МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки
Центр післядипломної освіти
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження методів штучного інтелекту для тестування програмного забезпечення

Виконав:
студент 2 курсу групи ІПЗмзд-21-2
Айлазян С.К.
(прізвище, ініціали)

Спеціальність	121 – Інженерія програмного забезпечення
Тип програми	Освітньо-наукова
Керівник	доц. Лановий О. Ф.

(посада, прізвище, ініціали)

Допускається до захисту
Зав. Кафедри _____ З.В. Дудар

2023

Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти
Кафедра Програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)

«__»_____2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Айлазяну Стефану Каримовичу
(прізвище ім'я по батькові студента)

1. Тема роботи «Дослідження методів штучного інтелекту для тестування програмного забезпечення», затверджена наказом університету від « 03 » квітня 2023 р. № 83 Стз.

2. Термін подання студентом роботи до екзаменаційної комісії: «15» травня 2023 р.

3. Вихідні дані до роботи електронні ресурси за обраною тематикою, методичні вказівки до виконання кваліфікаційної роботи магістра, пояснювальна записка.

4. Перелік питань, що потрібно опрацювати в роботіаналіз предметної області, постановка задачі, огляд існуючих методів та патентів, опис розробленої системи, математичне моделювання, опис проектних рішень, опис та аналіз експериментів.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд наукової та патентної літератури	20.02.2023	виконано
2	Постановка задачі	20.02.2023	виконано
3	Визначення вимог	04.03.2023	виконано
4	Планування експериментів	10.03.2023	виконано
5	Проектування та реалізація програмної системи	01.04.2023	виконано
6	Розробка плану проведення експериментальних досліджень	06.04.2023	виконано
7	Проведення експериментальних досліджень	12.04.2023	виконано
8	Аналіз отриманих результатів	16.04.2023	виконано
9	Підготовка пояснювальної записки	06.05.2023	виконано
10	Підготовка презентації та доповіді	11.05.2023	виконано
11	Перевірка на плагіат	11.05.2023	виконано
12	Нормоконтроль	11.05.2023	виконано
13	Архівування	16.05.2023	виконано
14	Попередній захист	18.05.2023	виконано
15	Допуск до захисту у зав. кафедри	18.05.2023	виконано

Дата видачі завдання 20 лютого 2023 р.

Студент _____

(підпис)

Керівник роботи _____ доц. Лановий О. Ф.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра: 78 с., 23 рис., 20 джерел.

ЗГОРТКОВА НЕЙРОННА МЕРЕЖА, МАШИННЕ НАВЧАННЯ, ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ФРЕЙМВОРК, ШТУЧНИЙ ІНТЕЛЕКТ, GITHUB, KERAS, TENSORFLOW.

Об’єкт – дослідження застосування методів штучного інтелекту при тестування програмного забезпечення.

Мета роботи – вдосконалення методів управління процесом тестування програмного забезпечення шляхом використанням методів штучного інтелекту для прогнозування можливих результатів тестування.

Методи дослідження – в дослідженні використовувалися методи машинного навчання та кластеризації, прогнозування результатів тестування для покращення ефективності процесу тестування.

SOFTWARE TESTING, ARTIFICIAL INTELLIGENCE, MACHINE LEARNING, CONVOLUTIONAL NEURAL NETWORK, FRAMEWORK, TENSORFLOW, KERAS, GITHUB

Object – study of the application of artificial intelligence methods in software testing.

The aim of the work is to improve the methods of managing the software testing process by using artificial intelligence methods to predict possible test results.

Research methods – the research used machine learning and clustering methods, predicting test results to improve the effectiveness of the testing process..

Умови публікації пояснювальної записки

Я, Айлазян Стефан Каримович
(прізвище, ім'я, по батькові)

студент(~~ка~~) групи ІПЗздм21-2, здобувач вищої освіти на другому (магістерському) рівні кафедри Програмної інженерії,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему: «Дослідження методів штучного інтелекту для тестування програмного забезпечення»,
(назва роботи)

що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(~~а~~) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

СКОРОЧЕННЯ

ЗНМ – згорткова нейронна мережа

МН – машинне навчання

НМ – нейронна мережа

ПЗ – програмне забезпечення

ШІ – штучний інтелект

ШНМ – штучна нейронна мережа

API – програмний інтерфейс програми

CI – безперервна інтеграція

ЗМІСТ

ВСТУП

Тестування програмного забезпечення є важливою складовою процесу розробки ПЗ, що дозволяє виявити і виправити помилки та недоліки в роботі програми. Проте традиційні методи тестування досить часто виявляються недостатньо ефективними у вирішенні складних завдань, а також вимагають великої кількості людських ресурсів і часу. З одного боку, зростають вимоги до якості програмного забезпечення, оскільки воно використовується в різних сферах, таких як медицина, банківський сектор, авіація, автомобільна промисловість та інші. Відповідно, недоліки програмного забезпечення можуть призвести до небезпечних ситуацій, що загрожують життю та здоров'ю людей або призводять до значних матеріальних збитків [1]. З іншого боку, швидкий темп розвитку інформаційних технологій вимагає від програмного забезпечення більшої функціональності та складності. Це призводить до збільшення обсягу тестування та зниження ефективності традиційних методів тестування, що змушує розглядати нові, більш ефективні методи.

Штучний інтелект, зокрема методи машинного навчання та нейронних мереж, достатньо широко використовуються для автоматизації процесу тестування та покращення ефективності виявлення помилок. Наприклад, інтелектуальні алгоритми можуть застосовуватись для визначення оптимальних шляхів проведення тестування програми, а також для прогнозування можливих проблеми в роботі програми, що дозволяє зменшити кількість помилок та скоротити час, необхідний для тестування програмного забезпечення.

У цьому контексті дослідження методів штучного інтелекту для тестування програмного забезпечення може допомогти у пошуку можливих шляхів покращення якості тестування. Інтелектуальні алгоритми машинного навчання та нейронні мережі можуть допомогти автоматизувати процес тестування, зменшити час, необхідний для його проведення та збільшити ефективність виявлення помилок. Також використання штучного інтелекту може підвищити точність прогнозування поведінки програмного забезпечення в різних умовах його функціонування [2].

Отже, актуальність дослідження полягає в тому, що дослідження методів штучного інтелекту для тестування програмного забезпечення є важливим напрямком, який дозволить підвищити ефективність процесу тестування, знизити витрати на тестування, а також забезпечити високу якість програмного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Місце тестування в розробці програмного забезпечення

За останні кілька років інженерія програмного забезпечення стала однією з найбільш швидкозростаючих галузей. Програмне забезпечення інтегроване в усі сфери повсякденного життя: від використання електронних пристроїв, таких як телефони та ноутбуки, до повсякденного використання різноманітного прикладного ПЗ для, наприклад, оплати покупок чи поїздок на роботу. Сьогодні майже неможливо знайти сферу застосування, яка б не була опосередкована програмним забезпеченням.

У зв’язку з цим бумом індустрія програмної інженерії повинна була швидко розвиватися. Враховуючи високу поширеність програмного забезпечення, воно повинно бути надійним та відмовостійким. Як наслідок, індустрія програмної інженерії прийняла вже існучий набір найкращих практик перевірки якості для забезпечення надійного використання своїх продуктів. Серед цих найкращих практик і, мабуть, найважливішою, є тестування програмного забезпечення. Очікується, що програмні проекти повинні мати комплексний набір тестів, який гарантує, що програмне забезпечення працює правильно в кожному випадку використання. Будь-який збій у тестуванні вважається критичною помилкою і досить не часто допускається у публічних версіях або у виробничому програмному забезпеченні.

Така важливість тестування програмного забезпечення зумовила необхідність розробки набору інструментів, які спрощують процес тестування. Зрештою, спрощення написання та запуску тестів гарантує, що всебічне тестування стане легкою звичкою для розробників. Фреймворки для тестування є важливими частинами програмного забезпечення, які допомагають у процесі написання тестів.

Підготовка та написання великого набору тестів – це лише одна частина тестування. Інша частина полягає в тому, що необхідно переконатися, що програмне забезпечення продовжує послідовно проходити ці ж самі тести після внесення до нього оновлень. Це досить важко для розробників ПЗ, що тестують

свій код вручну, але ще більш складною стає ситуація, коли очікується, що код буде працювати на різних «платформах». В цьому розумінні термін «платформа» розглядається як загальне визначення для позначення будь-якої варіації середовища тестування – наприклад, мобільні додатки можуть тестуватися на різних версіях iOS та Android, веб-додатки – на різних браузерах, а програмні бібліотеки – на різних версіях мови програмування, якою вони створені.

Подібно до того, як тести повинні послідовно виконуватись на будь-якій платформі, бібліотеки також повинні працювати на всіх цих платформах без виключення. Однак налаштування всієї множини платформ може бути складним у локальних середовищах розробки, що може суттєво сповільнити процес розробки ПЗ.

Безперервна інтеграція – це практика розробки програмного забезпечення DevOps, коли розробники регулярно об’єднують зміни програмного коду у центральному репозиторії, після чого автоматично виконується його збирання, тестування і запуск. Поняття безперервної інтеграції найчастіше застосовується до стадії збірки або інтеграції процесу випуску ПЗ і включає як компонент автоматизації (наприклад, сервіс безперервної інтеграції), так і компонент культури розробки (наприклад, навчання частої інтеграції). Головним завданням безперервної інтеграції є прискорення процесу знаходження та виправлення дефектів, покращення якості ПЗ та скорочення часових витрат на перевірку та випуск оновлень ПЗ.

На теперішній час сервіси безперервної інтеграції (CI) стали основною частиною робочого процесу DevOps з розробки ПЗ. Ці сервіси дозволяють розробникам налаштовувати сервери, роль яких полягає в тому, щоб приймати новий код, збирати його і запускати на ньому різні завдання для перевірки його функціональності. Ці завдання не обов’язково повинні бути пов’язані з тестуванням, і дозволяють розробникам запускати широкий спектр метрик. Тим не менш, найбільше поширення послуги CI знаходять саме в автоматизованому тестуванні, оскільки вони дозволяють запускати весь набір тестів щоразу, коли

у проект завантажується новий код. Крім того, сервери, які вони надають, легко конфігуруються, що дозволяє розробникам налаштовувати відповідні платформи на своїх серверах CI і забезпечувати належне виконання набору тестів на всіх підтримуваних платформах (рис.1.1).

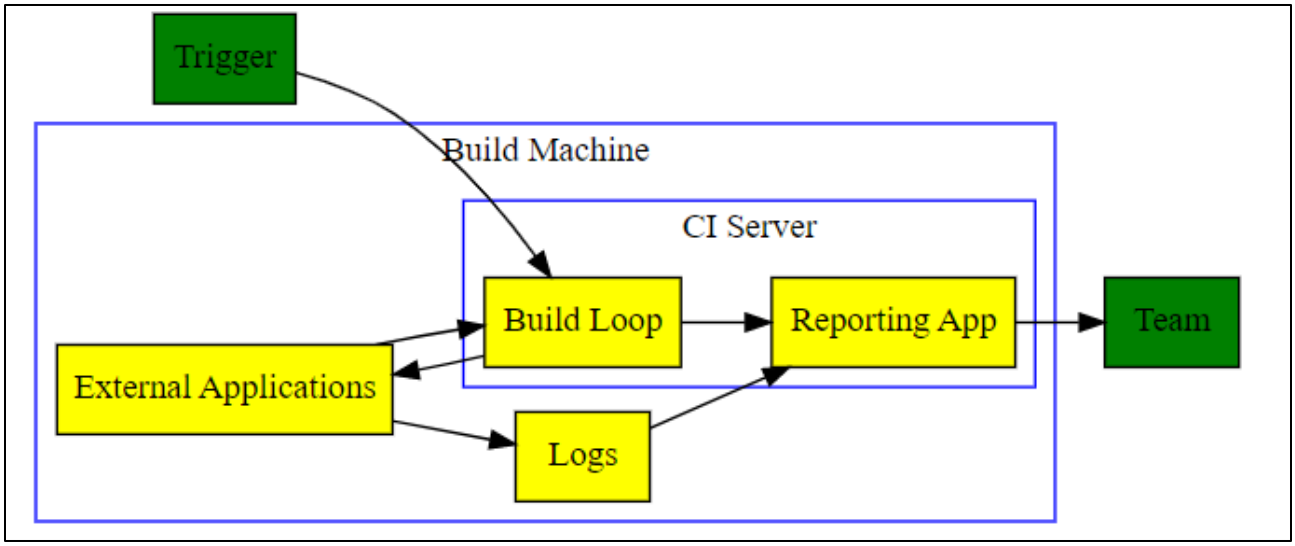

Рисунок 1.1 – Робочий процес безперервної інтеграції

Успішність та стрімке поширення таких сервісів як Windows Azure, Google App Engine, Heroku, Git та Perforce дозволило використовувати їх в проектах всіх форм і розмірів – від розробки масового промислового ПЗ до фрілансер-проектів.

Всі ці інструменти та сервіси мають на меті зробити процес розробки ПЗ та тестування максимально простим та зручним. Як результат: розробники та тестувальники вимушені писати багато тестів, оскільки вони прагнуть всебічно протестувати свій код. Це стає особливо очевидно, коли тестувальники не є розробниками і не знайомі з принципами побудови коду, що вимагає ще більшої кількості тестів, щоб переконатися, що всі частини функціональності ПЗ перевірені. Однак це має свою ціну: час. У простих випадках, коли кількість тестів зростає до сотень або навіть тисяч, потрібно все більше часу, щоб виконати їх усі. У проектах з декількома сотнями тестів повний прогін може зайняти кілька хвилин; у великих проектах час тестування може наближатися до критичної точки, коли він починає активно шкодить продуктивності

розробника [16]. Хоча, на перший погляд, витрата декількох хвилин на запуск тестового набору може здатися не надто проблематичним, будь-які затримки в робочому процесі розробника можуть швидко накопичуватися, особливо коли запуск всього тестового набору локально непрактичний і йому доводиться покладатися на автоматизоване тестування, яке виконує його CI, що додає додатковий час. Ситуація ще більше ускладнюється зростаючою складністю програмного забезпечення, широким розмаїттям платформ і більшою кількістю складних можливих варіантів використання, які необхідно моделювати за допомогою все більш складних і довготривалих тестів, щоб переконатися, що все працює в умовах різних сценаріїв використання. Таким чином, забезпечення швидкого виконання цих тестів є активною сферою роботи розробників, що вимагає від них забезпечення оптимізації тестів і тестових середовищ, які вони створюють, для досягнення максимальної ефективності тестування. Існує велика кількість посібників для налагодження уповільнень під час виконанні тестів [10], а самі інструменти CI надають відповідну функціональність для забезпечення максимально швидкого тестування.

Дуже важливо розуміти, чому швидкі чи скорочені набори тестів є важливими. Зрештою, чому розробник не може перейти до роботи над новою функцією, поки тести ще продовжують виконуватись? Вони не можуть цього зробити через можливість провалу тестових кейсів. Оскільки будь-яка помилка вважається неприйнятною в стандартній практиці розробки ПЗ, розробники повинні забезпечити, щоб помилки виявлялися якомога меншими витратами часу. Крім того, невдалий тест досить часто вказує на помилки в коді, які необхідно негайно виправити, щоб пізніші доповнення не були порушені помилковим початковим станом. Тому розробники найбільше зацікавлені у швидкому знаходженні невдалих тестових кейсів. Якщо про такі збої повідомляється на ранній стадії процесу тестування, то розробники мають більше шансів виявити, ідентифікувати та виправити свої помилки, а також надіслати нові версії коду для тестування. З іншого боку, якщо про ці збої повідомляється наприкінці процедури запуску тесту, то розробнику доведеться

чекати значно довше, перш ніж він зможе знову стати продуктивним і виправити ці проблеми. Таким чином, для сервісів тестування дуже важливо мати можливість розробляти методи для визначення тестових кейсів, які є найбільш схильними до збоїв, і ставити їх у чергу першими, забезпечуючи швидкий зворотній зв'язок зі своїми користувачами (рис.1.2).

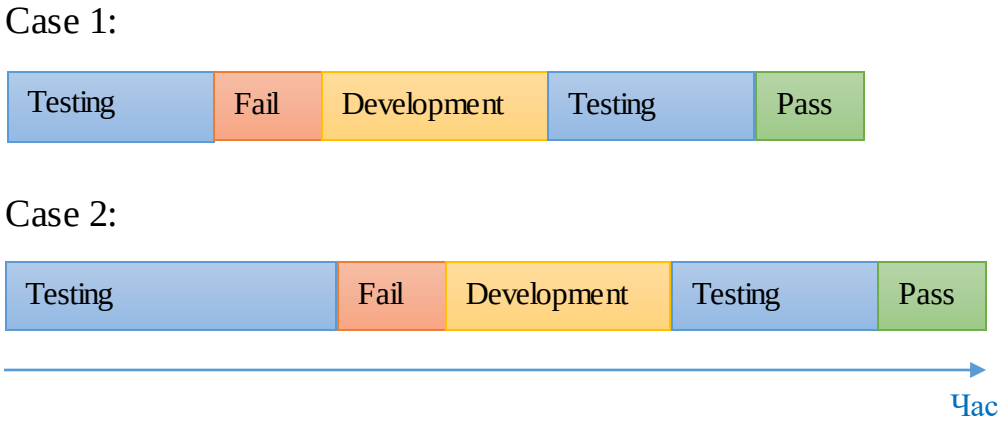

Рисунок 1.2 – Порівняння часу визначення дефектів

Рисунок ілюструє приклад того, як раннє виявлення збоїв може вплинути на ефективність професійної діяльності розробника. Якщо тест не пройшов на ранній стадії, розробник може швидше виправити дефект, повторити ітерації і швидше досягти бажаного результату. У тому випадку, якщо тест виявив дефекти ПЗ наприкінці процесу тестування, час розробки може зайняти значно більше часу.

1.2 Процеси управляння тестуванням ПЗ

Зазвичай процес управління тестуванням складається з трьох етапів: процес планування тестування, процес проведення тестування і, нарешті, процес завершення тестування [16]. Під час планування тестування проводиться детальний аналіз вимог, викладений в проектній документації, які в подальшому об’єднуються в тест-кейси та тестові сценарії. В процесі виконання тесту проходження етапів, які описані у вказаних артефактах, виконується для оцінки фактичної поведінки ПЗ для порівняння її з очікуваною поведінкою. Якщо під час тестування фактична та очікувана поведінка ПЗ

збігаються, то викликається процес завершення тесту, в іншому випадку повідомляється про дефект, який вказує на відхилення. В процесі завершення тесту перевіряються результати виконання всіх тестових кейсів, сценаріїв і скриптів, а також стан всіх зареєстрованих дефектів. На основі цього готується звіт про закриття тесту, який передує завершенню тестової діяльності. На рисунку 1.3 зображено процес динамічного тестування, який детально описує етапи процесу управління тестуванням.

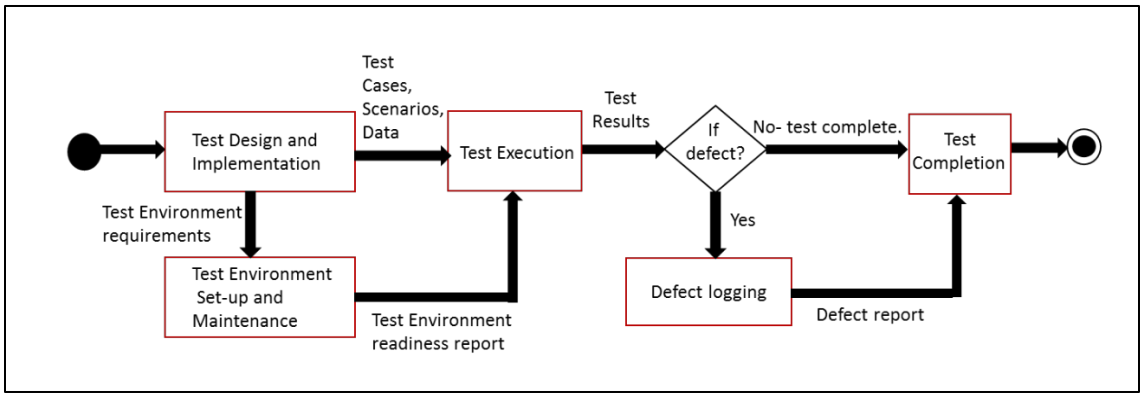

Рисунок 1.3 – Процес динамічного тестування

Іншим важливим аспектом тестування є вибір стратегії процесу тестування. Параметри якості, такі як безпека, надійність, функціональна придатність, зручність використання, безпека тощо підтверджуються та перевіряються декількома типами тестів, а саме: тестування продуктивності, функціональне тестування, тестування зручності використання тощо. Ці процеси тестування відбуваються на різних фазах життєвого циклу розробки ПЗ [1]. Тому весь процес тестування є комбінацією таких підпроцесів. Зв'язок типів тестування, рівнів тестування, параметрів якості з процесом тестування проілюстровано на рисунку 1.4.

Існує три способи організації тестування.

Тестування чорного ящика (black-box testing) – це метод тестування програмного забезпечення, при якому вхідні дані та результати не повинні залежати від внутрішньої структури програми. Тобто тестувальник не знає, як саме програма працює, і тестує її з точки зору функціональності, перевіряючи, чи працює програма правильно відповідно до вимог специфікації. Цей метод

тестування зазвичай використовують на ранніх стадіях розробки програмного забезпечення, коли ще не має достатньо інформації про внутрішню структуру системи. Для цього використовуються різноманітні методи тестування, такі як еквівалентність, граничні значення, випадкові тестові дані тощо.

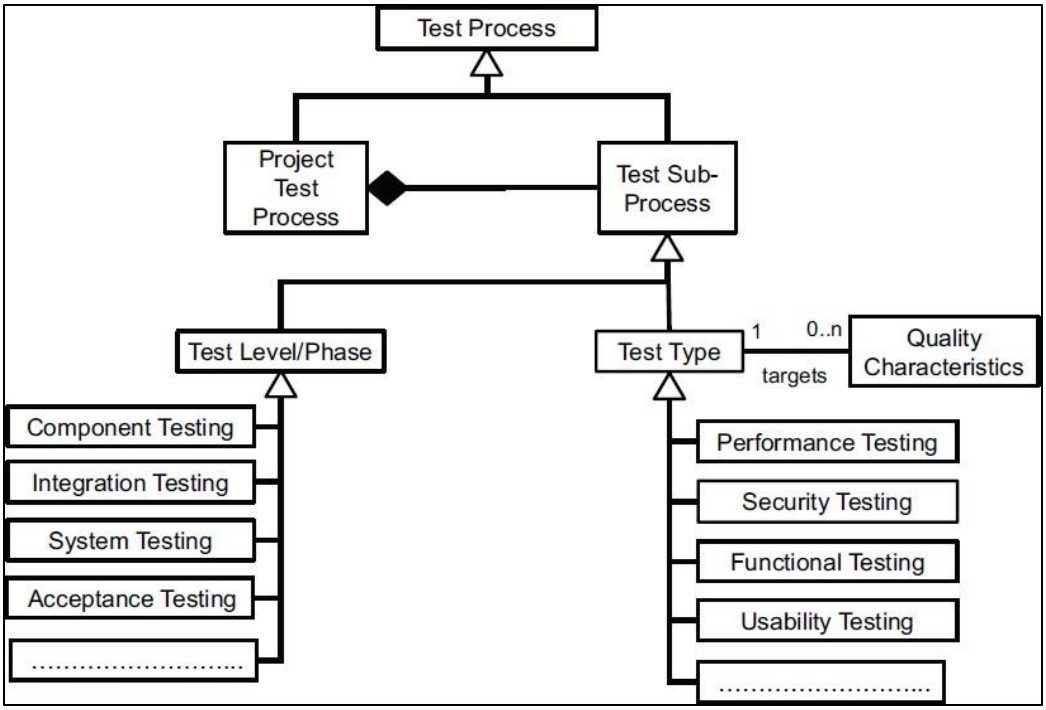

Рисунок 1.4 – Етапи та типи тестування

Метод тестування чорного ящика є ефективним інструментом для перевірки відповідності програми вимогам, а також для забезпечення якості програмного забезпечення в цілому.

Метод тестування білого ящика (white-box testing) є одним з методів тестування програмного забезпечення, який базується на аналізі внутрішньої структури програми та знанням її логіки роботи. Цей метод тестування дозволяє виявляти помилки в програмному коді та перевіряти правильність виконання умов та гілок коду.

Для проведення тестування білого ящика зазвичай використовують такі техніки, як тестування рівня операцій, тестування рівня рішень, тестування рівня умов, тестування граничних значень та тестування з використанням змінних. Тестування білого ящика можна проводити як вручну, так і з використанням автоматизованих тестових фреймворків.

Метод тестування білого ящика дозволяє виявляти більшу кількість помилок у програмному забезпеченні, оскільки тестувальник має доступ до всієї інформації про програму. Однак, цей метод вимагає значної кількості зусиль для аналізу вихідного коду та розробки тестових сценаріїв, тому він може бути менш ефективним за іншими методами тестування у випадку складних програмних систем.

Тестування сірої скриньки (grey-box testing) є комбінацією методів тестування білого та чорного ящика. При цьому тестувальник має деяку інформацію про внутрішню структуру системи, але не повністю ознайомлений з її роботою.

Зазвичай, тестування сірої скриньки використовується в тих випадках, коли необхідно забезпечити високу якість програмного забезпечення, але з обмеженими ресурсами. Тестування сірої скриньки дозволяє ефективно використовувати ресурси для проведення тестування, тому що тестувальник враховує внутрішню структуру системи під час розробки тестів [3].

При тестуванні сірої скриньки тестувальник проводить аналіз внутрішньої структури системи, щоб зрозуміти її роботу. Далі, тестувальник створює тести, які включають в себе знання про структуру системи та залежності між компонентами, а також про вхідні та вихідні параметри.

Основними методами тестування сірої скриньки є тестування інтерфейсу, тестування внутрішніх структур та тестування функцій. При тестуванні інтерфейсу тестувальник перевіряє, чи працює користувальницький інтерфейс правильно. При тестуванні внутрішніх структур тестувальник перевіряє, чи працюють окремі компоненти системи правильно. При тестуванні функцій тестувальник перевіряє, чи працює програмне забезпечення відповідно до вимог та специфікацій. На рисунку 1.5 представлено схематичне зображення цих трьох методів

На підсумку можна зазначити, що головна задача тестування полягає в тому, щоб перевірити, чи відповідає програмне забезпечення вимогам та очікуванням користувачів. Це означає, що тестування має перевірити, чи

працює програма правильно, чи відповідає вона специфікації, чи відповідає вона потребам користувачів, чи не має помилок та дефектів, і чи може вона працювати в різних умовах та середовищах.

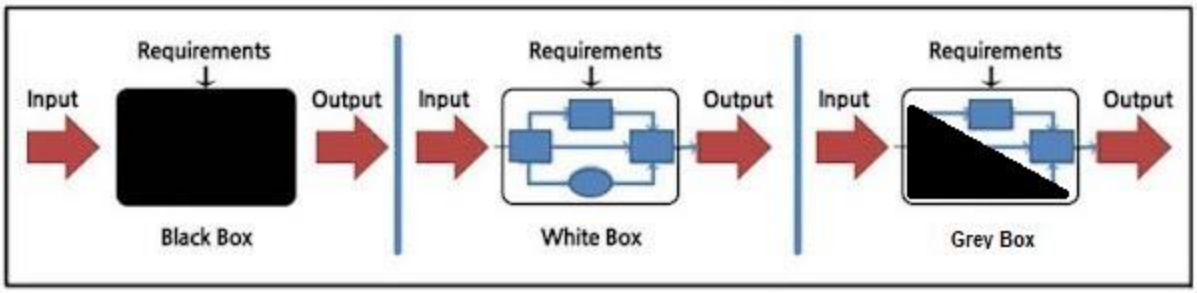

Рисунок 1.5 – Методи тестування

Головною метою тестування є забезпечення якості програмного забезпечення, що забезпечується шляхом виявлення та виправлення помилок та дефектів в програмі перед її випуском на ринок або перед подальшим використанням користувачами.

У цілому, головна задача тестування полягає в тому, щоб забезпечити якість програмного забезпечення та підвищити рівень довіри користувачів до продукту, але повністю виключити можливість появи збоїв та відмов в ПЗ неможливо.

1.3 Прогнозування збоїв та відмов

Концепція прогнозування збоїв ПЗ не є новою. Останні досягнення в обробці даних і машинному навчанні дозволили сформувати прогностичні моделі в найрізноманітніших сферах, і тестування програмного забезпечення не є винятком. Автоматичне виявлення помилок є великою сферою досліджень, і були побудовані успішні моделі, які можуть знаходити помилки, досліджуючи вихідний код [17]. Інші моделі здатні автоматично створювати тести на основі наявних у проекті функцій і підтримувати ці тести в актуальному стані, коли ці функції змінюються [4]. Нарешті, було проведено багато досліджень щодо прогнозування результатів тестування на основі вихідного коду проекту [18]. Ця остання група є найбільш привабливою з точки зору можливості

безпосередньо передбачати, які тести можуть не пройти, коли у проект додається новий код.

Разом з тим важливо зазначити, що ці роботи були орієнтовані на аналіз вихідного коду для проведення прогнозування. Їх моделі повинні переглядати або фактичні файли коду, або скомпільовані двійкові файли, щоб виокремити ті ознаки, які потім можуть бути використані для опрацювання прогнозними моделями. Необхідність пошуку якісних ознаках вимагає часу, витраченого на експерименти з різними тестовими наборами ознак, щоб знайти той, який може призвести до найкращих прогнозів. Навіть у тому випадку, якщо такий набір знайдено, то для іншого проекту потрібна подальша робота з повторного вилучення цих ознак з коду, який може бути в іншому форматі або відповідати іншому стилю кодування. Ці дії є досить складними і трудомісткими, і вимагають глибокого знання принципів побудови та організації програмного проекту.

Для подолання цього недоліка в роботі буде приділено увагу розробці системи для прогнозування результатів тестування виключно на основі минулих результатів тестування. Зокрема, один запуск набору тестів може бути використаний для створення матриці результатів, де кожен рядок представляє унікальний тестовий випадок, а кожен стовпець – платформу, на якій він тестується (див. рис.1.3). Таким чином, набір даних буде складатися з усіх запусків тестового набору за всю історію проекту (збірки, релізу), представлених у вигляді серії таких матриць.

	Platform 1	Platform 2	Platform 3	Platform 4	…
Test 1	PASS	PASS	PASS	PASS	
Test 2	PASS	PASS	PASS	PASS	
Test 3	FAIL	FAIL	FAIL	FAIL	
Test 4	FAIL	PASS	PASS	PASS	
…					

Рисунок 1.3 – Приклад матриці результатів тестування

За рахунок такого підходу формат набору даних є послідовним для різних проектів будь-якого типу, написаних будь-якою мовою: не потрібно додатково налаштовувати систему під різні функції в різних проектах, натомість можна використовувати єдиний інтерфейс для збору даних і навчання моделі для подальшого прогнозування. Такий інтерфейс також дозволяє досить легко розширювати систему.

1.4 Постановка задачі

Метою кваліфікаційної роботи є дослідження методів та інструментальних засобів на основі штучного інтелекту для підвищення ефективності та надійності тестування програмного забезпечення. Це дозволить покращити ефективність тестування та скоротити час, необхідний для виконання тестів.

Для досягнення поставленої мети роботи необхідно вирішити наступні локальні завдання:

- дослідити існуючі методи та підходи до тестування ПЗ з використанням штучного інтелекту;
- проаналізувати вимоги до програмної системи для тестування ПЗ, яка буде враховувати результати тестів, що були виконані у минулому;
- розробити модель програмної системи для тестування ПЗ з використанням методів штучного інтелекту;
- розробити алгоритми для оптимізації процесу тестування, використовуючи дану модель;
- провести експериментальне дослідження розробленої моделі та алгоритмів на реальних даних, що були зібрані під час тестування ПЗ.

В результаті виконання роботи очікується створення програмної системи для тестування ПЗ, яка зможе використовувати результати тестів, що були виконані у минулому, для оптимізації процесу тестування та зменшення часових витрат на тестування нових версій ПЗ. Модель програмної системи, яка розробляється, містить інструмент машинного навчання для покращення якості

тестування програмного забезпечення. Ця система буде використовувати у якості вхідних даних результати попередніх етапів тестування, як це було описано вище, для навчання своїх предиктивних моделей для прогнозування результатів тестування. Планується, що системі, яка розробляється, будуть необхідні лише ці дані, не вимагаючи жодних інших форм врахування особливостей написання вихідного коду.

На рисунку 1.4 наведено загальну схему системи, що підлягає розробці. З Git-репозиторію користувачі можуть отримувати дані про історію результатів тестування. Потім вони можуть використовувати ці дані для доповнення існуючого процесу запуску тестів предиктивною моделлю, яка може робити прогнози щодо майбутніх результатів тестування.

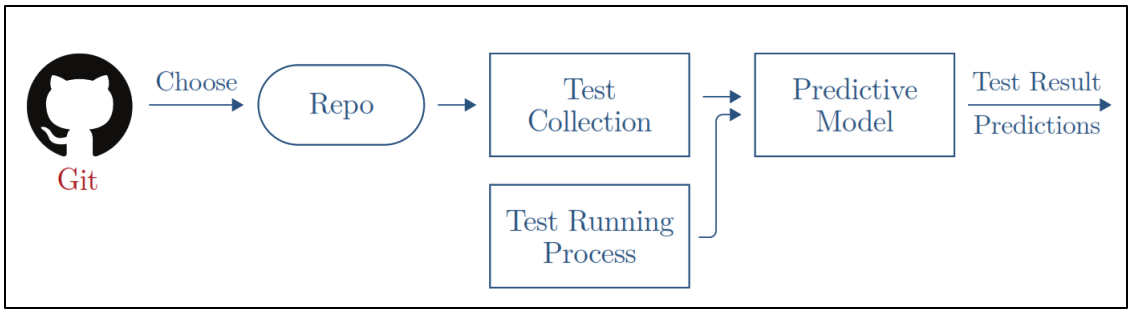

Рисунок 1.4 – Модель системи тестування

Інструментальна система для тестування методами штучного інтелекту буде орієнтована на використання у двох типах репозиторіїв:

– програмні проекти, які використовують Git або GitHub. Для Git-репозиторіїв ця система буде служити наскрізним інструментом, який зможе отримувати з Git-проекту дані про результати тестування за всю історію проекту, завантажувати ці дані в заздалегідь визначені моделі, навчати їх та використовувати для майбутніх прогнозів щодо нових проектів;
– інші репозиторії: користувачі будуть повинні самостійно підготувати дані про результатів тестування і відформатувати їх у відповідності до специфікації. Для аналізу даних вини також зможуть використовувати зовнішній API для створення власного інструменту для прогнозування

результатів тестування нових випусків або збірок.

Таким чином, програмна система повинна мати 2 основні модулі, які служать для:

– збору наборів даних про результати тестування з історії Git-проекту;
– навчання моделей, які виконують прогнозування результатів тестування.

Програмна система, що підлягає розробці, буде прототипом для застосування машинного навчання для прогнозування результатів тестування програмного забезпечення та інструментальним засобом щодо використання методів штучного інтелекту в тестуванні ПЗ. Технології, які будуть використовуватися для розробки програмного продукту, буде розглянуто більш детально у наступних розділах.

2 ДОСЛІДЖЕННЯ МЕТОДІВ РОЗВ'ЯЗАННЯ ЗАДАЧІ

2.1 Штучний інтелект та його місце в тестуванні ПЗ

Штучний інтелект (Artificial Intelligence) – це галузь комп'ютерних наук, яка займається створенням систем, що вміють виконувати завдання, які зазвичай вимагають людської інтелектуальної діяльності, такі як розпізнавання мови, розуміння природної мови, планування, прийняття рішень та інше. Штучний інтелект може використовувати різні методи, такі як машинне навчання, глибинне навчання, нейронні мережі, генетичні алгоритми та інші для досягнення своїх цілей. Взагалі, метою штучного інтелекту є розробка систем, що здатні розуміти та вирішувати завдання, що зазвичай вимагають людської інтелектуальної діяльності.

Аналіз існуючих методів тестування ПЗ з використанням штучного інтелекту показує, що такі методи можуть бути дуже ефективними, особливо в умовах швидкого розвитку технологій та збільшення складності програмних систем [2]. До найбільш популярних методів відносять наступні.

Генетичні алгоритми. Цей метод використовує ідеї еволюції для створення оптимальних тестових сценаріїв. Алгоритм постійно змінює параметри тестування, поки не знайде найбільш ефективний спосіб тестування програмного забезпечення.

Нейронні мережі. Метод використовує штучні нейронні мережі для навчання системи розпізнавати і передбачати помилки в програмному забезпеченні. Навчання моделі відбувається на основі результатів попередніх тестів.

Методи байесівської мережі. Використовує статистичні методи для моделювання відносин між різними факторами тестування та результатами тестів. Це дозволяє системі передбачати, які тестові сценарії найбільш ефективні для певної програмної системи.

Методи машинного навчання. Цей метод найбільш поширений для автоматичного створення тестових сценаріїв на основі попередніх результатів тестування. Модель навчається на основі відомих результатів тестування, а

потім застосовується для передбачення ефективності тестування на нових програмних системах.

Аналіз статичного коду. Метод використовує аналізатори коду для виявлення можливих помилок та дефектів програмного забезпечення на основі статичного аналізу коду.

Наведені методи ШІ дозволяють автоматизувати процес тестування ПЗ.

Під час використання методів ШІ для тестування ПЗ він будує шар абстракції над ПЗ, яке тестується, так само, як людський мозок створює абстракцію ПЗ та його очікуваної поведінки. «Ключовими перевагами тестування на основі методів ШІ є те, що воно є універсальним, багаторазовим, надійним, адаптивним і масштабованим» [17]. Підходи машинного навчання можна використовувати для тестування різних рівнів тестів, атрибутів якості та додатків у різних доменах. Завдяки незалежності одні й ті ж тести можуть використовуватися для декількох додатків в межах домену або між доменами [6].

Окремою галуззю ШІ є машинне навчання. Машинне навчання (Machine learning) вивчає, як комп'ютерні системи можуть навчатися із обраних даних та робити передбачення чи приймати рішення без явного програмування. У методах штучного інтелекту машинне навчання застосовується для розв'язання різноманітних завдань, таких як класифікація, регресія, кластеризація, виявлення аномалій та інше [14]. Машинне навчання (МН) використовує алгоритми та моделі, які здатні адаптуватися до нових даних та знаходити складні залежності між вхідними та вихідними даними. Застосування МН може значно полегшити та прискорити роботу з даними та зробити її більш ефективною та точною.

2.2 Типи машинного навчання

Категоризація методів МН базується на різних факторах [18]. Кожен варіант використання відповідає одному з нижчезазначених типів, і кожен тип

МН може бути реалізованим у певний спосіб в залежності від його застосування для тестування ПЗ.

2.2.1 Навчання під наглядом

Навчання під наглядом (Supervised Learning) – це один із видів МН, де алгоритм отримує вхідні дані разом з відповідними правильними відповідями, тобто із належно позначеними даними навчання. Задача полягає у тому, щоб побудувати модель, яка може передбачати відповіді на нові, раніше невідомі дані, з використанням знань, набутих на основі вхідних даних і правильних відповідей.

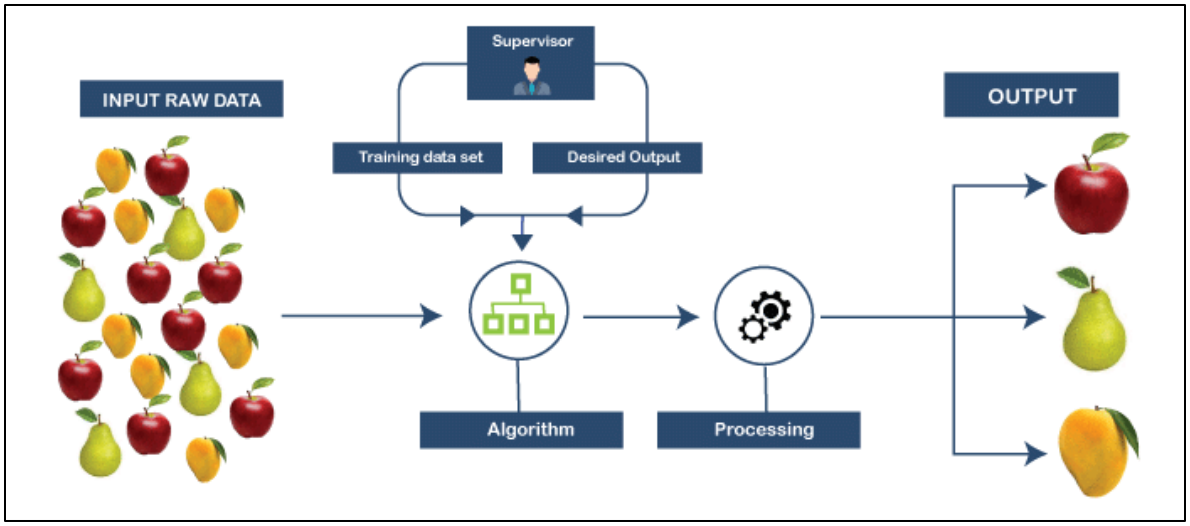

Рисунок 2.1 – Навчання під наглядом

При навчанні під наглядом вхідні дані і відповіді використовуються для побудови моделі, що здатна передбачати відповіді на нові вхідні дані. Процес навчання полягає в тому, щоб знайти такі параметри моделі, які мінімізують функцію втрат на вхідних даних навчання [7]. Після завершення навчання модель можна використовувати для передбачення відповідей на нові вхідні дані. У подальшому цей тип поділяється на 2 підтипи, а саме:

– класифікація: алгоритм повинен розрізняти заданий набір класів як мітку;
– регресія: метою алгоритму є прогнозування числових значень,

відсотків або ймовірностей в якості мітки.

2.2.2 Навчання без нагляду

Навчання без нагляду (Unsupervised Learning) є одним з видів МН, в якому модель навчається виявляти закономірності у наборі даних без попереднього навчання на мітках. На відміну від навчання під наглядом, де модель отримує набір даних з позначками, які пов'язані з очікуваним вихідним результатом, в навчанні без нагляду модель отримує тільки вхідні дані, не знаючи, які очікувані результати (див.рис.2.2).

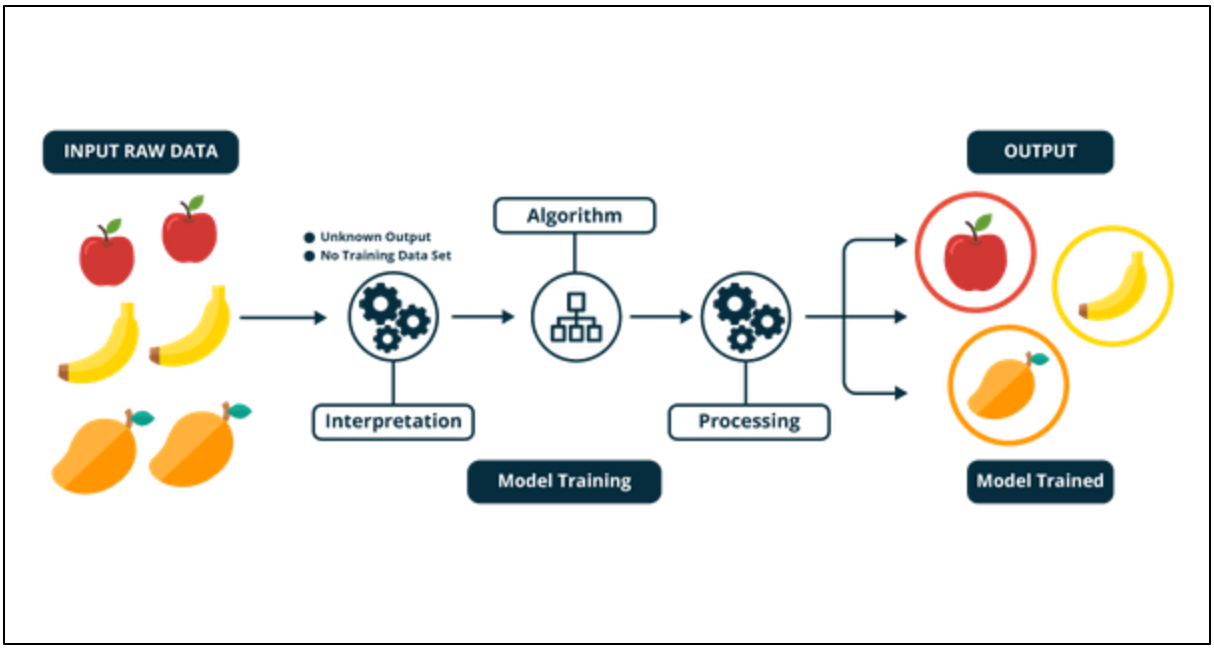

Рисунок 2.2 – Навчання без нагляду

Моделі навчання без нагляду використовуються для виявлення складних структур в даних, які можуть бути важко виявлені вручну. Наприклад, алгоритми кластеризації можуть розділити дані на групи на основі їх подібності, алгоритми зменшення розмірності можуть зменшити розмірність даних для полегшення їхньої обробки, а асоціативні правила можуть виявляти зв'язки між різними елементами даних.

Навчання без нагляду є важливим інструментом у багатьох галузях, включаючи обробку природних мов, комп'ютерний зір та аналіз даних. Цей тип поділяється на наступні типи:

- кластеризація – це метод МН, що використовується для групування схожих об'єктів разом в один клас або кластер на основі їх схожості між собою і відмінностей від об'єктів інших кластерів. Кластеризація може бути використана в багатьох сферах, таких як аналіз даних, комп'ютерний зір, маркетинг, біологія, інформаційна безпека та інші. У кластеризації об'єкти групуються на основі спільних ознак або характеристик, що дозволяє спрощувати і аналізувати великі обсяги даних. Кластеризація може бути виконана за допомогою різних методів, таких як метод к-середніх, ієрархічна кластеризація, DBSCAN, агломеративна кластеризація та інші;
- візуалізація – це процес вивчення даних та отримання нових знань за допомогою графічних зображень, діаграм, графіків та інших візуальних засобів. Цей тип зберігає оригінальну структуру даних недоторканою і призначає окремі кластери для розрізнення даних, що перекриваються, тим самим допомагаючи ідентифікувати невиявлені закономірності. Як правило, результат реалізується у вигляді 2D або 3D графіків [20];
- виявлення асоціативного правила – це процес виявлення статистично значимих зв'язків між різними елементами в наборі даних. Асоціативні правила використовуються для виявлення залежностей між різними елементами або подіями в даних, таких як покупки в магазині, кліки на веб-сайті або користування програмами. Вони допомагають зрозуміти, які елементи зазвичай супроводжуються іншими, і можуть бути використані для прийняття рішень щодо маркетингових стратегій, оптимізації процесів або покращення користувацького досвіду. Алгоритми виявлення асоціативних правил, такі як Apriori, використовуються в широкому спектрі застосувань штучного інтелекту та машинного навчання.

2.2.3 Напівконтрольоване навчання

Напівконтрольоване навчання (semi-supervised learning) – це тип МН, в якому модель навчається на даних, які містять як позначені (мітки класів), так і непозначені приклади. У порівнянні з повністю контрольованим навчанням, де весь навчальний набір має мітки класів, напівконтрольоване навчання використовує значно менше позначених даних.

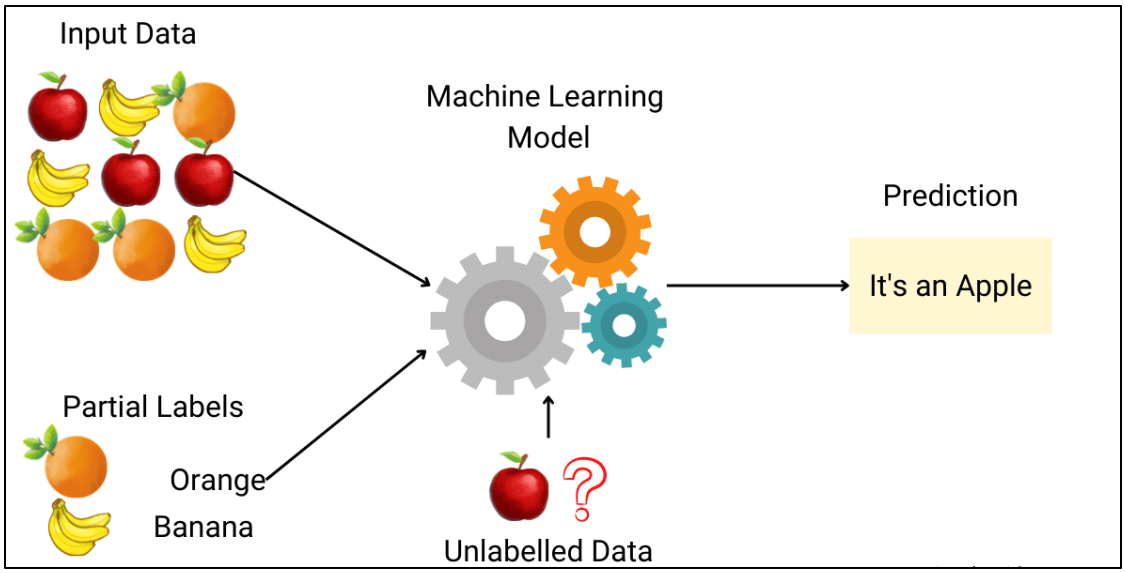

Рисунок 2.3 – Напівконтрольоване навчання

Напівконтрольоване навчання є корисним там, де маємо обмежену кількість позначених даних, але нам все ж потрібно створити модель, що буде здатна здійснювати передбачення на нових, непозначених даних. Це можливо завдяки використанню непозначених даних для побудови загальної структури даних та зменшення ризику перенавчання на занадто обмеженому наборі позначених даних.

Напівконтрольоване навчання застосовують у багатьох областях, зокрема, в обробці природних мов, комп'ютерному зорі, розпізнаванні голосу та інших.

2.2.4 Навчання з підкріпленням

Навчання з підкріпленням (reinforcement learning) – це один з підходів до МН, в якому агент навчається приймати рішення, спираючись на результати попередніх дій, які він виконував у певному середовищі. Агент отримує

інформацію про стан середовища та можливі дії, які він може здійснювати, і має на меті максимізувати нагороду (reward) за послідовність своїх дій та запобігти штрафу (penalty). Зазвичай нагороди визначаються заздалегідь і залежать від того, наскільки успішними були дії агента (рис.2.4). Наприклад, якщо агент грає в гру, то за правильні рухи йому нараховується певна кількість балів, а за неправильні – знімається деяка кількість балів.

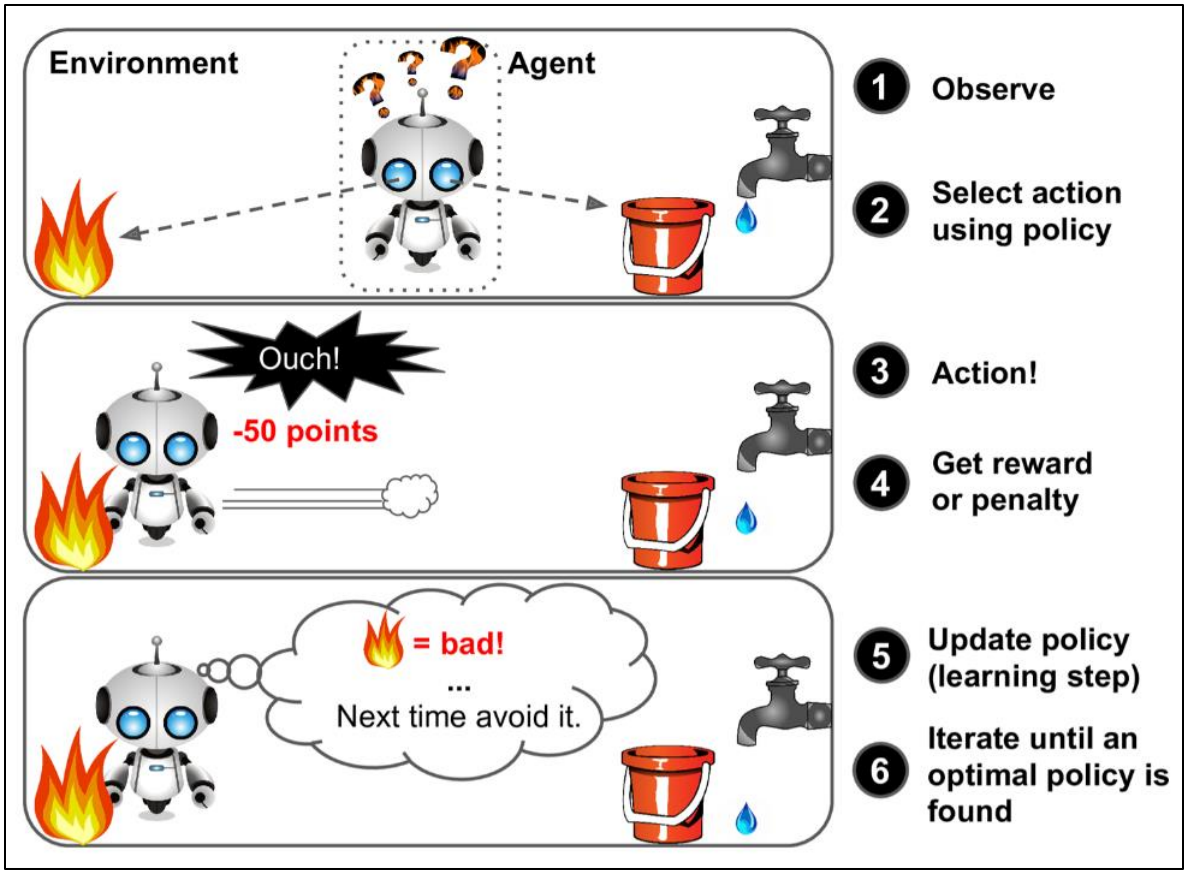

Рисунок 2.4 – Навчання з підкріпленням

Завданням алгоритму навчання з підкріпленням є пошук такої послідовності дій, яка максимізує очікувану нагороду в майбутньому. Для цього агент повинен проводити експерименти, взаємодіючи з середовищем та оцінюючи ефективність своїх дій на основі отриманих нагород.

Ключовими складовими навчання з підкріпленням є агент, середовище та функція нагороди. Агент приймає дії в середовищі, отримує нагороду за кожну взаємодію з середовищем і намагається максимізувати нагороду, вибираючи найбільш оптимальну дію на основі свого досвіду. Функція винагороди

визначає, які дії агента є корисними, а які – не корисні для досягнення мети навчання.

2.3 Алгоритми машинного навчання

Для розробки моделей машинного навчання використовуються різні алгоритми, зокрема:

Алгоритми класифікації: дерева рішень, метод найближчих сусідів, наївний Байєс, логістична регресія, метод опорних векторів тощо;

Алгоритми кластеризації: k-середніх, спектральна кластеризація, ієрархічна кластеризація тощо;

Алгоритми зменшення розмірності: головні компоненти, факторний аналіз, t-розподілена стохастична вкладеність тощо;

Алгоритми регресії: лінійна регресія, регуляризація, дерева рішень тощо;

Нейронні мережі: звичайні персептрони, звичайні глибокі нейронні мережі, згорткові нейронні мережі, рекурентні нейронні мережі, автокодувальні нейронні мережі тощо;

Алгоритми з використанням зв'язних марківських моделей: Hidden Markov Models, Conditional Random Fields тощо.

Це лише деякі з алгоритмів, які використовуються для розробки моделей машинного навчання. Вибір конкретного алгоритму залежить від типу задачі та характеристик даних. За результатами проведеного аналізу наукової літератури, присвяченої проблемі використання методів ШІ для тестування ПЗ [4-17], було прийнято рішення щодо використання нейронних мереж.

2.3.1 Нейронні мережі

Нейронні мережі (Neural Networks) – це математичні моделі, які підтримують обчислення, що наслідують принципи функціонування людського мозку. Вони складаються з взаємозалежних нейронів, які працюють разом, щоб розв'язувати завдання. Нейронні мережі використовуються в багатьох областях, включаючи комп'ютерне зорове сприйняття, машинне навчання, обробку мови

та інші. Вони можуть бути навчені на великих наборах даних, що дозволяє здійснювати розпізнавання образів, класифікацію, прогнозування та інші завдання, що потребують високої обчислювальної складності. Нейронні мережі вважаються однією з основних технологій штучного інтелекту [5].

Розглянемо модель нейрона, що лежить в основі НМ (рис.2.5).

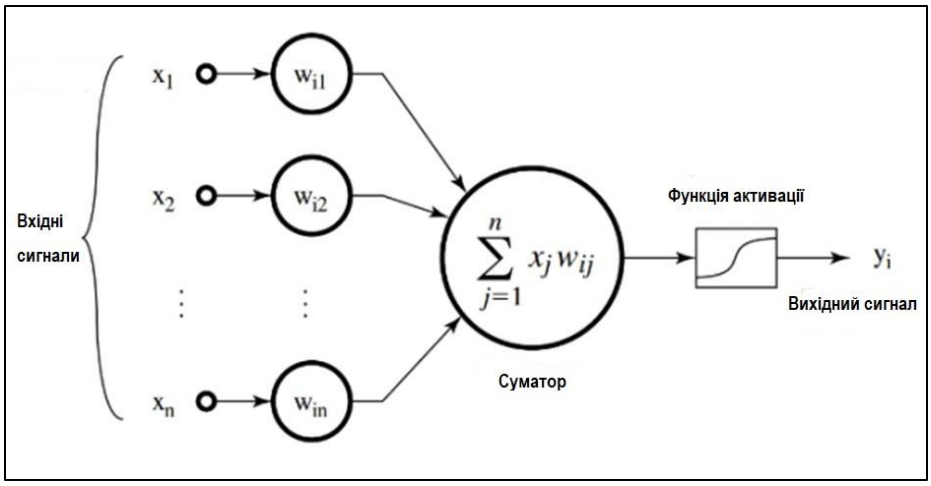

Рисунок 2.5 – Модель штучного нейрону

Штучні НМ являють собою пов’язану між собою систему взаємодіючих штучних нейронів, що взаємодіють один з одним за допомогою вхідних сигналів, які вони періодично отримують, і сигналами, якими вони періодично обмінюється з іншими нейронами. Тим самим штучний нейрон імітує в першому наближенні властивості біологічного нейрона: на вхід штучного нейрона надходить деяка множина сигналів, кожен з яких є виходом іншого нейрона. Кожен з входів множиться на відповідну вагу, яка є аналогом синаптичної силі реального нейрону, а результати визначають рівень активації нейрону [11].

Математичну модель функціонування нейрону можна представити за допомого системи рівнянь

$$net = \sum_{j=1}^{n} x_j w_{ij} \qquad (2.1)$$

$$y_i = \frac{1}{1+e^{-net}}$$

де x_1 , x_2 ,..., x_n – вхідні сигнали нейрону i;

w_{i1} , w_{i2} ,...,w_{in} – вагові коефіцієнти нейрону i;

y_i – вихідний сигнал нейрону i.

НМ можна розглядати як особливий спосіб завдання функції переходу і для її повноцінного використання необхідно навчити мережу.

Навчання НМ передбачає або підготовку навчальної вибірки, яка складається з множин пар вхідних та відповідних їм заздалегідь визначених вірних вихідних векторів, або використання алгоритму самонавчання НМ. Процес навчання НМ полягає в такий модифікації ваги нейронів для підбору оптимальних значень, яке допоможе зменшити функцію втрат та покращити якість передбачень моделі. Зазвичай процес навчання складається з декількох епох, кожна з яких включає кроки прямого та зворотного поширення помилки.

Процес прямого поширення полягає в тому, що дані вхідного шару нейронної мережі передаються вперед через приховані шари до вихідного шару, в результаті чого генерується передбачення. Після цього виконується розрахунок функції втрат (наприклад, середньої квадратичної помилки) для порівняння передбачень з правильними відповідями.

Процес зворотного поширення помилки полягає в тому, що втрата передається в зворотному напрямку від вихідного шару до вхідного шару, під час чого ваги нейронів модифікуються згідно до величини помилки, яку було виявлено на попередньому етапі. Цей процес повторюється для кожної з епох, доки не буде досягнуто визначеного рівня точності (рис.2.6).

Наступним етапом навчання НМ є перевірка результатів навчання.

Перевірка результатів навчання нейронної мережі – це процес, за допомогою якого оцінюються ефективність навчання нейронної мережі на тестових даних. Це важливий крок у процесі розробки нейронної мережі, оскільки він дозволяє визначити, наскільки точно мережа працює на нових даних, які вона раніше не бачила.

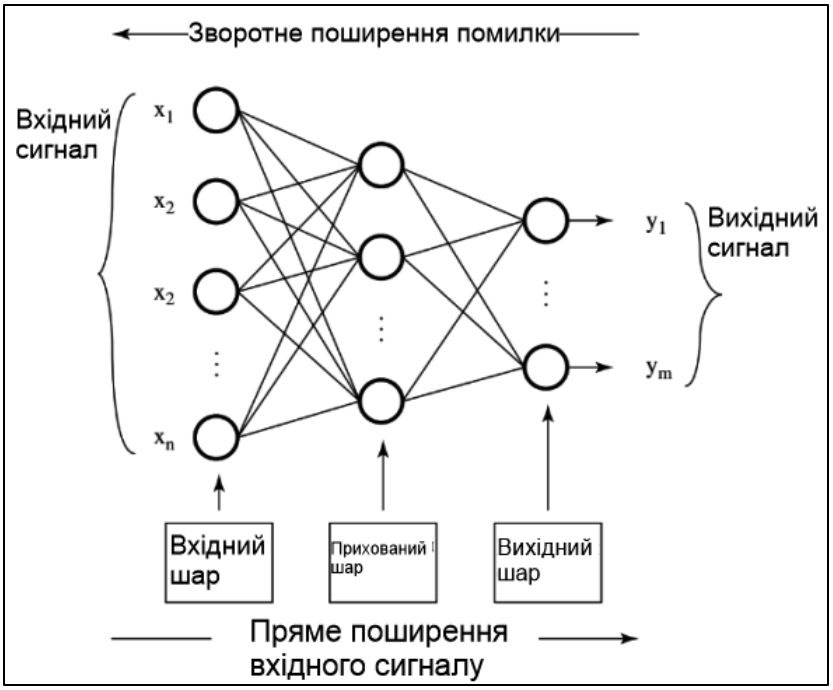

Рисунок 2.6 – Пряме та зворотне поширення помилки в НМ

Після того, як мережа навчена на тренувальному наборі даних, вона тестується на тестовому наборі даних. Ці дані повинні бути представленими у тому ж форматі, що і дані тренувального набору, але мережа не повинна бути навчена на цих даних. Тестовий набір даних використовується для оцінки точності прогнозів, зроблених мережею на нових даних.

Оцінка точності прогнозів може бути здійснена за допомогою різних метрик, таких як середня абсолютна помилка, середня квадратична помилка, коефіцієнт детермінації, точність, відновлення тощо. Ці метрики дозволяють порівняти прогнози, зроблені нейронною мережею, зі справжніми значеннями даних і оцінити ефективність навчання мережі.

Якщо точність прогнозів не відповідає вимогам, то можна використовувати різні методи для покращення результатів навчання, такі як зміна параметрів нейронної мережі, використання інших алгоритмів навчання, збільшення кількості тренувальних даних, видалення шуму тощо. Перевірка результатів навчання нейронної мережі – це ітеративний процес, що дозволяє покращувати точність прогнозів з кожним новим циклом тестування.

2.3.2 Застосування згорткових нейронних мереж в тестуванні ПЗ

Згорткові нейронні мережі (Convolutional Neural Networks) – це клас нейронних мереж, які зазвичай використовуються для обробки зображень, відео та аудіо. Вони використовують згортку для пошуку локальних особливостей у вхідному сигналі та підсумовують результати, щоб отримати більш складні особливості. Згорткові нейронні мережі є основою для багатьох застосувань штучного інтелекту, включаючи візуальне розпізнавання, обробку мовлення та рекомендації.

Основна структура ЗНМ складається зі вхідного шару, згорткових шарів, шарів підсумовування (pooling), повнозв'язного шару та вихідного шару. Згорткові шари використовують фільтри (ядра), які скользять по вхідному зображенню та виконують згортку (кореляцію) для виявлення локальних особливостей. Шари підсумовування використовуються для зменшення розмірності зображення та зниження обчислювальної складності. Повнозв'язний шар виконує класифікацію об'єктів на основі отриманих ознак, а вихідний шар визначає кількість класів та виводить результат.

ЗНМ зазвичай тренуються за допомогою методу зворотного поширення помилок (backpropagation), де ваги мережі оновлюються на основі різниці між вихідним значенням та очікуваним значенням. Ваги оновлюються так, щоб зменшити помилку мережі та підвищити її точність.

ЗНМ завжди були основою глибинного навчання. На теперішній час глибинні ЗНМ вважаються одним з найбільш ефективним інструментом для їх застосування в задачах класифікації зображень. На рисунку 2.7 наведено варіант архітектури ЗНМ, призначеної для розв’язання задачі класифікації графічного зображення десяткової цифри, для чого використовується 10 класів, належність зображення до одного з яких і перевіряє ЗНМ.

В тестуванні ПЗ ЗНМ буде проходити навчання на попередніх результатах тестування ПЗ, для цього будемо використовувати матриці результатів тестування збірок ПЗ для проекту, який розміщено на GitHub та який містить інформацію про різні параметри ПЗ та його функціональність.

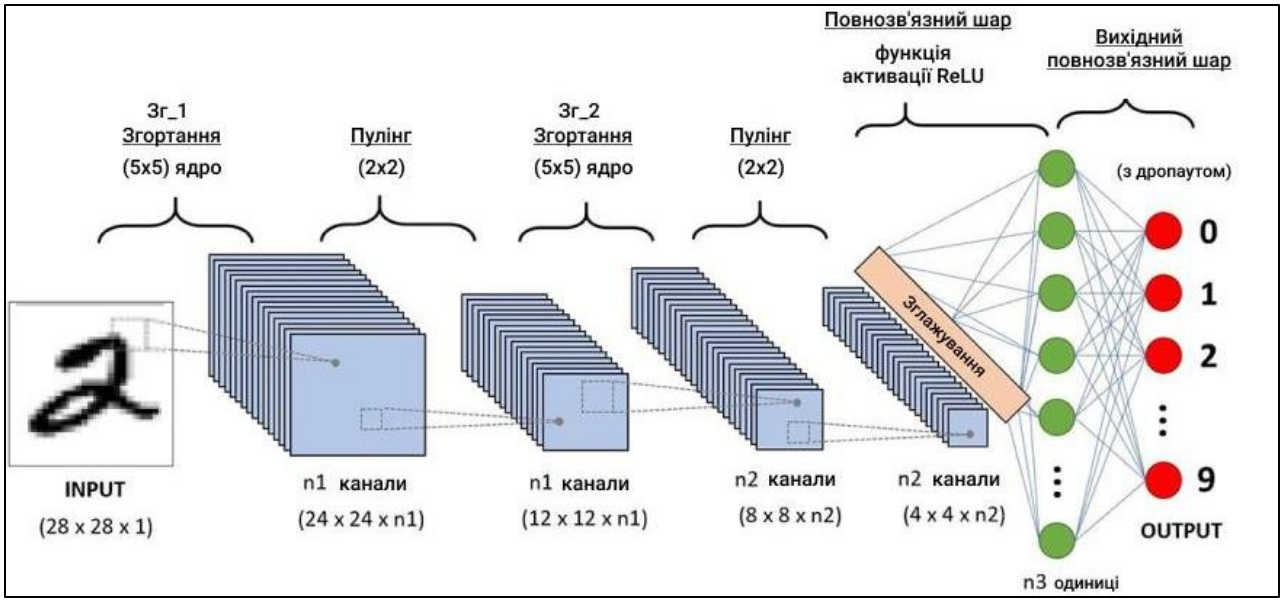

Рисунок 2.7 – Приклад згорткової нейронної мережі

Після навчання ЗНМ можна буде використовувати для автоматичної оцінки результатів тестування та прогнозування майбутніх результатів тестування. ЗНМ може виділяти найбільш важливі параметри ПЗ і відслідковувати їх зміни під час тестування.

Застосування такого підходу спроможне допомогти покращити ефективність тестування ПЗ та зменшити час тестування. Однак, для використання ЗНМ необхідно мати достатню кількість даних для навчання, тому для проведення досліджень будемо використовувати проект, розміщений на GitHub. Також було обрано бібліотеки Tensorflow та Keras для створення ЗНМ.

TensorFlow – бібліотека алгоритмів машинного навчання, яка використовується для систем МН у багатьох сферах, в тому числі для аналізу явищ, розпізнавання голосу, аналіз географічної інформації, розпізнавання образів та зображень, резюмування тексту, інформаційний пошук тощо [12]. В рамках побудови ПЗ для дослідження методів ШІ в тестуванні ПЗ вся архітектура ЗНМ (що складається з декількох шарів) використовує TensorFlow в якості бекенду, а також – для зміни форми даних при їх обробці.

Для підвищення швидкодії створення систем машинного навчання, що будуть навчати ЗНМ, будемо використовувати Keras, який дозволяє використовувати всі переваги масштабованості і крос-платформних можливостей TensorFlow. Основними структурами даних Keras є шари і моделі [13]. Всі шари, які присутні в моделі ЗНМ, реалізовані з використанням Keras. За рахунок перетворення вектору класів в бінарну матрицю класів при обробці даних він суттєво спрощує загальну модель системи.

3 ПІДГОТОВКА ТА ПРОВЕДЕННЯ ДОСЛІДЖЕНЬ

3.1 Використання Git

Моделі машинного навчання потребують великих обсягів даних для самонавчання. Для цього ми будемо використовувати результати тестування за всю історію певного програмного проекту. У цьому розділі детально описано основи управління версіями Git, як він дозволяє отримати доступ до необхідних нам даних про результати тестування, а також чому процес збору даних покладається на функціональність Git.

У системі контролю версій Git коміт (commit) – це дія, за допомогою якої зміни в файлі або наборі файлів фіксуються як нова версія. В GitHub, який є хостинговою платформою для Git-репозиторіїв, коміт є фіксацією змін у файлі або наборі файлів, зроблених користувачем з повідомленням про опис змін. Кожен коміт має унікальний ідентифікатор, що дозволяє відслідковувати історію змін у репозиторії. Коміти є основою роботи з Git та GitHub, тому їх використовують для відстеження роботи розробників та підтримки цілісності кодової бази.

На рисунку 3.1 наведено базову структуру комітів у Git-і. У цьому прикладі показано 3 коміти, позначені 3 хешами над блоками. Кожен коміт вказує на коміт, що був розроблений безпосередньо перед ним, тому 98ca9 є найстарішим комітом, а f30ab – найновішим. Кожен коміт містить декілька атрибутів, таких як дерево, автор, комітер та опис. Дерево вказує на хеш вмісту каталогу, в якому знаходиться коміт, автор і комітер вказують на ідентифікатори розробників, які написали зміни і зробили коміт, а опис – це опис змін, зроблених у коміті, вказаний комітером.

Git використовує деревоподібну модель для відстеження всіх комітів проекту, що дозволяє декільком комітам використовувати один і той самий попередній коміт. Коли два коміти розходяться від одного коміту, вони розглядаються на окремих «гілках». У стандартній практиці розробки програмного забезпечення існує основна головна гілка, яка представляє

«найреальнішу» версію кодової бази – тобто ту версію, яка є загальнодоступною для користувачів.

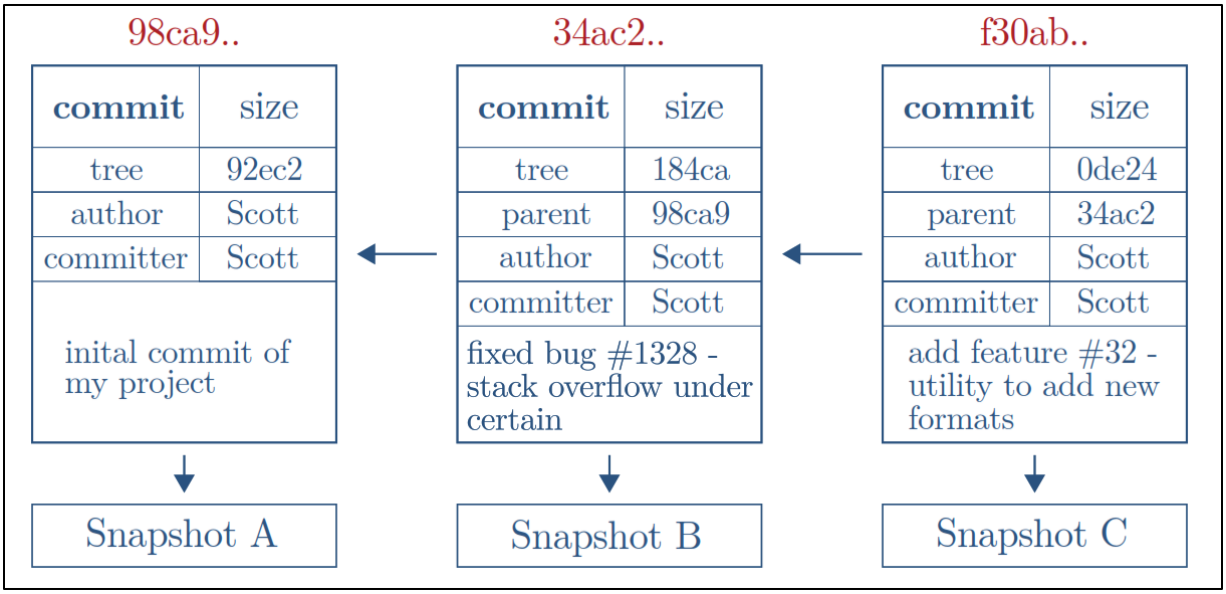

Рисунок 3.1 – приклад базової структури коммітів у Git

Щоб вносити зміни, розробники створюють відгалуження від цієї основної гілки. Коли вони завершили роботу над своєю гілкою, вони подають «pull request», щоб їх код було об'єднано з основною майстер-гілкою. Інші розробники можуть схвалити або відхилити ці запити, і в разі схвалення зміни будуть додані до головної гілки. Таким чином, стандартними гілками Git-проекту є головна гілка і всі гілки запитів на інтергацію, які були або не були об'єднані в майстер-гілку. Приклад такої структури наведено на рисунку 3.2.

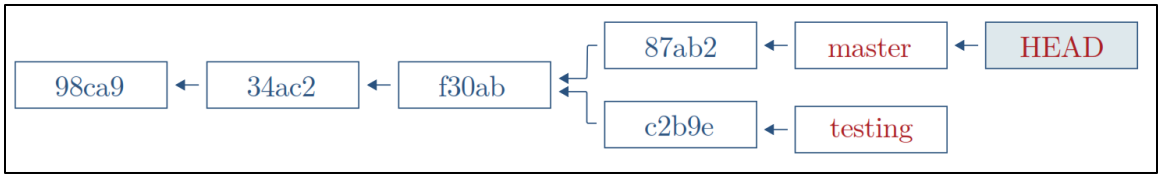

Рисунок 3.2 – Приклад базової структури коммітів у Git

Структура містить дві гілки: master та testing. Вони розходяться від коміту f30ab, причому master має коміт 87ab2, а testing має коміт c2b9e. На рисунку показано не лише хеш-ідентифікатори для кожного коміту, але й три додаткові ідентифікатори: HEAD, master та testing. Master і testing є ідентифікаторами для

двох гілок і вказують на останній коміт у кожній гілці. HEAD – це вказівник на коміт, який користувач в даний момент переглядає локально, тому на малюнку користувач переглядає останній коміт на майстер-гілці. Всі ці вказівники динамічно оновлюються при створенні нових комітів. Вони діють як альтернативний спосіб посилання на коміти поза хеш-рядками.

Коміти призначені не лише для запису змін у коді. Кожен коміт також записує «знімок» усього стану кодової бази. Користувачі можуть переглянути будь-який такий знімок, змінивши місце, на яке вказує HEAD. Оскільки HEAD це лише вказівник, його можна оновити, щоб він вказував на будь-який коміт, і таким чином можна переглянути будь-який такий знімок.

Команда для оновлення HEAD – git checkout id, де id – це ідентифікатор коміту. При цьому HEAD встановлюється на ідентифікатор коміту, а користувач повертається до стану кодової бази на момент створення коміту.

Це замінює весь вміст файлів, що зберігаються у проекті, на вміст цих файлів на момент створення коміту, дозволяючи користувачам переглядати і використовувати цю попередню версію коду. Оскільки кожен коміт відстежує попередній коміт, доступна вся минула історія комітів, тому можна переглянути будь-який завершений коміт. Це включає як попередні стани головної гілки, так і проміжні стани запитів pull request, над якими працювали інші розробники. Таким чином, ми маємо доступ до всієї історії розроблюваного програмного забезпечення, розділеної на окремі часові мітки в кожному коміті. Ці дії є оборотними, тому в будь-який момент можна просто повернутися до останнього стану кодової бази.

Як зрозуміло зі структури Git, ми можемо отримати будь-який попередньо зроблений знімок проекту на основі Git. Однак, дані про результати тестування не зберігаються явно при кожному коміті. Замість цього, всі тести можна запустити на цій старій версії – оскільки всі файли були замінені назад та повернуті до їх попереднього стану, то тепер тестується стара версія коду. Аналізуючи журнали, отримані в результаті виконання цих тестів, ми можемо зібрати дані, які нам потрібні для досягнення мети дослідження. Таким чином,

ми маємо алгоритм для збору великої кількості даних про результати тестування:

```
Generate a list of all commits
For each commit:
      Checkout the commit
      Run all tests
      Parse logs and accumulate data
```

Перейдемо до розгляду фактичної реалізації вищенаведеного алгоритму.

3.2 Отримання всіх комітів

Пошук всіх комітів сховища складається з двох компонентів: головної гілки master та усіх бічних гілок pull-request. Важливим є отримання даних про результати тестування з комітів обох компонентів, а не просто збільшити кількість отриманих даних. Оскільки головна гілка є найбільш публічною і відкритою для звичайних користувачів, дуже важливо, щоб коміти в ній були релізної якості і містили мінімальну кількість помилок, якщо вони є. Таким чином, майже всі тести з цих комітів будуть прохідними, які, хоча і будуть містити коректні дані, також будуть містити мінімальну кількість прикладів помилок. На противагу цьому, гілки запитів pull-request можуть мати проміжні коміти, які не є якісними для релізу, але ці коміти з набагато більшою ймовірністю можуть містити приклади з помилками, саме тому ми обов’язково перевіряємо і ці коміти.

Знайти всі коміти вздовж головної майстер-гілки дуже просто. Кожен коміт має вказівник на свого батька, і Git надає простий синтаксис для пошуку батьківських комітів: якщо ідентифікатор коміту – id, то ідентифікатор його батька має вигляд id~{1}, а ідентифікатор його батька – id~{2}, і так далі. Таким чином, ми можемо почати з останнього коміту на головному сховищі і перебирати батьківські коміти, поки не залишиться жодного батьківського коміту.

Отримання всіх комітів за pull-запитом є більш складним завданням. Спочатку ми модифікуємо файл .git/config у кореневому каталозі сховища, щоб витягнути всі гілки, позначені pr. Кожна з цих гілок матиме вигляд origin/pr/id, де id – це лічильник для кожного pull-запиту. Для кожної з цих гілок ми виконаємо наведену нижче команду, щоб знайти всі коміти в цій гілці, але не в головній гілці:

```
git log origin/pr/id --not
     $(git for-each-ref --format=\'%(refname)\' refs/heads/ |
     grep -v "refs/heads/origin/pr/id")
```

У вище наведеній процедурі є суттєва проблема – необхідність повторного запуску всіх тестів при кожному коміті, а цей процес може зайняти дуже багато часу. Для безпосереднього запуску програми потрібно налаштувати проект та його залежності у будь-якому середовищі, яке використовується, що ще більше ускладнюється зміною залежностей при дослідженні попередніх комітів.

Майже всі великі проекти використовують ту чи іншу форму безперервної інтеграції для автоматичного запуску всіх тестів щоразу, коли додається новий код. Ці інструменти CI надають загальнодоступні API, які дозволяють робити запити до історії проекту та «витягувати» різні журнали і метадані. З огляду на доступність цієї інформації, запити до будь-якого інструменту CI, який використовується в проекті для отримання інформації про тести, є ідеальним методом.

Однак, при більш детальному аналізі використання інструментів CI для отримання даних було виявлено критичні недоліки, які витікають з того факту, що їхні API не розроблені з урахуванням нашого конкретного випадку використання – отримання конкретних даних про результати тестування. Натомість, вони розроблені більше з точки зору загальної перспективи розробки, надаючи інформацію про всі процеси, які були налаштовані на запуск при додаванні нового коду.

В якості прикладу можна розглянути TravisCI, один з найпопулярніших інструментів CI, який також є безкоштовним для використання в проектах з відкритим вихідним кодом. Його API орієнтований на запити до збірок, які деталізують додавання нового коду до проекту, що складається з багатьох завдань. Ці завдання включають ведення журналу, який описує, що було зроблено у завданні. Звичайно, деякі з цих завдань пов'язані з тестуванням, і ми могли б використовувати їхні логи для вилучення даних, які нас цікавлять. На жаль, важко відрізнити, які з них пов'язані з тестуванням, а які ні, без ручного перегляду журналів, а оскільки ці API обмежені за швидкістю, спроба проаналізувати всі журнали є досить повільною.

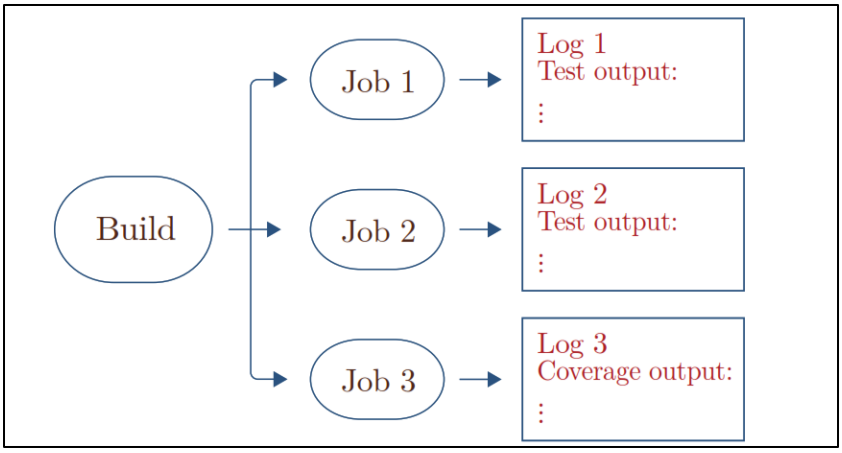

Рисунок 3.3 – Схема об'єктів, доступних через Travis CI API

TravisCI – це інструмент для автоматичної перевірки коду на GitHub. Він дозволяє автоматично запускати тестові скрипти при кожному коміті в репозиторії. Таким чином, ви можете бути впевнені, що ваш код проходить тестування на кожному етапі розробки.

Для аналізу результатів тестування за допомогою TravisCI потрібно налаштувати файл .travis.yml в кореневій директорії репозиторію на GitHub. У цьому файлі вказується, які команди повинні бути виконані для тестування вашого коду. TravisCI автоматично виконує ці команди і повідомляє про результати тестування.

Після налаштування файлу .travis.yml, ви можете додати значок стану TravisCI до свого репозиторію на GitHub. Це дозволить іншим користувачам

переглядати статус тестування вашого коду в реальному часі, чим ми й скористуємося при проведенні досліджень.

3.3 Формування матриць результатів тестування

Після того, як TravisCI успішно виконає тести, результати тестування будуть доступні через репозиторій у файлі results.txt. Для формування матриці результатів необхідно зчитати цей файл і обробити його відповідним чином. Обробку будемо виконувати з використанням бібліотеки csv-writer, яка дозволяє легко створювати та записувати дані у форматі CSV. Для формування матриці результатів можна дані з файлу results.txt записати у відповідні комірки таблиці CSV.

Наведемо код додатку на JavaScript для експорту результатів тестування.

```
const fs = require('fs');
const createCsvWriter = require('csv-
writer').createObjectCsvWriter;

// Зчитування результатів тестування з файлу results.txt
const results = fs.readFileSync('results.txt', 'utf-
8').split('\n');

// Підготовка даних для формування таблиці
const data = results.map((result) => {
 const [name, status] = result.split(':');
 return { name, status };
});

// Налаштування заголовків таблиці
const csvWriter = createCsvWriter({
 path: 'results.csv',
 header: [
 { id: 'name', title: 'Test Name' },
 { id: 'status', title: 'Status' },
 ],
});

// Запис даних до файлу
csvWriter.writeRecords(data).then(() => {
 console.log('Results saved to results.csv');
});
```

Для того, щоб отримати матрицю результатів за усіма комітами, код буде мати наступний вигляд:

```
const axios = require('axios');

// отримуємо список останніх комітів з GitHub
async function getCommits() {
 try {
 const response = await
axios.get('https://api.github.com/repos/owner/repo/commits');
 return response.data;
 } catch (error) {
 console.error(error);
 }
}

// отримуємо список файлів з останнього коміту
async function getFiles(commitSha) {
 try {
 const response = await
axios.get(`https://api.github.com/repos/owner/repo/commits/${commitSha}`
);
 return response.data.files;
 } catch (error) {
 console.error(error);
 }
}

// перевіряємо кожен файл на наявність тестів і повертаємо матрицю
результатів
async function checkFiles() {
 const commits = await getCommits();
 const latestCommitSha = commits[0].sha;
 const files = await getFiles(latestCommitSha);

 const testMatrix = [];

 files.forEach((file) => {
 if (file.filename.includes('test')) {
 testMatrix.push({
 file: file.filename,
 result: file.status === 'added' ? 'pass' : 'fail'
 });
 }
 });

 console.log(testMatrix);
}

checkFiles();
```

У цьому коді ми отримуємо список останніх комітів з репозиторію на GitHub, отримуємо список файлів з останнього коміту та перевіряємо кожен файл на наявність тестів. Якщо файл містить слово "test" в назві і був доданий до останнього коміту, ми позначаємо його як "pass", інакше – як "fail". Матриця

результатів тестування повертається у вигляді масиву об'єктів з властивостями "file" та "result".

Слід зазначити, що такий підхід до формування матриці результатів є мультиплатформним та не залежить від вибору мови програмування, яку було використано в проекті.

3.4 Практичне використання ЗНМ

Отримавши результати тестування у вигляді матриці результатів використаємо ці результати для первинного навчання та аналізу результатів тестування за допомогою ЗНМ. Для цього будемо використовувати TensorFlow.js, безкоштовну бібліотеку для використання ЗНМ у JavaScript-проектах, та Keras.js, безкоштовну бібліотека для машинного навчання, яка пропонує інтерфейс Keras для ЗНМ [12-13]. Оскільки значення у матриці результатів є категоріальними («PASS» або «FAIL»), то будемо використовувати функцію активації softmax.

В якості експериментальної платформи для проведення досліджень ефективності використання ЗНМ для тестування ПЗ було обрано проект Flask. Flask – це популярний веб-фреймворк на Python з відкритим вихідним кодом, який розміщений на GitHub та має тисячі комітів і сотні учасників. Проект було обрано для тестування з декількох причин:

- він має документацію, зокрема щодо налаштування та запуску тестів;
- він має достатню кількість тестів (близько 400), а також велику кількість комітів (понад 3000), що забезпечує належну кількість даних для навчання ЗНМ.

3.5 Метрики тестових даних

Статистика щодо загальної кількості результатів тестів, які були отримані через TravisCI, наведено у таблиці 3.1. Слід зазначити, що Flask підтримує не всі версії Python, а лише починаючі з Python 2.6.

Таблиця 3.1 – Кількість тестів

Версія Python	Кількість коммітів з тестами	Загальна кількість результатів тестів
2.6	2177	299317
2.7	1930	298781
3.3	1200	297316
3.4	1200	297316
3.5	1200	297316
3.6	1200	297316

З таблиці видно, що для версій 3.3–3.6 загальна кількість комітів з тестами співпадають, так саме, як і сама загальна кількість тестів – причиною слід вважати той факт, що підтримку версій 3.3–3.6 було додано одночасно. Однак за результатами звіту щодо кількості збоїв під час виконання тестів, який наведено у таблиці 3.2, видно, що у більшості версій коміти не мають збоїв. Також було виявлено, що загалом 825 комітів мали щонайменше один збій, і 1534 запуски тестового набору мали щонайменше один збій, де під запуском тестового набору розуміють лише одне виконання тестового набору на одній версії.

Таблиця 3.2 – Статистика відмов

Версія Python	Кількість відмов				
	0	1	2-10	11-100	>100
2.6	2099	30	29	7	5
2.7	1851	31	29	7	5
3.3	857	283	34	12	5
3.4	1040	40	98	8	5
3.5	1110	42	26	8	5
3.6	366	680	138	5	2

На підставі отриманих результатів та аналізу таблиць 3.1 і 3.2 було складено таблицю 3.3, в якій наведено розбивку частоти збоїв у відсотках.

Таблиця 3.3 – Відсоток тестів, що не пройшли, та комітів з помилками

Версія Python	Загальна кількість невдач	Відсоток тестів, які провалились	Відсоток комітів з помилками
2.6	1352	.0045%	.033%
2.7	1352	.0045%	.037%
3.3	1705	.0057%	.278%
3.4	1733	.0062%	.126%
3.5	1368	.0046%	.068%
3.6	1682	.0057%	.688%

Розглянемо особливості організації прогнозування результатів тестування.

3.6 Проблема прогнозування та можливі рішення

Проблему прогнозування результатів тестування в конкретному сенсі можна визначити наступним чином:

Дано коміти $C_{1...m}$ та матриці результатів тестування для кожного з них $T_{1...m}$, тоді існує можливість передбачити результати тесту T_{m+1} для C_{m+1}, де i, j – запис у матриці T, який містить результати для тесту i при його виконанні на платформі j. При побудові системи не використовується аналіз вихідного коду з коміту C_{m+1} або попередніх комітів.

На перший погляд, можна припустити, що, маючи новий коміт і попередню історію результатів тестів, ми можемо передбачити, які тести не пройшли з урахуванням попередньої історії. Однак у цьому випадку ми не

маємо жодної інформації про поточний коміт, оскільки не використовується поточний аналіз вихідного коду.

Тому використаємо підхід, який можна розглядати як задачу заповнення матриці. Задача полягає у наступному: для кожного нового коміту C_{m+1} обираємо деяку підмножину тестів, для яких спочатку маємо результати. На основі результатів цієї підмножини тестів робимо прогноз щодо результатів решти тестів. Таким чином під час формування матриці спочатку заповнимо частину матриці тестів їх реальними значеннями, потім, використовуючи нашу модель для створення прогнозів – для значень решти елементів матриці. Принцип полягає в тому, що існують тести, які мають спільні атрибути один з одним, тобто, якщо один тест пройде або не пройде, то це дає інформацію про те, чи пройдуть або не пройдуть інші тести з більшою ймовірністю. Таким чином, за результатами одного тесту ми отримуємо інформацію, на основі якої можемо передбачити результати інших тестів. Тоді задача заповнення матриці може бути розв'язана двома способами: одноразовий підхід та ітеративний [19].

При одноразовому підході після заповнення початкової частини матриці ми робимо прогнози щодо кожного із усіх інших елементів матриці (рис.3.6).

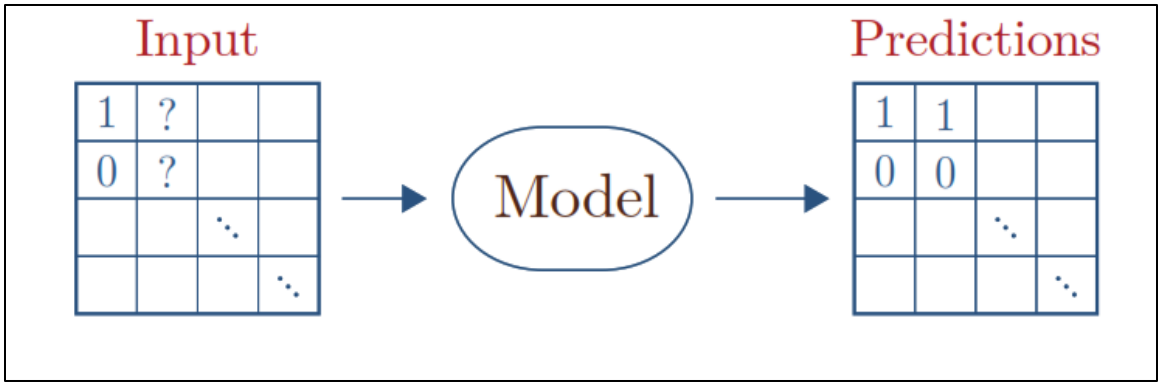

Рисунок 3.4 – Одноразовий підхід

При ітеративному підході спочатку заповнюється невеличка початкова частина матриці, а потім робиться прогноз. З цих прогнозів обираються лише ті, що мають найвищий рівень достовірності, після чого знаходимо реальні значення та порівнюємо з прогнозованими. Потім вносимо ці реальні значення в матрицю і повторюємо до вичерпання всіх прогнозів (рис.3.5).

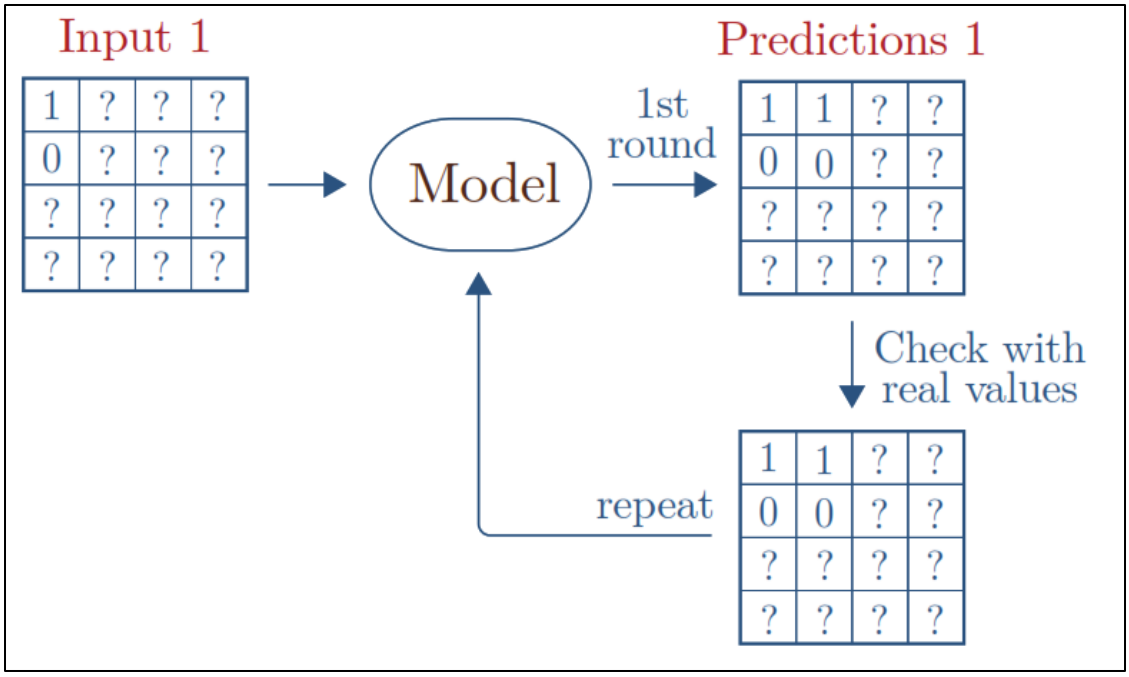

Рисунок 3.4 – Ітеративний підхід

Незалежно від того, який підхід використовується, обидва мають одну мету – передбачити незаповнені елементи частково заповненої матриці. Звернемо увагу на формат цієї матриці.

Матриці результатів тестування мають один рядок для кожного унікального тесту і один стовпець для кожної платформи, на якій він тестується. Кожен елемент матриці є результатом тесту, зазначеного в рядку, на платформі, зазначеній у стовпчику. Якщо тест пройшов на відповідній платформі, то значення елементу матриці дорівнює 0, а якщо не пройшов – 1. Такі значення були обрані тому, що метою є виявлення збоїв, тому розглядаємо результат проходження тесту як стан за замовчуванням, а відмови – як виключні події. Будь-який інший результат (нерозпізнаний стан або помилка тестування) буде мати невизначене значення NaN.

Значення NaN може зустрічатися лише у 3 випадках:

– тест має результат SKIPPED;
– тест має результат ERRORED;
– тест не існував у цьому коміті.

Таким чином, за допомогою описаної вище процедури існує можливість згенерувати матрицю для кожного коміту. Під час використання цих матриць в моделі прогнозування, ми повинні враховувати, які матриці вважаються «попередніми» для даної матриці, щоб ми могли використовувати ці «попередні» матриці для навчання ЗНМ. Для цього необхідно організувати впорядкування комітів на основі мітки часу, коли вони були зроблені. Потім ми можемо відсортувати увесь список матриць відповідно до цього порядку комітів таким чином, щоб матриці, що відповідають комітам з більш ранніми мітками, з'являлися перед комітами з більш пізніми мітками.

Задача сортування може бути розв’язана за допомогою колаборативної фільтрації. Колаборативна фільтрація – це метод рекомендаційного аналізу, який використовується для передбачення відносної оцінки або відгуку користувачів на продукт на основі даних про раніше задоволення продуктом та поведінку користувачів. У контексті тестування ПЗ колаборативна фільтрація може бути використана для передбачення, які тести мають бути запущені спочатку або які частини програми мають бути перевірені з більш високим пріоритетом.

Колаборативна фільтрація матриці результатів тестування ПЗ включає аналіз інформації про коміти та результати тестування. Результатом такого аналізу може бути матриця, яка містить передбачені оцінки тестів для кожного коміту. Для програмної реалізації колаборативної фільтрації може бути використана бібліотека TensorFlow, до складу якої входить SVDFeature.

SVDFeature – це бібліотека машинного навчання, яка використовується для рекомендаційних систем, зокрема для колаборативної фільтрації матриць. Вона забезпечує підтримку методу Singular Value Decomposition (SVD) для зменшення розмірності даних та поліпшення якості рекомендацій. SVDFeature також може бути використана для роботи зі звичайними табличними даними.

Однією з основних переваг SVDFeature є можливість використання різних метрик оцінки якості рекомендацій, що дозволяє знайти оптимальну модель для

конкретного застосування. Бібліотека також підтримує паралельну обробку даних, що дозволяє підвищити швидкість обчислень.

Для використання SVDFeature для розв’язання задачі використання методів ШІ для тестування ПЗ необхідно визначитись, які саме функції будемо використовувати. В контексті задачі прогнозування необхідно, щоб властивості відображали минулу історію матриць до поточного коміту, для якого робимо прогнози. До таких функцій відносяться частка випадків, коли тест не пройшов у всіх попередніх випадках, та частка тестів, які не пройшли на окремій платформі в усіх попередніх наборах тестів.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ

4.1 Формування вибірки для прогнозування

Для проведення досліджень всі матриці даних були поділені на домени відповідно до того, скільки відмов спостерігалось в окремому коміті. Було визначено 5 доменів: 1 відмова, 2 відмови, 3-10 відмов, 11-100 відмов та понад 100 відмов. Розподіл комітів між цими доменами у версії 3.6 наведено в таблиці 4.1.

Таблиця 4.1 – Кількість матриць у кожному домені

Кількість відмов				
1	2	3-10	11-100	>100
418	250	107	37	10

Зауважимо, що коміти, в яких під час тестування не було знайдено дефектів, були відкинуті, оскільки при використанні підходу, що базується на SVD, будь-яка матриця без збоїв (тобто яка містить всі позитивні результати тестування) буде факторизована на елементи, які дорівнюють 0, тому і всі прогнози будуть також дорівнювати 0. Таким чином, нас цікавлять лише матриці з відмовами. Також були відкинуті матриці, що містили нерозпізнані результати тестування (NaN).

4.2 Проведення досліджень

Після проведення формування матриць та проведення навчання ЗНМ було проведено прогнозування можливого передбачення результатів тестових випадків. Для проведення досліджень було обрано ітеративний підхід. Цей підхід дозволяє ітеративно робити прогнози щодо елементів матриці за допомогою SVDFeature. В рамках реалізації програмної системи було розроблено відповідний API, який дозволяє користувачеві легко працювати з SVDFeature, поступово заповнюючи матрицю тестів і вирішуючи, які тести запускати.

Робота з API передбачає використання обгортки, яку було розроблено навколо SVDFeature. Ця обгортка надає стандартизований список функцій, які очікує API, до яких відносяться наступні методи:

– initialize: отримує список минулих тестових матриць і матрицю для заповнення та виконує ініціалізацію моделі з цими параметрами;
– update: бере оновлену версію поточної тестової матриці та виконує оновлення моделі цією новою матрицею;
– predict: приймає конфігурацію, що визначає параметри моделі та повертає прогноз, зроблений моделлю з цим набором параметрів, де кожний прогноз – це відсоток того, що тест не пройде, і місце в матриці цього тесту, впорядковане таким чином, щоб тести з найбільшою ймовірністю не пройшли першими.

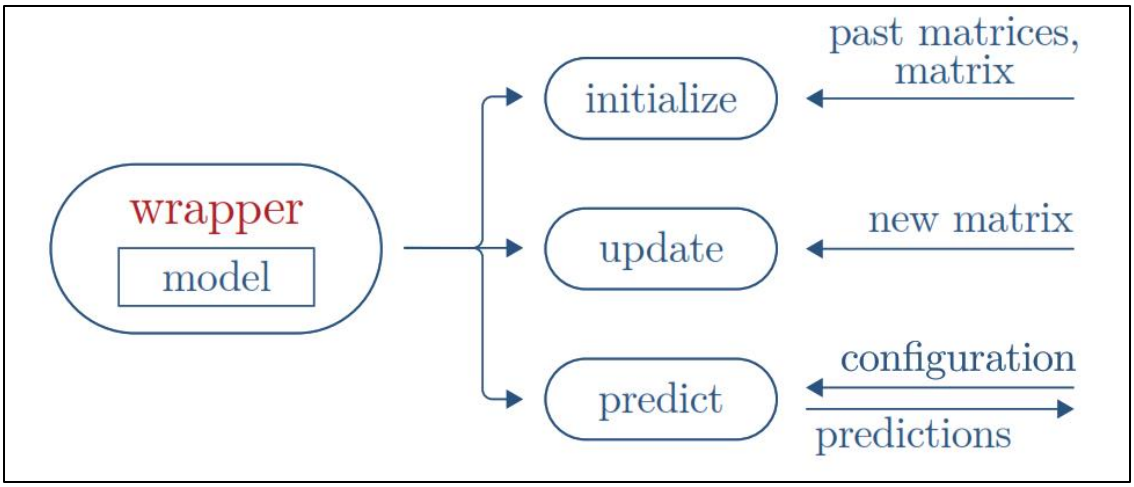

Рисунок 4.1 – Ітеративний підхід

Обгортку, функції якої зображено на рис.4.1, реалізовані за допомогою SVDFeature, серед яких:

– initialize: записує відповідний конфігураційний файл, обчислює всі ознаки з історії матриць тестування та зберігає отриману матрицю як features_matrix;
– update: оновлює features_matrix та виводить нову матрицю;
– predict: викликає SVDFeature для навчання моделі на основі вхідних файлів і генерації прогнозів.

Існує виключна ситуація для predict: якщо матриця features_matrix все ще порожня, то навчена на ній модель SVDFeature не матиме на чому вчитися, а отже, не зможе зробити жодних реальних прогнозів. Тому було прийнято рішення в цьому варіанті повертати тести, які зазнали найбільшої кількості невдач у минулому.

На рис.4.2 наведено алгоритм застосування програмної системи, призначеної для прогнозування тестів, поділений на дві частини: формування моделі та її використання.

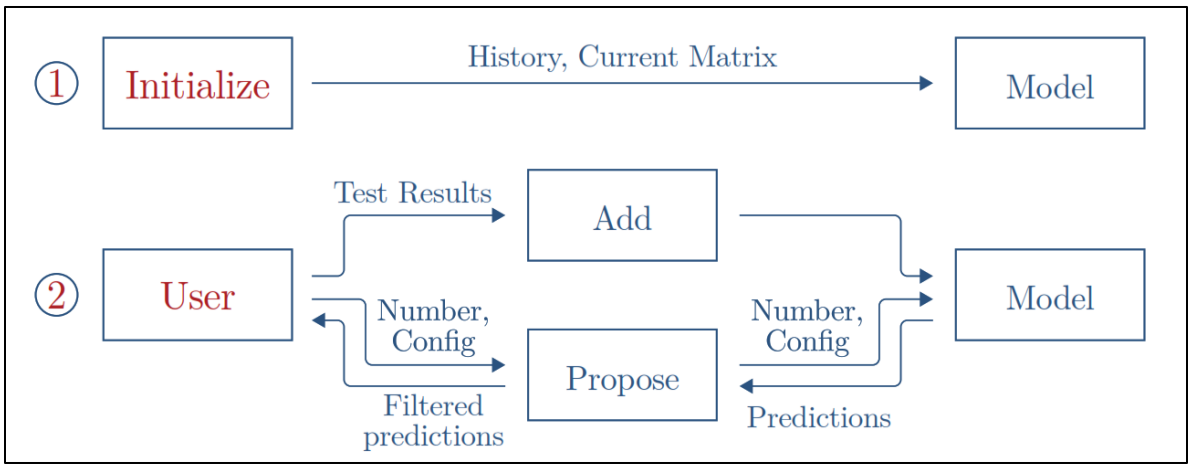

Рисунок 4.2 – Алгоритм застосування

Після отримання результатів перейдемо до їх аналізу з точки зору досягнення мети дослідження.

4.3 Аналіз отриманих результатів

Для оцінювання результатів прогнозування було використано домени, опис яких наведено у 4.1. Для кожного з доменів було обрано по 3 випадкових коміти, для яких було застосовано вищеописаний алгоритм. Алгоритм застосовується на кожній виборці 5 разів, кожного разу роблячи запит розмірністю 60, що дозволило розглянути загалом 300 тестів. Для кожного з комітів, для якого було виконано цю процедуру, було визначено частку всіх відмов та збоїв, які були запропоновані алгоритмом, отримані результати наведені в таблиці 4.2.

Таблиця 4.2 – Статистика прогнозування збоїв

Номер домену	Індекс коміту в історії	Загальна кількість тестів	Кількість відмов	Частка співпадіння прогнозів
1	2623	1738	1	0.0
2	2644	1738	1	0.0
3	2776	1834	1	0.0
4	2206	1422	2	0.5
5	2374	1475	2	0.5
6	2403	1631	2	0.5
7	2308	1475	8	0.86
8	2473	1643	8	0.25
9	3008	2257	10	0.4
10	2171	1394	54	0.94
11	2189	1406	36	0.89
12	2290	1469	14	0.85
13	2287	1457	190	0.89
14	2288	1457	190	0.85
15	2791	1809	126	0.75

Отримані результати показують, що в той час, як модель дуже ненадійно визначає відмови для комітів з 1 або 2 збоями, вона дуже добре справлялася з комітами з більшою кількістю збоїв. Причина того, що модель погано працює на комітах з дуже малою кількістю збоїв полягає у тому, що у неї дуже мало інформації, на основі якої вона може будувати свої прогнози. З іншого боку бачимо, що навіть при дуже низькому пороговому значенні кількості збоїв вдається ідентифікувати значну частину тестів, перевіривши менше за 25% всіх тестів. Таким чином, розглянутий метод використання ЗНМ з предиктивною моделлю навчання є обнадійливим початком для розробки ефективних методів швидкого виявлення збоїв під час реальних процедур тестування.

Аналізуючи отримані результати можна зробити наступні висновки:

– точність прогнозу, що генерує розроблена програмна система, суттєвим

чином залежить від кількості дефектів, що були виявлені під час попередніх спроб тестування та не може надати достовірний результат, якщо зафіксована кількість відмов під час виконання тестів менша за 10 (рис.4.2);

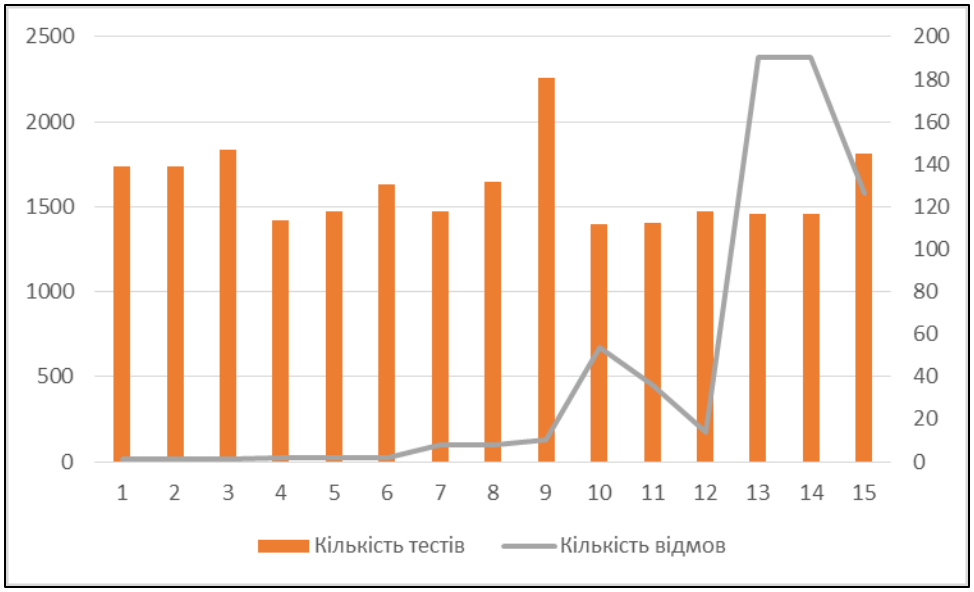

Рисунок 4.2 – Співвідношення між кількістю тестів та кількістю дефектів

– Б) На кількості виборок у 15 комітів, яку було досліджено, можна зробити висновок, що запропонований метод є більш ефективним для виконання процедури ранжування тестів (див. рис.4.3) з метою прискорення отримання результатів тестування, що дозволить ефективно вплинути на процес розробки програмної системи.
– В) Цей підхід може знайти своє застосування щодо визначення та рекомендації користувачам тестів для їх виконання, але потребує подальшого доопрацювання. На теперішній час доцільно використовувати цей підхід в статичних середовищах, де всі результати тестування вже відомі, і користувачі просто перевіряють, чи здатна обрана ними модель успішно робити прогнози щодо використання тестів.

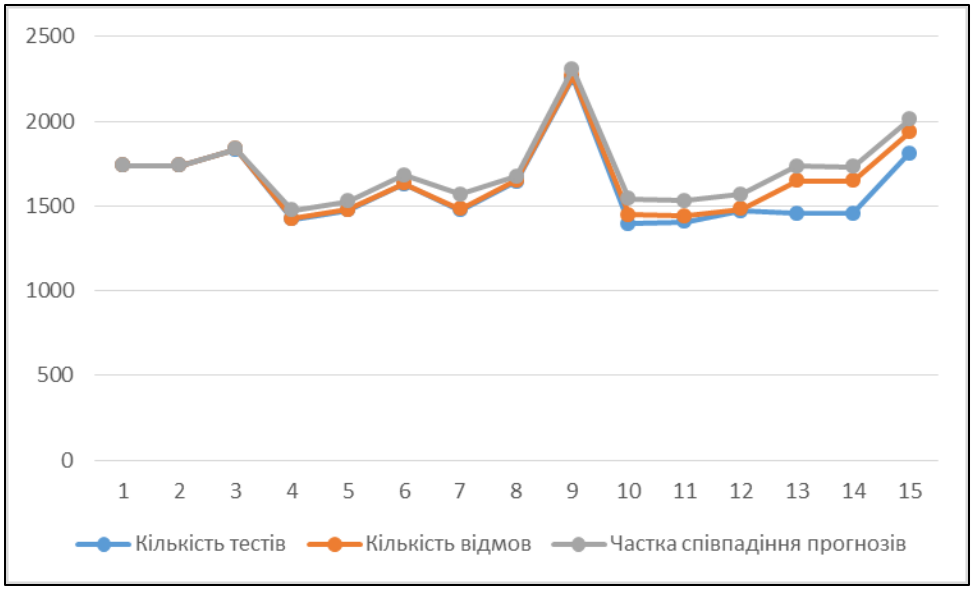

Рисунок 4.3 – Точність прогнозів

Запропонована модель використання методів штучного інтелекту для тестування програмного забезпечення орієнтована на зовнішні сховища, але цей підхід можна з легкістю адаптувати до будь-якої системи. Збір тестів зараз виконується послідовним способом, який, хоча і є простим, але може вимагати багато часу, особливо для великих проектів з відкритим вихідним кодом, для яких, запропонована модель програмного забезпечення може виявитися найбільш корисним. Більш висока продуктивність тестування дозволяє швидше завершувати розробку та впроваджувати нове програмне забезпечення.

ВИСНОВКИ

Під час написання кваліфікаційної роботи магістра були розглянуті актуальні питання вдосконалення підходів до організації тестування ПЗ за рахунок використання методів ШІ. В роботі було досліджено підходи щодо використання нейромережевих технологій для прогнозування результатів тестування програмних проектів з відкритим кодом зроблено висновки щодо доцільності застосування ЗНМ. За темою дослідження було проведено: аналіз першоджерел; виконано дослідження використання методів ШІ для тестування ПЗ; визначено можливі шляхи покращення якості тестування за рахунок використання ЗШМ прогнозування можливих результатів з подальшим ранжуванням тест-кейсів; виконано постановку задачі дослідження та визначено локальні задачі, які були успішно розв’язані під час проведення досліджень; розроблено та досліджено програмну модель системи прогнозування результатів тестування з використанням колаборативної фільтрації, яка входить до бібліотеки TensorFlow.

Слід зазначити, що використану в роботі функцію SVDFeature було використано лише з одним набором ознак, однак існує багато можливостей для подальшого розширення простору ознак, серед яких можна вказати на пошук явних кореляцій між тестами, які могли б відігравати важливу роль у прогнозуванні нових результатів тестів. Подальші дослідження за темою роботи можуть покращити якість моделі прогнозування та надавати більш ефективний результат застосування методів ШІ в тестуванні ПЗ.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Myers, Glenford J. The art of software testing / Glenford J. Myers ;Revised and updated by Tom Badgett and Todd Thomas, with Corey Sandler.—3nd ed.

2. Haenlein, Michael & Kaplan, Andreas. A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. California Management Review. URL: https://www.researchgate.net/publication/334539401_A_Brief_History_of_Artificial_Intelligence_On_the_Past_Present_and_Future_of_Artificial_Intelligence (дата звернення: 06.03.2023).

3. Lachmann, R. Machine Learning-Driven Test Case Prioritization Approaches for Black-Box Software Testing. Nuremberg, Germany, European Test and Telemetry Conference ettc2018. URL: https://www.ama-science.org/proceedings/getFile/ZwtmZt== (дата звернення: 22.02.2023).

4. Mabl Extends Low-code Test Automation Platform to Include Accessibility Testing. URL: https://www.prnewswire.com/news-releases/mabl-extends-low-code-test-automation-platform-to-include-accessibility-testing-301533708.html (дата звернення: 27.03.2023).

5. Arbon, J., 2019. The AI Testing Singularity | STAREAST 2019. [Online] URL: https://www.youtube.com/watch?v=9xLnHcMIhhk (дата звернення: 18.02.2023).

6. Gerrard, P. The New Model for Testing. [Online] URL: https://www.youtube.com/watch?v=1Ra1192OpqY (дата звернення: 20.02.2023).

7. A. Yerokhin, O. Zolotukhin. Fuzzy Probabilistic Neural Network in Document Classification Tasks. Interbranch collection of scientific papers «Information Extraction and Processing». National Academy of Sciences of Ukraine. -2019. http://vidbir.ipm.lviv.ua/

8. Turevska, O. , Shubin, I. Improving the automated testing of Web-based services by reflecting the social habits of target audiences. 2015 Information Technologies in Innovation Business Conference, ITIB 2015 - Proceedings, 2015, с. 93-96.

9. Лановий О.Ф. Про один підхід до функціонального тестування web-додатків // Полиграфические, мультимедийные и web-технологии. Т1. Тез. докл. 2-й Международ. науч.-техн. конф.(16-22 мая 2017) / редкол.: ВФ Ткаченко, ИБ Чеботарева и др.–Харьков: ХНУРЭ, 2017.–246 с.

10. Maksym Bekuzarov, Oleksandr Samantsov, Oksana Mazurova, Mariia Shirokopetleva. Neural Network Architecture Editor With Code Generation. Problem of Infocommunications. Science and Technology (PIC S&T′2020), Kharkiv, Ukraine.- 6-9 October 2020.

11. Neural Networks: A Comprehensive Foundation, by Simon HAYKIN; Macmillan College Publishing~ New York, USA; IEEE Press, New York, USA; IEEE Computer Society Press, Los Alamitos, CA, USA; 1994; 696 pp. – URL: http://amutiara.staff.gunadarma.ac.id/Downloads/files/14638/Neural_Networks._A_Compre_Found__2nd_.pdf (дата звернення: 29.04.2023).

12. Citing TensorFlow. URL: https://www.tensorflow.org/about/bib – (дата звернення: 28.03.2023).

13. Bing-wan Cao Augmenting the software testing workflow with machine learning, 2018 // URL: https://www.semanticscholar.org/paper/Augmenting-the-software-testing-workflow-with-Cao/9f0d7367ebe282e40226aebc6c2ad167d39adb4c (дата звернення: 28.03.2023).

14. О.Ф.Лановий, О.В.Золотухін. Застосування нейромережевого підходу для класифікації втручань в роботу комп'ютерних систем // Застосування інформаційних технологій у діяльності НПУ: матеріали наук.-практ. семінару (м.Харків, 21 грудня 2018 р.) / МВС України, Харк.нац.ун-т внутр.справ. Харків. ХНУВС, 2018.– С.78-79.

15. C. L. Prabha and N. Shivakumar, Software Defect Prediction Using Machine Learning Techniques, 2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184), Tirunelveli, India, 2020, pp. 728-733, doi: 10.1109/ICOEI48184.2020.9142909.

16. Mika Mantyla Eero Laukkanen. Build waiting time in continuous integration – an initial interdisciplinary literature review. In Rapid Continuous

Software Engineering, 2015. URL: https://dl.acm.org/doi/pdf/10.5555/2820678.2820680 (дата звернення: 27.03.2023).

17. Martin Rinard Fan Long. Automatic patch generation by learning correct code. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016. / URL: https://dl.acm.org/doi/10.1145/2837614.2837617 (дата звернення: 27.03.2023).

18. Mitt Shah, Nandit Pujara. Software Defects Prediction Using Machine Learning. / URL: https://arxiv.org/pdf/2011.00998 (дата звернення: 27.03.2023).

19. Jungho Kim, Joung Woo Ryu, Hyun-Jeong Shin, Jin-Hee Song. Machine Learning Frameworks for Automated Software Testing Tools: A Study / URL: https://doi.org/10.5392/IJoC.2017.13.1.038 (дата звернення: 27.03.2023).

20. Лановий О.Ф. Візуалізація в методах тестування програмного забезпечення // Харків, 6-а Міжнародна науково-технічна конференція «ICT-2017», ХНУРЕ, 11-16 вересня 2017 р., С.110-111.