ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ВИКОРИСТАННЯ АЛГОРИТМІВ МАШИННОГО НАВЧАННЯ ДЛЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

на здобуття освітнього ступеня магістр
за спеціальності 126 Інформаційні системи та технології
(код,найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Ярослав ЖИВАГО
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти
група ІСДМ-64 Ярослав Живаго
(ім'я, ПРІЗВИЩЕ)
Керівник Ольга ПОЛОНЕВИЧ
к.т.н. *(ім'я, ПРІЗВИЩЕ)*
Рецензент: _____
науковий ступінь, вчене звання *(ім'я, ПРІЗВИЩЕ)*

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти магістр
Спеціальність 126 Інформаційні системи та технології
Освітньо-професійна прорама Інформаційні системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедрою ІСТ
_____ Каміла СТОРЧАК

“____” _____2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Живаго Ярославу Станіславовичу
(прізвище, ім’я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Використання алгоритмів машинного навчання для тестування програмного забезпечення
керівник кваліфікаційної роботи: Ольга ПОЛОНЕВИЧ к.т.н., доцент
(ім’я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від “15” жовтня 2024 р. № 320

2. Строк подання кваліфікаційної роботи «27» грудня 2024 р.

3. Вихідні дані кваліфікаційної роботи:
1. Методи дослідження та аналітичної діяльності.
2. Методи прогнозування помилок.
3. Методи оптимізації.
4. Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Дослідження методів дослідження та аналітичної діяльності.
2. Огляд методів прогнозування помилок та методів оптимізації.
3. Аналіз результатів запропонованої моделі виявлення помилок.

5.Перелік ілюстраційного матеріалу: *презентація*

6. Дата видачі завдання «15» жовтня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Підбір технічної літератури	15.10-05.11.24	
2.	Дослідження методів дослідження та аналітичної діяльності	06.11-12.11.24	
3.	Дослідження методів прогнозування помилок та методів оптимізації	13.11-19.11.24	
4.	Результати запропонованої моделі виявлення помилок	20.11-03.12.24	
5.	Висновки по роботі	04.12-10.12.24	
6.	Розробка демонстраційних матеріалів, доповідь.	11.12-20.12.24	
7.	Оформлення магістерської роботи	21.12-27.12.24	

Здобувач вищої освіти _____ Ярослав Живаго
(підпис) *(ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Ольга ПОЛОНЕВИЧ
(підпис) *(ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття ступня магістр: 84 стор., 32 рис., 5 табл., 16 джерел.

Мета роботи – дослідження методів прогнозування помилок та методів оптимізації програмного забезпечення із використанням алгоритмів машинного навчання.

Об'єкт дослідження – процес тестування програмного забезпечення та виявлення помилок.

Предмет дослідження – алгоритми машинного навчання та їх застосування для оптимізації процесів тестування програмного забезпечення.

Короткий зміст роботи. У першому розділі виконано аналіз використання алгоритмів машинного навчання для тестування програмного забезпечення. Здійснено огляд наукової літератури, яка описує особливості застосування методів машинного навчання у забезпеченні якості ПЗ. Також розглянуто сучасні тенденції автоматизації тестування з акцентом на використанні штучного інтелекту для виявлення помилок. Представлено порівняльний аналіз традиційних та сучасних методів тестування програмного забезпечення. Проведено оцінку ефективності алгоритмів і їх застосування для прогнозування дефектів у коді.

Третій розділ присвячено моделюванню процесу тестування із використанням алгоритмів машинного навчання. Представлено експериментальну методологію, описано алгоритми автоматизації та оптимізації тестування. Представлені результати дослідження з використанням реальних даних, що підтверджують ефективність розробленої моделі.

КЛЮЧОВІ СЛОВА: АЛГОРИТМИ МАШИННОГО НАВЧАННЯ, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯБ ВИЯВЛЕННЯ ДЕФЕКТІВ, ОПТИМІЗАЦІЯ, РЕГРЕСІЙНЕ ТЕСТУВАННЯ, КЛАСИФІКАЦІЯ.

ABSTRACT

The text part of the qualifying work for obtaining a bachelor's degree: 84 pp., 32 fig., 5 tables, 16 sources.

The purpose of the work is to study methods for predicting defects and optimizing software using machine learning algorithms.

The object of the study is the process of software testing and defect detection.

The subject of the study is machine learning algorithms and their application for optimizing software testing processes.

Summary of the work. The first chapter analyzes the use of machine learning algorithms for software testing. A review of scientific literature is provided, describing the specifics of applying machine learning methods to ensure software quality. Modern trends in testing automation are also examined, with an emphasis on using artificial intelligence to detect defects.

A comparative analysis of traditional and modern methods of software testing is presented. The effectiveness of algorithms and their application for defect prediction in code is evaluated.

The third chapter is devoted to modeling the testing process using machine learning algorithms. The experimental methodology is outlined, and the algorithms for automation and optimization of testing are described. The study results, based on real data, confirm the effectiveness of the proposed model.

KEYWORDS: MACHINE LEARNING ALGORITHMS, TESTING AUTOMATION, SOFTWARE, DEFECT DETECTION, OPTIMIZATION, REGRESSION TESTING, CLASSIFICATION.

ЗМІСТ

ВСТУП

Актуальність теми. Процес тестування програмного забезпечення є одним із найважливіших етапів у забезпеченні його якості та надійності. У зв'язку зі стрімким розвитком інформаційних технологій, сучасні системи програмного забезпечення стають все більш складними, а кількість можливих помилок у коді значно зростає. Традиційні методи тестування часто є трудомісткими, займають багато часу та вимагають значних ресурсів. У цьому контексті актуальним стає використання алгоритмів машинного навчання для автоматизації тестування програмного забезпечення, що дозволяє підвищити ефективність процесу, зменшити витрати та мінімізувати людський фактор.

Алгоритми машинного навчання відкривають нові можливості для автоматизації процесів тестування, прогнозування дефектів у коді, оптимізації тестових сценаріїв та виявлення прихованих помилок. Крім того, їхнє застосування сприяє покращенню безпеки та стабільності програмного забезпечення, що є критично важливим у багатьох галузях, таких як фінанси, охорона здоров'я та транспорт.

Мета роботи – дослідження методів прогнозування дефектів і оптимізації процесу тестування програмного забезпечення із використанням алгоритмів машинного навчання.

Для досягнення мети, у магістерській роботі успішно виконано наступні завдання:

- Аналіз тенденцій розвитку автоматизації тестування програмного забезпечення;
- Огляд алгоритмів машинного навчання, які використовуються для прогнозування помилок та оптимізації тестування;
- Розробка моделі автоматизації тестування із використанням алгоритмів машинного навчання.

Об'єкт дослідження – процес тестування програмного забезпечення та виявлення помилок.

Предмет дослідження – алгоритми машинного навчання та їхнє застосування для оптимізації процесу тестування програмного забезпечення.

Методи дослідження. У ході виконання роботи застосовувались методи аналізу та синтезу, моделювання, а також алгоритми машинного навчання, зокрема методи класифікації, регресії та оптимізації.

Наукова новизна одержаних результатів. У ході дослідження запропоновано новий підхід до автоматизації процесу тестування, що базується на використанні гібридних алгоритмів машинного навчання. Продемонстровано переваги запропонованої моделі в порівнянні з традиційними методами тестування, зокрема у підвищенні точності виявлення дефектів.

Практична значущість одержаних результатів. Запропонована модель може бути застосована для автоматизації тестування програмного забезпечення у різних галузях, що сприятиме зменшенню витрат на тестування, підвищенню його якості та скороченню часу виходу продукту на ринок.

Апробація результатів магістерської роботи. Основні положення та результати роботи були представлені на наукових конференціях та семінарах, організованих Державним університетом інформаційно-комунікаційних технологій.

РОЗДІЛ 1 АНАЛІЗ АЛГОРИТМІВ МАШИННОГО НАВЧАННЯ ДЛЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Огляд методів тестування програмного забезпечення

Один із найбільших витрат у розробці програмного забезпечення припадає на його обслуговування. Тому важливо зрозуміти, що саме спричиняє потребу в обслуговуванні та чи можна це передбачити. Численні дослідження показали, що певні методи оцінки складності створених програм можуть дати корисні прогностичні моделі для визначення ймовірності необхідності обслуговування через збої програмного забезпечення. Як правило, це здійснюється перед релізом, і для створення таких моделей часто потрібні певні об'єктно-орієнтовані програмні метрики. Однак не завжди розробники програмного забезпечення мають доступ до таких метрик.

Машинне навчання застосовується до наявних даних для обчислення сукупного рівня збоїв у програмному забезпеченні. Техніка прогнозування залишкової дефективності програмного забезпечення за допомогою машинного навчання може розглядатися як рішення проблеми передбачення залишкових помилок. Програмні метрики та дані про дефекти були виділені зі статичного репозиторію вихідного коду. Метрики програмного забезпечення створюються на основі статичного коду, а для збору інформації про дефекти використовуються зареєстровані помилки у репозиторії. За допомогою методу кореляції були видалені метрики, які не мали зв’язку з даними про дефекти. Це дозволяє аналізувати всі дані без зупинки процесу програмування.

Основною проблемою великих та складних програм є те, що неможливо вручну контролювати всі аспекти, а вартість помилки може бути досить високою. У результаті розробники можуть пропустити помилки під час тестування, що збільшує витрати на обслуговування. Загальна мета полягає у знаходженні методу, який дозволить точно прогнозувати дефекти програмного забезпечення. Рисунок 1 показує зростання зацікавленості цією темою з 2002 року (лише одне дослідження

з цього зразка, яке було опубліковане в 1993 році, вийшло до цієї дати). Ми можемо побачити помірний, але зростаючий інтерес до 2010 року. Досягнення в галузі машинного навчання за останнє десятиліття призвели до значно більшого застосування машинного навчання у генерації тестів, особливо починаючи з 2018 року. Понад 70% публікацій опубліковано лише за останні п'ять років, з яких 30% за останні два роки. Це сфера, що викликає зростаючий інтерес і зрілість, і очікуємо, що кількість публікацій значно збільшиться в наступні кілька років.

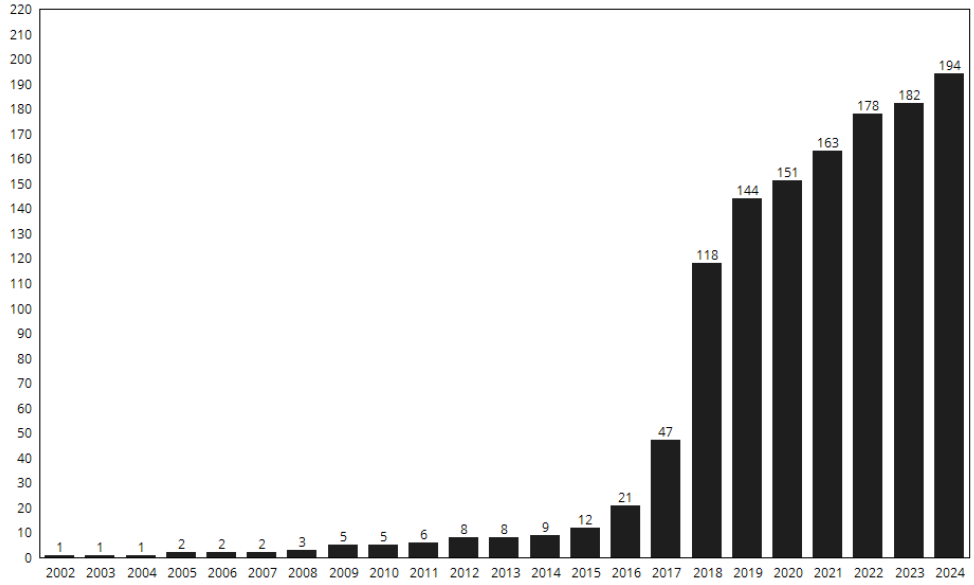

Рис 1.1 Кількість публікацій про машинне навчання на англомовних ресурсах

1.2 Сучасні тенденції автоматизації тестування

Автоматизоване тестування програмного забезпечення — це метод, який використовує спеціальні інструменти для запуску тестів, порівняння результатів і перевірки роботи додатків без втручання людини. На відміну від ручного тестування, автоматизоване тестування дозволяє швидше знаходити помилки та знижує ризик людських помилок. Це особливо важливо при тестуванні великих і складних програм, де велика кількість функцій, операцій та взаємозв’язків між

компонентами.

Основні переваги автоматизованого тестування:

1. Швидкість і продуктивність: Тестові випадки виконуються швидше, ніж при ручному тестуванні, що особливо важливо при частих релізах.
2. Повторюваність: Можливість багаторазово використовувати одні й ті сценарії для різних версій продукту.
3. Надійність: Завдяки стандартизованим тестам, які виключають фактор людського суб’єктивізму.
4. Зниження витрат на тривалих етапах: Хоча початкове налаштув автоматизації може бути витратним, у довгостроковій перспективі це суттєво економить ресурси.

Проте навіть при всіх перевагах автоматизованого тестування є значні виклики:

- Складність у створенні та підтримці тестових скриптів, що часто потребує суттєвих зусиль і знань.
- Висока ймовірність, що при зміні інтерфейсу користувача або коду ці скрипти потрібно оновлювати.
- Обмежена здатність автоматизованих тестів виявляти нетипові дефекти або адаптуватися до нових змін.

Машинне навчання (МН) може значно підвищити ефективність автоматизованого тестування завдяки своїм здатностям до:

1. Аналізу і прогнозування: МН моделі можуть аналізувати великі обсяги даних і виявляти закономірності. Наприклад, на основі попередніх тестових результатів та історії помилок модель може прогнозувати найбільш імовірні області, де можуть з’явитися дефекти.
2. Автоматичного генерації тестів: Завдяки алгоритмам машинного навчання можливо автоматизувати створення тестових випадків, які раніше створювались вручну, що скорочує час на написання тестів.

3. Оптимізації регресійного тестування: МН може визначати, які тести важливо виконувати в першу чергу, виявляючи пріоритетні сценарії, що знижує час на тестування.
4. Виявлення аномалій: Використання алгоритмів МН для визначення аномальних патернів в поведінці додатків допомагає виявляти дефекти, які можуть залишатися непоміченими при традиційному тестуванні.
5. Тестування UI та UX: Використовуючи комп'ютерний зір, МН моделі можуть аналізувати зображення для тестування інтерфейсів, перевіряти наявність елементів на екрані та правильність їхнього розташування.

Застосування алгоритмів МН в автоматизованому тестуванні дозволяє розв'язати деякі з найскладніших проблем і підвищити гнучкість процесів, що веде до зниження витрат, підвищення якості продукту та швидшого виходу на ринок.

Функція програмного забезпечення є його функціональним компонентом. Процес розробки програмного забезпечення часто зосереджений на створенні функцій. Розробка функцій здійснюється таким чином, щоб кілька команд не створювали одну чи кілька функцій одночасно. Помилки у програмі зазвичай розподіляються нерівномірно між її компонентами або функціями. Натомість певні функції мають тенденцію до частіших збоїв порівняно з іншими. Наступне питання: чи можливо виявити ці помилки до того, як вони почнуть спричиняти збої? Якщо так, це може дозволити більш ретельно аналізувати ці характеристики перед їх впровадженням для клієнтів.

Загальноприйнято, що дефекти неминучі під час проєктування великомасштабного програмного забезпечення. Тому дуже важливо мати спосіб усунення недоліків у процесі їх розвитку до серйозних збоїв. Некеровані збої можуть коштувати значних фінансових витрат і погіршити репутацію серед клієнтів. Водночас проактивне управління дефектами та грамотне управління збоями в цілому можуть зменшити витрати та зміцнити довіру клієнтів.

Як можна зменшити наслідки програмних збоїв? Для цього враховуються дві стратегії: по-перше, визначити, наскільки високим є очікуваний рівень збоїв у

випуску. По-друге, звузити фокус для визначення, які функції програмного забезпечення є більш схильними до помилок у майбутніх версіях. Це вимагає створення системи, яка може прогнозувати майбутні збої та оцінювати ймовірність виникнення помилок у розроблених функціях. Така система має бути об'єктивною, зрозумілою та максимально точною (наскільки це можливо і доцільно). Це необхідно, щоб було легко порівнювати результати в межах організації, просто впроваджувати та використовувати.

Методи машинного навчання підходять для завдань прогнозування дефектів на основі загальних методологій. На сьогодні існує більше досліджень, які демонструють використання алгоритмів машинного навчання для підтримки програмного забезпечення, ніж інших методів. Проте немає єдиного алгоритму, який можна застосувати до всього статичного програмного забезпечення. Через це часто важко використовувати одну й ту ж модель прогнозування для всього програмного забезпечення. Для забезпечення точності порівняльних досліджень слід створювати більш надійні методології.

Дослідження показують, що кількість помилок зменшується зі зменшенням розміру програмного модуля, проте інші дослідження демонструють, що кількість помилок пов’язана з так званим "ідеальним розміром", який не є ні надто великим, ні мінімальним. Ще дві роботи пояснюють, чому немає чіткої залежності між розміром модуля програмного забезпечення та кількістю або щільністю помилок у ньому.

Основними цілями є:

- Виявлення нових залежностей у характеристиках самого коду.
- Розробка моделей прогнозування для програмного забезпечення (визначення дефектів у процесі розробки для ідентифікації майбутніх обмежень).
- Застосування цих знань до інших програмних проєктів.

Нові результати цього дослідження мають полегшити розуміння поточних проблем у програмному забезпеченні та прогнозування збоїв у майбутньому. Загалом, використання метрик дозволяє менеджерам аналізувати складність

проєкту, що розробляється або вже створеного, оцінювати обсяг роботи, стиль програми та зусилля кожного розробника. Однак метрики можуть бути лише рекомендаційними показниками; вони не можуть бути повністю визначальними, оскільки розробники часто прагнуть мінімізувати або максимізувати певний показник для своєї програми, що може призвести до зниження продуктивності.

Наприклад, якщо програміст додав кілька рядків коду або зробив структурні зміни, це не обов’язково означає, що він нічого не досяг, а, можливо, вирішував надзвичайно складну проблему. Проте цю проблему частково можна вирішити за допомогою метрик складності, оскільки більш складна програма ускладнює пошук помилок. Важливо зазначити, що не існує універсальної метрики, яка підходить для всіх ситуацій.

Гібридні метрики також залежать від простіших показників і не можуть бути універсальними. Будь-які контрольовані аспекти програми повинні регулюватися або у взаємозв’язку один з одним, або залежно від конкретного завдання. Жодна метрика не може бути абсолютно об'єктивною і стати основою для рішень, оскільки будь-яка метрика є лише індикатором, який залежить від мови програмування та стилю програми.

Рисунок 1.2 демонструє приклад взаємозв'язку характеристик. На основі певного набору внутрішніх властивостей (базових програмних метрик) обчислюються якісні характеристики. Метрики отримуються та аналізуються для виявлення значущих зв'язків.

Основною метою аналізу даних є дослідження реальних залежностей між метриками, а також визначення причинно-наслідкових зв’язків між показниками, тобто наскільки зміна одного показника залежить від зміни іншого.

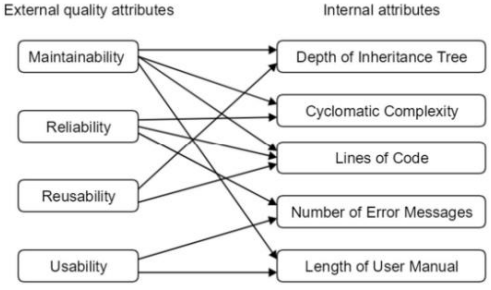

Рис1.2 Види програмних метрик

Залежності можуть бути прямими чи зворотними. У першому випадку зі зростанням показника x зростає і показник y, у другому — зі зменшенням x зменшується y. Рис.1.3 демонструє приклад діаграми розсіювання для метрик.

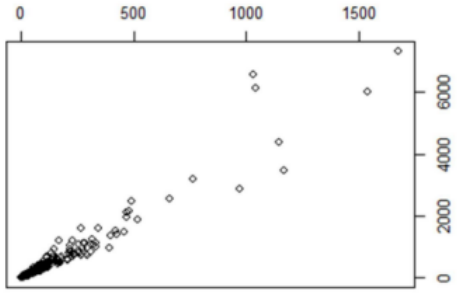

Рис1.3 Показники програмного забезпечення. Зважений метод підраховується як вісь X, а рядки коду – як розкид по осі Y.

Метою дипломної роботи є дослідження та аналіз методів застосування алгоритмів машинного навчання для підвищення ефективності та автоматизації процесу тестування програмного забезпечення. У роботі також розглядаються можливості оптимізації виявлення помилок, зменшення часу на тестування, а також підвищення точності тестування завдяки використанню алгоритмів машинного навчання.

1.3 Порівняння традиційних та сучасних підходів

Використання машинного навчання для оптимізації регресійного тестування веб-додатків. Об’єкт дослідження в цій роботі — це процес регресійного тестування веб-додатків, який є важливою частиною забезпечення якості програмного забезпечення. Регресійне тестування проводиться після внесення змін або оновлень у код з метою виявлення можливих помилок, що виникають у частинах програмного забезпечення, які вже працюють правильно. Тестування застосовується для перевірки того, чи не порушила зміна коду існуючі функціональні можливості програмного продукту. У випадку веб-додатків, цей процес набуває додаткової важливості через специфіку таких продуктів, які працюють на різноманітних платформах, зокрема браузерах і пристроях.

Веб-додатки є надзвичайно важливими для бізнесу, оскільки вони використовуються в різних сферах, від електронної комерції до соціальних мереж та бізнес-платформ. Веб-додатки постійно змінюються та оновлюються, і процес регресійного тестування дозволяє забезпечити, щоб ці зміни не призвели до помилок, які можуть вплинути на користувачів. Змінити лише одну частину коду в веб-додатку може бути достатньо для того, щоб пошкодити його основну функціональність. Тому регулярне регресійне тестування є необхідним для підтримки стабільної та безпечної роботи додатка в умовах постійних змін.

У контексті веб-додатків регресійне тестування не обмежується перевіркою лише внутрішніх функцій програми. Воно включає в себе також перевірку кросбраузерної сумісності, тобто перевірку того, як додаток працює на різних веб-браузерах. Враховуючи, що кожен браузер має свої особливості роботи з кодом, зміни в програмному забезпеченні можуть впливати на один браузер, але не обов’язково на інший. Це особливо важливо для програм, які орієнтовані на широку аудиторію, де необхідно забезпечити сумісність з різними браузерами, такими як Google Chrome, Mozilla Firefox, Safari, Opera тощо.

Ще одним важливим аспектом є перевірка сумісності з різними пристроями. Веб-додатки повинні бути адаптованими для мобільних телефонів, планшетів і комп’ютерів. Регресійне тестування допомагає перевірити, чи не порушить зміна коду функціональність на мобільних пристроях або на різних платформах, що важливо для додатків, що мають багатоканальне використання. Веб-додаток може виглядати і працювати по-різному на мобільному телефоні і на десктопному комп’ютері, тому важливо перевіряти, щоб оновлення не призвело до втрати функціональності на одному з пристроїв.

Однією з основних проблем веб-додатків є питання швидкості роботи. Оптимізація коду може впливати на час завантаження сторінок або час відгуку серверів. Швидкість роботи є важливим фактором для користувачів, оскільки навіть кілька секунд затримки можуть негативно вплинути на користувацький досвід, що може призвести до зниження кількості користувачів і вплинути на репутацію компанії. Регресійне тестування дозволяє вчасно виявляти потенційні

проблеми з продуктивністю додатка і забезпечити його стабільну роботу після змін.

Загалом, об’єкт дослідження охоплює весь процес регресійного тестування веб-додатків, від перевірки базових функцій до складних взаємодій з іншими програмними та апаратними компонентами. Це вимагає комплексного підходу до тестування, що включає не лише перевірку основного функціоналу, але й тестування на різних пристроях, у різних браузерах, а також з оцінкою швидкості роботи та навантаження.

Предметом дослідження є використання алгоритмів машинного навчання для оптимізації регресійного тестування веб-додатків. Оскільки традиційні методи регресійного тестування вимагають великої кількості часу та ресурсів, особливо для великих систем, застосування машинного навчання може значно оптимізувати цей процес. Машинне навчання дозволяє автоматизувати вибір тестових випадків, визначати пріоритети тестування та прогнозувати можливі дефекти, що допомагає знизити витрати часу і ресурсів.

Один із основних напрямків машинного навчання у тестуванні — це передбачення дефектів на основі історичних даних, зібраних під час попередніх тестувань. За допомогою алгоритмів машинного навчання можна виявляти закономірності в даних, які дозволяють точніше визначати, які частини коду найімовірніше спричинять помилки в майбутньому. Це дозволяє зосередити зусилля тестувальників на найбільш критичних ділянках програми.

Додатково, автоматизація вибору тестових випадків є ще одним важливим аспектом використання машинного навчання в регресійному тестуванні. Завдяки машинному навчанню, тестові випадки можуть бути автоматично відібрані на основі їх важливості та ймовірності виявлення дефектів, що зменшує кількість виконуваних тестів і таким чином знижує витрати часу та ресурсів. Це також дозволяє зменшити ризик людської помилки при відборі тестів.

Для ефективної пріоритизації тестових випадків також використовуються алгоритми машинного навчання, що дає змогу ранжувати тести за важливістю, забезпечуючи виконання найбільш критичних тестів спочатку. Такий підхід дозволяє не тільки скоротити час тестування, але й підвищити ефективність

виявлення помилок на ранніх етапах.

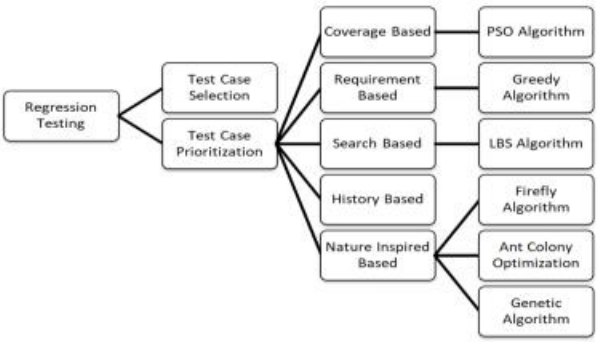

Рис.1.4. Методи пріоритезації тестових випадків

Застосування машинного навчання також має великий потенціал у прогнозуванні дефектів, що дозволяє не тільки вибирати найбільш важливі тести для виконання, а й заздалегідь визначати ділянки коду, які потребують найбільшої уваги. Це дозволяє спрогнозувати можливі проблеми до того, як вони проявляться в реальному середовищі. Таким чином, використання машинного навчання в регресійному тестуванні веб-додатків може значно покращити якість тестування, знизити витрати на обробку великих кодових баз і забезпечити більш точну і швидку перевірку. Цей підхід також може бути масштабований для великих проєктів, що дає змогу ефективно підтримувати якість програмного забезпечення навіть у найбільших системах. Машинне навчання дозволяє не тільки автоматизувати вибір і пріоритизацію тестів, а й адаптуватися до змін, які виникають під час розробки програмного забезпечення, дозволяючи своєчасно виявляти дефекти й оптимізувати процес тестування. Рішення за допомогою МН:

1. Збір даних: Спочатку збираються історичні дані про тестування, включаючи дані про зміни в коді, дефекти, які були виявлені, і результати тестів (успішні або провальні).

2. Навчання моделі: Використовується алгоритм машинного навчання (наприклад, Random Forest або нейронна мережа) для аналізу цих даних. Модель

навчається на основі того, які частини коду зазвичай схильні до появи дефектів і які тестові випадки їх виявляють.

3. Прогнозування ризикових модулів: Після внесення змін в код, модель аналізує їх та прогнозує, які модулі (або частини коду) мають найвищий ризик появи дефектів. Таким чином, тестувальники отримують рекомендації щодо пріоритетності тестування.

4. Оптимізація тестового набору: На основі прогнозів, модель формує список найбільш критичних тестів для проведення, що значно скорочує час на тестування, адже перевіряються саме ті компоненти, де ймовірність дефектів найвища.

Результати:

- Економія ресурсів: Завдяки скороченню кількості тестових сценаріїв, які потрібно виконувати, знижується навантаження на команду тестувальників і заощаджується час.

- Підвищення ефективності тестування: Застосування МН допомагає зосередити увагу на найуразливіших компонентах, що знижує ймовірність того, що дефекти залишаться непоміченими.

- Гнучкість і адаптивність: Модель адаптується до змін у коді, автоматично підлаштовуючись до нових сценаріїв без потреби в ручному втручанні.

Цей приклад показує, як алгоритми машинного навчання можуть оптимізувати регресійне тестування та підвищити ефективність у розробці веб-додатків, особливо в умовах частих змін і оновлень.

Методи дослідження:

Метод аналізу та синтезу

Аналіз передбачає розбиття складного об'єкта чи системи на менші компоненти для їх детального вивчення. У контексті тестування програмного забезпечення цей метод використовується для дослідження окремих модулів чи процесів.

Застосування аналізу в тестуванні програмного забезпечення:

1. Розбиття систем на модулі: Розподіл програмного забезпечення на функціональні частини, наприклад, автентифікація користувачів, обробка даних, створення звітів.

2. Аналіз журналів помилок: Вивчення логів роботи системи для визначення найбільш уразливих зон, які викликають збої.

3. Створення тестових сценаріїв: Розробка тест-кейсів, орієнтованих на конкретні функції чи модулі програми.

Синтез передбачає об’єднання результатів аналізу для формування повного уявлення про систему. Цей підхід допомагає інтегрувати результати тестування окремих компонентів, щоб оцінити загальну якість програмного забезпечення.

Застосування синтезу в тестуванні програмного забезпечення:

1. Інтеграція результатів тестування: Об’єднання результатів тестування окремих модулів для оцінки роботи всієї системи.

2. Оцінка продуктивності системи: Аналіз змін у одному компоненті системи та їх впливу на всю програму.

3. Створення підсумкових звітів: Підготовка звітів, що містять дані з функціонального, регресійного та продуктивного тестування.

Переваги методу аналізу та синтезу

1. Детальне розуміння компонентів: Дозволяє глибоко дослідити кожен модуль для виявлення дефектів та оптимізації.

2. Комплексна оцінка: Інтегрує результати тестування для формування загального уявлення про якість програмного забезпечення.

3. Покращене управління ресурсами: Дає змогу сфокусуватися на найбільш критичних зонах під час тестування.

4. Підтримка модульного тестування:

Узгоджується з практиками Agile та DevOps, дозволяючи поступове тестування та інтеграцію компонентів.

Проблеми використання

1. Складність: Розбиття взаємозалежних систем може бути досить складним.

2. Часозатратність: Як аналіз, так і синтез вимагають значних часових ресурсів.

3. Залежність від досвіду: Вимагає глибокого знання архітектури та функціоналу системи.

2. Моделювання

Моделювання — це метод, що використовується для створення абстрактного або спрощеного представлення системи, процесу чи явища. У тестуванні програмного забезпечення моделювання дозволяє відтворювати реальні або гіпотетичні сценарії використання системи для аналізу її поведінки та оцінки її надійності.

Моделювання передбачає створення моделей, які можуть бути:

1. Математичними: Використовуються формули та алгоритми для опису функцій програми.

2. Статистичними: Аналізуються ймовірності подій, пов’язаних із дефектами або збоями.

3. Симуляційними: Відтворюються умови роботи програмного забезпечення у штучному середовищі, що імітує реальні сценарії.

Етапи моделювання у тестуванні ПЗ

1. Постановка цілей: Визначення конкретної задачі тестування (наприклад, виявлення дефектів, оцінка продуктивності).

2. Збір даних: Аналіз логів роботи системи, історії змін у коді або статистики помилок.

3. Створення моделі: Вибір типу моделі (нейронні мережі, decision tree, математичні рівняння) та її розробка.

4. Перевірка моделі: Тестування моделі на контрольному наборі даних для оцінки її точності.

5. Аналіз результатів: Інтерпретація результатів моделювання та корекція недоліків моделі.

Застосування моделювання у тестуванні програмного забезпечення

1. Моделювання поведінки користувачів

Один із найпоширеніших напрямів моделювання в тестуванні ПЗ — відтворення сценаріїв використання програми користувачами.

2. Прогнозування дефектів

Моделі машинного навчання, такі як Random Forest або нейронні мережі, можуть використовуватися для аналізу історичних даних про дефекти та визначення зон ризику у коді.

3. Тестування продуктивності

Моделювання запитів до серверів або взаємодії між компонентами системи дозволяє оцінити, як програма працює під час пікових навантажень.

4. Автоматизація генерації тестів

Моделювання може використовуватися для автоматичного створення тестових сценаріїв. Це спрощує процес тестування великих систем, особливо у випадках, коли кількість сценаріїв використання занадто велика для ручного опрацювання.

Переваги методу моделювання

1. Ефективність:

Моделі дозволяють швидко відтворювати складні сценарії роботи програми, що суттєво знижує час тестування.

2. Гнучкість:

Можливість адаптувати моделі до різних типів систем і умов роботи (веб-додатки, мобільні додатки, серверні системи).

3. Прогнозування:

Завдяки аналізу даних моделі можуть передбачати потенційні зони ризику або точки відмови системи.

4. Масштабованість:

Моделювання дозволяє тестувати системи, які працюють із великою кількістю даних або високим рівнем одночасної активності користувачів.

5. Автоматизація:

Багато етапів моделювання, таких як генерація тестів або симуляція поведінки користувачів, можуть бути автоматизовані.

Обмеження методу моделювання

1. Складність створення моделі:

Створення якісної моделі вимагає значних ресурсів, зокрема часу на аналіз системи, вибір типу моделі та її налаштування.

2. Вимоги до даних:

Для створення моделей необхідні великі обсяги даних, що можуть бути недоступними на ранніх етапах розробки.

3. Точність моделі:

Результати моделювання залежать від якості моделі: якщо модель недосконала, її прогнози можуть бути хибними.

4. Витрати на підтримку:

Моделі потребують регулярного оновлення у зв’язку зі змінами в системі. Це може збільшити витрати на тестування.

Моделювання — це інструмент для тестування програмного забезпечення, який дозволяє відтворювати складні сценарії, прогнозувати дефекти та оцінювати продуктивність системи. Завдяки своїй ефективності та гнучкості метод моделювання є одним із найпопулярніших підходів у сучасному тестуванні ПЗ. Однак його застосування вимагає глибокого розуміння системи, доступу до даних і ресурсів для створення моделей.

4. Експериментальний метод

Експериментальний метод використовується для перевірки гіпотез, визначення причинно-наслідкових зв'язків та оцінки ефективності підходів шляхом практичних експериментів. У тестуванні програмного забезпечення експериментальний підхід дозволяє зібрати емпіричні дані, які підтверджують або спростовують ефективність певних рішень, методів чи моделей.

Експериментальний метод базується на контрольованих умовах, у яких змінюється один або декілька факторів (вхідні дані, параметри середовища, алгоритми) для оцінки впливу на результат. У тестуванні програмного забезпечення цей метод використовується для:

1. Вимірювання продуктивності системи у різних умовах.
2. Порівняння ефективності алгоритмів.
3. Визначення впливу змін у коді на стабільність роботи програми.

Ключові характеристики:

- Чітко визначені гіпотези та мета.
- Контрольоване середовище.
- Повторюваність для перевірки результатів.

Етапи експериментального методу

1. Формулювання гіпотези: чітко визначається, що саме необхідно перевірити. Наприклад: "Чи знижує використання алгоритму Random Forest кількість помилок у прогнозах дефектів порівняно з SVM?"

2. Вибір метрик: визначення показників для оцінки результатів, наприклад: точність (accuracy), повнота (recall), швидкість виконання тощо.

3. Підготовка середовища: створення контрольованого середовища для проведення експерименту, наприклад, тестового серверу або симуляційної платформи.

4. Проведення експерименту: зміна певних параметрів або умов (наприклад, типу алгоритму, вхідних даних) для аналізу їхнього впливу на результат.

5. Збір даних: фіксація всіх результатів та показників для подальшого аналізу.

6. Аналіз результатів: порівняння отриманих даних із гіпотезою. Наприклад, чи підтверджується передбачувана ефективність алгоритму.

Застосування експериментального методу у тестуванні ПЗ

1. Тестування продуктивності: експерименти дозволяють оцінити, як програма працює у різних умовах, наприклад, під час одночасного доступу тисячі користувачів.

2. Порівняння моделей машинного навчання: під час вибору оптимального алгоритму (наприклад, Random Forest, нейронні мережі або SVM) проводяться експерименти на одному наборі даних, щоб визначити, який підхід забезпечує найкращі результати.

3. Аналіз змін у коді: експерименти дозволяють визначити, як внесення змін у код впливає на його якість або продуктивність.

4. Автоматизація регресійного тестування: порівняння результатів тестів до і після оновлень системи для виявлення можливих регресійних помилок.

Переваги експериментального методу

1. Точність:

Забезпечує кількісні показники, що дають змогу об'єктивно оцінювати результати.

2. Повторюваність:

Експерименти можна повторити для перевірки стабільності результатів.

3. Виявлення закономірностей:

Дозволяє зрозуміти, які фактори впливають на якість програмного забезпечення.

4. Оптимізація:

Дає змогу знаходити найефективніші рішення та усувати слабкі місця.

Обмеження експериментального методу

1. Складність підготовки середовища:

Налаштування експериментального середовища може бути складним і часозатратним.

2. Залежність від обсягу даних:

Для проведення якісних експериментів часто потрібні великі обсяги даних.

3. Ресурсоємність:

Експерименти можуть вимагати значних обчислювальних ресурсів, особливо під час тестування продуктивності чи моделей машинного навчання.

Експериментальний метод є невід'ємною частиною тестування програмного забезпечення. Завдяки його застосуванню можна визначити ефективність нових рішень, оцінити продуктивність програм та виявити закономірності у виникненні дефектів. Однак ефективність цього підходу залежить від ретельної підготовки середовища, обсягу доступних даних і вибору правильних метрик для оцінки результатів.

4. Метод статистичного аналізу

Метод статистичного аналізу використовується для збору, обробки, інтерпретації та представлення даних у вигляді числових показників. У тестуванні програмного забезпечення статистичний аналіз дозволяє виявляти закономірності, оцінювати результати тестів та приймати обґрунтовані рішення на основі отриманих даних. Статистичний аналіз базується на застосуванні математичних і статистичних методів для оцінки характеристик програмного забезпечення. Він використовується для аналізу:

1. Частоти та типів дефектів у коді.
2. Продуктивності системи.
3. Ефективності змін у коді.
4. Кореляцій між метриками якості.

Метод допомагає зрозуміти, які фактори впливають на якість ПЗ, та оцінити результати тестування з точки зору кількісних показників.

Етапи статистичного аналізу

1. Збір даних: збір тестових результатів, логів роботи програми або статистики помилок.

2. Обробка даних: усунення "шуму" (некоректних або неповних записів), нормалізація даних (наприклад, перетворення у відсотки або стандартні відхилення).

3. Аналіз даних: використання статистичних методів, таких як обчислення середнього значення, медіани, кореляційного аналізу чи тестів значущості.

4. Візуалізація результатів: побудова графіків, діаграм, таблиць, що дозволяють спростити інтерпретацію даних.

5. Інтерпретація: висновки про ефективність тестування, оцінка якості коду або продуктивності системи.

Застосування методу статистичного аналізу у тестуванні ПЗ

1. Аналіз дефектів

Статистичний аналіз дозволяє визначити частоту, типи та розподіл дефектів у системі.

2. Оцінка продуктивності

Метод використовується для аналізу швидкості обробки запитів, часу відгуку сервера, пропускної здатності системи.

3. Порівняння результатів тестування

Статистичний аналіз дозволяє порівнювати результати різних підходів або версій програмного забезпечення.

4. Визначення залежностей між метриками

Кореляційний аналіз дозволяє визначити залежність між різними параметрами, наприклад, кількістю змін у коді та частотою помилок.

Переваги методу статистичного аналізу

1. Об'єктивність:

Дані аналізуються математично, що зменшує ризик суб'єктивних помилок.

2. Прогнозування:

Статистичний аналіз дозволяє передбачити ризики та вразливості системи на основі попередніх результатів.

3. Візуалізація:

Графіки, діаграми та таблиці допомагають зрозуміти великі обсяги даних.

4. Гнучкість:

Метод може застосовуватися як для великих, так і для малих проєктів.

Обмеження методу статистичного аналізу

1. Залежність від якості даних:

Результати аналізу залежать від точності та повноти даних. Некоректні або

неповні записи можуть призвести до хибних висновків.

2. Складність:

Використання складних статистичних методів вимагає високої кваліфікації спеціаліста.

3. Часові витрати:

Збір, обробка та аналіз великих обсягів даних можуть займати багато часу.

Приклади інструментів для статистичного аналізу

1. R: відкрита мова програмування для статистичного аналізу. Широко використовується для побудови графіків та обчислення статистичних метрик.

2. Python (бібліотеки pandas, matplotlib, seaborn): використовується для обробки великих наборів даних та їх візуалізації.

3. Excel: підходить для базового статистичного аналізу та створення діаграм.

4. SPSS: інструмент для складного статистичного аналізу у великих проєктах.

Метод статистичного аналізу є важливим інструментом у тестуванні програмного забезпечення. Він дозволяє виявляти закономірності, оцінювати продуктивність та ефективність змін у системі, приймати обґрунтовані рішення на основі даних. Проте його ефективність залежить від точності вихідних даних і кваліфікації спеціаліста, який проводить аналіз.

5. Метод порівняльного аналізу

Метод порівняльного аналізу — це дослідницький підхід, який використовується для оцінки об'єктів, процесів або методів через їх порівняння. У контексті тестування програмного забезпечення цей метод дозволяє оцінити ефективність різних підходів, інструментів або методологій для вибору найкращого рішення. Порівняльний аналіз базується на принципах систематичного зіставлення характеристик або показників. Його основна мета — визначення переваг, недоліків та придатності конкретних рішень для певного проєкту чи середовища.

Ключові елементи порівняльного аналізу:

1. Вибір критеріїв для порівняння (швидкість, точність, стабільність, витрати тощо).

2. Підготовка тестового середовища для забезпечення однакових умов тестування.

3. Порівняння результатів різних методів або інструментів.

Етапи порівняльного аналізу

1. Визначення мети аналізу:

Чітке формулювання питання, яке потрібно вирішити через порівняння (наприклад, "Який інструмент для автоматизації тестування є найефективнішим?").

2. Вибір об'єктів для порівняння:

Вибір двох або більше підходів, інструментів чи алгоритмів, які будуть оцінюватися.

3. Визначення критеріїв:

Встановлення метрик або показників для оцінки (наприклад, точність виявлення дефектів, час виконання тестів, вартість впровадження).

4. Проведення тестування:

Використання кожного об’єкта порівняння в однакових умовах (один набір тестів, ідентичне середовище).

5. Аналіз результатів:

Порівняння результатів за всіма критеріями та вибір найкращого варіанта.

6. Підготовка висновків:

Формулювання висновків щодо придатності та ефективності кожного рішення.

Застосування методу порівняльного аналізу у тестуванні ПЗ

1. Порівняння інструментів автоматизації тестування

Один із найпоширеніших напрямів — оцінка інструментів, таких як Selenium, Katalon Studio, TestComplete або Appium..

2. Порівняння алгоритмів виявлення дефектів

Вибір оптимального алгоритму машинного навчання (наприклад, Random Forest, SVM, нейронні мережі) для аналізу дефектів у коді.

3. Порівняння підходів до тестування (ручне vs автоматизоване)

Аналіз ефективності традиційного ручного тестування у порівнянні з автоматизованими тестами.

4. Оцінка продуктивності різних версій програми

Порівняння продуктивності системи до та після внесення змін у код.

Переваги методу порівняльного аналізу

1. Обґрунтованість вибору:

Дає змогу приймати рішення на основі реальних результатів, а не припущень.

2. Ефективність:

Дозволяє обрати найкращий інструмент чи підхід для конкретного проєкту.

3. Універсальність:

Метод підходить для порівняння інструментів, алгоритмів, підходів і навіть версій програмного забезпечення.

4. Поліпшення якості:

Порівняння різних варіантів допомагає визначити найефективніші способи підвищення якості програмного забезпечення.

Обмеження методу порівняльного аналізу

1. Складність вибору критеріїв:

Важливо правильно визначити, які показники найбільше впливають на успіх проєкту.

2. Часозатратність:

Проведення тестів для кожного варіанта може зайняти багато часу.

3. Необ’єктивність:

Якщо порівняння проводиться без чітких критеріїв або за умов, що відрізняються, результати можуть бути упередженими.

4. Вимоги до ресурсів:

Для порівняння інструментів або підходів можуть знадобитися додаткові ресурси (обладнання, ліцензії тощо).

Метод порівняльного аналізу — це потужний інструмент для прийняття обґрунтованих рішень у тестуванні програмного забезпечення. Завдяки

систематичному порівнянню інструментів, алгоритмів чи підходів, можна обрати найбільш ефективне рішення для конкретного проєкту. Однак успішне використання методу потребує чіткого визначення критеріїв оцінки та забезпечення однакових умов для порівнюваних об’єктів.

РОЗДІЛ 2 ВИКОРИСТАННЯ АЛГОРИТМІВ МАШИННОГО НАВЧАННЯ У ТЕСТУВАННІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Моделювання процесів тестування за допомогою машинного навчання

Машинне навчання стало інструментом для вирішення багатьох проблем в тестуванні програмного забезпечення, включаючи прогнозування дефектів, вибір тестових випадків та пріоритизацію тестів. Алгоритми машинного навчання дозволяють автоматизувати процеси, зменшити людський фактор та поліпшити ефективність тестування, що є особливо важливим для складних і масштабних систем. Серед найбільш використовуваних алгоритмів машинного навчання в тестуванні програмного забезпечення можна виділити методи класифікації, такі як Support Vector Machines (SVM), Random Forest, а також методи, які дозволяють вирішувати задачі регресії та класифікації, такі як Gradient Boosting Machines (GBM). Крім того, для вибору тестових випадків та їх пріоритизації застосовуються алгоритми, що базуються на нейронних мережах, таких як Convolutional Neural Networks (CNN) та Recurrent Neural Networks (RNN).

Використання цих методів дозволяє значно підвищити точність тестування і мінімізувати ймовірність помилок, що виникають через недосконаліть традиційних методів.У даному розділі ми проаналізуємо існуючі підходи та алгоритми машинного навчання, які використовуються в тестуванні програмного забезпечення, а також оцінимо їх ефективність та переваги у порівнянні з традиційними методами тестування.

Random Forest

Використовується для прогнозування дефектів у коді, класифікації модулів як "дефектний/бездефектний".

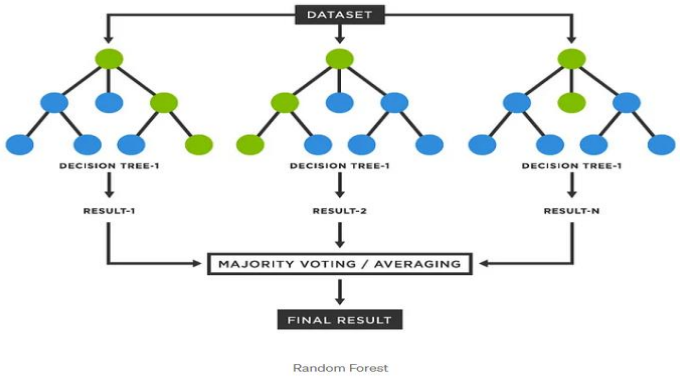

Рис.2.1. Random Forest algorithm

Random Forest є одним із найефективніших і найпоширеніших алгоритмів машинного навчання, що використовується для вирішення задач класифікації та регресії. Він належить до ансамблевих методів, які створюють моделі на основі поєднання кількох слабких моделей (у цьому випадку дерев рішень), щоб отримати сильніший прогноз. Цей підхід забезпечує більшу стійкість до шуму в даних і дозволяє моделі працювати навіть із великими та складними наборами даних.

Суть алгоритму полягає у побудові ансамблю дерев рішень. Для кожного дерева обирається випадкова підмножина даних методом "bagging" (bootstrap aggregating), що передбачає вибір з повторенням. Крім того, кожне дерево будується, використовуючи лише випадкову підмножину ознак, що забезпечує додаткову випадковість і допомагає уникнути перенавчання. Для класифікації кожне дерево голосує за певний клас, і клас, що отримав найбільшу кількість голосів, обирається як остаточний результат. Для регресії модель усереднює прогнози всіх дерев.

Однією з найважливіших переваг Random Forest є його здатність працювати з високою точністю навіть у складних умовах. Наприклад, він ефективно обробляє набори даних із пропущеними значеннями, шумом або великою кількістю ознак. Крім того, алгоритм забезпечує функцію оцінки важливості ознак, що дозволяє зрозуміти, які фактори найбільше впливають на результат. Це особливо важливо в задачах, де необхідна інтерпретація моделі.

Random Forest має широкий спектр застосувань. У тестуванні програмного

забезпечення алгоритм використовується для прогнозування дефектів, аналізуючи метрики коду, такі як кількість рядків, цикломатична складність або кількість викликів функцій. На основі історичних даних про помилки модель визначає модулі, які, ймовірно, можуть містити дефекти, що дозволяє командам тестувальників ефективніше розподіляти свої ресурси. У медицині Random Forest застосовується для виявлення захворювань, класифікації пацієнтів за групами ризику та прогнозування результатів лікування. У фінансах алгоритм допомагає оцінювати кредитні ризики, прогнозувати тенденції на ринку та виявляти шахрайські транзакції. У сфері комп'ютерного зору Random Forest використовується для класифікації зображень та розпізнавання об'єктів.

Алгоритм також демонструє високу стійкість до шуму в даних. Наприклад, якщо деякі дерева в ансамблі були побудовані на основі помилкових чи неточних даних, їхній вплив на кінцевий результат буде мінімальним, оскільки модель об'єднує прогнози багатьох дерев. Ця властивість робить Random Forest надзвичайно корисним для задач, де вхідні дані є неповними або містять помилки.

Однак Random Forest не позбавлений недоліків. Його основною проблемою є висока обчислювальна складність, особливо коли йдеться про побудову великої кількості дерев для великих наборів даних. Це може вимагати значних ресурсів і часу. Крім того, хоча окреме дерево рішення легко інтерпретувати, ансамбль із сотень або тисяч дерев стає "чорною скринькою", де зрозуміти внутрішню логіку прийняття рішень стає значно складніше. Для покращення продуктивності Random Forest може бути поєднаний із іншими алгоритмами. Наприклад, його можна інтегрувати з методами обробки тексту або часових рядів для аналізу неструктурованих даних, таких як журнали роботи програмного забезпечення або звіти про помилки. Це робить алгоритм ще більш універсальним і адаптивним для різних сфер.

Таким чином, Random Forest залишається одним із найнадійніших інструментів для вирішення багатьох практичних задач. Його здатність працювати з великими наборами даних, враховувати багато ознак і забезпечувати високу точність робить його незамінним у багатьох галузях, зокрема в

тестуванніпрограмного забезпечення, медицині, фінансах та аналітиці даних.

Support Vector Machines (SVM)

Для класифікації тестових випадків або визначення ризикових ділянок коду.

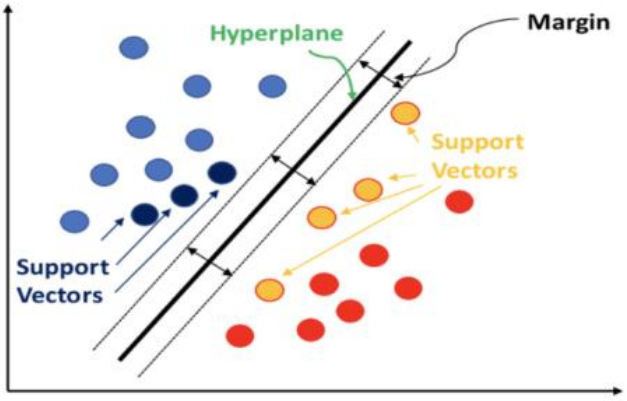

Рис.2.2. Support Vector Machines

Support Vector Machines (SVM) є алгоритмом машинного навчання, який часто використовується для задач класифікації та регресії. Цей алгоритм базується на принципі знаходження гіперплощини, яка оптимально розділяє дані на класи, забезпечуючи максимальний зазор (margin) між точками різних класів. SVM є ефективним для роботи з високовимірними даними та невеликими наборами даних, що робить його придатним для багатьох сфер, включаючи тестування програмного забезпечення.

Алгоритм шукає таку гіперплощину, яка максимізує відстань від найближчих точок кожного класу до гіперплощини, забезпечуючи стійкість до шуму та зменшуючи ймовірність помилкової класифікації. У випадках, коли дані не можуть бути лінійно розділені, SVM використовує ядрові функції (kernel functions), які перетворюють дані у вищий вимір для полегшення розділення. Серед популярних ядерних функцій — лінійне, поліноміальне, радіально-базове (RBF) та сигмоїдне ядра.

У тестуванні програмного забезпечення SVM знаходить широке

застосування, особливо для класифікації тестових випадків або визначення ризикових ділянок коду. Наприклад, на основі історичних даних про дефекти SVM може ідентифікувати модулі коду, які найімовірніше містять помилки. Це дозволяє тестувальникам зосередити свої зусилля на найбільш критичних компонентах системи, знижуючи витрати часу та ресурсів. Крім того, SVM використовується для аналізу тестових сценаріїв і їх пріоритизації, визначаючи, які сценарії мають найбільшу ймовірність виявлення дефектів.

SVM має кілька переваг. По-перше, алгоритм демонструє високу ефективність у роботі з великими розмірностями даних, що робить його придатним для сучасних складних систем. По-друге, він добре працює навіть у випадках, коли кількість ознак значно перевищує кількість прикладів. По-третє, ядрові функції дозволяють SVM бути надзвичайно гнучким і адаптивним для роботи з нелінійними даними.

Проте алгоритм має і свої обмеження. SVM може бути обчислювально затратним при роботі з дуже великими наборами даних, особливо якщо використовуються складні ядра. Крім того, налаштування параметрів (наприклад, вибір ядра чи коефіцієнта регуляризації) потребує значних зусиль і досвіду. Алгоритм також менш ефективний у випадках, коли дані містять багато шуму або є надзвичайно незбалансованими.

SVM широко використовується в задачах прогнозування дефектів у програмному забезпеченні. Наприклад, він може аналізувати метрики коду, такі як кількість змін у файлі, складність алгоритму або історію виправлення помилок, для класифікації файлів як "ризикові" чи "стабільні". У результаті це дозволяє автоматизувати процес визначення найбільш критичних частин системи та мінімізувати ймовірність дефектів у фінальному продукті.

Загалом, Support Vector Machines є ефективним інструментом для класифікації та прогнозування у сфері тестування програмного забезпечення. Завдяки своїй здатності працювати з нелінійними даними та забезпечувати високу точність результатів, SVM допомагає вдосконалити процеси забезпечення якості, роблячи їх швидшими та ефективнішими.

Gradient Boosting Machines (GBM)

Використовуються для створення моделей пріоритизації тестів.

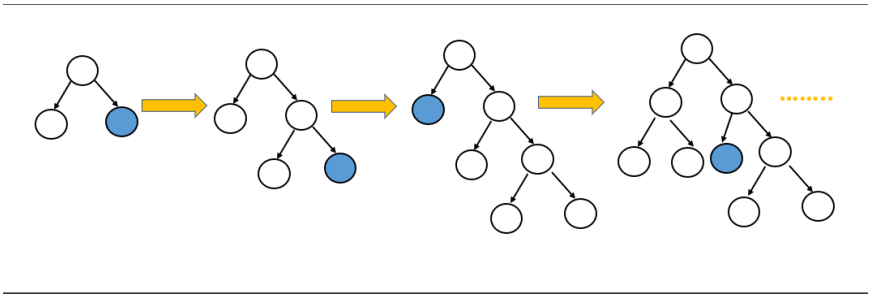

Рис.2.3. Overview of Gradient Boosting Machines

Gradient Boosting Machines (GBM) — це алгоритм машинного навчання, який широко використовується для створення високоточних моделей класифікації та регресії. GBM є методом ансамблевого навчання, який комбінує прості моделі (такі як дерева рішень) у сильнішу модель. Алгоритм ідеально підходить для задач, де потрібно визначати пріоритетність або передбачати найкритичніші об'єкти, як-от тестові сценарії у регресійному тестуванні програмного забезпечення.

GBM базується на ітеративному вдосконаленні моделі, де кожне наступне дерево в ансамблі виправляє помилки попередніх. На кожному етапі створюється дерево рішень, яке фокусується на спостереженнях, що були неправильно класифіковані раніше. Цей процес повторюється до досягнення оптимального результату або досягнення певного критерію зупинки. GBM має можливість налаштування через гіперпараметри, такі як кількість дерев, їхня глибина та швидкість навчання (learning rate), що дозволяє адаптувати модель до конкретної задачі.

Одним із ключових застосувань GBM є пріоритизація тестів у регресійному тестуванні програмного забезпечення. Алгоритм аналізує історичні дані про виконання тестів, включаючи інформацію про виявлені дефекти, покриття коду та час виконання. На основі цих даних GBM створює модель, яка визначає, які тести

найімовірніше виявлять дефекти. Це дозволяє тестувальникам зосередитися на виконанні найважливіших тестів, знижуючи загальні витрати на тестування і час виконання. Пріоритизація тестів є особливо важливою у великих проєктах із сотнями чи тисячами тестових сценаріїв, де виконання всіх тестів одночасно є неможливим через обмежені ресурси.

GBM також забезпечує високу точність у прогнозуванні, оскільки використовує механізм оптимізації втрат (loss function), що дозволяє мінімізувати помилки на кожній ітерації. Алгоритм добре працює з різними типами даних і може враховувати складні нелінійні взаємозв'язки між ознаками. Це робить його ідеальним для аналізу багатовимірних даних, таких як лог-файли тестування або складні програмні метрики.

Незважаючи на високу ефективність, GBM має деякі недоліки. Наприклад, алгоритм є обчислювально затратним через ітеративний процес навчання. Для досягнення високої точності потрібен ретельний вибір гіперпараметрів, що може бути складним і потребує великої кількості ресурсів. Крім того, GBM може бути чутливим до шуму в даних, якщо модель має недостатню регуляризацію.

Алгоритм знаходить широке застосування в інших сферах, таких як медицина, фінанси, електронна комерція та обробка тексту. Наприклад, у фінансах GBM використовується для прогнозування ризиків, у медицині — для класифікації пацієнтів за групами ризику, а в аналізі текстів — для класифікації документів.

Таким чином, Gradient Boosting Machines є інструментом для оптимізації процесів тестування програмного забезпечення, зокрема для пріоритизації тестів. Завдяки високій точності, гнучкості та здатності працювати з багатовимірними даними GBM допомагає скоротити час і витрати на тестування, підвищуючи ефективність забезпечення якості програмного забезпечення.

K-Means Clustering

Застосовується для групування схожих тестових випадків.

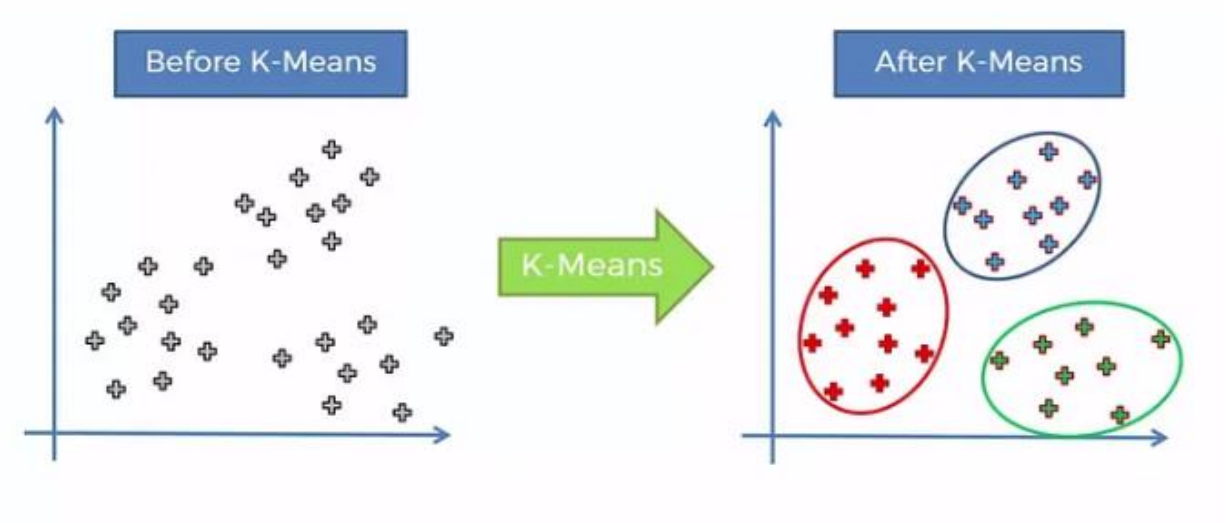

Рис.2.4. K-Means Clustering algorithm

K-Means Clustering — це популярний алгоритм машинного навчання, який використовується для задач кластеризації. Він належить до методів безконтрольного навчання і застосовується для групування подібних об'єктів у кластери на основі їхніх спільних характеристик. У тестуванні програмного забезпечення алгоритм K-Means знаходить своє застосування, зокрема, для групування схожих тестових випадків, що дозволяє оптимізувати процеси тестування та знизити витрати ресурсів.

Принцип роботи K-Means Clustering базується на визначенні центроїдів кластерів і поступовому уточненні їхніх положень для досягнення оптимального розподілу даних. Алгоритм починає роботу з випадкової ініціалізації k центроїдів (k — кількість кластерів, що задається заздалегідь). Потім для кожного об’єкта в наборі даних визначається найближчий центроїд, на основі чого об’єкти розподіляються по кластерах. Далі центроїди оновлюються на основі середніх значень об’єктів у кожному кластері. Цей процес повторюється, доки положення центроїдів не стабілізується або не досягне максимальна кількість ітерацій.

У контексті тестування програмного забезпечення K-Means Clustering може бути використаний для групування схожих тестових випадків на основі їхніх характеристик, таких як тривалість виконання, рівень покриття коду, тип перевірки

(функціональна, нефункціональна), історія результатів тестів, чи ймовірність виявлення дефектів. Це дозволяє автоматизувати процес аналізу тестів і зменшити дублювання зусиль. Наприклад, тестові сценарії з подібними характеристиками можуть бути об’єднані в один кластер, що дозволяє зосередитися на виконанні найбільш критичних тестів.

Застосування алгоритму в тестуванні має низку переваг. По-перше, це зменшує час і ресурси, необхідні для ручного аналізу великого набору тестових випадків. По-друге, кластеризація допомагає виявити дублікати або тестові сценарії, які не приносять додаткової цінності. Це дозволяє скоротити загальну кількість виконуваних тестів без втрати якості перевірки. По-третє, K-Means дозволяє адаптувати тестування до складних програмних систем, де традиційні підходи можуть бути менш ефективними.

Проте K-Means має свої обмеження. Одним із головних є необхідність задавати кількість кластерів (k) заздалегідь, що може бути складним завданням без попереднього аналізу даних. Крім того, алгоритм чутливий до вибору початкових центроїдів, і різні початкові положення можуть призводити до різних результатів. Ще одним викликом є те, що K-Means погано працює з даними, які містять нерівномірно розподілені кластери або шуми.

У сфері тестування програмного забезпечення K-Means може бути інтегрований із іншими алгоритмами машинного навчання для створення складніших моделей. Наприклад, результати кластеризації можуть використовуватися як вхідні дані для алгоритмів класифікації, таких як Random Forest чи SVM, що дозволяє створювати більш точні моделі прогнозування дефектів або пріоритизації тестів.

Загалом, K-Means Clustering є важливим інструментом для автоматизації й оптимізації процесів тестування програмного забезпечення. Його здатність групувати подібні тестові випадки дозволяє підвищити ефективність тестування, скоротити витрати ресурсів і забезпечити високу якість продукту навіть у складних системах із великою кількістю тестів.

Reinforcement Learning

Для адаптивного пріоритизації тестів залежно від ризиків.

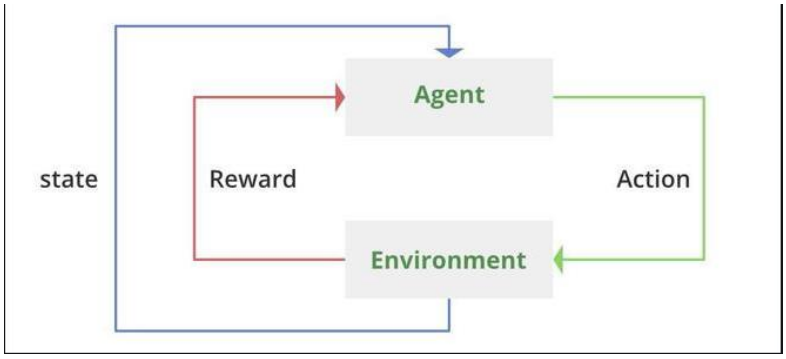

Рис.2.5 Reinforcement Learning algorithm

Reinforcement Learning (RL) є алгоритмом машинного навчання, що застосовується для адаптивної пріоритизації тестів у тестуванні програмного забезпечення, враховуючи ризики. Основна ідея RL полягає в тому, що модель, що виконує тестування, поступово навчається на основі зворотного зв'язку від своїх дій у середовищі, отримуючи винагороду чи штраф за кожну правильну чи неправильну дію. У контексті тестування програмного забезпечення середовище включає код, тестові сценарії та процес тестування, а дії агента — це вибір тестових випадків і їх пріоритизація.

Алгоритм RL адаптується в реальному часі до змін у коді та поведінці програми. Це дозволяє моделі з часом оптимізувати вибір тестів для найбільш ефективного виявлення дефектів у коді. За допомогою навчання на попередніх результатах тестування, агент визначає, які тести мають найбільший ризик виявлення помилок, і відповідно змінює пріоритети тестів.

Наприклад, коли система тестування виявляє, що певний модуль має тенденцію до частих помилок, RL може автоматично адаптувати пріоритет для тестів цього модуля, гарантуючи, що він перевіряється перед іншими компонентами. Це дозволяє зосередитись на більш критичних аспектах програми, знижуючи витрати часу і ресурсів, необхідних для виконання тестування.

Застосування RL у тестуванні програмного забезпечення забезпечує високий рівень адаптивності і гнучкості, дозволяючи системі автоматично коригувати свої стратегії пріоритизації в залежності від нових даних або змін у програмному середовищі. Це значно підвищує ефективність тестування, оскільки алгоритм з часом оптимізує вибір тестових випадків, що дозволяє тестувальникам зосередитися на найбільш важливих аспектах програми та швидше виявляти дефекти.

Попри всі переваги, використання RL може бути обчислювально затратним, оскільки вимагає значних ресурсів для навчання та коригування моделі в реальному часі. Однак цей підхід вже показав свою ефективність у зменшенні часу на тестування та підвищенні точності виявлення дефектів.

Neural Networks

Використовуються для автоматизації генерації тестових даних або для виявлення дефектів на основі складних залежностей.

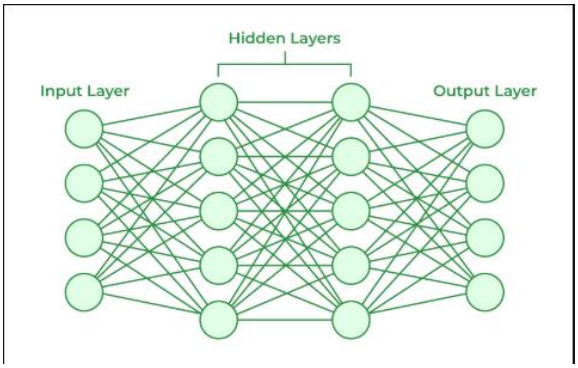

Рис.2.6 Neural Networks Architecture

Neural Networks — це математичні моделі, які імітують роботу нейронів людського мозку та здатні автоматично виявляти складні залежності в даних. Вони використовуються в різних областях, включаючи тестування програмного забезпечення, для автоматизації генерації тестових даних та виявлення дефектів. Завдяки здатності навчатися на великих обсягах даних і адаптуватися до нових ситуацій, нейронні мережі є дуже ефективними інструментами для оптимізації процесів тестування.

У тестуванні програмного забезпечення нейронні мережі часто застосовуються для створення тестових даних, що відповідають реальним сценаріям використання системи. Це особливо корисно, коли потрібно протестувати складні функціональні можливості, для яких важко або неможливо вручну створити достатньо варіантів тестових даних. Мережі можуть генерувати дані, що покривають широкий спектр можливих ситуацій, враховуючи як звичайні, так і екстремальні сценарії, тим самим підвищуючи якість тестування.

Ще одне застосування нейронних мереж — виявлення дефектів у програмному забезпеченні на основі складних залежностей, що існують між різними компонентами системи. Алгоритми нейронних мереж можуть аналізувати код і дані про помилки, виявляючи нетривіальні закономірності, які важко помітити традиційними методами тестування. Наприклад, нейронні мережі можуть виявляти дефекти, які з’являються тільки за певних умов або в результаті взаємодії різних компонентів програми, що значно підвищує точність тестування.

Нейронні мережі мають декілька переваг, включаючи здатність працювати з великими наборами даних і виявляти складні, нелінійні взаємозв’язки між різними ознаками. Вони можуть автоматично вдосконалюватися в процесі навчання, що дозволяє досягти високої точності і адаптивності. Однак їхнє використання вимагає значних обчислювальних ресурсів та великої кількості даних для навчання, а також наявності належних навичок для правильної налаштування та інтеграції в процес тестування.

Загалом, нейронні мережі дозволяють значно покращити процес тестування програмного забезпечення, зменшивши час, необхідний для генерації тестових даних, та підвищуючи точність виявлення дефектів.

Naive Bayes

Застосовується для аналізу журналів тестування та прогнозування дефектів. Наївний Баєсовий класифікатор — це один із найпростіших і найшвидших методів класифікації, який відмінно справляється з великими обсягами даних. Він успішно застосовується в багатьох завданнях, таких як фільтрація спаму, класифікація тексту, аналіз настроїв і рекомендаційні системи. Для прогнозування класів, які

неможливо визначити заздалегідь, використовується Баєсівська теорема ймовірностей.

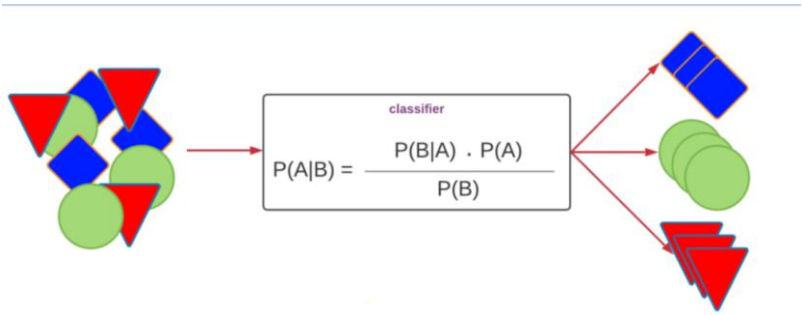

Рис.2.7. Naive Bayes Classifier

Наївний Баєсовий класифікатор — це простий, але ефективний ймовірнісний модельний підхід у машинному навчанні, який базується на впливі теореми Баєса.

Теорема Баєса — це формула, яка пропонує умовну ймовірність події A, якщо вже відбулася інша подія B. Її математична формула має вигляд:

$$P(A|B) = (P(B|A)\,P(A)) \,/\, P(B) \qquad (2.9)$$

Де:

- A і B — дві події.
- P(A|B)— ймовірність події A, за умови, що подія B вже відбулася.
- P(B|A) — ймовірність події B, за умови, що подія A вже відбулася.
- P(A) — незалежна (апріорна) ймовірність події A.
- P(B) — незалежна (апріорна) ймовірність події B.

В тестуванні програмного забезпечення алгоритм Naive Bayes може бути використаний для аналізу журналів тестування, де тестові результати записуються у вигляді подій або категорій. Кожен запис у журналі тестування може бути розглянутий як вектор ознак, і Naive Bayes дозволяє прогнозувати, чи є певні вхідні дані (наприклад, помилки, попередні результати тестів) індикатором дефекту. На основі наявних даних про попередні помилки або події, алгоритм може допомогти автоматично визначити ймовірність виникнення дефектів у майбутніх тестах або

навіть класифікувати журнали як дефектні чи бездефектні.

Naive Bayes є ефективним для прогнозування дефектів, оскільки добре працює з великими наборами даних, навіть якщо існує певний рівень шуму або невизначеності в даних. Він також добре підходить для задач, де існують великі набори вхідних ознак і де кожна ознака може бути незалежною від інших. Наприклад, алгоритм може використовувати історію виконання тестів і пов'язані з ними показники, такі як час виконання, тип помилки, модуль, на якому тестувалося програмне забезпечення, для визначення, які тести ймовірніше за все приведуть до дефектів.

Переваги Naive Bayes включають простоту реалізації та швидкість обробки даних, навіть якщо обсяг даних великий. Оскільки цей алгоритм не вимагає значних обчислювальних ресурсів і здатний ефективно працювати з пропущеними або неповними даними, він є популярним вибором для завдань з аналізу журналів тестування.

Однак Naive Bayes має свої обмеження. Наприклад, його припущення про незалежність ознак не завжди відповідає реальним умовам, оскільки деякі ознаки можуть бути залежними між собою. Це може знижувати точність класифікації у складних або сильно залежних даних.

Таким чином, Naive Bayes є корисним і швидким методом для прогнозування дефектів і аналізу журналів тестування, особливо в умовах великих і складних даних, де інші більш складні методи можуть бути занадто обчислювально затратними.

Logistic Regression

Для прогнозування ймовірності дефектів у модулі.

Логістична регресія — це статистична модель, яка часто використовується для аналітики прогнозування та класифікації. Вона розраховує ймовірність події (наприклад, голосування чи не голосування) на основі набору незалежних змінних. Оскільки результат є ймовірністю, діапазон залежної змінної — від 0 до 1.

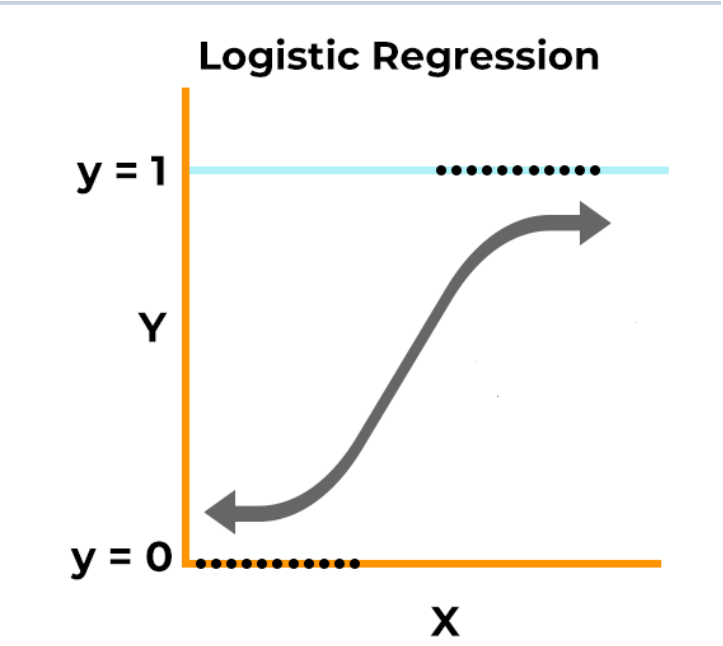

Рис.2.8. Key Assumptions for Implementing Logistic Regression

У логістичній регресії шанси (ймовірність успіху, поділена на ймовірність невдачі) трансформуються за допомогою логарифмічної функції (logit). Формули для логістичної функції виглядають так:

- $Logit\,(\pi) = 1 / (1 + exp\,(-\pi))$ (2.1)
- $ln\,(\pi / (1 - \pi)) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$ (2.2)

Logit(pi) є залежною або відповідною змінною у цьому рівнянні, тоді як X — незалежна змінна.

Найпоширенішим методом оцінювання параметра Beta у цій моделі є оцінювання максимальної правдоподібності (MLE). Цей підхід ітеративно оцінює різні значення Beta для знаходження найкращого співвідношення логарифмічних шансів.

Після визначення найкращого коефіцієнта (або коефіцієнтів, якщо є декілька незалежних змінних) можна обчислити умовні ймовірності для кожного спостереження, зареєструвати їх і скласти разом, щоб отримати прогнозовану ймовірність.

Критерій бінарної класифікації:

- Ймовірність < 0.5 прогнозує 0.

- Ймовірність > 0.5 прогнозує 1.

Після обчислення моделі рекомендується оцінити її якість (goodness of fit) або те, наскільки добре вона прогнозує залежну змінну. Тест Хосмера-Лемешоу є популярним методом для оцінки відповідності моделі.

Deep Neural Networks (DNN)

Для складних аналізів та прогнозів, включаючи тестування програмного забезпечення.

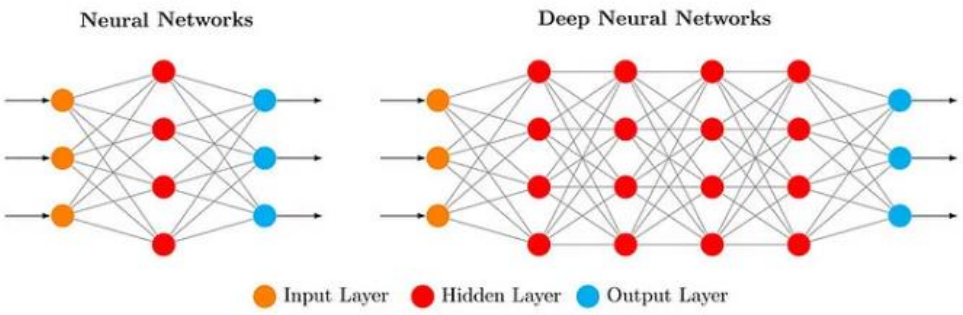

Рис.2.9. Comparison of Deep Neural Networks and Neural Networks

Deep Neural Networks (DNN) — це тип нейронних мереж, який складається з численних шарів нейронів, що дозволяє моделі виявляти складні та нелінійні залежності в даних. Вони є важливим інструментом у машинному навчанні, особливо для складних аналізів та прогнозів, де традиційні методи можуть бути менш ефективними. У контексті тестування програмного забезпечення DNN застосовуються для виконання високоточних прогнозів, виявлення дефектів та оптимізації тестових процесів.

DNN складаються з кількох шарів: вхідний шар, один або кілька прихованих шарів і вихідний шар. Кожен шар складається з безлічі нейронів, які обробляють інформацію та передають її далі. Це дозволяє DNN ефективно працювати з великими обсягами даних, автоматично виявляючи важливі патерни та закономірності без явного програмування кожної окремої ознаки.

Одним із основних застосувань DNN у тестуванні програмного забезпечення є виявлення дефектів. Завдяки здатності нейронних мереж виявляти складні,

нелінійні залежності, вони можуть аналізувати величезні обсяги даних про програму — включаючи журнал помилок, метрики коду, попередні тестові випадки — і на основі цього прогнозувати, які частини коду з найбільшою ймовірністю викличуть дефекти в майбутньому. Цей підхід значно знижує необхідність у ручному аналізі і дозволяє зосередитись на найбільш критичних частинах коду.

У тестуванні програмного забезпечення DNN також використовуються для автоматизації створення тестових даних. Нейронні мережі здатні генерувати нові тестові випадки на основі існуючих даних, що дозволяє покрити більше сценаріїв використання програми та знизити ймовірність пропуску важливих тестів. Наприклад, для складних веб-додатків, де може бути багато можливих варіантів введення та взаємодій користувача, DNN можуть генерувати тестові дані, що точно відповідають реальним сценаріям використання, враховуючи широкий спектр умов, включаючи екстремальні ситуації.

Іншим важливим застосуванням DNN є прогнозування та пріоритизація тестів. Замість того щоб виконувати всі тести, DNN можуть прогнозувати, які з тестів є найбільш ймовірними для виявлення дефектів, і таким чином дозволяють зосередити ресурси на найбільш критичних аспектах програми. Вони можуть також бути використані для класифікації тестових сценаріїв або визначення, які з тестів вже пройшли в минулому, і які слід виконати заново після змін у коді.

DNN можуть бути дуже ефективними, особливо коли мова йде про складні програми або великі проекти з багатьма модулями, оскільки ці моделі здатні працювати з великими наборами даних та враховувати багатофакторні залежності. Вони також добре підходять для роботи з неструктурованими даними, такими як текстові або зображення, що є важливими в контексті тестування програм, де лог-файли, звіти про помилки або навіть скріншоти можуть використовуватися для виявлення дефектів.

Проте, варто відзначити, що DNN є обчислювально складними і потребують значних ресурсів для навчання. Також процес навчання може бути часозатратним, особливо коли йдеться про великий обсяг даних, і необхідність налаштування великої кількості гіперпараметрів може ускладнити процес інтеграції таких

моделей у систему тестування. Враховуючи це, одним із важливих аспектів є наявність достатнього обсягу даних для навчання моделі і її регулярне оновлення для забезпечення актуальності прогнозів.

Загалом, DNN мають великий потенціал для оптимізації тестування програмного забезпечення. Вони здатні підвищити точність тестування, знизити витрати часу та ресурсів, а також забезпечити гнучкість у роботі з складними або новими програмними системами, що робить їх незамінними у багатьох випадках.

Convolutional Neural Networks (CNN)

Застосовується у візуальних задачах, може адаптуватися для тестування GUI-додатків.

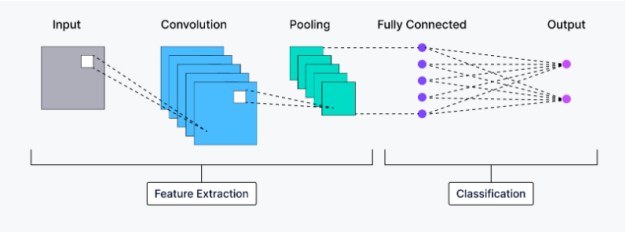

Рис.2.10. The Architecture of Convolutional Neural Networks

Convolutional Neural Networks (CNN) є одними з найпоширеніших моделей у галузі глибокого навчання, які застосовуються для аналізу візуальних даних. CNN ефективно виявляють патерни в зображеннях, що робить їх ідеальними для задач, пов'язаних з обробкою зображень, відео та іншими візуальними даними. Завдяки своїй здатності автоматично виділяти важливі характеристики на різних рівнях зображення, CNN можуть бути адаптовані для тестування графічних інтерфейсів користувача (GUI) в програмному забезпеченні, зокрема для автоматичного виявлення помилок в інтерфейсі, перевірки відображення елементів на різних пристроях та браузерах.

Принцип роботи CNN ґрунтується на застосуванні згорток (convolutions), які дозволяють виділяти важливі ознаки на зображеннях, такі як лінії, краї та текстури. Згорткові шари застосовуються до входу зображень, і кожен шар поступово виявляє дедалі більш складні патерни. Наприклад, початкові шари можуть виявляти прості контури, тоді як глибші шари можуть виявляти складніші форми,

такі як обличчя, об'єкти або сцени. Завдяки такому підходу, CNN можуть ефективно працювати з великими наборами візуальних даних, автоматично знаходячи важливі патерни, що можуть бути використані для тестування програмного забезпечення.

У контексті тестування графічних інтерфейсів користувача (GUI), CNN можуть бути використані для автоматичного виявлення дефектів в інтерфейсі додатку. Це може включати перевірку коректності розташування елементів інтерфейсу, таких як кнопки, поля вводу, тексти або зображення. Застосовуючи CNN, можна перевірити, чи правильно відображаються ці елементи на різних екранах і в різних браузерах. Це дозволяє автоматизувати тестування візуальних аспектів програм, що є трудомістким і часто схильним до людських помилок при ручному тестуванні. CNN також допомагають визначити, чи змінюються елементи інтерфейсу після внесення змін у код, що дозволяє швидко виявляти будь-які непередбачувані помилки в дизайні або функціональності.

Адаптація CNN для тестування GUI включає використання специфічних архітектур, які можуть працювати з багатоканальними вхідними даними, що є особливо важливим для візуальних тестів, де можуть бути використані зображення різних частин екрана. CNN можуть порівнювати візуальні елементи на екрані з еталонними зображеннями, що дозволяє виявити помилки або зміни, які можуть виникнути через зміни в коді чи інші фактори. Цей підхід особливо корисний для тестування веб-додатків і мобільних додатків, де є потреба в тестуванні на різних пристроях і браузерах, що мають різні характеристики екранів та підтримку технологій.

Однією з основних переваг CNN є їх здатність працювати з неструктурованими даними, такими як зображення, що дозволяє значно автоматизувати процес тестування. Завдяки здатності CNN виявляти навіть найдрібніші відмінності між елементами інтерфейсу, вони можуть бути використані для тестування не тільки відображення елементів, але й для перевірки їхньої функціональності, наприклад, перевірка клікабельності кнопок чи коректності відображення тексту.

Проте, використання CNN має деякі обмеження. Вони потребують великих обсягів розмічених даних для навчання, що може бути складним і затратним процесом. Крім того, процес навчання CNN є обчислювально інтенсивним, і для досягнення високої точності часто потрібні значні обчислювальні ресурси. Також можуть виникати проблеми з надмірним навчанням, якщо дані не є достатньо різноманітними або представлені в неправильному форматі.

Однак, незважаючи на ці обмеження, Convolutional Neural Networks є дуже ефективними для тестування графічних інтерфейсів користувача, допомагаючи автоматизувати перевірку візуальних елементів додатків, знижувати витрати на тестування та підвищувати точність виявлення помилок. Завдяки високій здатності до адаптації та аналізу великих обсягів візуальних даних, CNN можуть значно покращити процес тестування програмного забезпечення, забезпечуючи більшу стабільність і якість кінцевих продуктів.

Recurrent Neural Networks (RNN)

Використовуються для роботи з послідовними даними, наприклад, аналізу логів тестування.

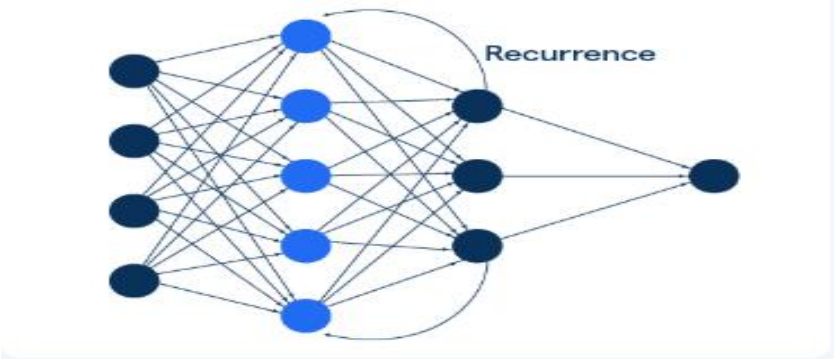

Рис.2.11. Recurrent Neural Networks

Recurrent Neural Networks (RNN) — це тип нейронних мереж, який здатний ефективно працювати з послідовними даними, де важлива залежність між елементами послідовності. Ця особливість робить RNN ідеальними для таких завдань, як аналіз журналів тестування, де необхідно враховувати послідовність виконання тестів і пов'язані з ними результати. Важливість цієї технології для

тестування програмного забезпечення полягає в тому, що журнали тестування, які містять записи про виконані тести, помилки та їхні взаємозв'язки, можуть бути важливим джерелом інформації для прогнозування дефектів у програмному забезпеченні.

Особливість роботи RNN полягає в тому, що вони можуть «пам’ятати» попередні стани, що дозволяє мережі враховувати минулі значення при прогнозуванні майбутніх. У класичних нейронних мережах кожен елемент обробляється незалежно від попереднього, тоді як у RNN виходи одного шару можуть впливати на входи наступного, що дозволяє зберігати контекст. Це дає змогу моделі ефективно працювати з послідовними даними, такими як журнали тестування, де один запис або результат тесту може залежати від попереднього.

У тестуванні програмного забезпечення RNN використовуються для аналізу даних, що містять послідовність тестових результатів або повідомлень про помилки. Завдяки своїй здатності зберігати історію тестів, ці мережі можуть не лише виявляти помилки, а й прогнозувати їх на основі аналізу даних з попередніх тестувань. Наприклад, RNN може виявити закономірності в тестах, що виконувалися раніше, і передбачити, де у наступних версіях коду можуть виникнути дефекти.

Використання RNN у тестуванні програмного забезпечення дає змогу не тільки автоматизувати процес аналізу журналів тестування, але й забезпечити гнучкість у прогнозуванні ризикових ділянок коду. Мережа може використовувати попередні тести для прогнозування майбутніх дефектів, що дозволяє тестувальникам більш ефективно планувати свої дії і фокусуватися на найбільш критичних ділянках коду, що має високу ймовірність помилок. Враховуючи, що журнал тестування часто містить багато звітів про виконання тестів, застосування RNN для автоматичної обробки цих даних дозволяє значно знизити навантаження на тестувальників.

Одна з важливих переваг RNN полягає в їхній здатності працювати з великими наборами даних і враховувати залежності, що виникають у процесі виконання тестів. Наприклад, якщо в результаті одного тесту виникає помилка в

певному модулі, це може вплинути на виконання наступних тестів. RNN дозволяють прогнозувати, як ці помилки можуть вплинути на інші частини програмного коду. Це дозволяє зосередити зусилля тестувальників на тих частинах коду, які мають найбільший ризик виникнення дефектів, на основі історії попередніх тестів.

Проте, для ефективного використання RNN необхідно забезпечити наявність достатньої кількості даних для навчання моделі. Це може бути складно, особливо в умовах обмежених ресурсів або коли дані є неповними. Крім того, RNN потребують значних обчислювальних ресурсів для навчання, особливо коли йдеться про великі обсяги тестових даних. Тим не менше, використання RNN в тестуванні програмного забезпечення надає величезний потенціал для автоматизації процесів тестування, підвищення точності виявлення дефектів та оптимізації використання ресурсів.

Загалом, RNN є ефективним інструментом для аналізу журналів тестування та прогнозування дефектів у програмному забезпеченні. Вони дозволяють моделі автоматично вивчати закономірності та залежності в даних, що допомагає тестувальникам більш точно оцінювати ризики та фокусуватися на найбільш важливих аспектах програми. Це не лише оптимізує тестування, але й дозволяє значно знизити витрати часу та ресурсів.

K-Nearest Neighbors (KNN)

Простий у реалізації алгоритм для класифікації тестових випадків.

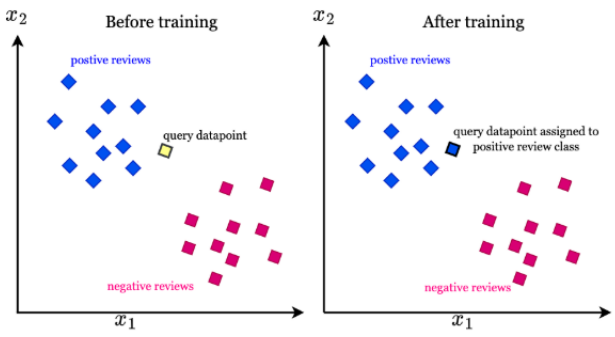

Рис.2.12. Use of KNN algorithm

Nearest Neighbors (KNN) — це один з найбільш простих і водночас ефективних алгоритмів машинного навчання, який широко застосовується для задач класифікації, зокрема для класифікації тестових випадків у тестуванні програмного забезпечення. Суть алгоритму полягає в тому, що для кожного нового тестового випадку він порівнюється з уже наявними в навчальній вибірці на основі визначеної метрики відстані, наприклад, евклідової відстані. Клас тесту визначається за допомогою найближчих "сусідів", що дозволяє створити на основі минулого досвіду (наприклад, результатів виконаних тестів) передбачення для нових даних.

Принцип роботи KNN можна пояснити так: кожен тестовий випадок (або зразок) у навчальній вибірці має певні характеристики (ознаки). Коли з'являється новий тест, алгоритм порівнює його з іншими тестами, що вже є у вибірці, і визначає клас для нового тесту на основі найближчих до нього тестових випадків. Чим менша відстань між новим тестом і іншими з навчальної вибірки, тим більша ймовірність, що новий тест матиме той самий клас, що й найближчі до нього сусіди. В результаті класифікації ми отримуємо, до якого класу належить новий тестовий випадок — наприклад, "дефектний" або "бездефектний".

У тестуванні програмного забезпечення KNN використовується для визначення того, які тести найбільш ймовірно призведуть до виявлення дефектів. У разі внесення змін до коду тестування знову проходить на основі раніше виконаних тестів, які мають подібні ознаки, такі як типи тестованих функцій, частоти помилок, історія змін тощо. KNN може порівнювати нові тестові випадки з попередніми та передбачати, де з найбільшою ймовірністю виникнуть помилки, що дає змогу тестувальникам зосередитись на найбільш ризикованих ділянках коду. Це дозволяє скоротити витрати часу та ресурсів, оскільки не потрібно виконувати всі можливі тести, а лише ті, що мають більшу ймовірність виявлення дефектів.

Застосування KNN дозволяє автоматизувати процес класифікації тестів, що значно підвищує ефективність тестування в умовах великих проектів, де є велика кількість тестових сценаріїв. Наприклад, при великій кодовій базі можна

здійснювати вибір тестів на основі найближчих до них за характеристиками тестів з попередніх релізів, що зменшує навантаження на тестувальників і дозволяє більше зосередитись на перевірці критичних компонентів.

Основною перевагою KNN є його простота у реалізації та відсутність необхідності в попередньому навчанні. Алгоритм не потребує побудови складних моделей, а працює безпосередньо на даних, що значно полегшує процес інтеграції в існуючі системи тестування. Крім того, KNN може бути використаний для розв'язання широкого спектру задач без необхідності складних налаштувань, що робить його універсальним інструментом.

Однак є й обмеження, пов’язані з використанням KNN. Оскільки алгоритм залежить від обчислення відстаней між всіма парами точок, при великих наборах даних він може бути повільним, що значно збільшує час обчислень. Це особливо критично при тестуванні великих програмних систем, де кількість тестових випадків може бути дуже великою. Крім того, KNN чутливий до вибору метрики відстані, оскільки результат класифікації може значною мірою залежати від того, як саме вимірюється відстань між тестами. Важливим аспектом є також необхідність нормалізації даних, оскільки алгоритм схильний до впливу різних масштабів ознак.

Також KNN має проблему з перенавчанням, якщо вибірка навчальних даних не є репрезентативною або коли є дуже багато "шуму" в даних. Це може призвести до того, що алгоритм буде приймати некоректні рішення, класифікуючи нові тестові випадки невірно.

K-Nearest Neighbors — це потужний і простий в реалізації інструмент для класифікації тестових випадків у тестуванні програмного забезпечення. Його застосування дозволяє значно скоротити час тестування, зменшити витрати ресурсів і підвищити точність виявлення дефектів, зосереджуючи увагу на найбільш важливих ділянках коду. Хоча алгоритм має деякі обмеження, пов’язані з великою кількістю даних та вибором метрики відстані, його простота і універсальність роблять його одним з найкращих варіантів для автоматизації тестування в ряді програмних систем.

AdaBoost і XGBoost

Для оптимізації класифікаційних задач та роботи з великими даними. AdaBoost та XGBoost є двома алгоритмами ансамблювання, які використовуються для класифікації та роботи з великими даними. Вони обидва працюють за принципом комбінування кількох слабких моделей для досягнення кращих результатів. Основна ідея AdaBoost полягає в тому, щоб послідовно покращувати слабкі класифікатори, зосереджуючи увагу на тих даних, які були неправильно класифіковані попередніми моделями. Кожен новий класифікатор вносить корекції у помилки попередніх, що дозволяє значно підвищити точність класифікації. Алгоритм присвоює більшу вагу тим елементам, які були неправильно класифіковані, що забезпечує покращення моделі на більш складних випадках. Оскільки AdaBoost адаптивно покращує точність за допомогою багатьох слабких моделей, він є потужним інструментом для роботи з класифікаційними задачами, де є різноманітність класів і важливість точності для виявлення дефектів.

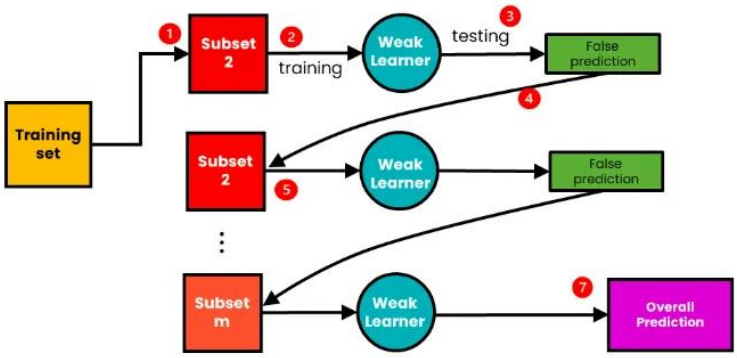

Рис.2.13. The Process of Boosting

У свою чергу, XGBoost є більш вдосконаленим варіантом Gradient Boosting, що активно використовується для класифікації та регресії. XGBoost забезпечує більш високу точність і швидкість навчання завдяки використанню регуляризації та паралельної обробки, що особливо важливо для великих наборів даних. Його ефективність у зменшенні похибок робить його популярним інструментом на практиці для обробки великих обсягів даних. Крім того, XGBoost є адаптивним до

змін у даних, що робить його ідеальним для задач тестування програмного забезпечення, де необхідно враховувати великий обсяг інформації про помилки і забезпечувати ефективну класифікацію тестових випадків. Однією з головних особливостей XGBoost є здатність адаптувати модель до великих наборів даних, використовуючи оптимізацію та регуляризацію для запобігання перенавчанню.

Обидва алгоритми, AdaBoost і XGBoost, використовуються для оптимізації класифікаційних задач в тестуванні програмного забезпечення, де важливо точно класифікувати тестові випадки, щоб виявити дефекти на ранніх етапах. Наприклад, AdaBoost може бути застосований для покращення точності класифікації тестів, що дозволяє зменшити кількість тестів і підвищити ефективність, зосереджуючи увагу на найбільш ймовірних дефектах. XGBoost, у свою чергу, ефективно працює з великими даними і може використовуватись для прогнозування ймовірних дефектів на основі аналізу великого обсягу тестових результатів. Використання обох алгоритмів дозволяє знизити час, витрачений на тестування, підвищуючи при цьому точність виявлення дефектів і покращуючи загальну ефективність тестування програмного забезпечення.

Clustering Algorithms DBSCAN

Для сегментації тестових сценаріїв.

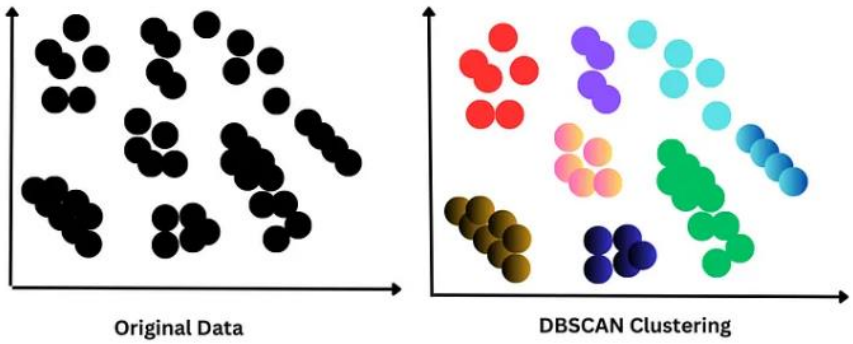

Рис.2.14. Density-Based Spatial Clustering of Applications with Noise

Алгоритми кластеризації, такі як DBSCAN (Density-Based Spatial Clustering of Applications with Noise), є методами, що використовуються для сегментації

тестових сценаріїв, особливо коли необхідно згрупувати схожі тестові випадки на основі їхніх характеристик. Ці алгоритми часто застосовуються в ситуаціях, коли тестові випадки не мають попередніх міток і потрібно зрозуміти їх взаємозв'язок або схожість.

DBSCAN зокрема є ефективним для задач кластеризації, коли дані містять шум або викиди, що є типовим для реальних тестувальних сценаріїв. На відміну від інших алгоритмів кластеризації, таких як K-means, які вимагають попереднього визначення кількості кластерів, DBSCAN визначає кластери на основі щільності точок даних. Це робить його надзвичайно корисним у випадках, коли дані можуть не мати чіткої або рівномірної структури. Завдяки здатності обробляти шум і викиди, DBSCAN є корисним інструментом для сегментації тестових випадків у тестуванні програмного забезпечення, особливо коли тестові випадки мають різні метрики, такі як час виконання, частота помилок або інші показники ефективності.

Перевагою кластеризаційних алгоритмів у контексті тестування програмного забезпечення є їх здатність автоматично визначати групи тестів, які можуть мати схожі характеристики або поведінку. Застосовуючи DBSCAN або інші алгоритми кластеризації, можна легко ідентифікувати тестові випадки, які пов'язані між собою функціонально або за типом помилок. Наприклад, тестові випадки, що перевіряють подібні модулі програмного забезпечення, або тестові випадки, що призводять до схожих видів помилок, можуть бути згруповані разом. Ця сегментація дозволяє оптимізувати процес тестування, фокусуючи зусилля на найбільш важливих або схильних до помилок ділянках програми.

Кластеризація також може бути корисною для оптимізації процесу регресійного тестування, де нові тести повинні бути пріоритизовані залежно від їх здатності виявляти раніше зустрічані дефекти або перевіряти взаємодії між уже перевіреними компонентами і новими змінами в коді. Завдяки тому, що алгоритми кластеризації можуть визначати, які тестові випадки є схожими або взаємопов'язаними, можна зменшити кількість тестів, що виконуються, фокусуючи увагу лише на найбільш релевантних випадках. Це дозволяє значно заощадити час і ресурси на тестування.

Таким чином, DBSCAN та інші алгоритми кластеризації є ефективним підходом для сегментації тестових випадків, виявлення патернів у даних та підвищення ефективності тестування, дозволяючи групувати схожі тести і оптимізувати роботу з ними. Ці методи можуть бути інтегровані з іншими техніками машинного навчання та аналізу даних, щоб покращити загальну ефективність і результативність тестування програмного забезпечення.

Generative Adversarial Networks (GANs)

Генерація тестових даних для симуляції користувацької поведінки.

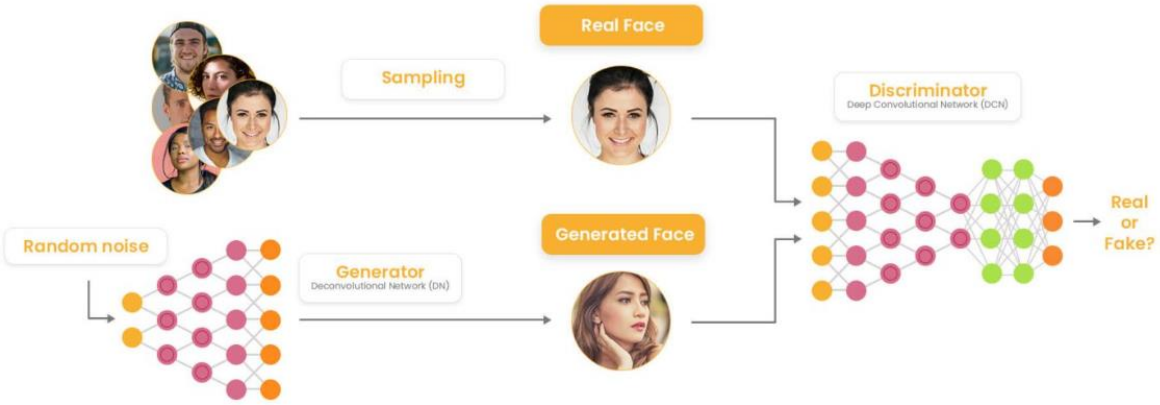

Рис.2.15. Simple Architecture of a GAN

Generative Adversarial Networks (GANs) — це клас алгоритмів машинного навчання, який здобув велику популярність завдяки своїй здатності генерувати нові, реалістичні дані, що схожі на дані з навчальної вибірки. В основі GAN лежить ідея двох нейронних мереж, що працюють змагально: генератор, який створює нові дані, і дискримінатор, що оцінює їх реалістичність. Генератор намагається створити такі дані, які будуть виглядати максимально схоже на реальні, а дискримінатор намагається відрізнити ці дані від справжніх. Обидві мережі навчаються одночасно, вдосконалюючи свої функції, що призводить до створення високоякісних синтетичних даних.

GANs можуть бути ефективно використані в тестуванні програмного забезпечення для генерації тестових даних, які імітують реальні сценарії

користувацької поведінки. Оскільки для тестування програмного забезпечення необхідно створити різноманітні та реалістичні дані, які відповідають реальним умовам використання додатка, GANs дозволяють автоматично генерувати такі дані без необхідності вручну створювати кожен тестовий випадок.

Для тестування веб-додатків або мобільних застосунків, GANs можуть бути використані для симуляції поведінки користувачів, що дозволяє перевірити, як система поводиться за різних умов. Наприклад, генератор може створювати трафік, схожий на реальний, включаючи запити до сервера, ввід даних користувачем, натискання кнопок або інші дії, що виконуються в процесі використання додатка. Дискримінатор, в свою чергу, оцінює ці дані, щоб переконатися, що вони дійсно схожі на реальні, надаючи таким чином можливість для перевірки працездатності програми під навантаженням.

Використання GANs дозволяє створювати великі обсяги даних для тестування без потреби у зборі реальних даних від користувачів, що є особливо важливим у випадках, коли таких даних неможливо зібрати через обмеження конфіденційності або обмеження доступу. Генерація тестових даних за допомогою GANs дозволяє створювати умови, які наближаються до реальних сценаріїв, зменшуючи ймовірність помилок, які можуть виникнути під час фактичного використання додатка.

Generative Adversarial Networks (GANs) забезпечують нові можливості для створення реалістичних тестових даних і є важливим інструментом для оптимізації процесу тестування програмного забезпечення. Вони допомагають автоматизувати процес створення тестів, зменшити витрати часу на підготовку тестових даних та покращити якість тестування за рахунок більш реалістичних і різноманітних сценаріїв, що імітують реальне використання програмного забезпечення. Principal Component Analysis (PCA)

PCA — це лінійний метод зменшення розмірності, який базується на техніках проєкції. Завдяки застосуванню PCA високовимірний евклідів простір проєктується у простір із нижчою розмірністю. Дано матрицю даних X, цільовий простір Y та проєкційну матрицю P. PCA описується наступним відображенням:

$$Y = PX$$

Ряди P утворюють нову базу для X. Оптимізаційна задача (максимізація варіації, зменшення коваріації між змінними) вирішується шляхом пропозиції сингулярного розкладу для розв'язання рівняння. Коваріаційна матриця C виражається як:

$$C = (1 / (n - 1)) PXX^{T}P^{T}$$

Через зміну базису у ортогональних базах та високу варіацію елементів, результуюче усічення дозволяє здійснити перехід до простору з меншою розмірністю, зберігаючи при цьому велику частину варіації вихідного набору даних.

Linear Discriminant Analysis (LDA)

Лінійний дискримінантний аналіз (LDA) є поширеним методом контрольованого зменшення розмірності. LDA досягає оптимального лінійного перетворення W, яке одночасно зменшує відстань між точками одного класу та збільшує відстань між класами. Критерій J(XW), який максимізується:

$$J(XW) = -LDA(XW) = -(W^{T}S^{B}W) / (W^{T}S^{B}W)$$

Де SB — матриця розсіювання між класами, а SW — матриця розсіювання в межах класу, які визначаються як:

$$SB = \sum(\mu c - \bar{x})(\mu c - \bar{x})^{T}$$
$$SW = \sum_{(c)} \sum_{(xi \in c)} (x_i - \mu c)(x_i - \mu c)^{T}$$

Де $\bar{x}$ — середнє значення всіх точок даних X, а μc — середнє значення точок даних, які належать до класу c.

Квадратичний дискримінантний аналіз (QDA)

Квадратичний дискримінантний аналіз (QDA) дуже схожий на лінійний дискримінантний аналіз (LDA), за винятком того, що припущення про рівність середнього та коваріаційної матриці для всіх класів було послаблено. Таким чином, для QDA потрібно обчислювати ці параметри окремо для кожного класу.

QDA відрізняється від LDA тим, що передбачається, що коваріаційна матриця може бути різною для кожного класу. У зв'язку з цим, ми оцінюємо коваріаційну матрицю Σ_k окремо для кожного класу k, де k=1,2,…,K.

Функція квадратичного дискримінанту:

$$\delta_k(x) = -½\, log\, |\Sigma_k| - ½\, (x - \mu_k)^T\, \Sigma_k^{-1}\, (x - \mu_k) + log\, \pi_k$$

2.2 Алгоритми оптимізації тестових сценаріїв

Оптимізація — це процедура знаходження та порівняння можливих рішень до моменту, поки не знайдеться краще рішення, або визначається як пошук оптимальних рішень серед набору альтернатив для отримання Парето-оптимального набору. У повсякденному житті люди використовують оптимізацію, часто навіть не усвідомлюючи цього, для простих речей, таких як подорож із одного місця в інше чи управління часом, а також для важливих рішень, таких як пошук найкращого поєднання навчання, роботи та інвестицій. Так само оптимізація знаходить багато застосувань в інженерії, науці, бізнесі, економіці тощо, за винятком того, що в цих сферах використовуються кількісні моделі та методи, на відміну від якісної оцінки вибору в повсякденному житті. Без оптимізації дизайну та операцій виробничі й інженерні діяльності не були б такими ефективними, як вони мають бути. Навіть у такому випадку все ще існує можливість оптимізації поточних промислових операцій, особливо в умовах постійно мінливого економічного, енергетичного та екологічного середовища [42].

У цій роботі методи оптимізації можна розділити на метаевристичні методи та традиційні методи. Традиційні методи гарантують знаходження оптимального рішення та доведення його оптимальності для кожного випадку задачі комбінаторної оптимізації з кінцевим розміром у залежності від часу виконання конкретного випадку. Метаевристика визначається як набір алгоритмічних концепцій, які можуть бути використані для визначення евристичних методів, що застосовуються до широкого спектру різних задач. Іншими словами, метаевристика може розглядатися як універсальний евристичний метод, призначений для

спрямування специфічної для задачі евристики (наприклад, алгоритму локального пошуку або евристики побудови) до перспективних областей пошукового простору, що містять високоякісні рішення. Таким чином, метаевристика є загальною алгоритмічною структурою, яка може бути застосована до різних задач оптимізації з відносно невеликими модифікаціями для їх адаптації до конкретної задачі. Метаевристики часто використовують досвід, накопичений під час попередніх пошуків (пам’ять), щоб спрямовувати нові пошуки. Метаевристики також можуть використовувати знання, специфічні для конкретної області, у формі евристик, що контролюються стратегією верхнього рівня. Приклади метаевристик включають генетичний алгоритм, ройову інтелектуальність, таку як оптимізація рою частинок, оптимізацію світлячків, алгоритм водяного циклу тощо. Усі метаевристики мають спільне те, що вони намагаються уникнути створення низькоякісних рішень, вводячи загальні механізми, які розширюють специфічні для задачі алгоритми одноразового виконання, такі як жадібні евристики побудови або ітеративне покращення локального пошуку. Відмінності між доступними метаевристиками стосуються методик, які використовуються для уникнення застрягання у субоптимальних рішеннях, і типу траєкторії, яку вони слідують у просторі часткових або повних рішень.

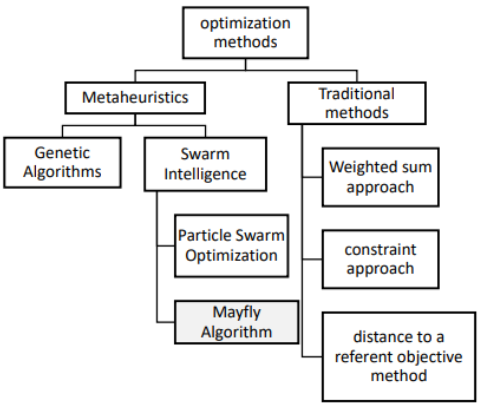

Рис.2.16. Методи Оптимізації

2.2.1 Традиційні методи

Існує кілька методів, які можуть бути використані для розв'язання багатокритеріальних задач за допомогою однокритеріальних апроксимацій. Одним із них є програмування цілей, яке спрямоване на пошук компромісного рішення, базуючись на відносній важливості кожної цілі. Існують два методи розв'язання програмування цілей: метод вагового сумування та метод пріоритетності. Ці два методи базуються на перетворенні кількох цілей в одну функцію. Обидва ці методи намагаються знайти оптимальний Парето-фронт, використовуючи різні методи апроксимації. У цьому розділі ми представимо підхід вагового сумування як один із традиційних методів.

2.2.2 Метаевристичні методи

Метаевристики вважаються високорівневими алгоритмічними стратегіями, які використовуються для спрямування інших евристик або алгоритмів у їхньому пошуку простору допустимих рішень до оптимального значення (у випадку однокритеріальної задачі) або до набору оптимальних значень (у випадку багатокритеріальної задачі). Усі метаевристики мають спільну рису: вони намагаються уникати створення низькоякісних рішень, впроваджуючи загальні механізми, що розширюють специфічні для задачі підходи. Різниця між доступними метаевристиками полягає у техніках, які використовуються для уникнення застрягання у субоптимальних рішеннях, та у типі траєкторії, якою вони слідують у просторі часткових або повних рішень.

На практиці використовуються кілька правил: максимальний час виконання на ЦП, максимальна кількість згенерованих рішень, відсоткове відхилення від верхньої/нижньої межі оптимуму та максимальна кількість ітерацій без покращення якості рішення. Ці техніки виконують наступні функціональні завдання.

Для розв'язання багатокритеріальних задач оптимізації багато традиційних методів скаляризують вектор цілей в одну ціль. У таких випадках отримане рішення сильно залежить від вагового вектора, що використовується у процесі

скаляризації, і вимагає від користувача знання про базову проблему. Крім того, при розв'язанні багатокритеріальних задач дизайнери можуть бути зацікавлені у наборі Парето-оптимальних точок, а не в одній точці.

Еволюційні алгоритми (EAs) — це алгоритми, які базуються на еволюції видів; зазвичай вони базуються на основній еволюційній теорії Чарльза Дарвіна. Хоча механізми еволюції реалізуються по-різному, основна ідея всіх цих варіацій є подібною. Еволюційні алгоритми характеризуються існуванням популяції індивідуумів, які піддаються впливу навколишнього середовища, що призводить до природного відбору, тобто виживання найбільш пристосованих, і, у свою чергу, до збільшення середнього рівня пристосованості популяції.

Пристосованість (fitness) є мірою ступеня адаптації організму до навколишнього середовища; чим більше значення пристосованості, тим краще організм пристосований до середовища. У загальному випадку еволюційні алгоритми зосереджуються лише на підмножині механізмів, визначених у біологічному еволюційному процесі. Оскільки еволюційні алгоритми (EAs) працюють із популяцією точок, здається природним використовувати EAs у багатокритеріальних задачах оптимізації для одночасного отримання кількох рішень. У цьому розділі ми представимо чотири алгоритми з усіма їхніми деталями. Два нових гібридних алгоритми PSO-RS та MA-RS будуть описані у п'ятому розділі.

Генетичні алгоритми (GA) були винайдені для імітації деяких процесів, що спостерігаються у природній еволюції. Генетичний алгоритм (GA) є метаевристикою, натхненною процесом відбору, яка належить до більшого класу еволюційних алгоритмів (EAs). Генетичні алгоритми зазвичай використовуються для створення високоякісних рішень у задачах оптимізації та пошуку, спираючись на біоінспіровані оператори, такі як мутація, схрещування та відбір.

Алгоритм оптимізації рою частинок (PSO) був запропонований Джеймсом Кеннеді та Расселом Еберхартом у 1995 році. PSO є одним із найбільш використовуваних еволюційних алгоритмів. Він мотивований соціальною поведінкою організмів, таких як зграї птахів або косяки риб. Алгоритм PSO,

роблячи коригування на основі "локальних" і "глобальних" найкращих частинок, схожий на операцію схрещування, що використовується у генетичних алгоритмах.

Алгоритм майфлая (MA) призначений для розв'язання задач оптимізації. Натхненний поведінкою польоту та процесом спарювання майфлаїв, запропонований алгоритм поєднує основні переваги ройової інтелектуальності та еволюційних алгоритмів. Для оцінки продуктивності запропонованого алгоритму використовуються 38 математичних тестових функцій, включаючи 13 тестів CEC2017, а результати порівнюються з результатами семи сучасних метаевристичних методів. Продуктивність MA також оцінюється через поведінку збіжності у багатокритеріальній оптимізації, а також через використання реальної задачі планування потокового виробництва. Результати порівняння демонструють перевагу запропонованого методу за швидкістю та якістю збіжності. Процеси "шлюбного танцю" та випадкового польоту покращують баланс між властивостями дослідження та використання алгоритму та допомагають уникнути локальних мінімумів.

2.3 Алгоритм оптимізації рою частинок (PSO)

Алгоритми оптимізації рою частинок (Particle Swarm Optimization, PSO) є метаевристиками, натхненними природою, заснованими на популяціях, які вперше були запропоновані Джеймсом Кеннеді та Расселом Еберхартом у 1995 році. Ці алгоритми імітують соціальну поведінку зграї птахів і косяків риб. Починаючи з випадково розподіленого набору частинок (потенційних рішень), алгоритми намагаються покращити рішення відповідно до міри якості (функції пристосованості). Покращення виконується шляхом переміщення частинок у просторі пошуку за допомогою простих математичних виразів, які моделюють деякі міжчастинкові комунікації.

У своїй найпростішій та базовій формі ці математичні вирази припускають переміщення кожної частинки до її найкращої позиції, досягнутої досі, і найкращої позиції всього рою, разом із деякими випадковими збуреннями. Щоб запровадити

єдину термінологію, нижче наведено визначення кількох технічних термінів, які часто використовуються:

- Рій: Популяція алгоритму.

Частинка: Член (індивідуум) рою. Кожна частинка представляє потенційне рішення задачі, що вирішується. Позиція частинки визначається рішенням, яке вона наразі представляє.

pbest (особистий найкращий результат): Найкраща особиста позиція заданої частинки, досягнута на даний момент. Це позиція частинки, яка принесла найбільший успіх (вимірюваний у вигляді скалярного значення, аналогічного до функції пристосованості в еволюційних алгоритмах).

gbest (глобальний найкращий результат): Позиція найкращої частинки всього рою.

Лідер: Частинка, яка використовується для спрямування іншої частинки до кращих областей у просторі пошуку.

Швидкість (вектор): Цей вектор керує процесом оптимізації, тобто визначає напрямок, у якому частинка повинна "летіти" (рухатися), щоб покращити свою поточну позицію.

Інерційна вага: Позначена як iii, використовується для контролю впливу попередньої історії швидкостей на поточну швидкість заданої частинки.

Фактор навчання: Представляє привабливість, яку частинка має щодо свого власного успіху або успіху своїх сусідів. Використовуються два фактори навчання: c1 і c2. c1 є когнітивним фактором навчання і представляє привабливість частинки до власного успіху. c2 є соціальним фактором навчання і представляє привабливість частинки до успіху її сусідів. Обидва фактори c1 і c2 зазвичай визначаються як константи.

PSO використовує популяцію індивідуальних частинок, кожна з яких має позицію, швидкість і пам’ять про місце свого найкращого пристосування, знайденого під час пошукового процесу. Кожна частинка оновлює свою швидкість і пам’ять, після чого пам’ять інших частинок ділиться у її околі. Під час оновлення швидкості частинка переміщується на нову позицію у просторі пошуку. У своїй

початковій формі PSO визначається наступним чином.

$$V_{id}^{t+1} = w \cdot v_{id}^{t} + c_1 \cdot D\left(P_{best,i}^{t} - x_{id}^{t}\right) + c_2 \cdot r_{2d}^{t} \left(G_{best}^{t} - x_{id}^{t}\right) \quad (2.10)$$

$$x_{id}^{t+1} = x_{id}^{t} + V_{id}^{t+1}, d = 1,2 \ldots\ldots n$$

Де:

v_{id}^{t}: швидкість частинки i у вимірі d у момент часу t.

x_{id}^{t+1}: позиція частинки i у вимірі d у момент часу t+1.

w: інерційна вага, яка контролює вплив попередньої швидкості на поточну швидкість.

$P_{best,i}^{t}$: особиста найкраща позиція частинки i у вимірі d, знайдена від початкової ініціалізації до моменту часу t.

G_{best}^{t}: глобальна найкраща позиція частинки i.

c_1, c_2: додатні константи прискорення, які регулюють внесок когнітивної та соціальної компонент відповідно.

r_{1d}^{t}, r_{2d}^{t}: випадкові числа з рівномірного розподілу U(0,1) у момент часу t.

Алгоритм PSO вважається одним із метаевристичних алгоритмів, які були розроблені в першу чергу для розв'язання задач оптимізації. Він є стохастичним підходом до оптимізації, що базується на пошуку в популяції. Індивідууми, які називаються частинками, групуються у рій, і кожна частинка у рої представляє допустиме рішення задачі в просторі пошуку. Продуктивність кожної частинки оцінюється відповідно до заздалегідь визначеної функції пристосованості, яка пов’язана із задачею, що вирішується.

У PSO імітується взаємодія групи в природі, коли члени групи прагнуть рухатися в напрямку до найкращого члена групи. Таким чином, поведінка кожного члена формується як особистою, так і соціальною інформацією, як показано на рис. (2.6). Було доведено, що це ефективний метод для багатьох задач оптимізації, і в деяких випадках він не має труднощів, з якими стикаються інші техніки еволюційного обчислення (Evolutionary Computing, EC).

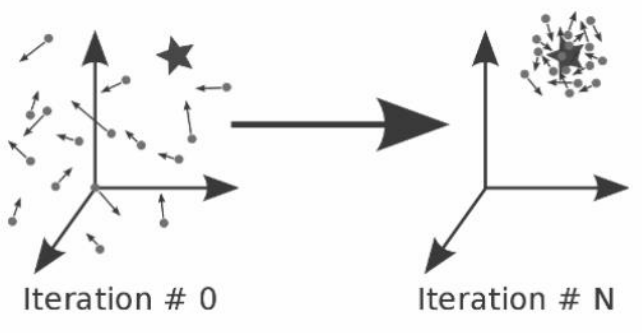

Рис.2.17 Процедура PSO

Крок 1: Ініціалізуйте рій індивідуумів, які називаються частинками, у просторі рішень. Призначте кожній частинці випадкову позицію та швидкість. Встановіть поточні позиції як найкращі позиції частинок (Pbest) і позицію найкращої оціненої частинки як глобальну найкращу позицію (Gbest). Тут і позначає номер частинки. Встановіть ітерацію (t) на 0 і перейдіть до Кроку 2.

Крок 2: Модифікуйте швидкості та позиції відповідно до рівнянь (2.10). Швидкість кожної частинки обмежена певною межею vmax, яка зазвичай встановлюється як частка простору рішень. Інерційна вага визначається за цим рівнянням:

$$W = W_{\max} - ((W_{\max} - W_{\min}) / t_{\max}) \cdot t \qquad (2.11)$$

В якому t позначає номер ітерації, а $t_{\max}$ — максимальну кількість ітерацій. Після цього перейдіть до Кроку 3.

Крок 3: Оцініть поточні позиції частинок. Порівняйте поточне значення функції пристосованості з найкращим значенням для кожної частинки. Якщо знайдено кращу позицію, оновіть найкращу позицію. Також оновіть глобальну найкращу позицію, якщо це необхідно.
Крок 4: Перевірте критерій завершення. Якщо умова виконана, завершіть алгоритм і надайте найкраще отримане рішення. В іншому випадку встановіть t→t+1 і

поверніться до Кроку 2.

Алгоритм 2: Алгоритм рою часток (PSO)

Input: Objective function min or max $\boldsymbol{f}(\boldsymbol{x})$

Output: the optimal solutions for each variable and the optimal cost

For each particle i

For each dimension d

Initialize position Xid^t

Initialize velocity vid^t

End for

End for

While (t < maximum number of iterations)

For each particle

Calculate fitness value

If the fitness value is better than $Pbest,i^t$ in history

Set current fitness value as the $Pbest,i^t$

End if

End for

Choose the particle having the best fitness value as the $gbest^t$

For each particle i

For each dimension d

Calculate velocity according to the equation:

$$vid^{t+1} = w \cdot vid^t + c_1 \cdot rand_1 d^t\ (Pbest,i^t - xid^t) + c_2 \cdot rand_2 d^t\ (gbest^t - xid^t)$$

Update particle position according to the equation:

$$Xid^{t+1} = Xid^t + vid^{t+1}$$

End for

End for

End while

2.3.1 Переваги алгоритму оптимізації рою частинок (PSO)

На основі експериментів із задачами оптимізації для тестування PSO бувстановлено, що

- PSO починає пошук із кількох точок, оскільки це алгоритм, що базується на популяції.
- PSO має швидший темп збіжності. Це зумовлено способом обміну інформацією. У PSO лише найкраща частинка в популяції ділиться інформацією з іншими, спрямовуючи їх до загальної найкращої позиції. Такий односторонній спосіб обміну інформацією дозволяє PSO швидко досягати оптимального рішення.
- Алгоритм має менше параметрів для налаштування.
- У процесі ітерації потрібні лише базові математичні оператори, тому загальна процедура обчислення PSO є простою та легко реалізується.
- PSO не потребує похідних, тому цільова функція безпосередньо використовується як функція пристосованості для спрямування пошуку.
- Алгоритм може бути застосований до дискретних задач або задач із змішаними цілими числами.

2.4 Алгоритм штучної нейронної мережі (ANN)

Штучна нейронна мережа (Artificial Neural Network, ANN) — це комп’ютерний метод, що має біологічне коріння і імітує поведінку та процеси навчання людського мозку. Цей метод повністю базується на історичних наборах даних вхід-вихід (набори прикладів), як показано на рис. (4.1), для вивчення взаємозв’язків між даними через навчання і не потребує явних знань про фізичні явища, що досліджуються.

ANN має вражаючу здатність до узагальнення, що дозволяє їй правильно прогнозувати вихідні дані для нових наборів вхідних даних, а також здатність працювати із шумовими даними та невизначеностями — це лише дві з багатьох переваг, які пропонують моделі, засновані на ANN.

Метод ANN широко застосовується у багатьох сферах, таких як інженерія, медицина, метеорологія, економіка, психологія та інші, завдяки своїм численним привабливим властивостям. У результаті численні дослідження вивчали використання ANN для оцінки багатошарових композитних панелей. Однак дослідження, які намагаються розробити узагальнену модель прогнозування, що може враховувати вплив кожного з геометричних параметрів сердечника у вигляді стільникових структур — товщина стінки осередка (t), кут стінки осередка (), довжина стінки осередка (a) та товщина сердечника (D) — є досить рідкісними, коли йдеться про композити багатошарових стільникових структур. Декілька експериментів, згаданих у літературі були обмежені модифікаціями геометрії осередків та врахованими геометричними параметрами.

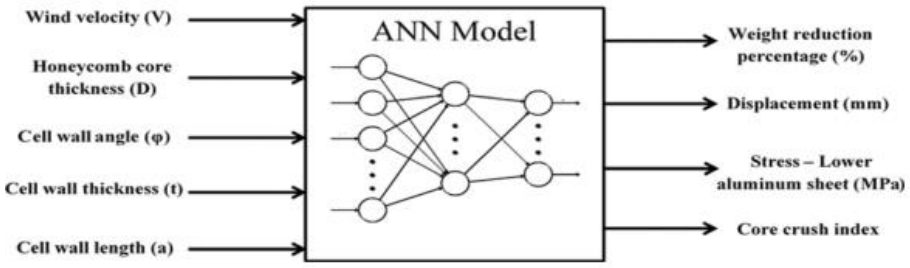

Рис.2.18 Входи та виходи режиму ANN

Складність і продуктивність розроблюваної моделі штучної нейронної мережі (ANN) визначаються кількістю нейронів у прихованому шарі, що є налаштовуваним параметром при розробці ANN. Наприклад, занадто мала кількість нейронів у прихованому шарі може призвести до зниження точності прогнозування (тобто мережа не здатна виявити нелінійні тенденції в наборі даних). Однак занадто велика кількість нейронів у прихованому шарі може спричинити проблеми, такі як перенавчання та тривалий час обчислення. Для

визначення кількості нейронів у прихованому шарі, які забезпечують найкращу продуктивність прогнозування, необхідно знайти компроміс.

Найпоширеніший спосіб оцінки продуктивності створеної моделі ANN — це обчислення середньоквадратичної помилки (MSE) (рівняння 2.12) і коефіцієнта детермінації (R2) між передбаченими моделлю вихідними значеннями та фактичними значеннями (рівняння 2.13). Коли MSE досягає мінімуму, а R2 є високим (R2>0.98~0.99), модель вважається такою, що має хороші прогнозувальні можливості, і ці показники були використані як метрики в цьому дослідженні.

$$MSE = \frac{1}{n}\sum_{j=1}^{n} * (Yact, i - Ypred, i)^2$$

$$R^2 = 1 - \frac{\sum_{j=1}^{n} * (Yact, i - Ypred, i)^2}{\sum_{j=1}^{n} * (Yact, i - Yavg)^2}$$

Де n — це кількість точок даних, Yact — це довжина стінки осередка, Ypred — це передбачене **значення**, отримане від створеної мережі, а Yavg — це середнє значення для Yact.

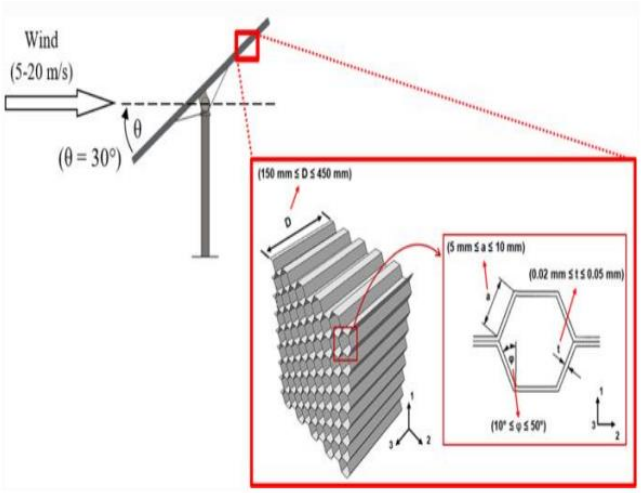

Рис.2.19 Конфігурації, що використовуються для вхідного набору даних проектування мережі та змінних дизайну оптимізації дослідження з їх діапазонами пошуку

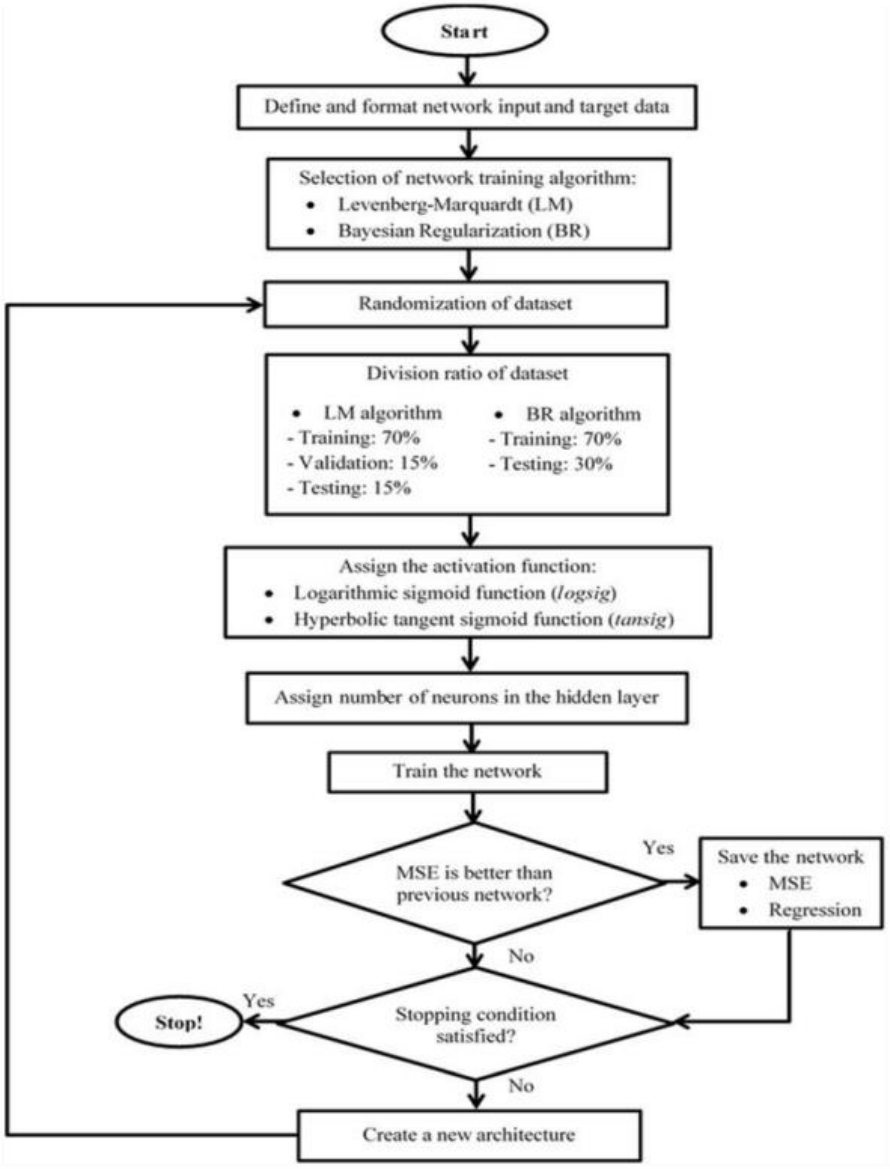

Рис.2.20 Блок-схема запропонованої методології ANN

РОЗДІЛ 3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ МОДЕЛЕЙ ТЕСТУВАННЯ

3.1 Дизайн моделі

Штучні нейронні мережі (ANN) та оптимізація рою частинок (PSO) були реалізовані в наступних дослідженнях із використанням трьох наборів даних для порівняння результатів традиційної класифікації, методів зменшення розмірності та гібридних алгоритмів.

3.1.1 Вибір моделі

Штучна нейронна мережа (ANN) — це комп'ютерний метод із біологічним корінням, що імітує поведінку та процеси навчання людського мозку. Цей метод повністю базується на історичних наборах даних вхід-вихід (набори прикладів), як показано на рис. (3.2), для вивчення взаємозв’язків між даними через навчання і не вимагає явного знання фізичних явищ, що досліджуються.

Ця методика має вражаючу здатність до узагальнення, що дозволяє правильно прогнозувати вихідні дані для нового набору вхідних даних, а також здатність працювати з шумовими даними та невизначеностями — це лише дві з багатьох переваг, які пропонують моделі на основі ANN.

Метод ANN широко використовується в багатьох сферах, таких як інженерія, медицина, метеорологія, економіка, психологія та багато інших, завдяки своїм численним привабливим властивостям. У результаті численні дослідження вивчали використання ANN для оцінки багатошарових композитних панелей. Однак дослідження, які намагаються розробити узагальнену модель прогнозування, що враховує вплив кожного геометричного параметра стільникового сердечника — товщини стінки осередка (t), кута стінки (), довжини стінки (a) та товщини сердечника (D) — зустрічаються рідко, коли йдеться про багатошарові композитні стільникові структури.

Декілька експериментів, згаданих у літературі, обмежувалися дифікаціями

геометрії осередків та враховували лише окремі геометричні параметри.

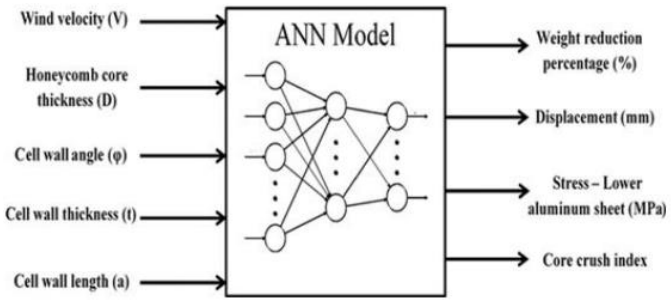

Рис.3.2 Вхідні та вихідні параметри ANN режиму

Оптимізація рою частинок (PSO) є особливо перспективною та ефективною технікою оптимізації для розв'язання задач із сильно обмеженими нелінійними та неконвексними умовами. Алгоритм PSO вперше був запропонований Кеннеді та Еберхартом і базується на кооперативній поведінці, що спостерігається у зграях птахів та косяках риб. Розташування точок (або частинок) у просторі дизайну представляє можливі рішення задачі оптимізації.

Відповідно до своєї індивідуальної оптимальної позиції та оптимальної позиції всього рою під час кожного покоління, кожна частинка оновлює своє положення. PSO має багато переваг порівняно з іншими підходами до оптимізації, зокрема, менше параметрів для налаштування, ніж у багатьох його конкурентів, швидкий час обчислень і можливість поєднувати різні техніки для створення гібридних інструментів. Крім того, фаза ітерації алгоритму PSO не залежить від початкового рішення.

3.2 Гібридні алгоритми

Запропонована модель (ANN-PSO):

Запропонована модель виконує задачу бінарної класифікації, використовуючи підхід оптимізації рою частинок (PSO) для вибору підмножини ознак із набору даних. Ось пояснення коду крок за кроком:

- Імпорт бібліотек: Імпортуються необхідні бібліотеки та модулі, включаючи warnings, pandas, tensorflow, scikit-learn (sklearn), seaborn, numpy, matplotlib і keras.

- Завантаження та попередня обробка даних: Набір даних завантажується з файлу CSV за допомогою pandas, і відображаються перші п’ять рядків. За допомогою LabelEncoder цільова змінна ('defects') кодується у числові значення, а також створюється новий стовпець. Стовпець 'defects' видаляється з набору даних, а результат зберігається в об'єкті df. Матриця ознак (data_X) і цільова змінна (data_y) витягуються з df. Дані поділяються на навчальну та тестову вибірки за допомогою функції train_test_split з sklearn.

- Встановлення випадкових початкових значень: Встановлюються випадкові значення для забезпечення відтворюваності результатів.

- Визначення цільової функції: Цільова функція 'f_per_particle' визначається як функція, яка приймає бінарну маску 'm'. Функція вибирає підмножину ознак на основі бінарної маски та виконує класифікацію за допомогою моделі нейронної мережі. Продуктивність моделі обчислюється за допомогою точності. Значення цільової функції 'j' обчислюється як зважене поєднання продуктивності класифікації та кількості обраних ознак.

- Цільова функція вищого рівня: Функція 'f' приймає весь рій 'x' і значення alpha як вхідні дані. Функція обчислює значення 'f_per_particle' для кожної частинки рою. Повертаються негативні значення цільової функції.

- Оптимізація рою частинок (PSO): Параметри PSO (опції) включають когнітивні та соціальні коефіцієнти (c1 і c2), інерційну вагу (w), максимальну швидкість (k) і p-значення. Оптимізатор PSO ініціалізується із заданими параметрами та кількістю вимірів (кількість ознак). Алгоритм PSO виконується для певної кількості ітерацій, оптимізуючи цільову функцію 'f'. Повертаються найкраща вартість і найкраща позиція (бінарна маска), отримані з оптимізації.

- Вибір ознак і навчання фінальної моделі: Обрані ознаки для навчання та тестування визначаються шляхом застосування найкращої позиції (бінарної

маски) до навчального та тестового наборів. Визначається та компілюється модель нейронної мережі, використовуючи обрані ознаки як вхідні дані. Модель навчається на навчальному наборі протягом заданої кількості епох. Прогнози виконуються на обраних ознаках тестового набору. Продуктивність моделі на тестовому наборі обчислюється і виводиться на екран.

Запропонована модель (ANN-PSO)

```
# Load and Preprocess Data
#raw_data = load_dataset('filename.csv')
#encoded_data = encode_target_variable(raw_data)
#df = drop_column(raw_data, 'defects')
#X, y = split_feature_target(df)
# Split Data into Train and Test Sets
#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
# Particle Swarm Optimization (PSO)
#best_position = particle_swarm_optimization(X_train, y_train)
# Select Features and Train Final Model
#selected_features_train = select_features(X_train, best_position)
#selected_features_test = select_features(X_test, best_position)
#model = create_neural_network_model(selected_features_train)
#train_neural_network(model, selected_features_train, y_train)
#predictions = predict_neural_network(model, selected_features_test)
#performance = compute_performance(predictions, y_test)
#print("Test performance:", performance)
```

Алгоритм 1: Запропонована модель (ANN-PSO)

```
# Start
# |--- Load and Preprocess Data
# |    |
# |    |-- Read dataset
# |    |-- Encode target variable
# |    |-- Create feature matrix and target variable array
# |--- Split Data into Train and Test Sets
# |    |
# |    |-- Split feature matrix and target variable into train and test sets
# |--- Particle Swarm Optimization (PSO)
# |    |
# |    |-- Initialize PSO parameters
# |    |-- Initialize PSO optimizer
# |    |-- Perform optimization using PSO
# |    |
# |    |-- For each iteration:
# |    |   |-- Update particle positions and velocities
```

```
# |    |   |-- Evaluate objective function for each particle
# |    |   |-- Update global best position and cost
# |    |
# |    |-- Return best position (binary mask)
# |--- Select Features and Train Final Model
# |    |-- Select features based on the best position obtained from PSO
# |    |-- Define and compile a neural network model
# |    |-- Train the model using selected features
# |    |-- Make predictions on the test set
# |    |-- Compute model performance on the test set
# |    |-- Print test performance
# End
```

3.2 Опис використаних алгоритмів та метрик

Дані були зібрані з репозиторію наборів даних PROMISE. Запропонований підхід був реалізований із використанням наборів даних CM1, PC1 та KC1. У таблиці (3.1) наведено опис різних атрибутів, використаних для аналізу. Крім того, ми розділили два набори даних на дві окремі частини: одну для навчання моделі, а іншу для оцінки результатів. 240 спостережень із набору даних CM1 використовувалися для навчання моделі, а 104 спостереження — для тестування. Модель навчалася на 1476 спостереженнях із набору даних KC1, а тестувалася на 633 записах. Для набору даних PC1 використовувалося 776 спостережень для навчання моделі та 333 спостереження для тестування.

Посилання на набори даних:

- http://promise.site.uottawa.ca/SERepository/datasets/pc1.arff
- http://promise.site.uottawa.ca/SERepository/datasets/cm1.arff
- http://promise.site.uottawa.ca/SERepository/datasets/kc1.arff

Attribute name	Description of attribute
LOC	Counts the total number of line in the module
Iv(g)	Design complexity analysis (McCabe)
Ev(g)	McCabe essential complexity
N	Number of operators present in the software module
v(g)	Cyclomatic complexity measurement (McCabe)
D	Measurement difficulty
B	Estimation of effort
L	Program length
V	Volume
I	Intelligence in measurement
E	Measurement effort
Locomment	Line of comments in software module
Loblank	Total number of blank lines in the module
uniq_op	Total number of unique operators
uniq_opnd	Total number of unique operand
T	Time estimator
Branchcount	Total number of branch in the software module
total_op	Total number of operators
Total_opnd	Total number of operators
Locodeandcomment	Total number of line of code and comments
Defects/Problems	Information regarding defect whether the defect is present or not

3.3 Налаштування параметрів

П’ятсот повторень були проведені для перевірки та тестування моделей. Вхідний лист для ANN базується на кількості спостережень. Оскільки був використаний ефективний **підхід**, було створено тисячі прихованих вузлів. Цей підхід вимагав більше прихованих вузлів, ніж традиційні алгоритми. Клас із двома категоріями був введений за допомогою одного вузла вихідного шару. Техніка оптимізації рою частинок (PSO) була розроблена для оптимізації гіперпараметрів, таких як (кількість оцінювачів, максимальна глибина тощо). Для оптимізації було використано 10 ітерацій, як показано в таблиці (3.2).

Model	Parameter	Values
ANN	Input nodes	based on No. Features
	Activation fun for output nodes	sigmoid
	Output nodes	1
	No. of Iterations	100
PSO	Number of particles	50
	Number of iterations	100
	Dimension	Number of features(21)
	α in fitness function	0.88
	β in fitness function	0.01
	Options	{‘c1’:0.5, ‘c2’: 0.5, ‘w’: 0.9 , ‘K’: 30, ‘p’: 2}

3.4 Експериментальна установка

Запропоновані методології реалізовані в Matlab та Python, і всі експерименти в цьому розділі проводилися на комп'ютері з процесором Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz та оперативною пам'яттю 8.0GB. Експерименти включають дослідження впливу параметра розміру популяції (NNN) та кількості ітерацій на ANN і PSO. У таблиці (3.1) наведено багато популярних значень параметрів для ANN, PSO та інших алгоритмів, які базуються на рекомендаціях з оригінальних статей про MA і PSO. Якщо ви бажаєте встановити справедливі порівняння та досягти надійного статистичного аналізу. Різні алгоритми класифікації (KNN, Logistic, SVM, ABC, PCA, LDA, DT, GBC, QDA) були реалізовані, і їх результати задокументовані та порівняні із запропонованими методами.

Критерії оцінки продуктивності

Запропоновані порівняльні моделі тестувалися на основі трьох змінних оцінювання. Ці параметри вимірювали точність, повноту та F1-показник як для класифікації, так і для дослідження. Параметри оцінки визначалися наступним чином:

Матриця плутанини:

Матриця плутанини була вказана як таблиця, яка використовується для визначення результатів класифікатора відповідно до ряду аналізів даних, щоб забезпечити реальні значення (тобто фактичні позитивні та негативні класи), які були враховані.

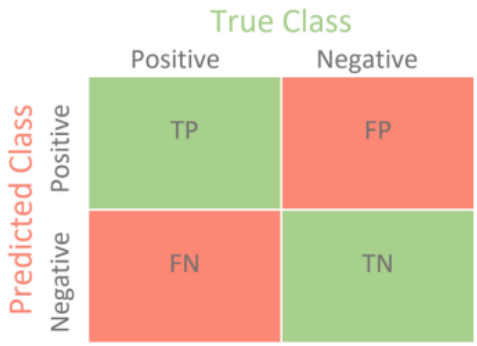

Рис.3.3 Матриця плутанини правил класифікації

Точність (Precision):

Точність обчислювалася як співвідношення результатів, отриманих системою. Вона точно визначала позитивні спостереження (True Positives) порівняно з усіма позитивними спостереженнями, отриманими системою, як правильними (True Positives), так і неправильними (False Positives). Формула для обчислення точності:

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \quad (3.1)$$

Повнота (Recall):

Повнота обчислювалася як співвідношення результатів, отриманих системою, до всіх реальних зразків з класу "позитивні" (Actual Positives), які, у свою чергу, правильно визначають позитивні спостереження (True Positives). Формула для обчислення повноти:

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \quad (3.2)$$

Точність класифікації (Accuracy):

Точність була одним із найбільш спостережуваних показників продуктивності. Це те, що багато хто вивчав у школі, поряд із такими показниками, як точність (Precision), повнота (Recall) та F1-міра.

У короткому вигляді точність класифікації визначалася як співвідношення правильно класифікованих результатів (як True Positives, так і True Negatives) до загальної кількості спостережень у наборі даних для дослідження. Формула для обчислення точності:

$$accuracy = \frac{true\ positives + true\ negatives}{true\ positives + false\ positives + false\ negatives + true\ negatives} \quad (3.3)$$

F1-міра (F1 Score):

F1-міра представляла зважене середнє (або збалансоване середнє) між точністю

(Precision) і повнотою (Recall). Це дозволяло врівноважити точність і повноту, беручи до уваги як помилково позитивні (False Positives), так і помилково негативні (False Negatives) значення. Формула для обчислення F1-міри:

$$F1\ score = \frac{2 * (precision * recall)}{precision + recall} \quad (3.4)$$

3.5 Результати експериментів та їх аналіз

У цьому розділі представлено порівняння між алгоритмами класифікації, штучною нейронною мережею (ANN) і згортковою нейронною мережею (CNN), базуючись на даних із робіт, згаданих у розділі "Попередні дослідження", із використанням набору даних PROMISE. Кожен із алгоритмів був застосований до наборів даних (PC1, CM1, KC1 та KC2).

У таблицях (3.1, 4.1, 5.1) наведено результати. Після реалізації моделей класифікації та запропонованої моделі ми порівняли нашу модель із попередніми дослідженнями на основі точності (Precision), повноти (Recall), F-міри (F-measure) та точності класифікації (Accuracy). Ми виявили, що PSO-ANN є більш точною, ніж інші моделі.

Під час аналізу набору даних KC1, згідно з рисунками (3.4, 3.5 і 3.6), ми отримали такі показники: точність 0.94, повнота 0.90, F-міра 0.95 і точність класифікації 86.00, що є суттєвим покращенням.

У разі аналізу набору даних CM1 ми отримали: точність 90.0, повнота 1.0, F-міра 95.0 і точність класифікації 90.88.

Для аналізу набору даних PC1 ми отримали: точність 95.0, повнота 1.0, F-міра 95.0 і точність класифікації 94.00.

Точність навчальних даних наведено на рисунку (3.4), а тестових даних — на рисунку (3.5).

Table 4.1: Classification algorithms and proposed model with CM1 dataset

		CM1							
		Train Dataset				Test Dataset			
		Acc	F1s	Pre	Rec	Acc	F1s	Pre	Rec
1	LogisticRegression	89.37	0.94	0.91	0.98	89.00	0.95	0.91	0.97
2	KNeighborsClassifier	89.95	0.95	0.91	0.99	89.67	0.95	0.91	0.97
3	DecisionTreeClassifier	84.49	0.91	0.92	0.91	82.00	0.90	0.90	0.90
4	RandomForestClassifier	89.66	0.95	0.90	0.99	89.33	0.94	0.90	0.97
5	GradientBoostingClassifier	88.22	0.94	0.91	0.96	87.33	0.93	0.90	0.96
6	AdaBoostClassifier	85.93	0.93	0.90	0.95	86.00	0.92	0.90	0.95
7	LinearDiscriminantAnalysis	88.23	0.94	0.92	0.95	87.33	0.93	0.90	0.97
8	QuadraticDiscriminantAnalysis	53.47	0.68	0.90	0.54	89.67	0.95	0.91	0.97
9	GaussianNB	52.64	0.66	0.94	0.51	86.67	0.93	0.91	0.94
10	SVC(Linear)	90.23	0.95	0.90	1.00	89.00	0.95	0.90	0.97
11	SVC(probability = true)	89.95	0.95	0.90	1.00	89.00	0.95	0.90	0.97
12	SVC(poly)	88.80	0.94	0.91	0.99	89.00	0.95	0.91	0.97
13	SVC(sigmoid)	89.95	0.95	0.92	0.97	89.00	0.95	0.90	0.97
14	PC-SVM	89.95	0.95	0.92	0.97	90.00	0.95	0.90	0.97
15	Proposed Model (PSO-ANN)	98.95	0.98	0.96	0.97	91.00	0.98	0.95	0.99

Рис.3.4. Точність навчальних даних

Table 4.2: Classification algorithms and proposed model with PC1 dataset

		PC1							
		Train Dataset				Test Dataset			
		Acc	F1s	Pre	Rec	Acc	F1s	Pre	Rec
1	LogisticRegression	92.66	0.96	0.94	0.99	92.79	0.96	0.94	0.95
2	KNeighborsClassifier	92.66	0.96	0.94	0.98	92.39	0.97	0.94	0.97
3	DecisionTreeClassifier	90.08	0.95	0.95	0.94	90.69	0.95	0.95	0.96
4	RandomForestClassifier	92.91	0.96	0.93	0.99	92.39	0.97	0.94	0.97
5	GradientBoostingClassifier	92.40	0.96	0.94	0.98	91.59	0.96	0.95	0.96
6	AdaBoostClassifier	92.53	0.96	0.94	0.98	92.29	0.97	0.96	0.98
7	LinearDiscriminantAnalysis	92.14	0.96	0.94	0.98	89.79	0.95	0.94	0.95
8	QuadraticDiscriminantAnalysis	37.65	0.51	0.94	0.35	92.99	0.97	0.94	0.97
9	GaussianNB	89.05	0.94	0.95	0.93	83.78	0.91	0.95	0.87
10	SVC(Linear)	92.78	0.96	0.93	1.00	92.99	0.97	0.94	0.97
11	SVC(probability = true)	93.30	0.97	0.93	1.00	92.99	0.97	0.94	0.97
12	SVC(poly)	92.91	0.96	0.94	0.94	92.79	0.96	0.94	0.98
13	SVC(sigmoid)	90.47	0.95	0.93	0.97	92.29	0.97	0.94	0.97
14	PC-SVM	89.95	0.95	0.92	0.97	93.00	0.95	0.90	0.95
15	Proposed Model (PSO-ANN)	97.90	0.98	0.96	0.97	94.00	0.99	0.98	1.00

Рис.3.5. Точність тестованих даних

Table 4.3: Classification algorithms and proposed model with KC1 dataset

		KC1							
		Train Dataset				Test Dataset			
		Acc	F1s	Pre	Rec	Acc	F1s	Pre	Rec
1	LogisticRegression	86.86	0.93	0.88	0.98	83.25	0.91	0.85	0.97
2	KNeighborsClassifier	85.64	0.92	0.89	0.96	81.52	0.90	0.84	0.96
3	DecisionTreeClassifier	84.01	0.91	0.90	0.91	75.99	0.86	0.85	0.86
4	RandomForestClassifier	86.58	0.93	0.88	0.98	82.46	0.90	0.84	0.97
5	GradientBoostingClassifier	87.19	0.93	0.89	0.97	81.67	0.90	0.85	0.94
6	AdaBoostClassifier	85.84	0.92	0.88	0.97	82.15	0.90	0.85	0.95
7	LinearDiscriminantAnalysis	86.38	0.92	0.89	0.96	81.36	0.89	0.85	0.94
8	QuadraticDiscriminantAnalysis	33.14	0.38	0.91	0.24	82.62	0.85	0.85	0.90
9	GaussianNB	83.20	0.90	0.90	0.91	81.20	0.89	0.87	0.91
10	SVC(Linear)	86.04	0.92	0.86	0.99	82.78	0.91	0.83	0.99
11	SVC(probability = true)	86.92	0.93	0.87	0.99	82.62	0.90	0.84	0.97
12	SVC(poly)	86.59	0.93	0.87	0.99	83.41	0.91	0.85	0.97
13	SVC(sigmoid)	79.74	0.88	0.88	0.89	78.20	0.87	0.85	0.90
14	PC-SVM	89.95	0.95	0.92	0.97	84.45	0.86	0.86	0.95
15	Proposed Model (PSO-ANN)	97.90	0.98	0.96	0.97	86.00	0.95	0.90	0.99

Рис.3.6. Класифікація

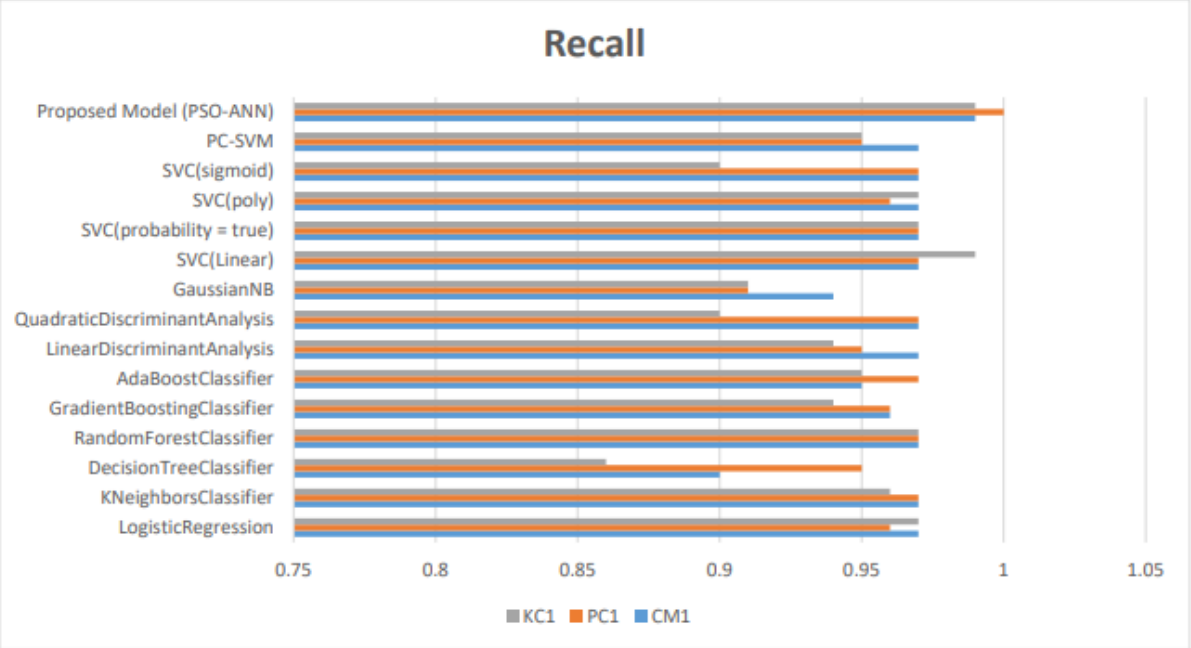

Рис.3.7 Повнота для наборів даних CM1, PC1 та KC1

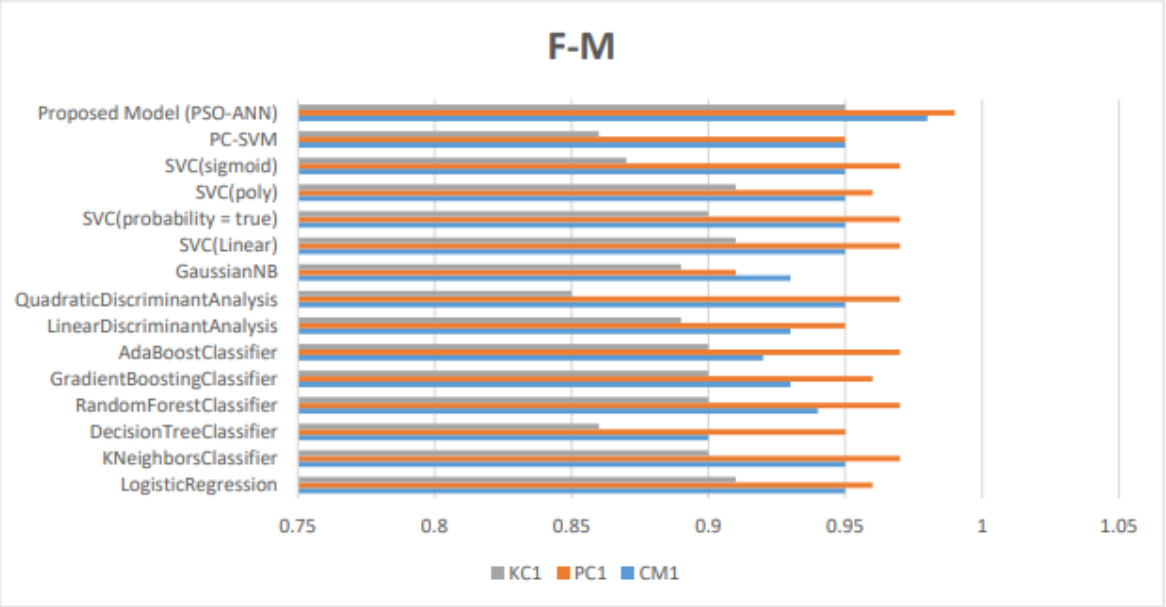

Рис.3.8 F-міра для наборів даних CM1, PC1 та KC1

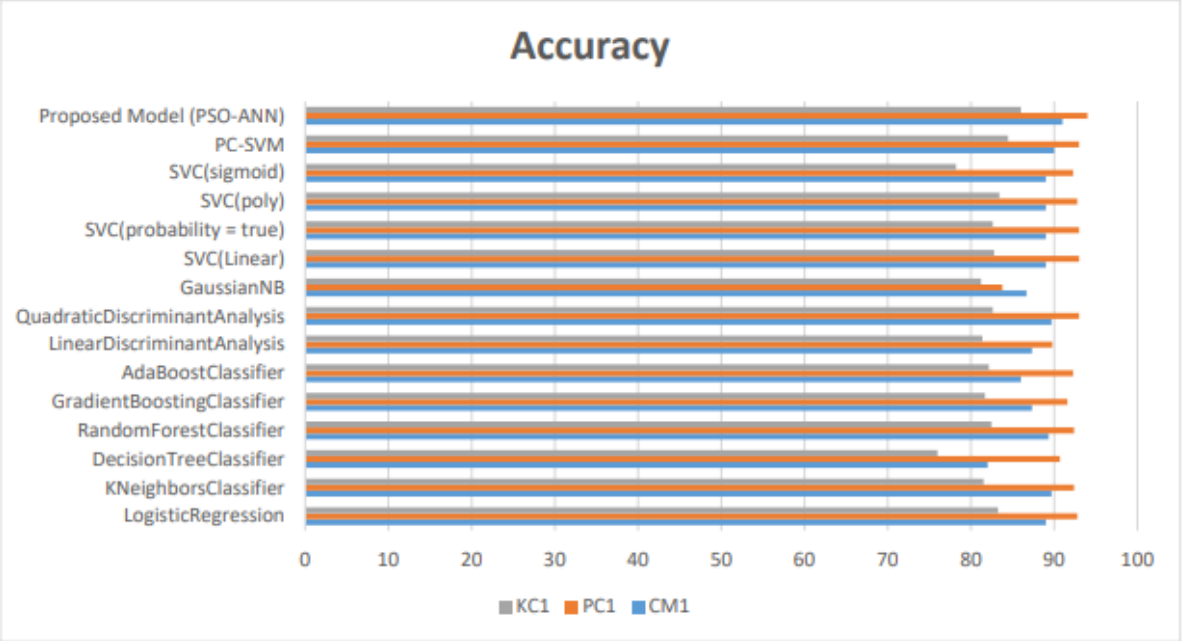

Рис.3.9 Точність класифікації для наборів даних CM1, PC1 та KC1

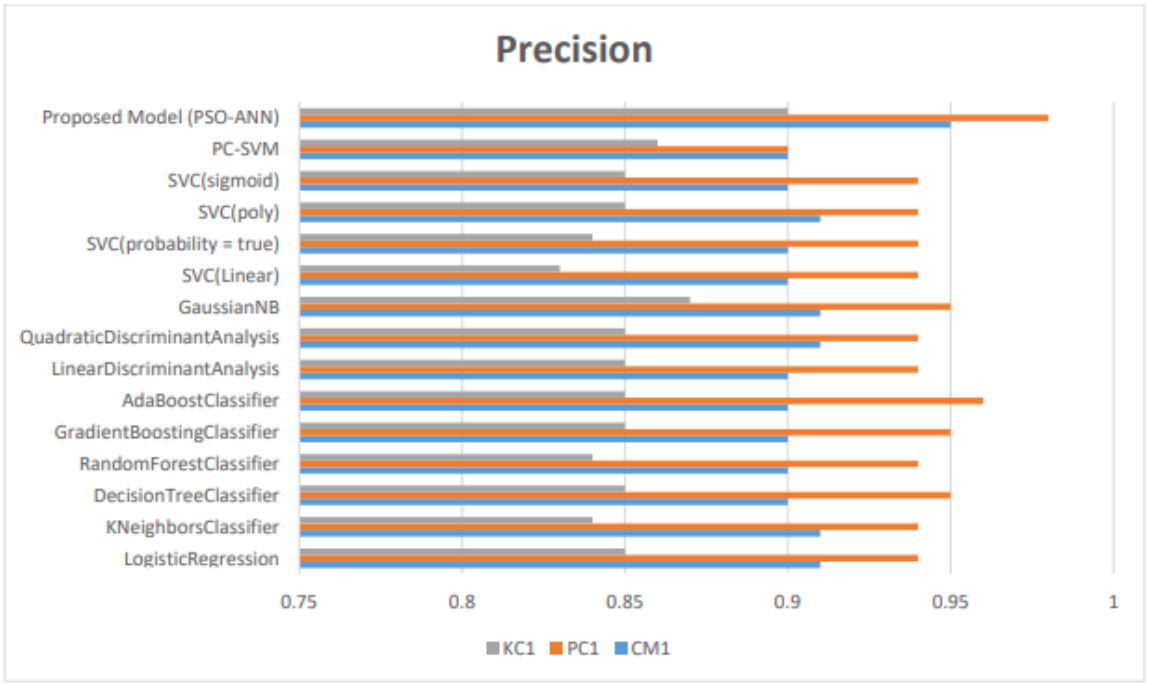

Рис.3.10 Точність для для наборів даних CM1, PC1 та KC1

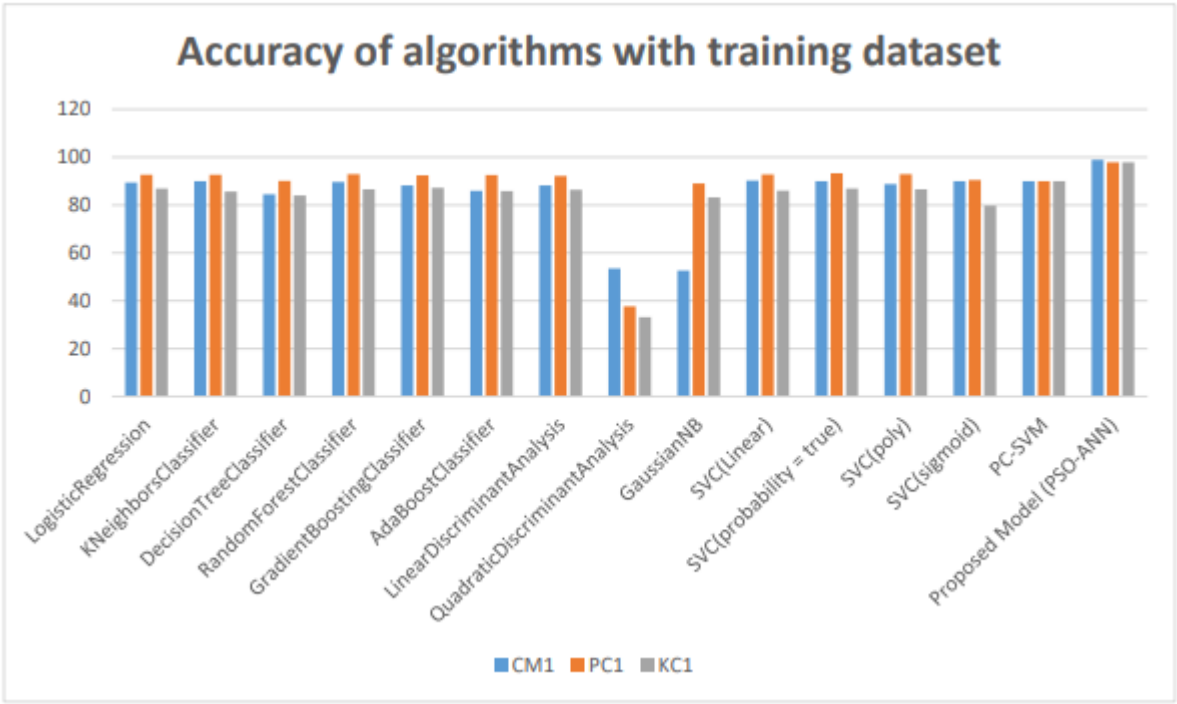

Рис.3.11. Точність алгоритмів на навчальному наборі даних

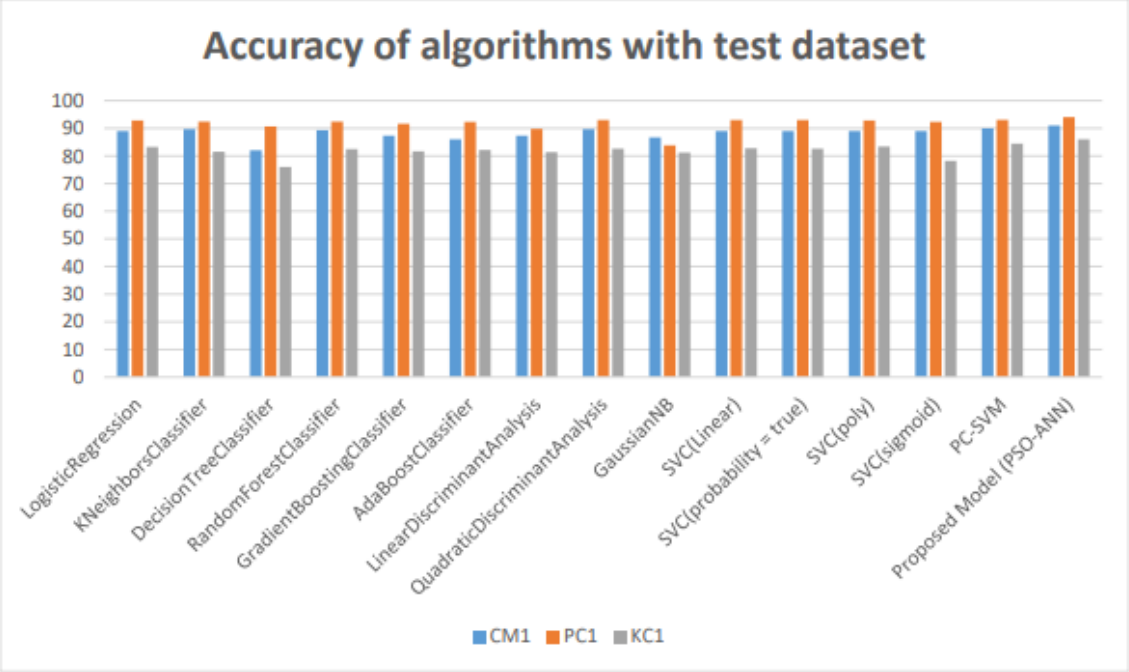

Рис.3.12 Точність алгоритмів на тестовому наборі даних

ВИСНОВКИ

Здатність точно та на ранніх етапах виявляти дефекти є ключовим фактором визначення якості програмного забезпечення. Різноманітні методи, розроблені дослідниками та науковцями, дозволяють виявляти дефекти програмного забезпечення на ранніх стадіях. Найефективнішим підходом є підхід на основі машинного навчання завдяки механізмам навчання класифікаторів. Ми дослідили продуктивність моделі ANN-PSO, реалізованої на основі репозиторію наборів даних PROMISE, використовуючи раніше розроблені традиційні методи.

Ми обрали ANN-PSO, щоб подолати проблему прокляття розмірності та зменшити обчислювальні вимоги до запропонованого завдання. Це була головна проблема попередньої методології, згаданої в дослідженні. Згодом ми використали PSO-ANN із багатошаровою архітектурою, яка є дуже потужною методологією класифікації в машинному навчанні. Ми отримали кращу точність (CM1: 91.0%, KC1: 86.0%, PC1: 94.0%), ніж за допомогою інших методів. Ми також виявили, що PSO-ANN є більш стійкою в багатьох випадках, оскільки вона забезпечує кращі результати з невеликими наборами даних і менш чутлива до викидів.

Деякі покращення моделей, що використовуються, можуть бути досягнуті за допомогою метаевристичної оптимізації (Cat Swarm Optimization (CSO), Ant Colony Optimization (ACO), Harris Hawks Optimization (HHO), Heap-Based Optimizer (HBO), Fire Hawk Optimizer (FHO), Flying Foxes Optimization (FFO)) та алгоритмів класифікації для знаходження найкращого рішення. Також планується створити новий набір даних із більшою кількістю атрибутів і рядків, після чого застосувати запропоновану модель та інші моделі для аналізу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Bejjanki K. K. Class imbalance reduction (cir): a novel approach to software defect prediction in the presence of class imbalance / K. K. Bejjanki, J. Gyani, N. Gugulothu // Symmetry. — 2020. — Vol. 12, No. 3. — c. 407.

2. Semi-supervised software defect prediction model based on tri-training / KSII Transactions on Internet and Information Systems. — 2021. — Vol. 15, No. 11.

3. Balogun A. O. Empirical analysis of rank aggregation-based multi-filter feature selection methods in software defect prediction / A. O. Balogun, S. Basri, S. Mahamad, [et al.] // Electronics. — 2021. — Vol. 10, No. 2. — c. 179.

4. Balogun A. O. Software defect prediction using wrapper feature selection based on dynamic re-ranking strategy / A. O. Balogun, S. Basri, L. F. Capretz, [et al.] // Symmetry. — 2021. — Vol. 13, No. 11. — c. 2166.

5. Zhu K. Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network / K. Zhu, N. Zhang, S. Ying, D. Zhu // IET Software. — 2020. — Vol. 14, No. 3. — c. 185–195.

6. Department of Computer Science, Virtual University of Pakistan, Lahore, Pakistan Software defect prediction using variant based ensemble learning and feature selection techniques / Department of Computer Science, Virtual University of Pakistan, Lahore, Pakistan, U. Ali, S. Aftab, [et al.] // International Journal of Modern Education and Computer Science. — 2020. — Vol. 12, No. 5. — c. 29–40.

7. Zheng S. Software defect prediction based on fuzzy weighted extreme learning machine with relative density information / S. Zheng, J. Gai, H. Yu, [et al.] // Scientific Programming. — 2020. — Vol. 2020. — c. 1–18. 68

8. Naseem R. Investigating tree family machine learning techniques for a predictive system to unveil software defects / R. Naseem, B. Khan, A. Ahmad, [et al.] // Complexity. — 2020. — Vol. 2020. — c. 1–21.

9. Zhang S. A software defect prediction approach based on bigan anomaly detection / S. Zhang, S. Jiang, Y. Yan // Scientific Programming. — 2022. — Vol. 2022. — c. 1–13.

10. Zhao Y. Cross-project defect prediction method based on manifold feature transformation / Y. Zhao, Y. Zhu, Q. Yu, X. Chen // Future Internet. — 2021. — Vol. 13, No. 8. — c. 216.

11. Murphy K. P. Machine learning: a probabilistic perspective / K. P. Murphy. — Cambridge, MA : MIT Press, 2012. — ISBN 978-0-262-01802-9.

12. Data classification: algorithms and applications / ed. C. C. Aggarwal. — Boca Raton, Fla. : CRC Press/Chapman & Hall, 2014. — ISBN 978-1-4665-8675-8.

13. Hastie T. The elements of statistical learning: data mining, inference, and prediction / T. Hastie, R. Tibshirani, J. H. Friedman. — New York, NY : Springer, 2009. — ISBN 978-0-387-84857-0.

14. Russell S. J. Artificial intelligence: a modern approach / S. J. Russell, P. Norvig. — Hoboken : Pearson, 2021. — ISBN 978-0-13-461099-3.

15. Kuhn M. Applied predictive modeling / M. Kuhn, K. Johnson. — New York : Springer, 2013. — ISBN 978-1-4614-6848-6.

16. Imbalanced learning: foundations, algorithms, and applications / ed. H. He, ed. Y. Ma. — Hoboken, New Jersey : John Wiley & Sons, Inc, 2013. — ISBN 978- 1-118-07462-6.