УДК 004.42:004.8

КОМБІНОВАНІ ПІДХОДИ ДО СТАТИЧНОГО АНАЛІЗУ КОДУ З ВИКОРИСТАННЯМ НЕЙРОННИХ МЕРЕЖ

DOI 10.36994 / 2788-5518-2023-01-05-20

Вохранов І.А. *Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", Київ, Україна. vokhranov@gmail.com*

Булах Б.В.*, к.т.н., доц. Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", Київ, Україна. bogdan.bulakh@gmail.com*

Анотація: *Ця стаття презентує огляд можливих підходів до застосування нейронних мереж в процесі статичного аналізу коду. У ній досліджується поточний стан речей у наявних підходах до покращення аналізу програм за допомогою методів машинного навчання із використанням пост-обробки сповіщень статичних аналізаторів, попередньої обробки коду, чи безпосереднього використання машинного навчання для аналізу вихідного коду. Окрім того, у статті досліджуються основні напрямки застосування підходів кожної категорії. Як класичні підходи, так і методи машинного навчання в аналізі програм мають чітко виражені сильні та слабкі сторони, які необхідно враховувати при їх практичному впровадженні. Однією з основних тез цього дослідження є те, що для побудови високоякісної системи вкрай важливо розуміти можливості комбінування даних підходів, використовуючи гнучкість, яку пропонують нейронні мережі, та зберігаючи при цьому достатній рівень надійності, що забезпечується класичними алгоритмами. Стаття покриває наступні три базові напрямки використання нейронних мереж для статичного аналізу коду. Перший напрямок – це коригування специфікацій: уточнення специфікацій, отриманих від класичного статичного аналізатора коду (усунення, кластеризація, ранжування попереджень або просто спрощення аналізу попереджень вручну, тощо). Другий напрямок – визначення специфікацій, прихованих у коді (вилучення та відбір властивостей, перетворення коду зі збереженням його поведінки, наприклад, з метою зробити його більш підходящим для класичних інструментів статичного аналізу). Третій напрямок – аналіз вихідного коду як "чорного ящика" з метою виявлення та виправлення дефектів (синтаксичних чи семантичних, або ж вразливостей) у коді, асистування при перевірці коду вручну, автоматичного форматування чи виявлення коду, що "погано пахне" (в цьому напрямку для аналізу використовується лише модель машинного навчання, тренування якої відбувається прямо на вихідному коді). Стаття окреслює коло можливих напрямків подальших досліджень, що фокусуються на розвитку та комбінуванні розглянутих підходів.*

Ключові слова: *статичний аналіз коду; нейронні мережі; машинне навчання; аналіз коду; виявлення дефектів; виправлення дефектів; коригування специфікацій; постобробка попереджень; визначення специфікацій; попередня обробка коду; навчання на вихідному коді.*

COMBINED APPROACHES TO THE STATIC CODE ANALYSIS USING NEURAL NETWORKS

Illia Vokhranov*. National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine. vokhranov@gmail.com*

Bogdan Bulakh*, Ph.D., Ass. Prof. National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine. bogdan.bulakh@gmail.com*

Abstract: *This article presents an overview of possible approaches to the application of neural networks in the process of static code analysis. It explores the current state of affairs in existing approaches to improving program analysis using machine learning methods, including postprocessing of static analysis alerts, preprocessing of source code, or direct use of machine learning for analyzing source code. Additionally, the article examines the main directions for applying approaches from each category. Both classical approaches and machine learning methods in program analysis possess distinct strengths and weaknesses that should be considered when implementing them in practice. One of the main theses of this research is that understanding the capabilities of combining these approaches, leveraging the flexibility offered by neural networks while maintaining a sufficient level of*

reliability provided by classical algorithms, is crucial for building a high-quality system. This article covers the following three basic directions of the application of neural networks for the static source code analysis. The first direction is a specification tuning: a refinement of specifications produced by a 'classic' static code analyzer (a removal, clustering, ranking of warnings or just assistance in manual warning analysis, etc.). The second direction is a specification inference, to find specifications hidden in code (feature extraction, selection, or code transformation retaining its behaviour, e.g. to make it more suitable for the 'classic' static analysis tools). The third way is a black box analysis to discover and fix code defects (syntactic, semantic ones or vulnerabilities), to assist in manual code checking, to format the code automatically or to find code smells (in this direction only a machine learning model is used, its training is performed on the source code directly). The article outlines directions for the future research which will focus on the development and combining of the approaches covered here.

Keywords: *static code analysis; neural networks; machine learning; code analysis; defect detection; defect correction; specification tuning; postprocessing of alerts; specification inference; preprocessing of source code; learning on source code.*

Вступ

Забезпечення відсутності дефектів в процесі розробки програмного забезпечення є надзвичайно важливим для стабільної, безперебійної роботи програмних систем. Статичний аналіз програмного коду є одним із ключових інструментів пошуку дефектів у програмному коді. Під час пошуку помилок та вразливостей традиційні підходи статичного аналізу покладаються на математичні методи, які надають чіткий, детермінований результат з підтвердженням чи спростуванням певного твердження про властивість програми. Через це такі методи не можуть навчитися покладатися на шаблони, які присутні у вихідному коді, або ймовірнісно обробляти неоднозначну інформацію, яка часто трапляється в реальному коді.

Основною проблемою, що виникає при застосуванні класичних математичних методів у статичному аналізі, є недостатня їх гнучкість і адаптивність. Через детермінованість результатів така система перевірки має вищий показник хибно позитивних результатів (повідомляє користувачеві про можливі дефекти чи вразливості, коли вони відсутні). Водночас, через нездатність таких методів працювати з шаблонами в коді, складніші проблеми залишаються поза увагою.

Використання нейронних мереж у процесі статичного аналізу може бути потенційним розв'язанням проблем, присутніх у класичних підходах. На основі багатьох досліджень, проведених у цьому напрямку за останні роки, зрозуміло, що використання нейронних мереж в якості інструмента для аналізу коду має потенціал. Але, разом з тим, такий підхід має і певні труднощі та часто не здатний повністю замінити класичні підходи.

Виконання досліджень

Статичний аналіз коду загалом — це процес виявлення помилок і дефектів у вихідному коді програмного забезпечення. Він походить від одного з найстаріших і найефективніших методів виявлення вад – *перевірки коду вручну* (англ. *code review*). Принцип перевірки коду полягає в перегляді вихідного коду іншими розробниками (не автором) і наданні рекомендацій щодо його вдосконалення. Цей процес дозволяє групі розробників виявляти помилки та проблеми у вихідному коді, які можуть призвести до збоїв і вразливостей програми в майбутньому.

Основна проблема ручної перевірки коду полягає в тому, що вона досить дорога: розробникам доводиться витрачати багато часу та зусиль на перевірку будь-яких нових змін, внесених у код програми.

Статичний аналіз коду у вужчому трактуванні – це спрощена, автоматизована перевірка коду. Його застосування не вимагає виконання програми, не залежить від середовища розробки та може бути виконаним безпосередньо на етапі написання коду. Це набагато швидше, ніж перевірка вручну, і забезпечує повне охоплення коду. Своєчасне надання інформації про проблеми у вихідному коді суттєво знижує витрати часу та грошей у комерційній розробці. Національний Інститут Стандартів і Технологій (NIST) зазначає, що відносна вартість виправлення дефектів, виявлених на різних етапах розробки програмного забезпечення, кратно зростає з кожним наступним етапом розробки. Так, наприклад, оцінюється, що вартість виправлень коду, виконаних після публічного випуску програми, можуть досягати 30-кратної вартості порівняно з виправленнями на етапі її розробки [1].

Форми статичного аналізу та завдання, які він виконує, можуть бути досить різними, наприклад: виявлення вразливостей, допоміжні перевірки коду, де-дуплікація коду, оптимізація тощо. Інструменти статичного аналізу є високоефективними у забезпеченні певного рівня якості розробки, завдяки здатності автоматично виявляти типові помилки програмування (як ділення на нуль, вихід за межі масиву та багато інших). Під час цієї процедури, вихідний код піддається серії формальних тестів, які спираються на певну базу даних вразливостей та помилок, і, на основі цього, аналізатор надає список виявлених проблем. Однак, розв'язання цих проблем ще не гарантує абсолютного захисту від помилок у коді, оскільки кожна процедура статичного аналізу здатна визначати наявність проблеми у вихідному коді, але не може повністю гарантувати її відсутність.

Якщо інструменти статичного аналізу не можуть визначити, чи є об'єкт розгляду помилкою, вони видають попередження, яке сповіщає користувача про *потенційну* помилку в даному місці. Через ряд причин, таких як абстракції, що використовуються цими інструментами, та компроміс між точністю та масштабованістю аналізу, вони генерують досить велику кількість попереджень. Користувачеві потрібно вручну перевірити ці попередження та виділити справжні помилки серед інших – хибно позитивних результатів (помилкових спрацьовувань). Відповідно до дослідження, проведеного у [2], велика кількість помилкових попереджень та зусилля, необхідні для їх ручної перевірки, вважаються основними проблемами, пов'язаними з використанням інструментів статичного аналізу на практиці.

Останнім часом дослідники почали працювати в напрямку створення більш інтелектуальних систем аналізу коду, які могли б не тільки повідомляти про помилки на основі наявної бази знань, але й вчитися на помилках, зроблених у попередніх раундах аналізу, щоб підвищити свою точність у майбутніх. Використання алгоритмів машинного навчання в процесі статичного аналізу коду потенційно може бути підходом, який допоможе покращити можливості аналізу програмного забезпечення. Основною метою даної статті є аналіз можливих підходів до залучення методів машинного навчання для удосконалення традиційних механізмів статичного аналізу коду.

Аналіз програм на основі машинного навчання дає можливість виявляти проблеми на основі неоднозначної та часткової інформації. Вивчаючи загальні моделі програмного коду, такі методи можуть запропонувати гнучкі ймовірнісні результати для аналізу різних аспектів поведінки програми. При цьому, методи машинного навчання не обов'язково намагаються витіснити традиційні методи статичного аналізу. Натомість використання таких інструментів, як нейронні мережі, може бути цінним доповненням до загального набору методологій аналізу програмного забезпечення. Сфера використання машинного навчання в аналізі коду ще відносно молода, але наявні результати показують її потенціал.

Традиційні підходи статичного аналізу зазвичай проводять аналіз на основі чітких формальних методів і набору попередньо визначених інструкцій. Однак, визначення такого набору правил є виснажливим ручним завданням, яке досить важко масштабувати до широкого діапазону властивостей і програм. Формальні методи є незамінним інструментом під час роботи з програмами, де безпека має вирішальне значення, але водночас існує широкий спектр програм, які втрачають можливість отримати переваги від більш гнучких підходів аналізу програмного забезпечення. Аналіз програм на основі машинного навчання має на меті розв'язати цю проблему, але жертвує здатністю надавати гарантії. Перш за все, машинне навчання може допомогти впоратися з двома найпоширенішими джерелами неоднозначності: прихованими специфікаціями та неоднозначними контекстами виконання. Поєднання задач статичного аналізу, в класичному його розумінні, та методів машинного навчання, може мати досить різні форми. В рамках своєї роботи [3], M. Allamanis розділяє всі підходи із застосуванням методів машинного навчання в статичному аналізі програмного коду на 3 категорії: ***коригування специфікацій*** *(англ. specification tuning),* ***визначення специфікацій*** *(англ. specification inference)* та ***аналіз "чорного ящика"*** *(англ. black box analysis learning)*. Перші дві категорії є комбінованими підходами, що поєднують в собі застосування машинного навчання і традиційних інструментів, а третя категорія базується лише на використанні машинного навчання. У даній роботі, дана класифікація була взята за основу розгляду підходів застосування нейронних мереж в статичному аналізі коду, і в наступних трьох підрозділах кожну із вказаних форм буде розібрано детальніше.

Коригування специфікацій — це підхід, у якому модель нейронної мережі використовується, щоб уточнити та покращити специфікації, отримані в результаті "класичного" статичного аналізу

коду. В літературі даний підхід також часто зустрічається як "*постобробка (англ. postprocessing) попереджень статичного аналізу*". Метою коригування специфікацій є надання більш лаконічних і точних уявлень про очікувану поведінку програми, з точки зору її потенційних проблем та вразливостей.

Коли експерт має намір створити високонадійний аналіз програми та налаштовує занадто чутливі правила, це зазвичай призводить до високого рівня хибно позитивних результатів (помилкових попереджень про наявність проблем в коді). Виклик великої кількості помилкових попереджень призводить до ігнорування користувачами істинно позитивних результатів, що сильно знижує користь такого аналізу [3]. Через це, з метою розв'язання проблем хибно позитивних результатів, почали з'являтись дослідження із використанням методів машинного навчання для коригування (постобробки) аналізу програми шляхом вивчення того, які аспекти формального аналізу можуть бути відкинутими для підвищення фінальної точності [4].

Таким чином, коригування специфікацій полягає саме в обробці попереджень після їх генерації, а його основними цілями є зменшення кількості отримуваних попереджень та зусиль, необхідних для їх ручної перевірки. Зазвичай, застосування обробки попереджень статичного аналізу в даному підході приймає одну з наступних форм [4]:

- зменшення кількості попереджень перед наданням їх користувачеві;
- зменшення зусиль, необхідних для ручної перевірки, шляхом розширення попереджень додатковою інформацією;
- спрощення процесу ручної перевірки, шляхом асистування чи надання допоміжних інструментів під час перевірки попереджень.

Враховуючи всі перспективи, наявні дослідження пропонують широкий набір підходів до постобробки попереджень, методи реалізації яких досить різноманітні. В дослідженні публікацій сфери постобробки попереджень статичного аналізу, виконаному в роботі [2], автори розділяють всі підходи на шість категорій, які будуть розглянуті нижче.

З метою зменшення загальної кількості попереджень, які потрібно переглянути для виявлення загальних шаблонів чи безпосередніх причин дефектів, може застосовуватись кластеризація попереджень. Цей підхід полягає в об'єднанні схожих попереджень в декілька груп (кластерів) на основі подібності або кореляції між ними. Подібність попереджень може визначатись за такими критеріями, як тип дефекту, розташування в коді або серйозність попередження.

Для забезпечення кращого управління списком попереджень, його можна ранжувати відповідно до їх пріоритету чи важливості. Таке упорядкування може виконуватись на основі різних характеристик попереджень, таких як: розміщенні у вихідному коді, критичності впливу дефекту на систему, ймовірністю істинно позитивного результату, попередній історії виправлення помилок, зусиллями, необхідними для його усунення тощо. Такий підхід може допомогти користувачеві зосередитися спершу на найбільш релевантних попередженнях, ігноруючи менш актуальні.

Зменшення загальної кількості попереджень також може відбуватись шляхом усунення певної їх частини. За такого підходу, усі попередження класифікуються на дві категорії: дійсні та недійсні, після чого недійсні попередження прибираються. Таким чином, усуваються ті попередження, які вважаються зайвими, нерелевантними чи нецікавими з певних причин, наприклад: дублікати, конкретні типи дефектів, дефекти, що не мають критичного впливу тощо. Усунення попереджень може відбуватись, як на основі синтаксичного чи семантичного аналізу, так і на основі налаштувань самого користувача. Завдяки звуженню діапазону попереджень, які потребують опрацювання, даний підхід може допомогти зберегти час та ресурси користувача.

Менша кількість попереджень може досягатись шляхом зниження рівня хибно позитивних результатів. Для цього, отримані попередження перевіряються за допомогою додаткових методів, що спираються на додаткову інформацію в коді й здатні автоматично ідентифікувати та усунути хибно позитивні результати. Так помилкові попередження можуть бути усунуті без ручної перевірки, що, своєю чергою, збільшує довіру користувача до решти отриманих попереджень.

. Покращення якості отриманих результатів може також відбуватись за допомогою динамічного аналізу. В такому випадку, результати інструментів статичного аналізу передаються на обробку інструментами динамічного аналізу для створення тестів, що підтверджують наявність помилки. Цей підхід дозволяє підвищити точність та комплексність виявлення дефектів, і зменшити кількість хибно позитивних та хибно негативних результатів.

Ручна перевірка попереджень може бути полегшена наданням додаткової інформації чи забезпеченням користувача інструментальною підтримкою. Така підтримка може приймати вигляд пояснень, рекомендацій, візуалізації, інтерактивних інтерфейсів тощо, і може здійснюватись за допомогою різних методів, таких як генерація природної мови, системи підтримки прийняття рішень, надання графічного інтерфейсу, або гейміфікації. Такий підхід може допомогти підвищити ефективність та результативність ручної обробки попереджень, а також зберегти час та зусилля користувача.

Визначення специфікацій — це підхід, у якому модель машинного навчання використовується для того, щоб краще виділяти специфікації, присутні в програмному коді. Метою визначення специфікацій є створення більш точного та повного представлення очікуваної поведінки програми. Вважається, що більшість коду відповідає деяким прихованим специфікаціям, і саме тому можуть бути корисними моделі машинного навчання, що здатні вивчати шаблони в програмному коді та виводити приховані форми цих специфікацій [3]. Таким чином, даний підхід передбачає застосування певних перетворень чи модифікацій до вихідного програмного коду перед використанням інструментів статичного аналізу, з метою покращення якості, або ефективності виявлення дефектів.

В задачах проведення попередньої обробки програмного коду, серед методів машинного навчання на сьогодні найбільш результативними є *рекурентні* та *графові нейронні мережі*.

Рекурентні нейронні мережі (англ. Recurrent neural networks, або RNN) дуже добре справляються з обробкою послідовностей, в тому числі таких, як природна мова чи вихідний програмний код. Вони не потребують заздалегідь заданих параметрів, а натомість здатні працювати з послідовностями токенів: необроблений вихідний код розбивається на невеликі одиниці (наприклад, "*int*", "*func*", "*{*" та "*}*") і представляється в моделі у вигляді їх послідовності. RNN можуть захоплювати як синтаксичну, так і семантичну інформацію коду та використовувати її для виконання різноманітних завдань, наприклад, класифікації [5, 6].

Графові нейронні мережі (англ. Graph neural networks, GNN) є типом нейронних мереж, призначених для обробки даних з графовою структурою, таких як абстрактні синтаксичні дерева (англ. abstract syntax trees, AST) або графи управління потоком програми (англ. control flow graphs, CFG). Вони навчаються на основі вузлів та ребер графу, захоплюючи структурну та контекстуальну інформацію коду, використовуючи її для класифікації. GNN використовуються для моделювання структури та залежностей програмного коду, і можуть бути поєднані з інструментами статичного аналізу, щоб покращити точність аналізу програмного забезпечення. Наприклад, система ASTGNN [7] використовує графову нейронну мережу для моделювання AST програмного коду та виявлення помилок програмного забезпечення [3, 5].

Попередня обробка програмного коду за допомогою машинного навчання може приймати різні форми. В загальному, їх можна розділити на три основні групи: *вилучення властивостей (англ. feature extraction)*, *відбір властивостей (англ. feature selection)*, та *перетворення коду (англ. code transformation)*.

Вилучення властивостей – вилучення відповідних функцій чи атрибутів із вихідного коду, на основі яких інструменти статичного аналізу матимуть можливість ефективніше та точніше визначати дефекти. Наприклад, методи машинного навчання можуть використовуватись для вилучення синтаксичних, семантичних або структурних властивостей коду за допомогою таких структур, як абстрактні синтаксичні дерева (AST), графи потоку даних (англ. data flow graphs, DFG), графи потоку керування (CFG) і т.д.

Відбір властивостей – вибір підмножини функцій, які є найбільш відповідними або інформативними для виявлення дефектів, та відкидання решти. Такий підхід може допомогти зменшити розмірність і складність обробки коду та покращити продуктивність інструментів статичного аналізу, а також сфокусувати їх на найбільш важливих ділянках коду. Наприклад, методи машинного навчання можуть використовуватись із застосуванням методів фільтрації (англ. filter methods) або обгортання (англ. wrapper methods) для вибору властивостей на основі певних критеріїв, таких як кореляція, взаємна інформація або точність класифікації.

Перетворення коду – модифікація вихідного коду без зміни його функціональності та поведінки, з метою зробити його більш зручним чи сумісним для інструментів статичного аналізу. Тобто, підвищення ефективності статичного аналізу відбувається внаслідок поліпшення представлення та якості коду. Наприклад, методи машинного навчання можуть

використовуватись для застосування методів рефакторингу коду, нормалізації коду, обфускації або синтезу коду, з метою надання коду більш придатної для обслуговування форми.

Аналіз "чорного ящика" — це підхід, при якому для аналізу програми використовується лише модель машинного навчання, тренування якої відбувається безпосередньо на вихідному коді. В такому підході, модель діє за принципом "чорного ящика", генеруючи попередження в якості результатів аналізу програми, але при цьому ніколи явно не формулює конкретні специфікації.

Така форма аналізу програм є дуже гнучкою та може виходити за межі можливостей багатьох традиційних підходів. Однак, в обмін на свою гнучкість, аналіз "чорного ящика" жертвує ясністю та не надає жодних гарантій. Відсутність прозорості в процесі прийняття рішень моделі може ускладнити розуміння того, чому виникли певні попередження, або які кроки слід вжити для їх вирішення. А відсутність конкретних специфікацій означає відсутність гарантій, що отримані попередження точні, або що вони повністю покривають усі необхідні вимоги щодо потенційних проблем в коді. Тому, під час практичного застосування такого підходу, необхідно враховувати ці особливості, особливо у випадках, коли потрібен високий рівень впевненості в результатах аналізу. [3]

Як уже зазначалось, однією з особливостей застосування нейронних мереж в аналізі програмного коду є їх гнучкість та здатність виходити за рамки можливостей традиційних підходів. Завдяки цьому, вони мають досить широкий спектр можливостей практичного застосування, частину з яких розглянемо далі.

Автоматизоване виявлення дефектів (англ. automated defect detection). Основною задачею статичного аналізу є виявлення дефектів. Тому, одним із найбільш очевидних способів застосування методів машинного навчання в аналізі програмного коду є використання моделі безпосередньо для виявлення дефектів у ньому. Такий процес передбачає побудову класифікаторів, які здатні визначати області коду, що потенційно містять дефекти, спираючись на шаблони в програмному коді та використовуючи таку інформацію, як синтаксис, складність коду, семантику чи історію змін.

У контексті статичного аналізу програмного коду, під *дефектами* розуміються будь-які відхилення між очікуваною та реальною поведінкою програми. З точки зору комплексності проблем та направленості аналізу, в загальному, дефекти можуть бути розділені на три рівні (категорії): *синтаксичні дефекти*, *семантичні дефекти* та *вразливості*. Нижче, у таблиці 1, описано кожен із цих рівнів [6]. *Автоматизоване виправлення дефектів (англ. automated defect correction).* Наступним, більш амбіційним етапом після виявлення дефектів є їх автоматизоване виправлення. У такому процесі передбачається, що модель має автоматично визначити правки, необхідні для виправлення конкретної помилки, які надалі можуть бути застосовані з мінімальним втручанням, або й без втручання людини. У порівнянні з виявленням, виправлення є набагато складнішою задачею, яка стала реалістичною темою дослідження лише нещодавно, завдяки можливості використання останніх напрацювань сфери *обробки природних мов* (*англ. natural language processing*, *NLP*). За такого підходу навчання моделей найчастіше виконується або на вибірках з парами неправильних та правильних елементів програм, де модель навчається перетворювати одні на інші, або на вибірках з правильними екземплярами, де модель навчається перетворювати програми, які мають від них відхилення [6]. Одним із прикладів, що демонструють перспективи застосування методів обробки природних мов до програмного коду, є нещодавній революційний проєкт у сфері NLP – *Chat GPT* [8], який іноді здатен демонструвати досить хороші результати у питаннях генерації та виправлення коду, хоч все ще і потребує значного ручного контролю результатів.

Асистування при перевірці коду вручну (англ. code review assistance). Одним із важливих інструментів у забезпеченні якості розробки програмного забезпечення є *перевірка коду вручну*, яка призначена не лише для пошуку помилок та покращення загальної якості програмного забезпечення, а й для постійної передачі знань між розробниками. У великих проєктах якісна перевірка коду зазвичай є складним завданням. Оскільки вона сильно залежить від індивідуальних можливостей рецензентів, часто залишається певний набір змін вихідного коду, що не був перевірений належним чином [9]. Підняття якості та швидкості ручної перевірки коду може досягатися шляхом використання методів машинного навчання, які будуть автоматично визначати фрагменти коду, що потребують особливої уваги та додаткової перевірки. Такий спосіб поєднання автоматизації з перевіркою коду вручну може покращити якість рецензування коду, зберігаючи при цьому можливість підтримувати передачу знань між розробниками [10].

Асистування при ручній перевірці коду часто може опиратись на підходи, що використовуються при автоматизованому виявленні дефектів. Однією з перспективних технік є використання машинного навчання в поєднанні з системою контролю версій і системою відстеження помилок. Передбачення дефектів на основі ітерацій змін коду може використовуватись для зосередження більшої уваги розробників на потенційно проблемних ділянках коду під час етапу ручної перевірки. Завдяки такому асистуванню може підтримуватись постійний базовий рівень якості перегляду коду, що буде зменшувати ймовірність непоміченого проходження помилок в кодову базу, і, як результат, – суттєво покращувати кінцеву якість програмного забезпечення з меншими експлуатаційними витратами [9; 10]. *Автоматичне оформлення / форматування коду (англ. automatic code stylizing / formatting).* Підтримка сталості стилю коду є дуже важливим аспектом розробки програмного забезпечення. Такі елементи форматування, як відступи, найменування, коментування, порожні рядки та потік керування (порядок команд, інструкцій та викликів) мають досить сильний вплив на зрозумілість вихідного коду [11]. Автоматичні інструменти форматування можуть значно покращити якість оформлення коду та зменшити зусилля розробників необхідні для його підтримки. Такі інструменти особливо актуальні у великих проєктах, де постійний контроль будь-яких змін коду є дуже складним. Здебільшого, системи автоматичного форматування коду побудовані на основі правил, проте, вони також можуть будуватись на основі машинного навчання [12; 13].

Таблиця 1.
Категорії дефектів програмного коду

Категорія дефекту	**Де проявляється**	**Приклади**
Синтаксичні дефекти *(порушення граматики мови)*	На етапах ***компіляції*** та ***запуску*** коду	Відсутні чи неправильно розміщені дужки, оператори, помилки в назвах ключових слів або змінних
Семантичні дефекти *(помилки в логіці)*	Під час виконання програма не відповідає передбаченій ***поведінці*** (висока складність виявлення)	Використання змінних, структур даних, алгоритмів не за призначенням
Вразливості	***Семантичні*** дефекти, які загрожують ***безпеці***, ***цілісності*** або ***доступності*** системи чи її даних	Переповнення буфера, переповнення цілих чисел, міжсайтовий скриптинг, "завислі" вказівники

Вибір стилю форматування коду часто є суб'єктивним питанням, яке може викликати дискусії між розробниками. Через це, інструменти форматування повинні підтримувати досить широкі можливості налаштувань, щоб бути здатними задовольняти потреби різних команд розробки. Робота з системами на основі правил часто потребує чимало зусиль. Оскільки кожна специфікація передбачає форматування для конкретного стилю, кожен новий стиль вимагає складних модифікацій або створення нових наборів правил. Навіть незначні зміни у граматиці можуть потребувати адаптації правил форматування. При цьому наявність величезної кількості правил у таких системах є досить типовою ситуацією на практиці.

Механізми форматування, засновані на машинному навчанні, зазвичай працюють у два етапи: навчання та форматування. Навчання моделі відбувається безпосередньо на базі корпусу

коду із бажаним форматуванням. В результаті даного етапу, формується статистична модель, яка представляє стиль форматування виділений із навчальних даних. На етапі форматування модель будує прогнози щодо того, як автор корпусу відформатував би певний контекст, повертаючи директиву форматування [13].

Виявлення "коду, що погано пахне" (англ. code smell detection). В процесі підтримки та покращення програмного забезпечення, розробники повинні регулярно вносити в нього зміни, щоб відповідати новим вимогам, поліпшувати функціональність та виправляти виявлені дефекти. Проте, обмежений час та інші фактори можуть сильно ускладнювати можливість підтримки якісної архітектури та знаходження оптимальних рішень. В результаті цього, часто порушується початковий дизайн системи та виникає так званий "код, що (погано) пахне". "Запах" коду був визначений М. Фаулером та К. Беком [14] як ознаки, що вказують на неоптимальний дизайн або варіанти реалізації у вихідному коді. Вони представляють одну із найсерйозніших форм технічної заборгованості (*англ. tech debt*) й не лише знижують загальну зрозумілість коду, а й роблять його більш схильним до появи помилок. Таким чином, "код, що пахне" є усталеною характеристикою вихідного коду, що вказує на потенційну наявність більших проблем. Їх усунення здійснюється за допомогою рефакторингу [15; 16; 17].

Задача виявлення "коду, що пахне" є дуже подібною до задачі виявлення дефектів, а тому, аналогічно до неї, має схожий набір викликів, пов'язаний із використанням інструментів виявлення на основі правил. Перш за все, важливим питанням є коректність налаштування правил. Розробники можуть мати різні погляди на допустимість певних властивостей у коді, тому метрики, що використовуються для виявлення запахів коду різними інструментами, можуть суттєво різнитися. Крім того, навіть якщо метрики однакові, їх порогові значення можуть змінюватися. Це може призводити до збільшення або зменшення кількості виявлених "запахів". Окрім цього, не менш важливою проблемою є точність результатів. Як і при виявленні дефектів, системи, що базуються на правилах, часто схильні до хибно позитивних результатів при виявленні "запахів" коду. З усіх цих причин, нещодавньою тенденцією є поява нових підходів до виявлення запахів коду на основі методів машинного навчання. Основною перевагою таких підходів є можливість навчати модель використовуючи достатньо велику кількість даних, наявних в тому ж, або в інших подібних проєктах, замість того, щоб опиратись на попередньо визначені порогові значення метрик [15].

На рис. 1 представлено загальну структуру категоризації підходів до застосування машинного навчання в статичному аналізі коду.

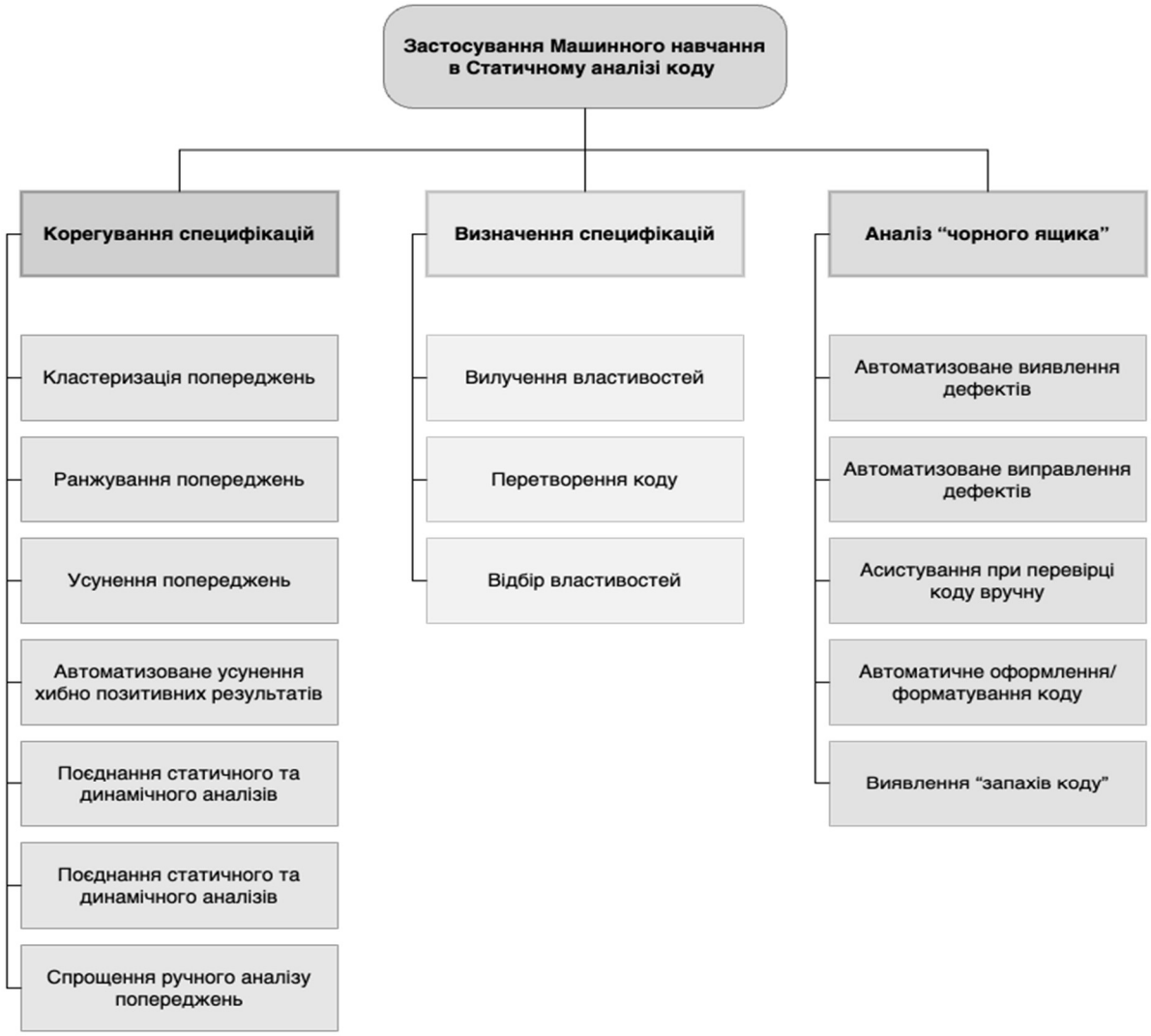

Рис. 1. Підходи до застосування машинного навчання в статичному аналізі

Враховуючи всі особливості розглянутих підходів до використання нейронних мереж в статичному аналізі коду, можна сказати, що *коригування специфікацій* та *аналіз "чорного ящика"* виглядають як більш перспективні рішення з точки зору практичного застосування. Підходи із категорії *визначення специфікацій*, на поточний момент, не виглядають як ті, що можуть запропонувати певні унікальні переваги, порівняно з іншими. При цьому, вони не уникають більшості викликів та ускладнень, пов'язаних із навчанням нейронних мереж безпосередньо на вихідному коді та не надто спрощують складність побудови статичних аналізаторів з огляду на необхідність конфігурації наборів правил. Звісно, успішні реалізації підходів з цієї категорії потенційно можливі, проте порядок застосування нейронних мереж та класичних алгоритмів статичного аналізу в них не виглядає оптимальним, що також підтверджується досить малою кількістю дослідницьких робіт у цьому напрямку.

Як *коригування специфікацій*, так і *аналіз "чорного ящика"* мають власні переваги та недоліки. В той час, як перша категорія орієнтована на піднятті точності результатів зі збереженням надійності класичних підходів, друга – спрямована на отримання більшої гнучкості комплексного аналізу, допускаючи при цьому зниження рівня зрозумілості та надійності. Завдяки тому, що підходи з категорії *коригування специфікацій* орієнтуються на вдосконаленні статичного аналізу з використанням наявних аналізаторів на базі правил, вони мають досить хороший потенціал для отримання нових інструментів аналізу, які матимуть вищу точність, не поступаючись при цьому їм у надійності. Особлива увага у подальших дослідженнях має приділятись застосуванню ансамблевого навчання (англ. *ensamble learning*) на базі кількох статичних аналізаторів. У такій формі, ансамблеве навчання передбачає поєднання результатів кількох інструментів статичного аналізу та використання моделі нейронної мережі для

постобробки і порівняння результатів кожного із них. Такий підхід, в перспективі, повинен покращувати точність аналізу завдяки "взаємному" підтвердженню аналізаторами повідомлень про потенційні дефекти, знижуючи таким чином вплив помилок окремих інструментів.

Висновки

Активна розробка програмного забезпечення – це неперервні зміни програмного коду. Для підвищення його цінності, розробникам необхідно вносити новий чи покращувати наявний функціонал, виправляти виявлені помилки та вдосконалювати архітектуру рішень. Усе це несе потенційну небезпеку появи нових дефектів у програмній системі. При роботі з великою кодовою базою майже неможливо знайти та усунути всі помилки, проте критично важливо зводити їх присутність до певного прийнятного мінімуму. Головне питання, яке виникає під час усунення несправностей вихідного коду, полягає в тому, як зробити це найбільш ефективно.

Враховуючи те, що з кожним наступним етапом розробки витрати на внесення виправлень зростають, логічним висновком є те, що основні зусилля для виявлення дефектів повинні прикладатись якомога раніше. Саме тому статичний аналіз коду є одним із найефективніших інструментів покращення процесу виявлення несправностей, оскільки здатен працювати вже на етапі появи перших зразків коду.

Класичні підходи статичного аналізу базуються на алгоритмах зіставлення шаблонів. Це означає, що вони сканують вихідний код, намагаючись виявити помилкові шаблони, порівнюючи їх із тими, що присутні в їхній базі даних. Недоліком такого підходу є його нездатність навчатися та адаптувати свою поведінку, коли виявлення не вдається, що може призводити до високого рівня хибно позитивних результатів і, своєю чергою, знецінювати корисність інструменту для розробників.

Використання таких підходів машинного навчання, як нейронні мережі, має хороші перспективи підвищення гнучкості систем та розширення їх можливостей. Підтримка методів машинного навчання в таких системах має забезпечувати їм здатність навчатися на попередніх виявленнях та зменшувати кількість хибно позитивних результатів, отримуючи високий рівень істинно позитивних результатів. При цьому, важливо відзначити, що використання нейронних мереж у статичному аналізі далеко не завжди означає відмову від класичних підходів, які добре зарекомендували себе у багатьох задачах. Відповідно до проведеного дослідження, методи машинного навчання можуть як повністю заміняти традиційні підходи (*аналіз "чорного ящика"*), так і використовуватись у комбінації з ними, для забезпечення більшої точності та ефективності (*коригування та визначення специфікацій*). У будь-якому випадку, побудова більш досконалих систем статичного аналізу змушує шукати нові рішення в області поєднання гнучких за природою нейронних мереж та детермінованих та надійних класичних алгоритмів.

Література

1. Тессі, Г. (2002). Економічні наслідки неадекватної інфраструктури для тестування програмного забезпечення. Технічний звіт. NIST, 309 стор.

2. Муске Т., Серебреник А. (2022). Огляд підходів до постобробки сповіщень статичного аналізу. ACM Computing Surveys (CSUR), № 55, стор. 1 - 39, 10.1145/3494521

3. Алламаніс, М. (2022). Графічні нейронні мережі в програмному аналізі. Wu, L., Cui, P., Pei, J., Zhao, L. (eds) Graph Neural Networks: Foundations, Frontiers, and Applications. Springer, Сінгапур, стор. 483 - 497. 10.1007/978-981-16-6054-2_22

4. Муске Т. Б. (2020). Постобробка тривог статичного аналізу. [Phd Thesis 1 (Research TU/e / Graduate TU/e), Mathematics and Computer Science]. Технологічний університет Ейндховена, 174 с.

5. Еррамредді, С., Мордал, А., Кок, У., Вей, С., Фостер, Дж., Карпуат, М., Портер, А. (2023). Емпірична оцінка підходів машинного навчання для сортування звітів інструментів статичного аналізу. Емпірична інженерія програмного забезпечення, вип. 28. 10.1007/s10664-022-10253-z

6. Мар'янов Т., Пащенко І., Массаччі Ф. (2022). Машинне навчання для виявлення вразливостей у вихідному коді: що працює, а чого ще немає. IEEE Security & Privacy, том. 20, № 5, С. 60-76. 10.1109/MSEC.2022.3176058

7. Чжан, З., Лу, С., Хуан, З., Чжао, З. (2022). ASGNN: графічні нейронні мережі з адаптивною структурою. ArXiv, 10.48550/arXiv.2210.01002

8. ChatGPT. OpenAI. Отримано з: https://openai.com/blog/chatgpt

9. Лал Х., Пахва Г. (2017). Аналіз аналізу коду програмної системи з використанням методів машинного навчання. 11-та Міжнародна конференція з інтелектуальних систем і управління (ISCO), Коїмбатор, Індія, стор. 8 - 13. 10.1109/ISCO.2017.7855962

10. Мадера, М., і Томон, Р. (2017). Випадкове дослідження моделі машинного навчання для експертної системи перевірки коду в програмній інженерії. Федеративна конференція з комп'ютерних наук та інформаційних систем (FedCSIS), стор. 1357-1363.

11. Міара Р., Масселман Дж., Наварро Дж., Шнейдерман Б. (1983). Програмний відступ і зрозумілість. Комун. ACM, № 11, вип. 26, стор. 861-867. 10.1145/182.358437

12. Рейс, С. (2007). Автоматична стилізація коду. Матеріали 22-ї міжнародної конференції IEEE/ACM з автоматизованої розробки програмного забезпечення, 10.1145/1321631.1321645

13. Парр Т., Вінджу Дж. (2016). На шляху до універсального форматувальника коду через машинне навчання. Матеріали міжнародної конференції ACM SIGPLAN 2016 року з розробки мови програмного забезпечення, 10.1145/2997364.2997383

14. Фаулер М., Бек К., Брант Дж., Опдайк В., Робертс Д. (1999). Refactoring: Improving the Design of Existing Code, 1st ed. Addison-Wesley Professional, 464 стор. ISBN 978-0201485677

15. Фонтана, Ф., Ментюля, М., Заноні, М., Маріно, А. (2016). Порівняння та експериментування методів машинного навчання для виявлення запаху коду. Емпірична інженерія програмного забезпечення, № 21, стор. 1143-1191. 10.1007/s10664-015-9378-4

16. Азім, М., Паломба, Ф., Ши, Л., Ван, К. (2019). Методи машинного навчання для виявлення запаху коду: систематичний огляд літератури та мета-аналіз. Інф. програмне забезпечення техн., С. 115-138. 10.1016/J.INFSOF.2018.12.009

17. Деванган С., Рао Р., Мішра А., Гупта М. (2022). Виявлення запаху коду за допомогою алгоритмів ансамблевого машинного навчання. Прикладні науки, 12(20), 10.3390/app122010321

References

1. *Tassey, G.* (2002). The economic impacts of inadequate infrastructure for software testing. *Technical Report. NIST*, 309 p.

2. Muske, T., Serebrenik, A. (2022). Survey of Approaches for Postprocessing of Static Analysis Alarms. *ACM Computing Surveys (CSUR)*, No 55, pp. 1 - 39, 10.1145/3494521

3. Allamanis, M. (2022). Graph Neural Networks in Program Analysis. *Wu, L., Cui, P., Pei, J., Zhao, L. (eds) Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer, Singapore, pp. 483 - 497. 10.1007/978-981-16-6054-2_22

4. Muske, T. B. (2020). Postprocessing of static analysis alarms. *[Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Eindhoven University of Technology,* 174 p.

5. Yerramreddy, S., Mordahl, A., Koc, U., Wei, S., Foster, J., Carpuat, M., Porter, A. (2023). An empirical assessment of machine learning approaches for triaging reports of static analysis tools. *Empirical Software Engineering*, vol. 28. 10.1007/s10664-022-10253-z

6. Marjanov, T., Pashchenko, I., Massacci, F. (2022). Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet. *IEEE Security & Privacy*, vol. 20, No. 5, pp. 60-76. 10.1109/MSEC.2022.3176058

7. Zhang, Z., Lu, S., Huang, Z., Zhao, Z. (2022). ASGNN: Graph Neural Networks with Adaptive Structure. *ArXiv*, 10.48550/arXiv.2210.01002

8. ChatGPT. OpenAI. Retrieved from: https://openai.com/blog/chatgpt

9. Lal, H., Pahwa, G. (2017). Code review analysis of software system using machine learning techniques. *11th International Conference on Intelligent Systems and Control (ISCO)*, Coimbatore, India, pp. 8 - 13. 10.1109/ISCO.2017.7855962

10. Madera, M., & Tomon, R. (2017). A case study on machine learning model for code review expert system in software engineering. *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1357-1363.

11. Miara, R., Musselman, J., Navarro, J., Shneiderman, B. (1983). Program indentation and comprehensibility. *Commun. ACM*, No. 11, vol. 26, pp. 861-867. 10.1145/182.358437

12. Reiss, S. (2007). Automatic code stylizing. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 10.1145/1321631.1321645

13. Parr, T., Vinju, J. (2016). Towards a universal code formatter through machine learning. *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering,* 10.1145/2997364.2997383

14. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. (1999). Refactoring: Improving the Design of Existing Code, 1st ed. Addison-Wesley Professional, 464 p. ISBN 978-0201485677

15. Fontana, F., Mäntylä, M., Zanoni, M., Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, No. 21, pp. 1143-1191. 10.1007/s10664-015-9378-4

16. Azeem, M., Palomba, F., Shi, L., Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Inf. Softw. Technol.*, pp. 115-138. 10.1016/J.INFSOF.2018.12.009

17. Dewangan, S., Rao, R., Mishra, A., Gupta, M. (2022). Code Smell Detection Using Ensemble Machine Learning Algorithms. *Applied Sciences*, 12(20), 10.3390/app122010321