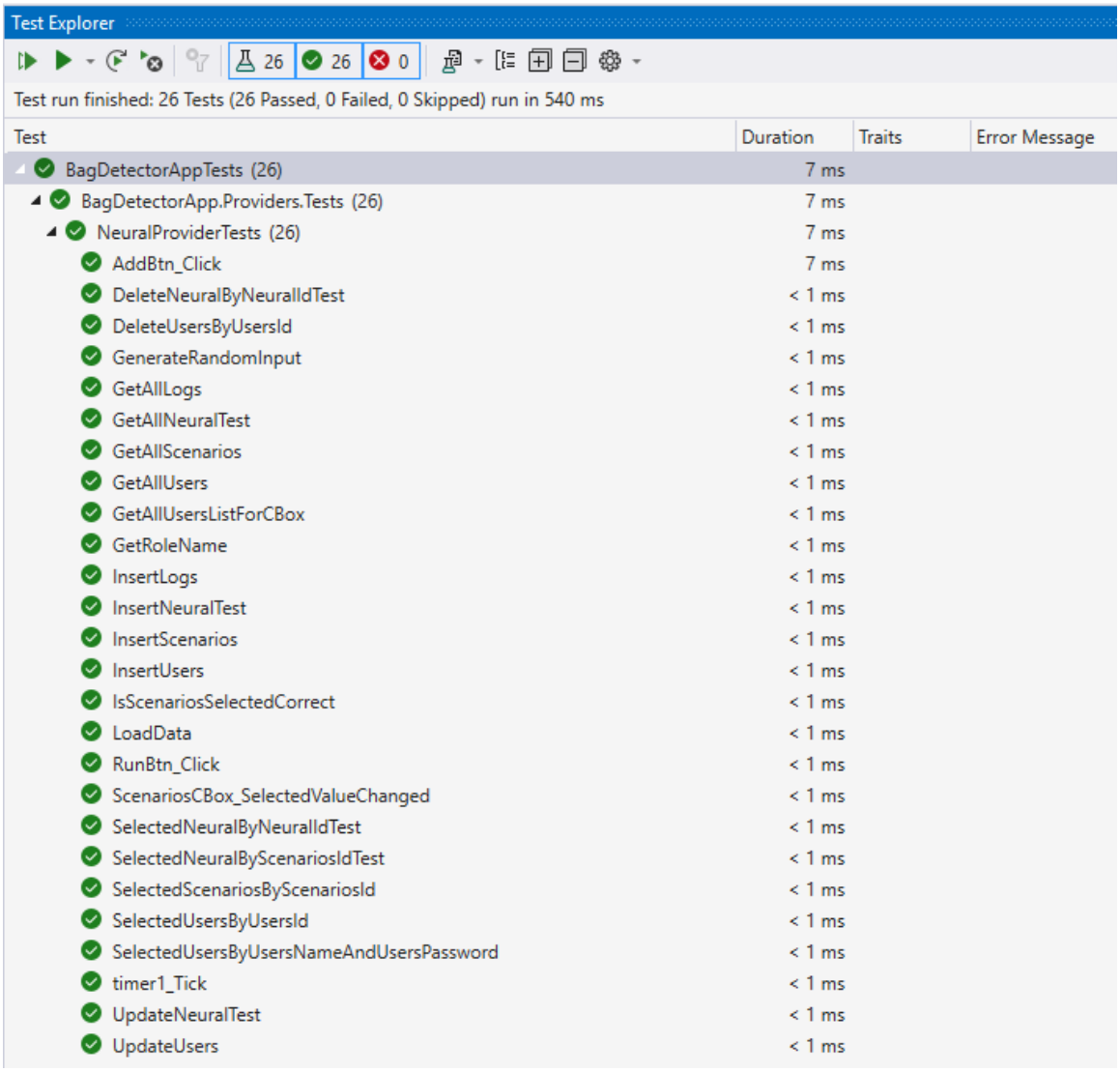

Рисунок 3.23 – Результати проведеного модульного тестування

Всі ці заходи забезпечили глибокий і всебічний підхід до тестування, результати якого були узагальнені та представлені у вигляді візуальних звітів. Ці звіти, які відображено на відповідному рисунку, ілюструють рівень покриття тестами критичних сегментів системи та ефективність виявлення потенційних вразливостей або помилок.

3.5.3 Тестування системи методом припущення про похибку

У рамках тестування методом припущення про похибку створено тестові сценарії, які імітують типові помилки, що допускаються в ході розробки

Сценарій №1 "Перевірка валідності даних".

Мета: Перевірити реакцію форми на введення неправильних або неповних даних під час навчання моделі.

Процедура: Спроба зберегти модель нейронної мережі без попереднього навчання або без вказівки необхідних параметрів, таких як ім'я моделі або вибраний сценарій.

Очікуваний Результат: Форма виводить повідомлення про помилку, вказуючи на неправильне введення даних, та блокує процес збереження до виправлення помилок.

Сценарій №2 "Перевірка реакції на видалення даних".

Мета: Визначити, чи коректно форма обробляє запит на видалення нейронної мережі зі списку.

Процедура: Вибір нейронної мережі зі списку та натискання кнопки "Видалити", після чого має з'явитися запит підтвердження дії.

Очікуваний Результат: Форма коректно видаляє обрану нейронну мережу після підтвердження та оновлює список без видаленого елемента.

Сценарій №3 "Валідація вибору сценарію"

Мета: Переконатися, що система коректно реагує на вибір сценарію, який не має відповідної нейронної мережі.

Процедура: Вибрати зі списку сценарій, для якого не було створено нейронну мережу, і спробувати запустити симуляцію.

Очікуваний Результат: Система повинна вивести повідомлення, що немає доступної нейронної мережі для обраного сценарію, і запобігти запуску симуляції.

Сценарій №4 "Проведення симуляції з випадковими даними"

Мета: Перевірити, наскільки точно система прогнозує результати на основі випадково згенерованих даних.

Процедура: Згенерувати випадковий набір вхідних даних за допомогою вбудованого генератора і запустити симуляцію для отримання прогнозу.

Очікуваний Результат: Система має коректно обробити згенеровані дані і надати прогноз наявності помилки, який можна візуально перевірити на панелі результатів.

Розроблені тестові сценарії дозволили ефективно перевірити відповідність роботи інтерфейсу користувача встановленим вимогам і виявити недоліки в логіці. Вони забезпечили важливе впевнення в тому, що симуляція сценаріїв відбувається згідно з очікуваннями і що система належно реагує на нестандартні ситуації, такі як відсутність даних для обраного сценарію. Крім того, завдяки тестуванню з випадковими даними, було можливо оцінити точність і надійність прогнозування нейронної мережі, а також забезпечити коректну обробку введення та відображення результатів.

Висновки до розділу 3

У рамках даного розділу була розроблена система для прогнозування появи помилок у програмному коді на основі трьохрівневої архітектури, що забезпечує чітке розділення функціональності, спрощення процесу розробки і масштабування. Визначені та реалізовані ключові компоненти системи включають рівень доступу до даних (DAL), який пропонує інтерфейси для взаємодії з базою даних, рівень бізнес-логіки, який обробляє вхідні дані та виконує алгоритми прогнозування, та рівень представлення, що забезпечує відображення інформації користувачам. Розроблені класи для кожного рівня, такі як LogsProvider для обробки логів, NeuralProvider для управління моделями машинного навчання, ScenariosProvider для управління сценаріями тестування і UsersProvider для управління даними користувачів, виконують специфічні функції, що покращують ефективність та надійність системи. Впровадження принципів об'єктно-орієнтованого проектування, таких як принцип єдиного обов’язку та принцип відкритості/закритості, дозволило створити гнучку, масштабовану та легку в підтримці систему, що лягла в основу ефективного прогнозування помилок в програмному коді.

Застосування методів модульного тестування дозволило забезпечити високу надійність і стабільність роботи системи. Ретельне тестування бізнес-логіки, доступу до даних та користувацького інтерфейсу з використанням автоматизованих сценаріїв перевірки забезпечило виявлення та виправлення помилок на ранніх стадіях розробки, що суттєво підвищило якість кінцевого продукту.

4 АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ПРОГНОЗУВАННЯ ПОМИЛОК У ПРОГРАМНОМУ КОДІ

4.1 Підготовка до експерименту

4.1.1 Опис використаного програмно-апаратного середовища

Основою для проведення експерименту є комплексне програмно-апаратне середовище, яке включає в себе наступні ключові компоненти:

– комп'ютерна платформа. Експерименти проводилися на персональному комп'ютері з процесором Intel Core i5, 16 ГБ оперативної пам'яті та твердотільним диском ємністю 512 ГБ. Таке обладнання забезпечує достатню обчислювальну потужність та швидкість обробки даних для ефективного проведення досліджень;

– операційна система. Для реалізації експерименту використовувалася операційна система Windows 10 Pro, що забезпечує стабільне та зручне середовище для розробки та тестування ПЗ;

– розробка та тестування. Використання інтегрованого середовища розробки (IDE) Microsoft Visual Studio 2022, яке є одним із провідних інструментів для розробки програм на мові C#. Visual Studio надає широкі можливості для написання коду, його відлагодження та тестування;

– машинне навчання та аналіз даних. Для обробки та аналізу даних використовувалася бібліотека Microsoft ML.NET, яка забезпечує інструментарій для реалізації машинного навчання у програмах на C#. Вона дозволяє створювати, тренувати, оцінювати та використовувати моделі машинного навчання;

– база даних. Використання системи управління базами даних Microsoft SQL Server забезпечує ефективне зберігання та обробку великих обсягів інформації, необхідної для аналізу та прогнозування помилок у програмному коді.

Це середовище було вибране за його надійність, широкі можливості для розробки, тестування та аналізу даних. Воно забезпечує необхідні умови для

ефективного проведення експериментів та отримання достовірних результатів дослідження.

4.1.2 Опис підходів для визначення точності та повноти методів прогнозування

Для забезпечення дослідження прогнозування появи помилок у програмному коді, особлива увага приділялася оцінці точності та повноти алгоритмів машинного навчання. Для цього використовувався метод логістичної регресії із стохастичним спуском (SdcaLogisticRegression), реалізований у середовищі ML.NET. Цей метод дозволяє аналізувати великі набори даних та ефективно виявляти потенційні помилки в коді.

Для визначення точності та повноти прогнозування було вибрано декілька ключових метрик, зокрема:

– точність (Accuracy) визначає відсоток правильно класифікованих випадків від загальної кількості випадків. Ця метрика є важливою для оцінки загальної ефективності моделі, але не завжди може відображати деталізовану картину, особливо у випадку нерівномірно розподілених класів;

– повнота (Recall) вимірює відсоток реальних випадків помилок, які були правильно ідентифіковані моделлю. Ця метрика критична для розуміння, наскільки ефективно модель виявляє існуючі помилки;

– F1-скор, який є гармонічним середнім між точністю та повнотою;

– площа під кривою оператора прийняття рішень (AUC-ROC), яка дозволяє оцінити, наскільки добре модель відрізняє два класи (помилка/не помилка) незалежно від порогу класифікації.

В експерименті буде використано набір даних, який містить характеристики програмного коду, такі як кількість методів у класі, зв'язність класів, глибина дерева наслідування тощо. Ці дані будуть використані для тренування моделі логістичної регресії.

Після тренування моделі буде проведена її оцінка на тестових даних, яка дозволить визначити точність та повноту прогнозування. Результати цих оцінок дозволять провести аналіз ефективності методу та визначення його придатності для виявлення помилок в програмному коді.

4.1.3 Вплив налаштувань середовища виконання на результати прогнозування

Було звернено особливу увагу на налаштування програмно-апаратного середовища, оскільки ці параметри мають прямий вплив на результати прогнозування. Конкретні налаштування та вибір параметрів, що були зроблені в ході підготовки до експерименту, включають:

– конфігурація обладнання. Для експериментів використовувався комп'ютер з процесором Intel Core i5, 16 ГБ оперативної пам'яті та SSD на 512 ГБ. Ця конфігурація забезпечила достатню обчислювальну потужність для обробки даних і тренування моделі;

– операційна система. Вибір операційної системи пав на Windows 10, оскільки вона забезпечує широку сумісність із різними інструментами та бібліотеками для машинного навчання;

– параметри моделі ML.NET. Використовуючи ML.NET, було налаштовано ключові параметри моделі логістичної регресії. Серед них швидкість навчання була встановлена на середньому рівні, щоб забезпечити баланс між швидкістю тренування та точністю моделі. Кількість ітерацій була визначена експериментально для досягнення оптимального результату;

– обробка та підготовка даних. Вхідні дані були ретельно підготовлені, включаючи нормалізацію числових змінних та обробку категорійних даних за допомогою One-Hot Encoding. Це забезпечило точність введення даних в модель;

– тестування та валідація моделі. Використання крос-валідації та різних наборів даних для тестування допомогло уникнути перенавчання моделі та перевірити її здатність до узагальнення на нових даних.

Проведені налаштування створили міцну основу для проведення надійних та об'єктивних експериментів, спрямованих на оцінку точності та повноти методів прогнозування помилок у програмному коді.

4.1.4 Вплив зовнішніх факторів на результати прогнозування

У процесі підготовки до експерименту була приділена значна увага мінімізації впливу зовнішніх факторів на результати. Для цього були вжиті наступні конкретні заходи, зокрема:

– стабілізація обчислювального середовища. Забезпечено, що весь експеримент проводився на одному й тому ж обладнанні зі сталою конфігурацією, що виключало варіативність у обчислювальних ресурсах. Також були встановлені оновлення для всього програмного забезпечення, щоб уникнути нестабільності через випадкові збої або помилки;

– контроль версій бібліотек і залежностей. Всі використані бібліотеки та фреймворки, включаючи ML.NET, були фіксовані до конкретних версій. Це дозволило уникнути несподіваних змін у поведінці системи через оновлення або зміни в залежностях;

– моніторинг зовнішніх умов. Проведено моніторинг температури та вологості в приміщенні, де знаходилася обчислювальна техніка, щоб виключити вплив зовнішніх умов на фізичне обладнання;

– стандартизація вхідних даних. Виконано ретельну підготовку та нормалізацію вхідних даних, що забезпечило їх однорідність та знизило вплив випадкових відхилень у даних;

– використання репрезентативного набору даних. Для тренування моделі було вибрано репрезентативний набір даних, що відображає реальні умови використання програмного коду. Це допомогло забезпечити, що модель буде ефективною у різних сценаріях використання;

– проведення попередніх тестів. Перед основним експериментом були проведені попередні тестування системи, щоб виявити та усунути можливі проблеми, які могли вплинути на результати.

Ці заходи були спрямовані на те, щоб максимально зменшити вплив зовнішніх чинників та забезпечити достовірність та повторюваність результатів експерименту з прогнозування помилок у програмному коді.

4.2 Проведення експерименту

У рамках проведення експерименту з прогнозування появи помилок в програмному коді, основною умовою було використання якісних та релевантних даних. Для цього було обрано спеціалізований набір даних з відкритого ресурсу [42], який містить широкий спектр метрик, пов'язаних з програмним кодом, включаючи кількість методів у класі, глибину дерева наслідування, кількість дочірніх класів, зв'язність класів тощо. Цей набір даних надав можливість розглядати різноманітні аспекти програмного коду та його потенційні помилки.

Перед проведенням експерименту була проведена детальна підготовка даних. Спочатку виконано аналіз на наявність аномалій та видалено неактуальні чи пошкоджені записи. Потім була виконана нормалізація даних, щоб усі вимірювання мали однаковий масштаб та не впливали на результати аналізу непропорційно. Також була забезпечена консистентність форматів даних, що має важливе значення для точності обробки даних алгоритмами машинного навчання.

Для експерименту використовувався метод логістичної регресії зі стохастичним спуском у середовищі ML.NET. Було виконано попереднє налаштування параметрів алгоритму, щоб забезпечити його ефективне навчання на підготовлених даних. Це включало вибір швидкості навчання, кількості ітерацій і регуляризацію для оптимального навчання моделі без перенавчання.

У подальшому, було проведено процес тренування моделі, під час якого модель навчалася розпізнавати потенційні помилки в програмному коді. Навчання моделі базувалося на використанні репрезентативних прикладів з набору даних, що дозволило виявляти широкий спектр потенційних помилок.

На рис. 4.1 представлено фрагмент інтерфейсу з підготовленими даними для проведення навчання моделі, демонструючи візуалізацію вхідних даних та їх підготовку до введення в модель. Це дало можливість візуально оцінити якість та структуру вхідних даних перед їх використанням у процесі машинного навчання.

. Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

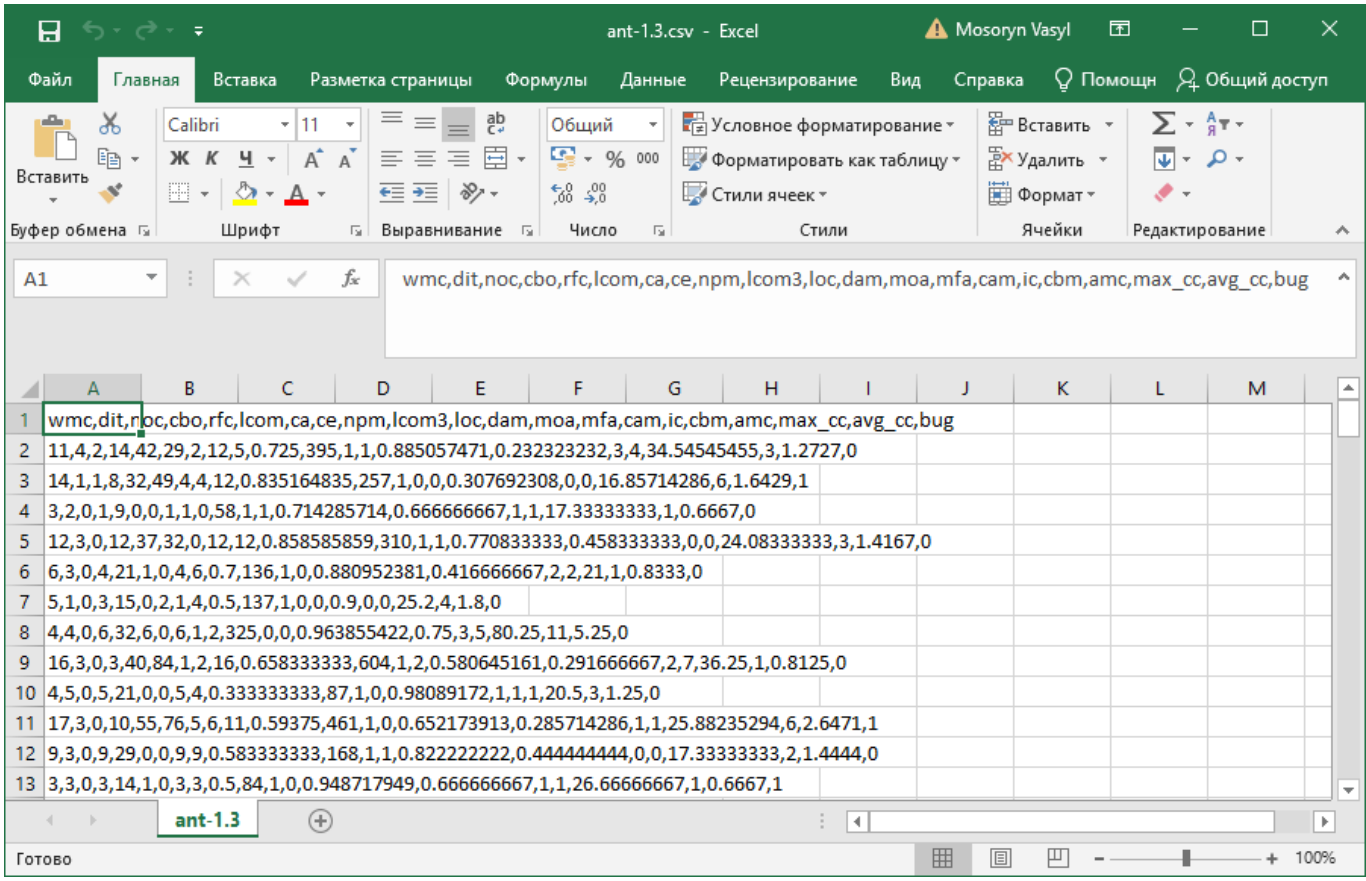

Рисунок 4.1 – Фрагмент підготовлених даних навчання моделі

Для тренування моделі у застосунку було створено сценарій навчання із назвою «Сценарій №1» (рис. 4.2). . Всі малюнти повинні бути на відстані не менш 1 см від лівого краю

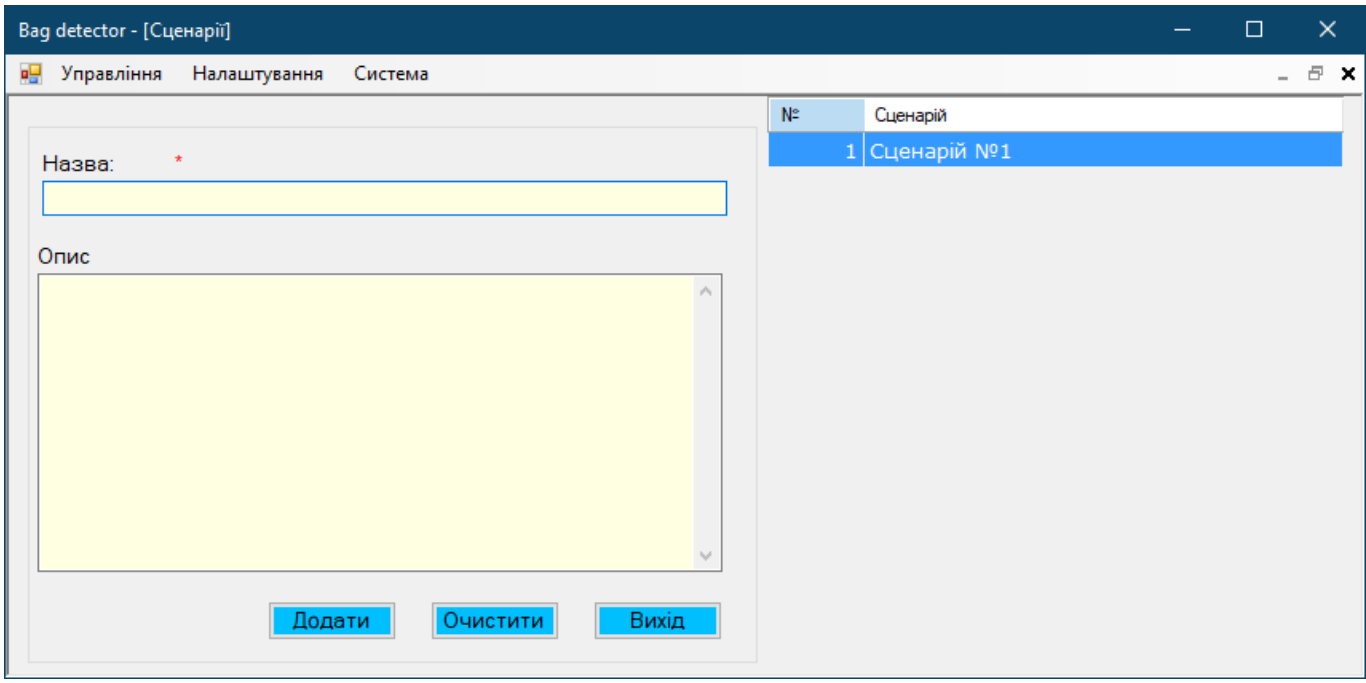

Рисунок 4.2– Створення сценарію навчання

Для тренування нової моделі створеного сценарію згідно підготовленого датасету, було здійснено перехід по меню програми «Налаштування» –> «Навчання моделі» та обрано створений сценарій «Сценарій №1». Після обрання підготовленого датасету у діалоговому вікні навчання моделі почалось автоматично. Результат навчання представлено на рис. 4.3.

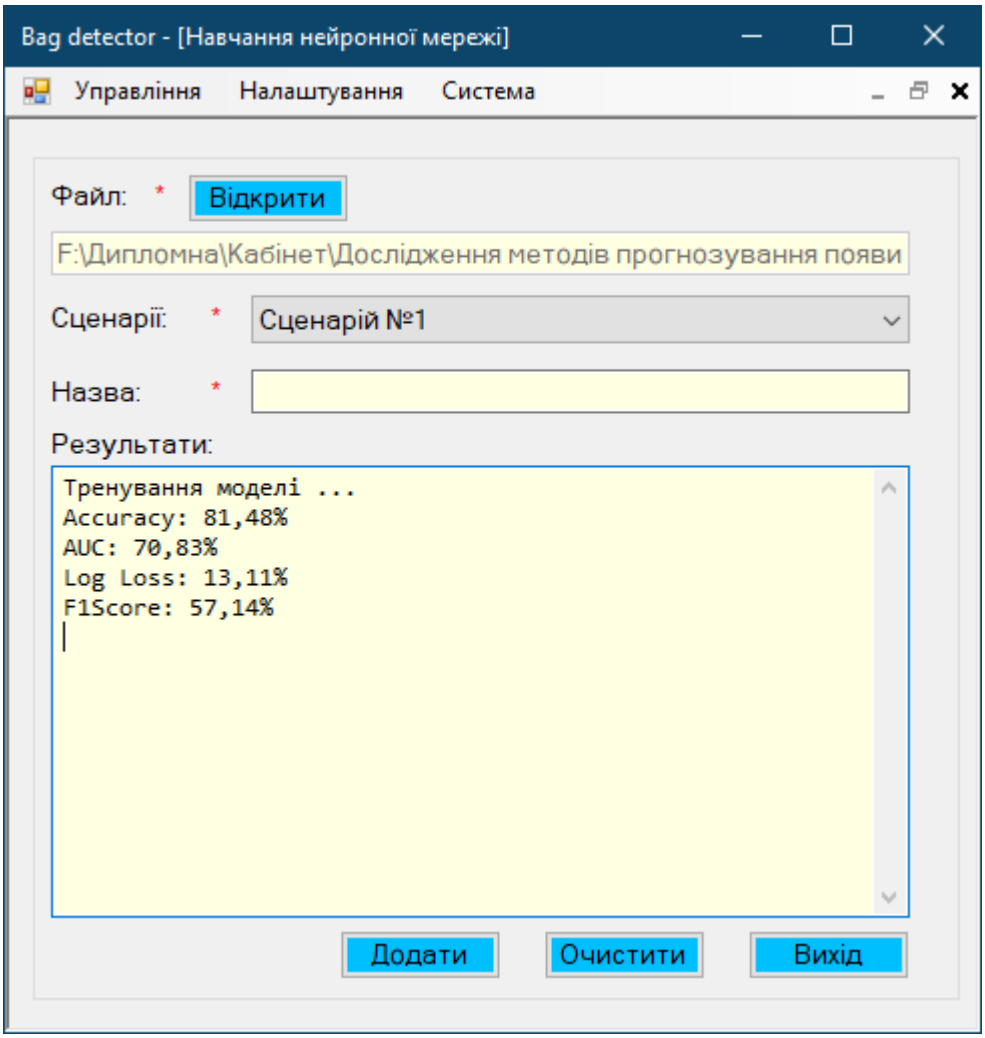

Рисунок 4.3 – Результат навчання НМ

Процес навчання моделі супроводжується виведенням повідомлення «Тренування моделі ...» на початку тренування та виведенням значень метрик навчання після закінчення процесу:

– Accuracy - 81,48%. Ця метрика вказує на загальну частку правильно класифікованих випадків (як помилок, так і не помилок) від усіх випадків у тестовому наборі даних. Висока точність у 81,48% свідчить про те, що модель здатна ефективно розпізнавати наявність помилок у більшості випадків. Однак, важливо пам'ятати, що сама по собі висока точність не завжди є ознакою ефективності моделі, особливо у випадку дисбалансу класів;

– AUC - 70,83%. AUC оцінює, наскільки добре модель здатна розрізняти між класами. Значення в 70,83% вказує на те, що модель має певну здатність розрізняти помилки від не помилок, але існує потенціал для покращення, оскільки ідеальне значення AUC становить 100%;

– Log Loss - 13,11%: Ця метрика використовується для оцінки якості прогнозів, які подаються у вигляді ймовірностей. Значення в 13,11% вказує на досить гарну якість прогнозів, адже чим менше Log Loss, тим краще. Це свідчить про те, що модель ефективно прогнозує ймовірності класів;

– F1Score - 57,14%: F1-скор є важливою метрикою у випадках, коли потрібен баланс між точністю (Precision) та повнотою (Recall). Значення F1-скору в 57,14% є помірним, що може вказувати на деякі проблеми з точністю або повнотою моделі. Оптимально, щоб ця метрика була близькою до 100%, тому є простір для покращення..

У цілому, метрики показують, що модель має добру загальну точність і непогане значення F1Score, але існують області, де можливе покращення, особливо у плані збільшення AUC та зниження Log Loss.

Після цього навчену модель було збережено у системі, для проведення подальшого експериментального дослідження (рис. 4.4).

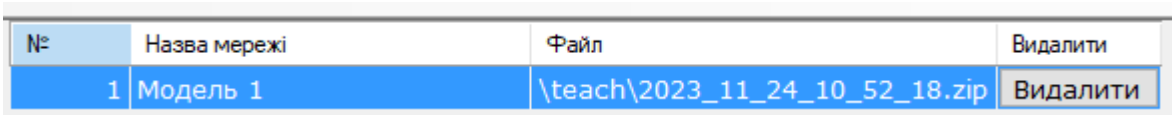

Рисунок 4.4 –Збереження моделі

Після цього проведено експеримент з використанням навченої моделі. Для цього, в меню програми обрано опцію "Тестування моделі", що відкриває форму, де можна виконати моделювання потенційних помилок у програмному коді. Користувач мав можливість вибрати будь-який сценарій із випадаючого списку.

Процес моделювання та прогнозування помилок проходив у кілька етапів:

– генерація даних. Використовуючи спеціально розроблений клас (GenerateRandomInput), програма генерувала різноманітні варіанти програмного коду, імітуючи різні аспекти та характеристики коду. Це дозволяло моделі аналізувати широкий спектр потенційних помилок;

– прогнозування за допомогою моделі. Отримані дані передавалися моделі, яка аналізувала код та визначала ймовірність наявності помилки;

– аналіз результатів прогнозування. Після обробки даних моделлю програма аналізувала отримані результати, оцінюючи кожен потенційний випадок на предмет наявності помилки;

– візуалізація та звітування. Результати моделювання відображалися в інтерфейсі програми. Кожен запис містив інформацію про симульований код, його характеристики та результати прогнозування.

Мета цього експерименту полягала не тільки в тому, щоб продемонструвати здатність моделі виявляти помилки, але й у тестуванні її ефективності та надійності у різних сценаріях.

Для виявлення точності навченої моделі було згенеровано декілька потенційних випадків за допомогою генератора сценаріїв та детально їх проаналізовано. На рис. 4.5 зображено перший згенерований випадок.

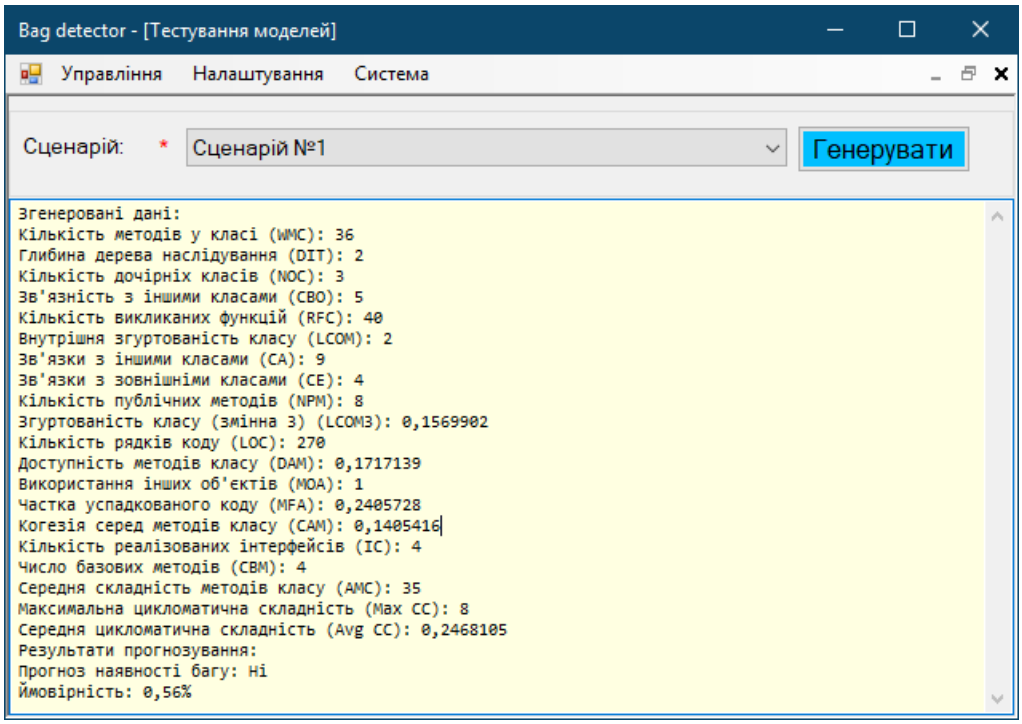

Рисунок 4.5 –Результат прогнозування першого випадку

Аналізуючи наведений сценарій прогнозування, можна зробити наступні висновки:

1. Загальна оцінка коду:

– кількість методів у класі (36) та кількість викликаних функцій (40) є значними, що може свідчити про високу складність класу;

– глибина дерева наслідування (2) та кількість дочірніх класів (3) вказують на помірну ієрархічну складність;

– зв'язність з іншими класами (CBO – 5) і зв'язки з зовнішніми класами (CE – 4) на рівні, що свідчить про достатню інтеграцію класу з іншими частинами системи;

– наявність внутрішньої згуртованості класу (LCOM – 2) та згуртованості класу (LCOM3 – 0,1569902) на низькому рівні, що може вказувати на добру внутрішню структуру класу.

2. Метрики, які вказують на потенційні проблеми:

кількість рядків коду (270) є значною, що може ускладнювати розуміння та підтримку коду;

максимальна цикломатична складність (8) та середня цикломатична складність (0,2468105) в межах прийнятних значень, але все ж вимагають уваги, особливо у випадках розширення функціоналу.

3. Прогноз наявності багу та ймовірність:

– модель прогнозує відсутність помилки (багу) з ймовірністю 0,56%. Це дуже низька ймовірність, що вказує на високу впевненість моделі в коректності даного класу коду.

На підставі цього аналізу можна зробити висновок, що, хоча клас має деякі ознаки високої складності, в цілому він не містить помилок. Однак, через велику кількість рядків коду та інші фактори, що впливають на складність, може виникнути потреба в подальшій оптимізації та рефакторингу для поліпшення читабельності та підтримки коду.

На рис 4.6 зображено скріншот результат виконання 2-го випадку.

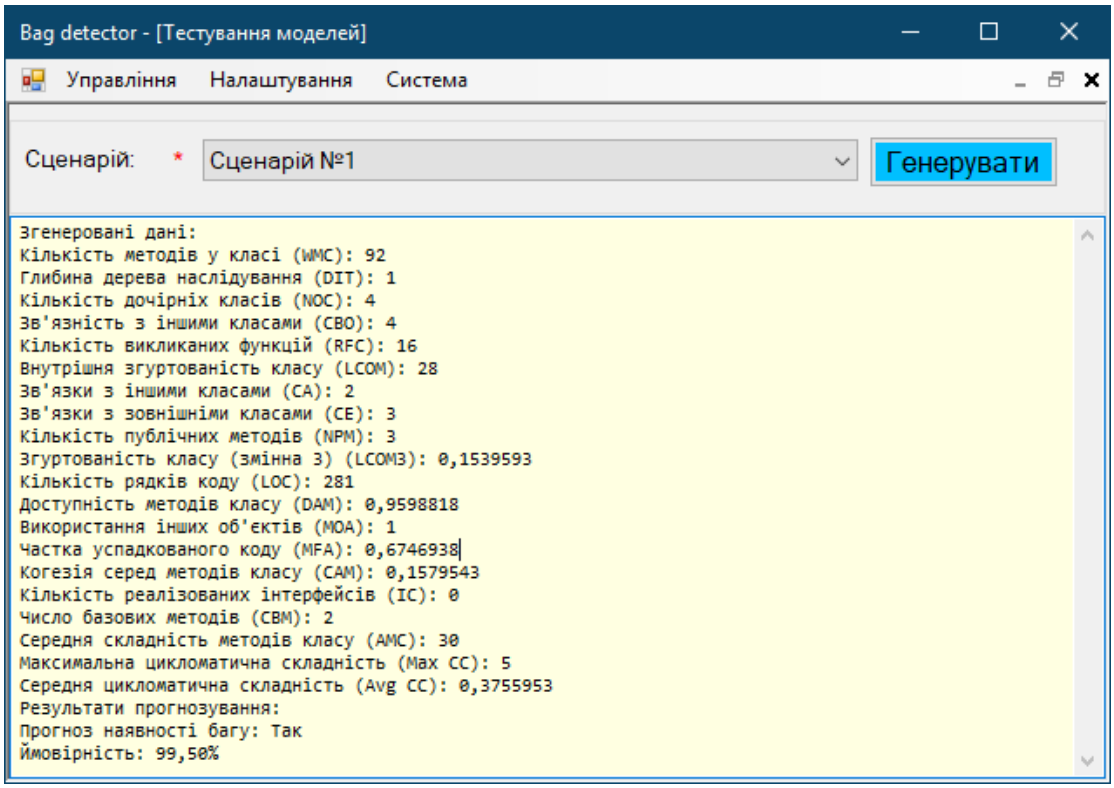

Рисунок 4.5 –Результат прогнозування другого випадку

Аналіз другого випадку прогнозування дозволяє виділити декілька ключових аспектів:

1. Комплексна оцінка класу:

– висока кількість методів у класі (WMC: 92) та велика кількість рядків коду (LOC: 281) свідчить про значну складність класу. Така складність може спричиняти труднощі у підтримці та розширенні класу;

– глибина дерева наслідування (DIT: 1) та кількість дочірніх класів (NOC: 4) вказують на відносно просту ієрархічну структуру, але із деякою тенденцією до розширення;

– значення зв'язності з іншими класами (CBO: 4) та зв'язків з зовнішніми класами (CE: 3) є не високими, що може вказувати на адекватний рівень інтеграції класу.

2. Показники, які можуть бути ознаками проблем:

– внутрішня згуртованість класу (LCOM: 28) та його згуртованість (LCOM3: 0,1539593) на рівнях, що можуть вказувати на недостатню когезію серед методів класу;

– частка успадкованого коду (MFA: 0,6746938) є високою, що може вказувати на значну залежність від базових класів;

– максимальна та середня цикломатична складність (Max CC: 5, Avg CC: 0,3755953) в межах прийнятних значень, але можуть вимагати уваги при змінах коду.

3. Прогноз наявності багу та ймовірність:

– модель вказує на високу ймовірність наявності помилки (99,50%). Це свідчить про високий ризик багу в даному класі, особливо з огляду на його велику складність та інші зазначені вище характеристики.

Цей згенерований випадок свідчить про високу ймовірність наявності помилки в класі з високою складністю та низкою інших факторів, які можуть сприяти появі помилок. Особлива увага має бути приділена оптимізації та рефакторингу такого класу для зниження ризику багів.

4.3 Аналіз результатів експерименту

У рамках дослідження було проведено серію експериментів, спрямованих на аналіз впливу різних характеристик програмного коду на ймовірність появи помилок. Ці експерименти були ретельно сплановані та виконані з метою збору вичерпних даних, які б допомогли зрозуміти ключові аспекти, що впливають на якість коду. Результати цих експериментів були детально зафіксовані та представлені у таблиці. Ця таблиця містить важливу інформацію, яка дозволяє оцінити взаємозв'язок між різними метриками та потенційними ризиками для програмного забезпечення.

Аналіз цих даних може допомогти у виявленні тенденцій та закономірностей, які можуть бути використані для покращення процесів розробки та тестування програмного забезпечення. За допомогою цього аналізу можна виявити ключові фактори, які впливають на надійність і стабільність коду, та розробити стратегії для їх оптимізації. Таким чином, результати, представлені у табл. 4.1, є не тільки відображенням проведеної роботи, але й цінним джерелом інформації для подальших досліджень та розвитку.

Таблиця 4.1 – Результати проведення експериментів

Характеристика	№ експериментальної перевірки моделі									
	1	2	3	4	5	6	7	8	9	10
WMC	11	3	82	96	30	6	40	6	99	92
DIT	4	1	3	4	3	2	1	1	2	2
NOC)	2	2	1	1	2	2	4	1	4	3
CBO)	2	5	5	3	9	4	1	4	6	4
RFC	14	42	27	39	30	1	18	7	20	36
LCOM	98	76	0	81	30	55	28	20	61	81
CA	3	1	8	3	3	2	5	0	2	2
CE	2	0	3	5	6	7	1	9	4	1
NPM	9	0	6	4	9	2	4	4	0	9
LCOM3	0,96	0,85	0,66	0,52	0,35	0,85	0,81	0,3	0,65	0,06
LOC	186	24	17	154	58	88	215	252	123	43
DAM	0,75	0,79	0,27	0,74	0,58	0,92	0,08	0,67	0,22	0,57
MOA	2	2	2	1	4	2	4	4	2	2
MFA	0,35	0,99	0,89	0,61	0,08	0,81	0,46	0,79	0,86	0,93
CAM	0,26	0,55	0,52	0,05	0,01	0,71	0,38	0,82	0,9	0,25
IC	1	2	0	3	0	1	2	1	4	3
CBM	0	2	0	0	4	4	2	2	4	3
AMC	28	25	40	15	40	9	12	8	1	24
Max CC	8	5	2	2	7	6	3	7	9	8
Avg CC	0,5	0,077	0,91	0,89	0,43	0,11	0,83	0,27	0,6	0,6
Прогноз наявності багу	Ні	Ні	Ні	Так	Ні	Ні	Ні	Ні	Ні	Так
Ймовірність (%)	0,11	47,84	14,61	75,92	6,36	11,62	2,88	10,34	14,44	55,51

Аналізуючи результати проведених експериментів, можна зробити декілька важливих спостережень стосовно прогнозування появи помилок у програмному коді. Зокрема, у двох експериментах, номер чотири та десять, прогнозується наявність багу. Цікаво, що експеримент номер чотири відрізняється високою ймовірністю помилки, становлячи 75,92%, тоді як у експерименті номер десять ця ймовірність є помірно високою, а саме 55,51%.

Далі, розглядаючи характеристики цих експериментів, можна виявити, що в експерименті номер чотири спостерігаються високі значення WMC і LOC, що може вказувати на велику складність і обсяг коду. У той же час, експеримент номер десять хоч і має високий LOC, але має нижчі показники WMC порівняно з четвертим експериментом.

Це контрастує з експериментами, де ймовірність помилок була низькою, наприклад, у номерах один та шість. У цих випадках можна бачити нижчі значення WMC і LOC, що може свідчити про меншу складність коду.

Виявлено, що певні метрики, такі як WMC і LOC, мають зв'язок з високою ймовірністю виникнення помилок, що вказує на можливу складність та обсяг коду. Це відкриває двері для подальших досліджень щодо впливу цих факторів на якість програмного продукту. Додатково, спостерігається, що характеристики з нижчими значеннями, як правило, корелюють з меншою ймовірністю помилок, підкреслюючи важливість збалансованості та чистоти коду. Ці результати демонструють значення аналізу метрик коду та їх використання як індикаторів потенційних ризиків у розробці програмного забезпечення.

Отже, аналіз проведених експериментів з прогнозування появи помилок у програмному коді дозволяє зробити наступні висновки:

– ефективність моделі логістичної регресії. Використання моделі логістичної регресії із стохастичним спуском в середовищі ML.NET продемонструвало здатність до адекватного прогнозування наявності помилок у програмному коді. Модель змогла ефективно аналізувати вхідні метрики і видавати відповідні прогнози;

– оцінка точності та повноти прогнозування. За результатами експериментів модель показала різні рівні точності та повноти залежно від характеристик вхідних даних. Випадки з високою складністю коду, великою кількістю методів і рядків коду були схильні до вищої ймовірності помилок. Натомість, простіші випадки з меншою кількістю методів і нижчою складністю частіше класифікувалися як безпомилкові;

– аналіз конкретних сценаріїв. Розгляд окремих сценаріїв прогнозування виявив, що модель ефективно виявляє потенційні помилки у складних випадках, надаючи високу ймовірність наявності багу. Такий підхід дозволяє зосередити увагу розробників на потенційно проблемних ділянках коду;

– важливість налаштувань середовища. Експерименти підкреслили значення правильної конфігурації середовища ML.NET, включаючи вибір алгоритмів, параметрів обробки даних та налаштувань тренування. Правильно налаштоване середовище забезпечує більш точні та надійні результати.

Отже, проведені експерименти продемонстрували потенціал методів машинного навчання у виявленні помилок у програмному коді. Однак, для підвищення точності та надійності прогнозування можна розглянути можливості подальшого вдосконалення моделі, такі як використання додаткових даних для тренування, зміна алгоритмів машинного навчання та подальша оптимізація параметрів моделі.

Висновки до розділу 4

У рамках даного розділу було проведено експерименти з прогнозування появи помилок у програмному коді, використовуючи модель логістичної регресії зі стохастичним спуском у середовищі ML.NET. Основною метою експерименту було визначення ефективності цієї моделі в розпізнаванні потенційних помилок на основі різних метрик програмного коду.

Аналіз результатів експериментів показав, що модель має здатність ефективно аналізувати вхідні метрики і видаляти відповідні прогнози. Модель продемонструвала різні рівні точності та повноти залежно від характеристик вхідних даних. Виявлено, що випадки з високою складністю коду, великою кількістю методів і рядків коду були схильні до вищої ймовірності помилок. Навпаки, простіші випадки з меншою кількістю методів і нижчою складністю частіше класифікувалися як безпомилкові.

Детальний аналіз окремих сценаріїв прогнозування виявив, що модель ефективно виявляє потенційні помилки у складних випадках, надаючи високу ймовірність наявності багу. Це дозволяє зосередити увагу розробників на потенційно проблемних ділянках коду.

Експерименти також підкреслили важливість правильної конфігурації середовища ML.NET, включаючи вибір алгоритмів, параметрів обробки даних та

налаштувань тренування. Правильно налаштоване середовище забезпечує більш точні та надійні результати.

Таким чином, проведені експерименти підтвердили потенціал методів машинного навчання у виявленні помилок у програмному коді. Однак для підвищення точності та надійності прогнозування необхідно розглянути можливості подальшого вдосконалення моделі, такі як використання додаткових даних для тренування, зміна алгоритмів машинного навчання та оптимізація параметрів моделі.

ВИСНОВКИ

Проведення критичного аналізу сучасного стану досліджень у сфері помилок у програмному коді було ключовим етапом у цій роботі. Цей аналіз зосередився на всебічному вивченні наукових та технічних публікацій, що охоплюють історію виникнення помилок у програмуванні, їхню еволюцію та сучасні дослідження. Вивчення історичних аспектів дало можливість оцінити, як зміни в технологіях та методологіях програмування впливали на характер та складність помилок. Також, це дозволило оцінити, як методи виявлення та виправлення помилок еволюціонували протягом часу, відповідаючи на виклики, які виникали з розвитком програмних мов та підходів до розробки.

Детальний огляд сучасних досліджень виявив зростаючу потребу в розробці більш ефективних методів для раннього виявлення помилок. Це виявилося важливим у контексті забезпечення надійності та безпеки програмних продуктів, особливо у світлі зростаючої складності сучасного програмного забезпечення та важливості його ролі в повсякденному житті. Вивчення різноманітних випадків, де помилки в програмному коді призвели до серйозних наслідків, як фінансових, так і в плані безпеки, підкреслило значення розробки надійних інструментів для раннього виявлення та виправлення помилок.

Аналіз програмно-апаратного забезпечення, включаючи застосування програмно-апаратних рішень для виявлення помилок, виявив значні переваги цих рішень у підвищенні продуктивності та ефективності. Цей аналіз охопив широкий спектр інструментів, включаючи інструменти статичного аналізу (наприклад, SonarQube, ESLint, PMD) та інструменти динамічного аналізу (такі як JUnit, Selenium, LoadRunner), а також оцінку можливостей сучасних інтегрованих середовищ розробки (IDE), як-то Visual Studio, IntelliJ IDEA, та Eclipse.

Оцінка виявила, що інтеграція програмно-апаратних засобів значно підвищує швидкість та точність у виявленні помилок, знижуючи час та ресурси, необхідні для тестування та відлагодження коду. Використання цих інструментів

також дозволяє автоматизувати багато процесів, що покращує загальну ефективність розробки.

Проте, аналіз також виявив певні прогалини в існуючих системах, особливо у контексті складної обробки даних та адаптації до нових вимог. Це включає обмеження щодо гнучкості налаштувань, необхідних для вирішення специфічних задач, і відсутність інтеграції між різними інструментами та платформами. Також було виявлено, що деякі існуючі інструменти не повністю відповідають сучасним вимогам до швидкості обробки та адаптивності, особливо у випадках, коли виникає потреба швидко реагувати на зміни у коді або коли працюється з великими обсягами даних.

Розробка системи для прогнозування помилок у програмному коді на основі трьохрівневої архітектури забезпечила чітке розділення функціональності, спрощення процесу розробки та масштабування. Визначені та реалізовані ключові компоненти системи включали рівень доступу до даних, рівень бізнес-логіки, та рівень представлення.

В розробленій системі ключовим елементом бізнес-логіки став алгоритм логістичної регресії із стохастичним спуском. Цей алгоритм був обраний за його здатність ефективно обробляти великі набори даних та виявляти залежності у складних наборах ознак, які характерні для програмного коду. Логістична регресія використовувалася для аналізу ймовірності появи помилок на основі різних ознак у коді, таких як частота використання певних конструкцій, залежності між модулями коду та інші.

Стратегія стохастичного спуску, в свою чергу, забезпечила оптимізацію процесу навчання моделі, дозволяючи швидко та ефективно адаптуватися до нових даних та змін у коді. Це забезпечило більшу точність прогнозування та зниження впливу шуму в даних, що є критично важливим для точного виявлення потенційних помилок.

Реалізація цього алгоритму в трьохрівневій архітектурі системи забезпечила, що кожен рівень виконував свої ключові функції, оптимізуючи процес виявлення та прогнозування помилок. Це, у свою чергу, сприяло

створенню гнучкої, масштабованої та легкої у підтримці системи, яка ефективно використовує переваги машинного навчання для підвищення якості програмного забезпечення.

Експерименти з прогнозування помилок за допомогою моделі логістичної регресії в середовищі ML.NET були спрямовані на визначення ефективності цієї моделі у розпізнаванні потенційних помилок на основі різних метрик програмного коду. Аналіз результатів показав, що модель має здатність ефективно аналізувати вхідні метрики і видаляти відповідні прогнози.

Модель продемонструвала різні рівні точності та повноти, залежно від характеристик вхідних даних. Було виявлено, що випадки з високою складністю коду, великою кількістю методів і рядків коду були схильні до вищої ймовірності помилок. Навпаки, простіші випадки з меншою кількістю методів і нижчою складністю частіше класифікувалися як безпомилкові.

Детальний аналіз окремих сценаріїв прогнозування виявив, що модель ефективно виявляє потенційні помилки у складних випадках, надаючи високу ймовірність наявності багу. Це дозволяє зосередити увагу розробників на потенційно проблемних ділянках коду.

Експерименти також підкреслили важливість правильної конфігурації середовища ML.NET, включаючи вибір алгоритмів, параметрів обробки даних та налаштувань тренування. Правильно налаштоване середовище забезпечує більш точні та надійні результати.

Таким чином, проведені експерименти підтвердили потенціал методів машинного навчання у виявленні помилок у програмному коді. Однак, для підвищення точності та надійності прогнозування необхідно розглянути можливості подальшого вдосконалення моделі, такі як використання додаткових даних для тренування, зміна алгоритмів машинного навчання та оптимізація параметрів моделі.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Ситник Л.Ю. Система оцінки якості програмного коду веб-проектів// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування”. – К.: НАУ, 2020. – 415 с.

2. ML.NET : веб-сайт. URL: https://dou.ua/forums/topic/34961/ (дата звернення: 27.11.2023).

3. What is Debugging? : веб-сайт. URL: https://aws.amazon.com/what-is/debugging/?nc1=h_ls (дата звернення: 27.11.2023).

4. Difference Between COBOL and FORTRAN : веб-сайт. URL: https://www.geeksforgeeks.org/difference-between-cobol-and-fortran/ (дата звернення: 27.11.2023).

5. C++ : веб-сайт. URL:http://www.znannya.org/?view=Cpp_basics (дата звернення: 27.11.2023).

6. Розробка на Java. URL: https://dou.ua/lenta/articles/how-to-learn-java/ (дата звернення: 27.11.2023).

7. On the use of deep learning in software defect prediction: https://www.sciencedirect.com/science/article/pii/S0164121222002138#:~:text=Predicting%20defect,reported%20in%20the%20literature (дата звернення: 05.12.2023).

8. Improved software fault prediction using new code metrics and machine learning algorithms. URL: https://www.sciencedirect.com/science/article/abs/pii/S2590118423000631(дата звернення: 05.12.2023).

9. An Empirical Study on Software Defect Prediction Using CodeBERT Model. URL: https://www.mdpi.com/2076-3417/11/11/4793#:~:text=We%20perform%20empirical%20studies%20using,defect%20prediction%20using%20CodeBERT%20models (дата звернення: 05.12.2023).

10. Introduction of Deadlock in Operating System. URL: https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/ (дата звернення: 27.11.2023).

11. Static code analysis tools: a systematic literature review, Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, Sonja Ristić January 2020, 11 с.

12. Dynamic Analysis vs. Static Analysis. URL: https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/inspector_xe/documentation/en/help/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm (дата звернення: 27.11.2023).

13. Соменко, Д. В.; Трифонова, О. М.; Садовий, М. І. Штучний інтелект та нейромережі в освітньому процесі: переваги та недоліки. Актуальні проблеми та перспективи технологічної і професійної освіти, 2023. – 178 с.

14. IDE. URL: https://learn.microsoft.com/uk-ua/visualstudio/extensibility/internals/ide-defined-commands-for-extending-project-systems?view=vs-2019 (дата звернення: 27.11.2023).

15. Встановлення IDE (Інтегрованого Середовища Розробки). URL: https://acode.com.ua/urok-4-vstanovlennya-ide-integrovanogo-seredovyshha-rozrobky/ (дата звернення: 27.11.2023).

16. Mondal D. Hemmati H. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. : веб-сайт. URL: http://www.cs.umanitoba.ca/~durocher/research/pubs/durocherICST2015.pdf

17. Bath G. Veenendaal E. Improving the Test Proces. URL: https://www.pdfdrive.com/download.pdf?id=158100139&h=dd82727a8f9065ea3ee68a94450f3f51&u=cache&ext=pdf (дата звернення: 27.11.2023).

18. Advantages and disadvantages of existing hardware and software complexes. URL: https://www.researchgate.net/figure/Advantages-and-disadvantages-of-using-the-various-hardware-platforms-for-training-and_tbl1_333712393 (дата звернення: 27.11.2023).

19. Perry W. E. Effective Methods Of Software Testing. URL: https://www.softwaretestinggenius.com/download/EMFST.pdf (дата звернення: 27.11.2023).

20. Srivastava P. Test Case // Journal of Theoretical and Applied Information Technology IEEE. URL: https://www.researchgate.net/publication/235799411 (дата звернення: 27.11.2023).

21. Бахрушин В.Є. Методи аналізу даних : навчальний посібник для студентів / В.Є. Бахрушин. - Запоріжжя: КПУ, 2011. – 268 с.

22. Dynamic program analysis. URL : https://training.qatestlab.com/blog/technical-articles/static-and-dynamic-testing-methods/ (дата звернення: 27.11.2023).

23. Abdullah, A., & Khan, M. Z. (2019). Software defect prediction using supervised machine learning and ensemble techniques: a comparative study. Journal of Software Engineering and Applications. URL: https://doi.org/10.4236/jsea.2019.125007 (дата звернення: 27.11.2023).

24. Проекти та методи гібридних розробок. URL : https://visuresolutions.com/uk/requirements-management-traceability-guide/hybrid-development (дата звернення: 27.11.2023).

25. Горюк Н. В. Засоби інтеграції технології статичного аналізу безпеки вихідного коду у середовище розробки програмного забезпечення / Горюк Н. В. // «Сучасний захист інформації» –2020. – 324 с.

26. SonarQube 10.3 Documentation. URL : https://docs.sonarsource.com/sonarqube/latest/ (дата звернення: 27.11.2023).

27. ESLint statically analyzes your code to quickly find problems. URL : https://eslint.org/ (дата звернення: 27.11.2023).

28. Selenium. URL : https://www.selenium.dev/ (дата звернення: 27.11.2023).

29. LoadRunner. URL : https://azuremarketplace.microsoft.com/en-us/marketplace/apps/micro-focus.ms-loadrunner?tab=overview (дата звернення: 27.11.2023).

30. Visual Studio. URL: https://codeguida.com/post/1754 (дата звернення: 27.11.2023).

31. IntelliJ IDEA. URL: https://www.jetbrains.com/idea/ (дата звернення: 27.11.2023).

32. Eclipse. URL: https://www.eclipse.org/ (дата звернення: 27.11.2023).

33. Software defect prediction dataset. URL: https://www.kaggle.com/datasets/nazgolnikravesh/software-defect-prediction-dataset (дата звернення: 27.11.2023).

34. Downey A. Think Python 2e: How to Think Like a Computer Scientist. - O'Reilly Media, 2015. – 292 с.

35. Horstmann C.S. Core Java SE 9 for the Impatient. - Addison-Wesley Professional, 2017. – 576 с.

36. Albahari J., Albahari B. C# 7.0 in a Nutshell: The Definitive Reference. - O'Reilly Media, 2017. – 1088 с.

37. Richter J. CLR via C#: Applied .NET Framework 4.5. - Microsoft Press, 2012. – 896 с.

38. What is .NET. URL: https://www.trustradius.com/products/net/reviews (дата звернення: 27.11.2023).

39. What is three-tier architecture. URL: URL:https://www.ibm.com/ topics/three-tier-architecture (дата звернення: 27.11.2023).

40. MVP (Model-View-Presenter) — патерн розробки інтерфейсу користувача. URL: https://brander.ua/technologies/mvp (дата звернення: 27.11.2023).

41. software defect prediction dataset. URL: https://www.kaggle.com/datasets/nazgolnikravesh/software-defect-prediction-dataset (дата звернення: 27.11.2023).

42. Testing .NET Code in Visual Studio 2019. URL: https://www.pluralsight.com/courses/visual-studio-testing-dotnet-code?utm_source=google&utm_medium=paid-search&utm_campaign=upskilling-and-reskilling&utm_term=ssi-emea-xyz-dynamic&utm_content=free-trial&gad_source=1&gclid=CjwKCAiAsIGrBhAAEiwAEzMlCwsHFEIT8zymRgdP

oQIHlgiiwQ_T9DDF09eSBKk0JI6kQZG60YormxoCjiEQAvD_BwE (дата звернення: 27.11.2023).