БАКАЛАВРСЬКА РОБОТА

БР. ІП - 21.00.00.000 ПЗ

Група ІП-21-4

Гусля Віталій

2025

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Гусля Віталій Васильович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Використання машинного навчання для автоматизації процесів генерації коду та його оптимізації
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____ Гусля Віталій Васильович_____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Романишин Юлія Любомирівна, проф., д.т.н_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2024 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Гуслі Віталію Васильовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Використання машинного навчання для автоматизації процесів генерації коду та його оптимізації"

керівник проекту (роботи) д.т.н., проф. Романишин Ю.Л.

затверджені наказом вищого навчального закладу від “ 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Моделі представлення мовних структур
2. Підходи до автоматичної генерації програм
3. Дослідження методів побудови ознак і навчання моделей ..

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Приклад генерації коду QVTr M2M (рис.2.1, ст.23).
2. Метамодель дерев розбору (рис.2.2, ст.30).
3. Огляд системи пошуку ознак. (рис.3.1, ст.38)
4. Машинне навчання, що використовується для визначення якості функції (рис.3.2, ст.39)..
5. Спрощена підмножина автоматично створеної граматики (рис.3.15, ст.58)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обґрунтування теми роботи	15.02.2025	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	25.02.2025	виконано
3	Побудова моделі або алгоритму власного рішення	15.03.2025	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	25.04.2025	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2025	виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 60 сторінок, 17 рисунків, список використаних джерел із 20 найменування,

Метою роботи є дослідити сучасні підходи до використання машинного навчання в автоматизації генерації програмного коду, проаналізувати існуючі мовні моделі, методи побудови ознак і навчання, а також оцінити ефективність цих підходів для створення якісного та оптимізованого програмного забезпечення.

Об'єкт дослідження: Процес автоматизації програмування, зокрема генерації та оптимізації програмного коду.

Предмет дослідження: Методи машинного навчання та мовні моделі, що використовуються для автоматичної генерації та оптимізації коду, включаючи представлення мовних структур, побудову ознак і навчання моделей.

Результати дослідження: у дослідженні проаналізовано мовні моделі та методи машинного навчання, що застосовуються для автоматичної генерації й оптимізації програмного коду, а також проведено експериментальну оцінку їх ефективності.

Першийу розділ присвячений аналізу мовних моделей, які дозволяють формалізувати синтаксис і семантику програмних мов для подальшої генерації коду.

В другому розділі висвітлено сучасні методи створення коду за допомогою машинного навчання та прикладних алгоритмів синтезу.

Третій розділ присвячено практичній реалізації підготовки даних, побудові ознак та експериментальному навчанню моделей генерації коду

Висновок: Машинне навчання, зокрема моделі глибокого навчання, демонструє високу ефективність у задачах генерації та оптимізації коду, що підтверджує його перспективність для автоматизації процесів програмування.

КЛЮЧОВІ СЛОВА: МАШИННЕ НАВЧАННЯ, ГЕНЕРАЦІЯ КОДУ, ГЛИБОКЕ НАВЧАННЯ, МОВНІ МОДЕЛІ, ТРАНСФОРМЕРИ, ГРАМАТИКА.

ABSTRACT

The bachelor's thesis contains 60 pages, 17 figures, a list of used sources with 20 names,

The purpose of the work is to investigate modern approaches to the use of machine learning in the automation of program code generation, to analyze existing language models, methods of feature construction and training, and to evaluate the effectiveness of these approaches for creating high-quality and optimized software.

Object of research: The process of programming automation, in particular the generation and optimization of program code.

Subject of research: Machine learning methods and language models used for automatic code generation and optimization, including the representation of language structures, feature construction and model training.

Research results: the study analyzed language models and machine learning methods used for automatic code generation and optimization, and also conducted an experimental assessment of their effectiveness.

The first section is devoted to the analysis of language models that allow formalizing the syntax and semantics of programming languages for further code generation.

The second section highlights modern methods of code generation using machine learning and applied synthesis algorithms.

The third section is devoted to the practical implementation of data preparation, feature construction and experimental training of code generation models

Conclusion: Machine learning, in particular deep learning models, demonstrates high efficiency in code generation and optimization tasks, which confirms its potential for automating programming processes.

KEYWORDS: MACHINE LEARNING, CODE GENERATION, DEEP LEARNING, LANGUAGE MODELS, TRANSFORMERS, GRAMMAR

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

MDE - Інженерія, керована моделями

UML - Уніфікована мова моделювання

ML - машинне навчання

AST - абстрактними синтаксичними деревами

OCL - мови обмежень об’єктів

DSL - Доменно-орієнтовані мови

AST - абстрактні синтаксичні дерева

TSG - граматика заміни дерева

MLP - багатошаровий персептрон

GRU - стробовані рекурентні одиниці

LSTM - довготривала короткочасна пам’ять

EGL - Epsilon Generation Language

SGD - стохастичний градієнтний спуск

ЗМІСТ

ВСТУП

Актуальність теми дослідження. Інженерія, керована моделями (MDE), має багато потенційних переваг для розробки програмного забезпечення, як засіб для представлення та керування основними бізнес-концепціями та правилами у вигляді моделей програмного забезпечення, таким чином гарантуючи, що ці активи електронного бізнесу протягом тривалого часу зберігаються незалежно від платформи. Незважаючи на довгострокові переваги MDE, багато підприємств і організацій не бажають застосовувати його через високі початкові витрати та необхідні спеціальні навички. Наше дослідження має на меті усунути ці перешкоди, дозволивши фахівцям загального програмного забезпечення застосовувати методи MDE за допомогою використання спрощених нотацій і забезпечуючи підтримку ШІ для процесів MDE. Однією з сфер, де були особливі проблеми для промислових користувачів

MDE, є визначення та обслуговування генераторів коду [32]. Генерація коду в контексті інженерії, керованої моделлю (MDE), передбачає автоматизований синтез ¬виконуваного коду, як правило, для цільової мови третього покоління (3GL), такої як Java, C, C++, Go або Swift, із специфікації програмного забезпечення, визначеної як одна або більше моделей на мові моделювання, такій як Уніфікована мова моделювання (UML) з формальними обмеженнями мови обмежень об’єктів (OCL) [28] або в доменно-специфічній. мова моделювання (DSL). Генерація коду MDE має потенційні значні переваги у зниженні вартості виробництва коду та покращенні якості коду завдяки забезпеченню використання систематичного архітектурного підходу в реалізації системи. Однак ручна конструкція таких генераторів коду може потребувати значних зусиль і потребуватиме спеціальних знань у мовах перетворення, які використовуються. Наприклад, для створення одного генератора коду UML для Java було потрібно кілька людино-років роботи [7]. Щоб зменшити знання та людські ресурси, необхідні для розробки генераторів коду, ми визначаємо новий підхід символічного машинного навчання (ML) для автоматичного створення правил генерації коду на основі прикладів перекладу. Ми називаємо цей підхід

створенням коду на прикладі або CGBE. Основою CGBE є вивчення міждеревних відображень між абстрактними синтаксичними деревами (AST) прикладів вихідної мови та відповідних прикладів цільової мови. Набір стратегій пошуку використовується для постулювання та перевірки потенційних відображень дерева до дерева між мовними AST. Як правило, вихідна мова є підмножиною уніфікованої мови моделювання (UML) і мови обмежень об'єктів (OCL), а цільовою мовою є мова програмування, наприклад Java або Kotlin. Однак ця методика в принципі застосовна для вивчення відображень між будь-якими мовами програмного забезпечення, які мають точні визначення граматики. Підхід CGBE для UML/OCL було оцінено на широкому діапазоні цільових мов програмування, і результати показали, що велика частина відповідних генераторів коду зазвичай може бути синтезована з прикладів, таким чином значно зменшуючи ручні зусилля та досвід, необхідні для створення генератора коду.

Метою роботи є дослідити сучасні підходи до використання машинного навчання в автоматизації генерації програмного коду, проаналізувати існуючі мовні моделі, методи побудови ознак і навчання, а також оцінити ефективність цих підходів для створення якісного та оптимізованого програмного забезпечення.

Для досягнення поставленої мети в роботі необхідно вирішити **наступні завдання.** Проаналізувати сучасні мовні моделі, зокрема предметно-орієнтовані та граматичні, які застосовуються для опису програмних структур, дослідити можливості моделей глибокого навчання для моделювання мовних послідовностей у контексті генерації коду, Розглянути існуючі технології автоматичної генерації програмного коду та підходи до синтезу генераторів із прикладів, Вивчити методи представлення мов програмування, які дозволяють ефективно здійснювати аналіз і трансформацію коду, РРзробити та реалізувати підхід до побудови ознак для навчання моделей машинного навчання, орієнтованих на генерацію коду, Провести експеримент з навчання моделей на основі згенерованих даних і оцінити якість автоматичної генерації програмних

фрагментів, Оцінити ефективність використаних підходів до генерації та оптимізації коду з точки зору точності, структурної коректності та узгодженості з вхідними даними.

Об'єктом виступає процес автоматизації програмування, зокрема генерації та оптимізації програмного коду.

Предмет дослідження: методи машинного навчання та мовні моделі, що використовуються для автоматичної генерації та оптимізації коду, включаючи представлення мовних структур, побудову ознак і навчання моделей.

В роботі були використанні такі **методи дослідження** аналіз літературних джерел, порівняльний аналіз, моделювання, формалізація, аналіз результатів.

Щоб підсумувати наш внесок, ми запропонували нову техніку для автоматизації створення генераторів коду за допомогою нового застосування символьного машинного навчання. Ми також оцінили CGBE на реалістичних прикладах завдань генерації коду, щоб встановити, що він ефективний для таких завдань. Ми розглядаємо існуючі підходи до генерації коду MDE та дослідження, а « Ідіоми генерації коду » описуємо поширені ідіоми, які виникають під час обробки генератором коду. Ці ідіоми використовуються для керівництва та обмеження стратегій пошуку CGBE. « Дослідницькі запитання » представляють дослідницькі питання, на які ми прагнемо відповісти. « CST L » описує мову генерації коду CSTL, яку ми використовуємо для вираження правил генерації коду. « Синтез генераторів коду з прикладів » описує детальний процес з використанням символічного ML.

Бакалаврська робота містить 60 сторінки, 17 рисунків, 3 розділи, список використаних джерел із 20 найменуванням.

РОЗДІЛ 1. МОДЕЛІ ПРЕДСТАВЛЕННЯ МОВНИХ СТРУКТУР

1.1 Предметно-орієнтовані мовні моделі

Доменно-орієнтовані мови (DSL) часто використовуються для визначення правил параметризації та станів для створення структури програми. Моделі, засновані на DSL, створюють різні граматичні правила для загальних інструкцій коду (наприклад, керування потоком, коментарі та дужки). У порівнянні з мовою програмування загального призначення граматичний розмір DSL зазвичай менший, що робить його більш ефективним для конкретних завдань генерації коду. Модель на основі DSL була вивчена Gulwani та ін. [27] і Джа та ін. [15]. Гверо та ін. [22] зменшив простір пошуку для пропозицій виразів Scala, використовуючи обчислення коротких типів і розробив функцію вищого порядку для завершення коду. Оскільки процес генерації можна інтерпретувати людиною, цей тип моделі може бути багатообіцяючим підходом у SE.

У індукції програм DSL визначає простір програм-кандидатів (шаблон програми), тоді як приклади пар введення-виведення відомі як специфікація. За цим сценарієм проблема відома як індуктивне логічне програмування (ILP) [16]. Двома класичними сімействами розв’язання ILP є підходи «знизу вгору», конструювання програм із прикладів функцій, і підходи «зверху вниз», тестування прикладів із поколінь і відповідне коригування правил [24]. Враховуючи точну та комбінаторну природу індукції, індукцію зазвичай називають проблемою задоволення обмежень (CSP) [22]. Це відноситься до низхідної сімейства ILP [24]. Формальне визначення проблеми можна знайти в роботі Pu et al. [14]. Така проблема CSP може бути вирішена за допомогою вирішувача обмежень, такого як Z3 [28]. Проблема цього підходу полягає в тому, що розв’язувач завжди є евристичним і його масштабованість погана. Часто ці системи не можуть обробляти зашумлені, помилкові чи неоднозначні дані. Керовані моделі DSL можуть охоплювати структуру конкретної мови програмування; однак вони вимагають глибоких знань предметної області для

створення детальних синтаксичних і семантичних правил. Одним із можливих способів підвищення гнучкості є представлення DSL за допомогою ймовірнісної моделі.

У формальних мовах правила виробництва можуть бути використані для визначення всіх можливих поколінь рядків (операторів коду). Контекстно-вільні граматики (CFG) - це набір продукційних правил, які можна застосовувати незалежно від контексту, тобто ліва частина продукційного правила містить лише один нетермінальний символ. CFG - це поширений спосіб визначення структури мови програмування, який потім можна використовувати для перетворення вихідного коду в абстрактні синтаксичні дерева (AST) [103]. Імовірнісна контекстно-вільна граматика (PCFG) є розширенням CFG, у якому правила створення контекстно-вільної мови пов’язані з імовірнісною моделлю. Bielik та ін. [25] узагальнив ідею PCFG для завдання завершення коду в інших неконтекстно-вільних мовах, таких як JavaScript. Пізніше Райчев та ін. [220] розширив попередню роботу над PCFG, вивчаючи дерево рішень для побудови імовірнісної моделі з використанням AST запропонованого DSL, а саме TGen. Ці роботи досягли значних результатів для деяких завдань завершення коду. Інше розширення CFG, а саме граматика заміни дерева (TSG), використовує фрагменти дерева замість послідовності символів для правил виробництва.

Такі правила TSG визначені Деревоподібною граматикою [17], яка може створити більшу гнучкість і краще представляти складні лінгвістичні структури [29]. Щоб обмежити складність моделі та розріджену граматику, дослідники часто використовують непараметричний метод Байєса для виведення розподілів [8, 29]. Ці моделі підходять для аналізу шаблонів, оскільки їх здатність автоматичного вибору моделі дозволяє відкривати складніші структури. Однак непараметричні байєсовські методи часто надзвичайно повільні в обчисленні та важко масштабуються. Хоча імовірнісні граматики досягають високої продуктивності для предметно-орієнтованих мов, вони все ще вимагають розроблених вручну правил для моделювання локальності коду та повторного використання.

1.2 Інші грамні моделі мови

Окрім двох синтаксичних моделей, наведених вище, однією простою, але ефективною моделлю статистичної мови є n-gram. Більш конкретно, ця модель припускає, що кожна лексема/слово умовно залежить від попередніх n - 1 лексем/слів, як описано в наступному рівнянні: P(W) = П t P(w t \w t _(n-1),..., t _Д де P (w t \w t _(n-1),..., t _ 1) можна просто обчислити шляхом підрахунку появи всіх n-грам у навчальному наборі. Пряма перевага n-грам над синтаксичними моделями (наприклад, імовірнісними граматиками та керованими моделями DSL) полягає в тому, що їх легше узагальнити, оскільки залежності та правила мов програмування вивчаються автоматично з вихідного коду. Hindle та ін. [10] виступили з ініціативою використати n-gram для створення мовної моделі для вихідного коду. З тих пір, окрім завершення коду [13, 22], моделі n-грам також застосовувалися для інших завдань, таких як аналіз ідіом [7] , виявлення синтаксичних помилок [16], аналіз вихідного коду [14] і обфускація коду [16]. Однак прості мовні моделі, такі як n-gram, не можуть охопити парадигми програмування високого рівня.

Щоб вирішити цю проблему, у ряді робіт було покращено мовні моделі, щоб вони були більш адаптованими до місцевої інформації. Bielik та ін. [26] доповнили модель на основі DSL за допомогою n-грами та продемонстрували сильні емпіричні результати для моделювання мовою програмування. Отримана модель також була більш ефективною в навчанні та висновках порівняно з нейронними моделями. Hellendoorn та ін. [18] додав локальний кеш n-грам і об’єднує прогнози локальних і глобальних моделей. Обидві моделі, як стверджувалося, перевершують аналоги DL (наприклад, RNN і LSTM) на момент їх публікації, таким чином, ці дві моделі будуть хорошою базою для моделювання вихідного коду.

Слід зазначити, що n-gram погано моделює довгострокові залежності (див. Лістинг 1 і 2) між токенами. Усічення n-грами відкидає довготривалу позиційну інформацію. Інші статистичні моделі, такі як приховані моделі Маркова, також

не можуть закодувати довгострокову історію [24], оскільки її простір станів стає експоненціально великим при кодуванні кількох попередніх токенів історії в один стан. Крім того, ще одним обмеженням моделі n-грам є розрідженість векторного представлення слова або кодового токена, що спричинено великим словниковим запасом вихідного коду. Цю проблему розрідженості можна вирішити за допомогою розподіленого представлення нейронних мовних моделей.

Замість того, щоб явно включати частоту кожного попереднього токена, моделі вбудовування нейронної мережі з одним прихованим шаром спочатку перетворюють одноразове кодування слова в проміжний вектор вбудовування слова з набагато меншою довжиною порівняно з розміром словника (наприклад, 100–1000). Ця ідея також відома як розподілене представлення слів. У своїй оригінальній роботі [23] вбудовування слів до n -1 попередніх токенів по відношенню до поточного слова було подано до повністю зв’язаної нейронної мережі з одним прихованим шаром. На вихідному рівні була застосована функція softmax для обчислення ймовірності наступного слова. Однак серйозним недоліком цієї оригінальної моделі є висока обчислювальна вартість прихованого шару. Тому для вирішення цієї проблеми було представлено логарифмічний білінійний [18] шляхом заміни нелінійних активацій контекстною матрицею для визначення контекстного вектора щодо поточного слова. Потім було обчислено подібність між вектором контексту попередніх токенів і поточного слова.

Пізніше було запропоновано декілька методів для прискорення часу навчання та прогнозування за допомогою ієрархічної архітектури [12]. У цій попередній роботі було продемонстровано, що логарифмічно-білінійна модель перевершує традиційну повністю зв’язану нейронну мережу та моделі мови n-gram для завдання моделювання APNews. Пізніше ідея моделі вбудовування нейронної мережі була прийнята дослідниками для моделювання вихідного коду, які можна назвати моделями простих нейронних програм. Більш конкретно, Меддісон та ін. [13] об’єднав логарифмічно-білінійну техніку обходу пошуку в

глибину дерева (тобто, логарифмічно-лінійні моделі обходу дерева), щоб створити зрозумілий людині вихідний код. Алламаніс та ін. [9] розширив підхід Меддісона для отримання фрагментів вихідного коду із запитів природної мови і навпаки. Алламаніс та ін. [4] також використовував логарифмічно-білінійну модель, щоб рекомендувати імена методів і класів для об'єктно-орієнтованого програмування на Java, і ця модель перевершила модель n-грам в обох завданнях. Подібно до моделей n - грамів, знання логарифмічних білінійних моделей обмежено попередніми n-1 токенами. Таким чином, у наведених вище роботах необхідно було чітко визначити глобальний і локальний контекст для логарифмічних білінійних моделей, щоб охопити короткострокові, довгострокові залежності та послідовність властивостей вихідного коду. Список контекстів все ще створений людьми та неповний, що обмежує його застосування в нових областях. Однак, завдяки своїй простоті, прості моделі нейронних програм використовуються як (попередньо навчені) вхідні функції для різних завдань Big Code.

1.3 Моделі глибокого навчання для моделювання послідовності

Два основних класи моделей DL для моделювання послідовності, а саме (i) рекурентні та (ii) нерекурентні нейронні мережі. Потім розглядаються три техніки для побудови більш надійних моделей, включаючи (i) механізм уваги, (ii) зовнішню пам'ять і (iii) пошук за променем. Зауважимо, що багатошаровий персептрон (MLP) [27] (він же повнозв'язана/пряма нейронна мережа) не розглядається в цьому розділі, оскільки він є розширенням нейронної мережі. Таким чином, MLP все ще обмежений у захопленні залежностей і послідовних властивостей послідовності, і не широко використовується для моделювання послідовності, якщо не поєднувати його з більш просунутими методами, такими як механізм уваги.

Рекурентні нейронні мережі. Рекурентна нейронна мережа (RNN) та її варіанти широко використовуються для побудови мовних моделей [13, 17, 11].

RNN — це особливий тип глибокої нейронної мережі, в якій блок параметрів використовується спільно та повторюється багато разів у різних частинах послідовності, в результаті чого створюється глибокий обчислювальний граф [5]. Ця архітектура допомагає мережі навчатися за допомогою вводу/виводу різної довжини, чого не можуть MLP.

Однак ванільні (прості) RNN важко навчити [20], і вони не здатні зберігати минулу інформацію з різних масштабів часу. Гейтовані RNN, такі як довготривала короткочасна пам’ять (LSTM) [10] і стробовані рекурентні одиниці (GRU) [4], моделюють механізми збереження та забування явно за допомогою сигмоїдних активацій, а саме воріт. LSTM має три вентилі для керування введенням, виводом і забуванням відповідно. Крім того, існує стан комірки пам'яті для створення прихованих станів.

Одиниці RNN можна зробити глибокими для кодування більш складних переходів [5]. Шари магістралей [24] були введені для стабілізації градієнтів навчання в рекурентних мережах магістралей [28]. Щоб охопити довгострокові залежності в часових рядах і представити ієрархічну інформацію, рівні RNN можуть бути складені з різною частотою оновлення [13]. RNN зі стробованим зворотним зв'язком [7] дозволяють мережі вивчати власні тактові частоти за допомогою додаткових вентилів. Найсучаснішою RNN для мовної моделі є Fast-Slow RNN [18], яка поєднує в собі сильні сторони глибоких і багатомасштабних RNN.

Нерекурентні нейронні мережі. Тимчасова згортка або одновимірна згортка в часі є іншим типом нейронної мережі, яка може фіксувати довгострокові зв’язки з ієрархічною архітектурою [25]. Він застосовувався для аналізу настроїв, класифікації речень, машинного перекладу та метанавчання [17]. Згорткові нейронні мережі (CNN) використовувалися в кількох завданнях моделювання речень. У 2013 році Kalchbrenner і Blunsom [12] використовували CNN як кодер і RNN як декодер для генерації діалогу. Через рік Blunsom та ін. запропонував Dynamic CNN [27] для семантичного моделювання речення, де змінна довжина та виявлення зв’язків були включені за допомогою

максимального об’єднання.

Однак у цих попередніх роботах не вдалося досягти аналогічної продуктивності LSTM. Нещодавно знову з’явилися нерекурентні структури з такою ж продуктивністю, як і RNN, але вони швидші для обчислення. Замасковані шари згортки використовуються як декодер у системі нейронного машинного перекладу [12]. Герінг та ін. [3] запропонував ConvS2S, який привернув пропуск з’єднання [5] та звернув увагу [16] на моделювання речень і досяг сучасної продуктивності перекладу. Поєднання рекурентних і згорткових одиниць також корисно. Він та ін. [97] посилили кореляцію вхід-вихід, додавши міжшарові згортки до складених RNN.

Васвані та ін. [2] запропонували модель уваги з декількома головами під назвою «Трансформатор», яка покладається на увагу до себе та позиційне кодування для обчислення представлень послідовності. Transformer дозволяє декодеру звертати увагу на інформацію на довільній відстані та значно скорочує час навчання без втрати якості. Подібно до CNN, причинно-наслідкова структура утримується шляхом маскування пізніших результатів для авторегресійної факторизації. Нещодавно модель глибокого мовного представлення BERT [9], яка використовує нову двонаправлену підготовку трансформатора, досягла найсучасніших результатів в одинадцяти завданнях НЛП, таких як класифікація/позначення речень, відповіді на запитання та розпізнавання іменованих сутностей. Пізніше BERT було розширено до мовного моделювання [33] та генерації мови [12] шляхом усунення його обмежень контекстів фіксованої довжини та двонаправленої природи, відповідно. Слід зазначити, що Transformers можна використовувати для контекстного (одного й того самого токена з різними використаннями) вбудовування вихідного коду.

Ще одним цікавим напрямком є прискорення послідовної генерації. Слід зазначити, що всі авторегресійні моделі генерують вибірки лише послідовно, оскільки вони використовують вибірку предків. Таким чином, необхідні альтернативні архітектури для швидкого, паралельного формування вибірки. Гу та ін. [3] увімкнули неавторегресійне навчання шляхом вибірки латентної

змінної, що представляє народжуваність, тобто використання кожного вихідного слова в декодуванні, яке потребує контролю зовнішньої системи вирівнювання. Інверсно-авторегресивні потоки (IAF) [13] можуть генерувати високовимірні зразки паралельно з латентних змінних. Оорд та ін. [19] об'єднав WaveNet з IAF для створення паралельного WaveNet, який дискретизувався з вищою точністю, але працював у 3000 разів швидше при генеруванні аудіо.

Механізм уваги. Однією з проблем початкової структури кодера-декодера є те, що декодер може отримати доступ лише до одного вектора контексту. Людина розуміє рядки тексту, постійно звертаючи увагу на різні частини послідовності. Щоб імітувати цю поведінку, Bahdanau et al. [16] використав послідовність як контекст і запропонував механізм звернення уваги для адаптації ваг контексту, пов'язаного з певним вихідним етапом, і нав'язування явного вирівнювання між вхідними та вихідними токенами. Точніше кажучи, із закодованою послідовністю F і на кожному кроці t прихований стан h t обчислюється за допомогою моделі RNN із вектором джерела вхідних даних c t, створеним механізмом уваги як додатковий вхід: h t = RNN(h t-1 , [e t-1 ; c t]). Цей спосіб залучення уваги відомий як раннє зв'язування. Крім того, увагу можна розглянути безпосередньо перед генерацією вихідного маркера. Типова м'яка увага, подібна до Bahdanau та інших [16], може бути обчислена таким чином:

(1) З попереднім прихованим станом h t-1 енергія уваги обчислюється за допомогою оцінювальної функції на основі вмісту u t = score(F, h t-1).

(2) Підведіть до степеня та нормалізуйте u t до 1: a t = softmax(u t).

(3) Обчисліть вектор вхідного джерела c t = F a t . Найпростішим способом визначення оцінки є скалярний добуток, оцінка (F, h t-1) = F T h t-1 .

Або очікуване вбудовування вхідних даних V може бути визначено так, що score(F, h t-1) = F T Vh t-1 . В оригінальній статті [16] енергія уваги обчислюється за допомогою багатошарового персептрона (MLP) наступним чином: scor e(F , h t-1) = v T tanh(WF + Vh t-1).

Енергія уваги на основі вмісту обчислюється шляхом оцінювання

кожного елемента окремо, що ускладнює розрізнення елементів із подібним вмістом у різних місцях. Увага, чутлива до місця розташування [5] подолала це обмеження, генеруючи напруги авторегресійним способом. Однак розгортання через інший RNN під час зворотного поширення може значно збільшити час обчислення. Васвані та ін. [25] представили мультиголову самоувагу для представлення відповідного контексту кожного слова у вхідній послідовності в різних місцях. Завдяки поєднанню моделювання послідовності та механізму уваги було досягнуто найсучасніших результатів для нейронного машинного перекладу [16].

Нейронні мережі з розширеною пам’яттю. Увага тісно пов'язана із зовнішньою пам'яттю. Разом -вони стали важливими будівельними блоками для нейронної мережі. Зовнішні пам’яті використовуються як внутрішні стани, які можна оновлювати за допомогою механізму уваги для вибіркового читання та оновлення. Класична мережа пам’яті (Mem NN) [26] намагалася імітувати пам’ять з довільним доступом (RAM) і використовувати м’яку увагу як диференційовану версію адресації. Мережа пам'яті зазвичай приймає такі входи:

- Запит q — це останнє висловлювання мовця під час загального діалогу або запитання в налаштуваннях QA.
- Вектор пам'яті m - це діалогова історія моделі. Знання можуть бути настільки великими, наскільки вся кодова база або документація є достатньо потужною для моделі.

І цей тип мережі має такі модулі для обробки вхідних даних: • Кодер перетворює q у вектор за допомогою RNN [14] або простішого вбудовування слів [17].

- Модуль пам'яті M знаходить найкращу частину m, пов'язану з q. Це етап адресації.
- Модуль контролера C надсилає q до M і зчитує відповідну пам’ять, додаючи її до поточного стану. На практиці ми завжди повторюємо цей процес, щоб уможливити складне міркування.
- Декодер генерує вихідні дані з кінцевих станів.

Навчання MemNN повністю контролюється, у якому мітка найкращої частини пам’яті дається на кожному етапі адресації пам’яті. Її подальша мережа наскрізної пам’яті [25] використовує м’яке увагу для адресації пам’яті, щоб тренувати увагу у зворотному поширенні та послабити нагляд лише за виходом. Слід зазначити, що використання однієї одиниці пам’яті для відповідності як запиту, так і стану обмежує виразність. Розділивши пам'ять на пару ключ-значення, Millor et al. [16] закодували попередні знання та отримали кращі результати. Recurrent Entity Network [19] демонструє, як покращити блок пам’яті, дозволивши агенту навчитися читати та записувати пам’ять для відстеження фактів. Уестон [26] узагальнив цю модель на неконтрольоване навчання, додавши новий етап для генерування відповідей і прогнозування відповідей. Цей тип моделі оцінюється на різних завданнях, таких як базове міркування за двадцятьма завданнями bAbI [26], читання дитячих книжок [10], розуміння справжніх діалогів із фільмів [20]. З усіма завданнями можна ознайомитися на сайті bAbI projec t 2. Останнім часом у багатьох роботах намагалися зробити традиційні парадигми пам’яті диференційованими, щоб моделі можна було оптимізувати за допомогою SGD. Такі наскрізні моделі, які можна навчати, використовувалися для вирішення завдань алгоритмічного навчання та міркувань, таких як розуміння мови та індукція програми.

Променевий пошук. Пошук найкраще декодованого результату з найвищою ймовірністю обчислювально важко піддається управлінню. Іншими словами, може бути експоненціально велика кількість згенерованих речень у NLP або вихідного коду у Big Code. Одним із рішень було б вибрати слово/лексему з найвищою ймовірністю виведення після кожного кроку в часі під час процесу декодування. Однак цей жадібний процес, швидше за все, призведе до неоптимального результату. Тому в машинному перекладі пошук за променем широко використовується як евристичний метод пошуку [248]. Замість того, щоб безпосередньо брати наступне слово з найвищою ймовірністю, зберігається список попередніх, найімовірніших, часткових перекладів, і ці вибрані слова/маркери розширюватимуться для кожного перекладу на

поточному кроці та змінюватимуть їх рейтинг. Довжина списку на кожному кроці часу відома як розмір променя. Цей метод часто покращує переклад, але продуктивність тісно залежить від розміру променя [13].

1.4 Висновки по розділу

У моделюванні мов програмування використовується широкий спектр підходів - від предметно-орієнтованих мов (DSL) до статистичних n-грамових моделей і простих нейронних мовних моделей. DSL-моделі забезпечують точний контроль над структурою коду та дозволяють явно описувати семантику завдяки граматикам (CFG, PCFG, TSG), але вимагають ручної розробки правил і мають обмежену гнучкість та масштабованість. N-грамові моделі - прості у реалізації та здатні автоматично вивчати локальні залежності з даних, але погано відображають довгостроковий контекст і страждають від розрідженості. Нейронні моделі, зокрема логарифмічно-білінійні, покращують узагальнення та зменшують розмірність, але залишаються залежними від обмеженої кількості попередніх токенів і вимагають додаткового моделювання контексту. Загалом, кожен підхід має свої переваги й обмеження, і ефективне моделювання коду часто вимагає їхньої комбінації або доповнення більш потужними архітектурами, такими як трансформери.

Загалом, поєднання глибокого навчання, механізмів уваги, зовнішньої пам’яті та евристичних методів пошуку дає змогу створювати високоефективні та масштабовані моделі для роботи з послідовностями як у природній мові, так і в програмному коді.

РОЗДІЛ 2 . ПІДХОДИ ДО АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ПРОГРАМ

2.1 Технології генерації коду

Існує три основні підходи до створення коду в контексті MDE. Генератори коду можуть безпосередньо створювати текст на цільовій мові з моделей, тобто підходи від моделі до тексту (M2T), або створювати модель, яка представляє цільовий код, тобто підходи від моделі до моделі (M2M) [9 , 23]. Нещодавно були визначені мови генерації коду тексту в текст (T2T) [24]. Зараз генерація коду часто виконується за допомогою мов M2T на основі шаблонів, таких як Epsilon Generation Language (EGL). і Acceleo . У них вказуються шаблони для елементів цільової мови, таких як класи та методи, з даними створених шаблонів, які обчислюються ¬за допомогою виразів, що включають елементи вихідної моделі. Таким чином, розробник генератора коду на основі шаблону повинен розуміти метамодель вихідної мови, синтаксис цільової мови та мову шаблону. Ці три ¬мови змішуються в текстах шаблонів із роздільниками, які використовуються для розділення синтаксису різних мов. Концепція ¬подібна до використання JSP для створення динамічних веб-сторінок із бізнес-даних. На малюнку 1 показано приклад сценарію EGL, який поєднує текст фіксованого шаблону та динамічний вміст , а також створений у результаті код. Динамічний вміст у шаблоні укладено в розділові дужки [% і %].

Підхід до генерації коду M2M розділяє генерацію коду на два етапи: (i) перетворення моделі з метамоделі вихідної мови в метамодель цільової мови [9] та (ii) створення тексту з цільової моделі. У цьому випадку автор генератора коду повинен знати як метамоделі вихідної, так і цільової мови, мову перетворення моделі та синтаксис цільової мови. На рисунку 2.1 показано приклад генерації коду M2M у QVTr із [9].

Підхід T2T до генерації коду визначає переклад з вихідної мови на цільову в термінах конкретного синтаксису або граматики вихідної та цільової мов і не залежить від метамоделей (абстрактного синтаксису) мов. Автор T2T

повинен знати лише граматику вихідної мови ¬та синтаксис цільової мови, а також мову T2T.

```
relation Package2Package_top{
    packName : String;

    enforce domain uml uml_model : uml::Model{
        nestedPackage = uml_pack: uml::Package{
            nestingPackage = uml_model
            , name = packName
        }
    };
    enforce domain java java_model : java::Model{
        ownedElements = java_pack : java::Package{
            model = java_model
            , name = packName
            , proxy = false
            --make sure no proxy package matches
        }
    };
    where{
        AuxPackage2Package(uml_pack, java_pack);
        Package2Package_nested(uml_pack, java_pack);
    }
}
```

Рисунок 2.1 - Приклад генерації коду QVTr M2M

Генерація коду включає в себе певні форми обробки та перетворення: як правило, модель або синтаксичне дерево, що представляють вихідну версію системи програмного забезпечення, обходять для побудови моделі або тексту, що представляє цільову версію системи програмного забезпечення. Може знадобитися багаторазовий обхід вихідних даних разом із створенням внутрішніх допоміжних даних у спосіб, подібний до обробки, що виконується компіляторами [10]. Досліджуючи кілька різних генераторів коду M2M, M2T і T2T, ми визначили набір характерних ідіом, які використовуються в їх обробці: Розробка: вихідний елемент зіставляється з цільовим елементом, а також деякими додатковими структурними/постійними елементами. Наприклад, для кожного конкретного класу UML C ми могли б створити додаткову операцію глобального конструктора createC() для створення екземплярів C. Перевпорядкування: елементи джерела змінюються в порядку цільового.

Java. Спрощення: елементи вхідних даних відкидаються під час отримання виходу. Наприклад, умови специфікації pre : і post : операції OCL можна відкинути під час перекладу з OCL на Java, і лише явне поведінкове визначення операції використовуватиметься для створення коду.

Заміна: Частини джерела функціонально замінюються на цільові частини. Наприклад, символи групи операторів (і) можуть узгоджено замінюватися символами блоку { і } у відображенні процедурних операторів OCL на Java.

Умовна генерація: перекладіть окремі елементи альтернативними способами на основі їхніх властивостей. Наприклад, числовий вираз ділення a /b буде перекладено на a//b у Python, якщо і a, і b мають цілий тип, інакше — a/b.

Генерація з урахуванням контексту: перекладайте окремі елементи альтернативними способами на основі їхнього контексту. Наприклад , атрибути та операції інтерфейсу будуть перекладені на Java іншим способом, ніж атрибути загального класу.

Ітеративна генерація: обробка елементів списку шляхом послідовного застосування того самого перекладу до кожного елемента списку по черзі. Наприклад, переклад літералів визначення перерахування в UML у відповідні літерали в мові програмування enum.

Накопичення: перебирайте структуру вихідної моделі, збираючи разом усі елементи з певними властивостями, які зустрічаються. Наприклад, зібрати разом усі безпосередньо належні та рекурсивно успадковані функції класу UML.

Горизонтальне розбиття: виконуйте альтернативну обробку елементів у колекції залежно від їхніх властивостей, розділяючи їх на непересічні групи в цільовому об’єкті. Наприклад, у перекладі функцій класу UML на Go статичні атрибути визначаються як глобальні змінні за допомогою var поза будь-яким класом (структурою), тоді як нестатичні атрибути стають полями всередині структури.

Вертикальне розбиття: генеруйте два або більше цільових елементів з одного вихідного елемента. Наприклад, створення відповідного файлу заголовка C і коду для класу UML.

Існують також специфічні мовні ідіоми для певних типів вихідних/цільових мов: Експрес-типи за допомогою ініціалізації:

у випадках, коли вихідна мова повністю набрана, але цільова мова не має явної типізації (наприклад, Python або JavaScript), інформація про тип для вихідних змінних може бути виражена вибором значення ініціалізації змінної в цільовій. Наприклад, у картографії ім'я змінної: рядок; на Python, ми могли б написати назва = ""

Цілі числа будуть ініціалізовані 0, подвійні - 0,0, а логічні значення - False тощо.

Виконайте висновок типу: у протилежній ситуації, коли вихідна мова має неявні типи, а цільова - явні типи, можна використати стратегію висновку типу, щоб ідентифікувати фактичний тип елементів даних, де це можливо.

Замінити успадкування асоціацією: якщо вихідна мова має успадкування, а цільова мова - ні, один із прийомів представлення успадкування полягає в тому, щоб вставити посилальний супер в екземпляр підкласу, пов’язавши його з екземпляром суперкласу. Доступи та оновлення атрибутів суперкласу f реалізуються як доступи та оновлення до супер. f . Подібним чином виклики успадкованих операцій виконуються через super. Ця стратегія використовується в перекладі UML2C [23].

Попередня нормалізація: щоб полегшити генерацію коду, вихідну модель/специфікацію може знадобитися переписати в певну форму. Наприклад, винесення на множники загальних підвиразів ¬в арифметичних виразах [10]. Дерево до послідовності: ця ідіома вирівнює структуру вихідного дерева , так що окремі піддерева вихідного дерева стають підпослідовності результуючої послідовності. Він використовується у випадках, коли цільова мова має плоску структуру, таку як мова асемблера або мова PLC, а джерело має блочну структуру.

2.2 Синтез генераторів коду з прикладів

Метою нашої процедури машинного навчання є автоматичне отримання генератора коду CSTL g, що відображає мову програмного забезпечення L 1 на

іншу мову L 2 на основі набору D прикладів відповідних текстів з L 1 і L 2. D має бути дійсним, тобто функціональним від джерела до цільового, і кожен приклад має бути дійсним відповідно до граматики своєї мови. Згенерований g має бути правильним щодо D, тобто він має правильно транслювати вихідну частину кожного прикладу d є D у відповідну цільову частину d. Крім того, g також має бути в змозі правильно транслювати вихідні елементи набору даних перевірки V з (L 1, L 2) прикладів, не перетинаючихся з D. Ми називаємо цей процес генерації коду прикладом або CGBE.

Набори даних D будуть організовані таким же чином, як і набори даних парних текстів для ML перекладачів природної мови: 5 * кожен рядок D містить одну пару прикладів, а вихідний і цільовий тексти розділені одним або кількома символами табуляції \t . Оскільки мови програмного забезпечення, як правило, організовані ієрархічно на підмови, наприклад, щодо типів, виразів, операторів, операцій/функцій, класів тощо, D буде типово розділено на частини, що відповідають основним поділам вихідної мови.

Машинне навчання (ML) можна визначити як технологію, яка спрямована на автоматизацію вивчення знань із екземплярів навчального набору, щоб потім отримані знання можна було застосувати для отримання інформації про інші екземпляри. Це визначення охоплює широкий спектр методів, включаючи статистичні підходи, традиційні та рекурентні нейронні мережі та підходи до символічного навчання, такі як дерева рішень [29] або індуктивне логічне програмування (ILP) [26]. Машинне навчання зазвичай використовує фазу навчання , під час якої знання виводяться з навчального набору, і фазу перевірки , де точність отриманих знань перевіряється на новому наборі даних, для якого відомі очікувані результати.

Фаза навчання може проходити під наглядом або без нагляду (або поєднувати обидва). Контрольоване навчання відбувається, коли навчальні дані позначаються очікуваним результатом для кожного пункту. Наприклад, класифікація зображення як зображення людини чи ні. Навчання без контролю відбувається, коли така інформація не надається; натомість категорії

класифікації або інші результати створюються за допомогою алгоритму ML із набору даних. Наш підхід CGBE також застосовує поділ на фази навчання та перевірки та використовує контрольоване навчання: кожному прикладу вихідної мови додається відповідний приклад цільової мови, який є очікуваним результатом відображення навченого перекладу.

Фаза навчання може проходити під наглядом або без нагляду (або поєднувати обидва). Контрольоване навчання відбувається, коли навчальні дані позначаються очікуваним результатом для кожного пункту. Наприклад, класифікація зображення як зображення людини чи ні. Навчання без контролю відбувається, коли така інформація не надається; натомість категорії класифікації або інші результати створюються за допомогою алгоритму ML із набору даних. Наш підхід CGBE також застосовує поділ на фази навчання та перевірки та використовує контрольоване навчання: кожному прикладу вихідної мови додається відповідний приклад цільової мови, який є очікуваним результатом відображення навченого перекладу.Основний розподіл між підходами ML полягає між несимволічними підходами, такими як нейронні мережі, де отримані знання представлені лише неявно, і символічними підходами, де знання явно представлені в символічній формі. Нещодавно було проведено значні дослідження щодо використання несимволічних методів машинного навчання для вивчення перетворень моделі та інших перекладів мов програмного забезпечення, наприклад [1 , 3 , 4 , 12 , 16 , 27]. Ці підходи зазвичай адаптовані з несимволічних підходів ML, які використовуються в машинному перекладі природних мов, таких як нейронні мережі LSTM [14], посилені різними механізмами уваги . Ці підходи не підходять для нашої мети, оскільки навчені транслятори представлені не явно ¬, а лише неявно у внутрішніх параметрах нейронної мережі. Таким чином, важко офіційно перевірити перекладачі або адаптувати їх вручну. Крім того, підходи нейромережі ML зазвичай вимагають великих навчальних наборів даних (наприклад, понад 100 МБ) і тривалого навчання. Це погіршує маневреність, а також має наслідки для ресурсів і навколишнього середовища.

Підходи символічного машинного навчання були застосовані до прикладу перетворення моделі (MTBE) [2] (з використанням ILP) і [21] (з використанням методів на основі пошуку). Вони, як правило, використовують значно менші (тобто в масштабі KB) навчальні набори даних порівняно з несимволічним ML і створюють чіткі правила. Одним із недоліків ILP є те, що на додаток до позитивних прикладів необхідно надати контрприклади стосунків, які потрібно вивчити. Підхід [21] використовує лише позитивні приклади. Він здатний вивчати окремі функції String-to-String і Sequence-to-Sequence з невеликої кількості (зазвичай менше 10) прикладів функції. Це дуже загальний інструмент синтезу перетворень, який може генерувати перетворення M2M кількома цільовими мовами (QVTr, QVTo, ETL і ATL). У цьому документі ми адаптуємо та розширюємо цей підхід MTBE для вивчення функцій, що стосуються дерев розбору мови програмного забезпечення, а отже, для синтезу генераторів коду T2T ¬.

Підхід [21] приймає як вхідні метамоделі для вихідної та цільової мов трансформації, а також початкове структурне відображення вихідних метакласів у цільові класи, виражене мовою трансформації абстрактної моделі TL [19]. У нашій адаптації MTBE ми використовуємо граматичні категорії мови L 1 і L 2 як вихідну та цільову метамоделі. Початкове відображення визначається , щоб вказати, які категорії L 1 відображаються в категоріях L 2. Наприклад, у перетворенні, що відображає UML/OCL на Java, можуть існувати категорії мови OCL OclLambdaExpression і OclConditionalExpression (підкатегорії OclExpression), а також категорія Java JavaExpr із відображеннями категорій мови структури OclLambdaExpression i—> JavaExpr OclConditionalExpression i—> JavaExpr На практиці ми використовуємо незалежні від мови категорії ProgramExpression, ProgramStatement тощо для цільової мови, щоб те саме структурне відображення можна було використовувати для різних цільових мов. Процес MTBE також приймає як вхідні дані модель m , що містить приклади екземплярів з вихідної та цільової мов, а також відношення відображення ^, що визначає, які вихідні та цільові елементи відповідають. Наприклад, використання

текстових представлень елементів:

```
xl : вираз OclLambda
xl.text = "лямбда x: Рядок y x+x"

yl : JavaExpr
yl.text = "x->(x+x)"

xl |-> yl
```

Лістинг 2.1 – Текстові представленя елементів

Щоб зробити висновок про функціональні відображення з вихідних даних, таких як OclLambdaExpression::text , на цільові дані, сумісні з типом, такі як JavaExpr::text, принаймні два приклади відповідності кожної категорії мови SC і—> TC повинні бути присутніми в моделі. За допомогою підходу MTBE [21] можна виявити відображення рядок у рядок кількох форм: де цільові дані формуються шляхом префіксування, інфіксування або додавання постійного рядка до вихідних даних; сторнуванням вихідних даних; шляхом заміни окремих символів фіксованим рядком тощо. Подібним чином, функції інших типів даних можуть бути запропоновані на основі відносно небагатьох прикладів. MTBE працює, постулюючи функції джерело-ціль на основі прикладів, а потім перевіряє, чи постульована функція дійсна для прикладів. Запропоновані функції вибрано з репертуару функцій, які, як виявлено, зустрічаються на практиці в специфікаціях перетворення. Ми використовуємо той самий принцип для CGBE та постулюємо схеми генерації коду на основі ідіом « ідіом генерації коду ¬». Потім ці відображення перевіряються на дані навчання.

2.3 Представлення мови програмного забезпечення

Для CGBE можна використовувати різні форми представлення програмного забезпечення:

- Текст, наприклад: “sq[i] + k” • Послідовності токенів, наприклад: 'sq', '[', 'i', ']', '+', 'k'
- Дерева абстрактного синтаксису/розбору (AST):

(OclBinaryExpression (OclBasicExpression sq [(OclBasicExpression i)]) + (OclBasicExpression k))

- Дані моделі екземпляра метамоделі мови:

```
ba0 : OclBasicExpression
ba0.type = Integer
ba0.data = "i" ba1 :
OclBasicExpression
ba1.type = Integer
ba1.data = "sq"
ba1.arrayIndex = ba0 ba2
: OclBasicExpression
ba2.type = Integer
ba2.data = "k" be1 :
OclBinaryExpression
be1.type = Ціле число
be1.operator = "+"
be1.left = ba1 be1.right
= ba2
```

Лістинг 2.2 - моделі екземпляра метамоделі мови

Ці уявлення стають дедалі багатшими та детальнішими. Порівняно з представленнями тексту чи токенів, дерева синтаксичного аналізу виражають більш детальну інформацію про програмний елемент, зокрема його внутрішню структуру з точки зору граматики мови. Як обговорюється [4], це представлення, отже, забезпечує більш ефективну основу для ML мовних відображень порівняно з послідовностями токенів або необробленим текстом. Наприклад, важко зробити висновок про відображення рядка в рядок між виразами OCL і виразами Java, представленими як необроблений текст, як демонструє приклад лямбда-виразу з попереднього -розділу.

Рисунок 2.2 показана метамодель, яку ми використовуємо для дерев розбору (тобто терміни AST). Ця метамодель має широке застосування не тільки для представлення та обробки мовних артефактів програмного забезпечення, а й для артефактів природної мови. ASTSymbolTerm представляє окремі символи, такі як '[' у наведеному вище прикладі. ASTBasicTerm

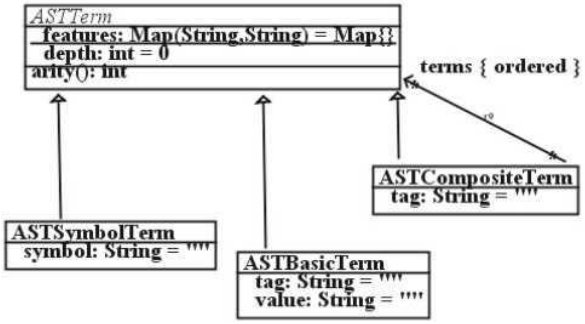

Рисунок 2.2 - Метамодель дерев розбору

представляє інші вузли дерева аналізу терміналу. Наприклад, (OclBasicExpression i). ASTCompositeTerm представляє нетермінальні вузли, такі як кореневий вузол OclBinary Expression прикладу. Тег основного або складеного терміна є ідентифікатором, який стоїть безпосередньо після початкового (, у текстовому представленні дерев синтаксичного аналізу . Арність символу дорівнює 0, базового терміна - 1, а складеного терміна - це розмір термінів (прямих підвузлів вузла дерева). Тег використовується як назва синтаксичної категорії дерева, коли дерево обробляється сценарієм CSTL , тобто правилами в Набір правил із цією назвою перевіряється на їхню застосовність до дерева: 0 для символьних термінів, 1 + max(terms^collect (depth)) для складених термінів.

Щоб використовувати MTBE для CGBE, ми отримуємо файл моделі m із текстових прикладів у наборі даних D. Кожен із вихідного та цільового прикладів ex з D виражається як елементи моделі mx у m, причому ці елементи мають атрибут ast: ASTTerm , значенням якого є дерево синтаксичного аналізу (відповідно до відповідної граматики мови) програмного елемента ex. Наприклад, елемент моделі x 1:

```
xl : OclLambdaExpr xl.ast =
(OclUnaryExpression лямбда x :
            (OclType String) y
            (OclBinaryExpression
                (OclBasicExpression x) +
                (OclBasicExpression x)))
```

представляє *лямбда x : Рядок y x + x .*

Лістинг 2.3 - Елемент моделі x 1

Повторювані моделі (наприклад, RNN) для моделювання послідовності важко навчати [20] і, подібно до інших типів нейронних мереж, легко схильні до переобладнання. Як і інші архітектури DL, RNN та його варіанти зазвичай навчаються за допомогою спеціальної форми зворотного поширення, а саме зворотного поширення в часі. Існує кілька методів оптимізації функції втрат моделі. Серед них стохастичний градієнтний спуск (SGD) або міні-пакетний градієнтний спуск переважно використовується завдяки його ефективним обчисленням і розпаралелюванню на графічних процесорах [5]. Меріті та ін. [17] застосував усереднений SGD із немонотонним запуском [16] для мовного моделювання та досяг чудових результатів. Слід зазначити, що більшість останніх робіт з DL повідомляють про їхні моделі, навчені сучасними версіями SGD, а саме RMSProp [21] або Adam [19].

Регулярізація моделі також має значний вплив на ефективність узагальнення. Ми розглядаємо чотири поширені типи регулярізації для більш ефективного навчання моделей DL, включаючи: (i) випадання, (ii) нормалізацію, (iii) регулярізацію активації та (iv) структурну регуляризацію. Dropout [243] випадковим чином вимикає кілька позицій активації за розподілом Бернуллі. Однак його застосування до прихованих станів RNN порушує здатність моделі зберігати довготривалі залежності [283]. Було прийнято два способи збереження інформації. Перший спосіб полягає в тому, щоб обмежити відсоток вилучення прихованих станів шляхом збереження попередньої інформації. Zoneout [139] випадково копіює попередні значення активацій, а не обнулює їх. Семенюта та ін. [23] застосував відключення на вхідному шлюзі, щоб запобігти втраті пам’яті. Другий — блокований відрив, тобто використання тієї самої маски вилучення для повного проходу вперед. Цей метод зберігає норми активації замість поступового скидання інформації. Гал та ін. [9] пов’язав заблоковане вилучення з варіаційним висновком Байєса та використав його для вбудовування вилучення. Крім того, заблокований DropConnect [25] на прихованих вагах призвів до суттєвих покращень [17]. Нормалізація обмежує активацію різних часових кроків для дотримання стабільного розподілу. Натхненний

нормалізацією партії [8], було вивчено кілька методів нормалізації, налаштованих для повторюваних структур, таких як регулярна нормалізація партії [5], нормалізація ваги [2] і нормалізація шару [3]. Існують також методи нормалізації, спрямовані на стабільність градієнта. Наприклад, спектральна нормалізація [9] була розроблена, щоб підтримувати градієнт обмеженим, обмежуючи активації неперервними за Ліпшицем.

Регуляризація може бути застосована до ваг і активацій моделей DL. Регуляризація L 2 на вагових коефіцієнтах називається розпадом вагових коефіцієнтів. Регулярізація активації карає активації за допомогою aL 2 (m ■ h t), де a — член регуляризації, m — масштабний коефіцієнт і h t — прихований стан, відповідно. Регуляризація часової активації [17] штрафує значні зміни в прихованому стані нейронної моделі за допомогою в L 2 (h t - h t _i), де в є коефіцієнтом масштабування, h t і h t-1 є прихованими станами в момент часу t і t - 1 відповідно. Структурна регулярізація запобігає розриву або зникненню градієнтів шляхом обмеження структури моделі. Обмеження структури моделі можна здійснити шляхом примусового унітарного рекурентної матриці [13] або використання поелементних взаємодій. Строго типізовані RNN [17] використовують узгоджені за типом операції для рекурентних одиниць. Іншими спрощеннями є Quasi-RNN [31] і Simple Recurrent Unit [14]. Ці методи регулярізації важливі для зменшення надмірного оснащення та покращення продуктивності узагальнення глибоких моделей вихідного коду, оскільки вивчені моделі можуть стати надто складними через необхідність представлення різних типів правил.

Для багатьох завдань глибокого вихідного коду вхідні дані приймають форму послідовності, як-от фрагменти коду, коментарі чи описи, де ми покладаємося на глибокий модуль для захоплення семантики та контексту введення для подальшої обробки. Ми називаємо такі модулі моделями глибокого кодування. Найпоширенішими моделями кодерів є послідовні моделі, такі як RNN та його варіанти. Для майнінгу сховищ програмного забезпечення загального призначення ефективність RNN була перевірена [26]. Однак

послідовні моделі можуть бути не настільки ефективними для моделювання та генерації коду через такі обмеження:

• Синтаксичний контекст погано представлений у послідовній моделі, що може призвести до порушення правил граматики мови програмування.

• Великий словниковий запас коду, що призводить до проблеми із словниковим запасом, впливає на можливість узагальнення моделі глибокого коду.

• Повторювані моделі (наприклад, RNN) страждають від вузького місця прихованого стану, у якому розмір вектора прихованого стану обмежує інформацію, яку модель може передати в часі.

Для усунення цих недоліків було запропоновано багато методів. Це (i) структурне (на основі дерева/графа) представлення, (ii) модель відкритого словника та (iii) механізм уваги. Структурне представлення. Абстрактне синтаксичне дерево (AST) - це природний спосіб охопити синтаксичну структуру програми. У AST програма розбирається на ієрархію нетермінальних і кінцевих (листових) вузлів на основі синтаксису мови програмування. Щоб використовувати AST для представлення коду, найпростішим способом буде використання пошуку в глибину для перетворення AST у послідовність [12, 4, 14]. Інші дослідження пропонують моделі DL (рекурсивні нейронні мережі [26], Tree-LSTM [26] або CNN [18]) для безпосередньої роботи з ієрархічною структурою дерева аналізу. Нещодавно Zhang et al. [28] показали, що розбиття AST на піддерева кодових операторів може покращити продуктивність представлень на основі дерева.

Останні роботи (наприклад, code2vec [11] і code2seq [10]) також запропонували використовувати шляхи AST як представлення для коду, в якому витягнуті шляхи будуть агреговані за допомогою глибокої нейронної мережі на основі уваги. Нещодавно Allamanis et al. [6] представили нові нейронні мережі Gated Graph [15] для представлення вихідного коду як спрямованого графа. Зокрема, вони включили інформацію про дані/потік керування змінними в AST, щоб більш ефективно отримувати синтаксичні та семантичні структури

вихідного коду. Відзначено, що спільнота Big Code викликає зростаючий інтерес до структурного представлення вихідного коду. Існує також вичерпний огляд [3] про вбудовування вихідного коду. Відкрита словникова модель.

Словник вихідного коду відкритий, а не фіксований. Thus, it is impractical to train a classifier using the whole vocabulary, so it is more common to truncate it by keeping only the most frequent 1,000 or 10,000 terms and replacing the others with an Out- of-Vocabulary (OoV) token (ie, <unk>). Недоліком цього скорочення є те, що лексеми OoV (неологізми [4]) з набору для тестування неможливо передбачити. Щоб вирішити проблему OoV, Karampatsis et al. [4] запропонували нову модель нейронної мови з відкритим словником для моделювання вихідного коду. У цій роботі GRU використовував для побудови моделі нейронної мови на основі одиниць підслова (підпослідовності символів кодових токенів), згенерованих алгоритмом кодування пари байтів [7]. Масштабні експерименти показали, що запропонована модель нейронної програми підслова була кращою, ніж найсучасніша модель n -gram, а також більш надійною проти проблеми OoV у різних мовах програмування та проектах. Символьні моделі DL [12,18] також є альтернативою підсловам для вирішення проблеми OoV. Нещодавно Cvitkovic et al. [2] розширив представлення коду на основі графів [6], щоб включити відкритий словник за допомогою Graph-Structured Cache, в якому нові слова/лексеми будуть додані як кешовані [7] вузли до існуючого AST.

Механізм уваги. Механізм звернення уваги можна використовувати як для вирішення проблеми OoV, так і для вузького місця прихованого стану RNN. Bhoopchand та ін. [24] використовували мережу покажчиків для копіювання OoV-токенів з недавнього минулого під час завершення коду. Мережа вказівників - це м'яка увага до попередніх вбудовувань вводу. Контролер створює скаляр, щоб вирішити, чи вибрати з позиції копіювання чи розподілу мовної моделі. Лі та ін. [14] нещодавно запропонував подібну модель, але звернув увагу на попередні приховані стани. Подібно до рівнів уваги в мережах декодерів, вихід уваги об’єднується у вхід. Увага, що використовується в цих моделях, обчислюється за допомогою багаторівневого персептрона [16].

Недоліком цього підходу є те, що модифікація кешу прихованого стану та обчислення уваги є дуже інтенсивними обчислювальними засобами. Ми також помітили, що додавання копіювання маркерів може призвести до зниження точності прогнозів маркерів у порівнянні з мережею без покажчика. Поділ прихованих станів на окремі частини для кодування мережі вказівників і контексту вирішує цю проблему. Механізм уваги також використовується деякими нерекурентними моделями, наприклад, Das і Shah [56] використали блок із блокуванням над вбудованими словами для прямої нейронної мережі. Ця модель використовувалася для представлення деяких типів значень, які часто зустрічаються, наприклад імен змінних ітераторів для контекстного завершення коду.

2.4 Висновки по розділу

Генерація коду в MDE здійснюється трьома основними підходами: M2T (модель → текст), M2M (модель → модель) і T2T (текст → текст), кожен з яких вимагає різного рівня знання моделей, синтаксису та мов трансформацій. У процесі широко застосовуються ідіоми обробки, подібні до компіляторних технік, для ефективного перетворення моделей у код з урахуванням структури, контексту та семантики цільової мови.

CGBE (Code Generator By Example) — це підхід до автоматичного синтезу генераторів коду з прикладів трансляції між мовами програмування L1 і L2 на основі контрольованого машинного навчання. Він використовує структуровані представлення, зокрема дерева синтаксичного аналізу, і символічні методи (як-от MTBE), що дозволяє навчатися з невеликих датасетів і створювати інтерпретовані транслятори.

РОЗДІЛ 3. ДОСЛІДЖЕННЯ МЕТОДІВ ПОБУДОВИ ОЗНАК І НАВЧАННЯ МОДЕЛЕЙ

3.1 Генерація даних та машинне навчання

Подібно до існуючих методів машинного навчання, ми повинні зібрати кілька прикладів вхідних даних для евристики та з'ясувати, якою має бути оптимальна відповідь для цих прикладів. Кожна програма скомпільована різними способами, кожна з яких має різну евристичну цінність. Ми вимірюємо час виконання скомпільованих програм, щоб з'ясувати, яке евристичне значення є найкращим для кожної програми. Через внутрішню мінливість часу виконання на цільовій архітектурі ми запускаємо кожну скомпільовану програму кілька разів (100 запусків), щоб зменшити сприйнятливість до шуму. Ми також витягуємо з компілятора внутрішні структури даних, які описують програми. Компонент пошуку об'єктів генеруватиме зведення цих даних як об'єктів-кандидатів. Типовими даними будуть абстрактне синтаксичне дерево, циклічні структури, ланцюжки визначення використання тощо. Будь-яка інформація, яку можна отримати, повинна бути записана, включно з будь-яким аналізом, виконаним за допомогою будь-якої існуючої евристики. Цей процес детально описано в розділі . Ці два аспекти генерації даних зображено на рисунку 3.1.

Компонент пошуку ознак, показаний на малюнку 6, підтримує сукупність виразів ознак, представлених у вигляді дерева вибору s 3. Вирази походять із сімейства, описаного граматикою, автоматично отриманою з IR компілятора. Оцінка функції в програмі генерує одне дійсне число; сукупність цих чисел для всіх програм формує вектор значень ознак, які пізніше використовуються підсистемою машинного навчання. Компонент пошуку використовує еволюційний пошук по деревах вибору з операторами пошуку. Вони дозволяють генетичні мутації та спаровування

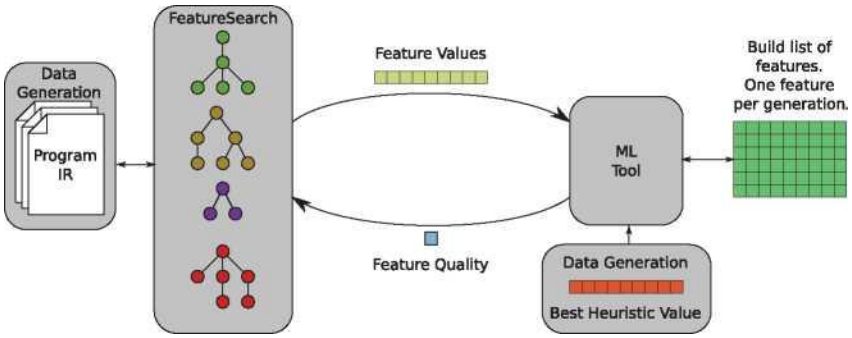

Рисунок 3.1 - Огляд системи пошуку ознак.

вибір дерев. Популяція вибраних дерев зберігається та сортується відповідно до функції відповідності. Після ранжування дерев створюється нова популяція такого ж розміру. Кожен член нової популяції створюється шляхом мутацій і спарювання або, рідше, простого копіювання членів попередньої популяції. Процес відбору для визначення осіб, які братимуть участь у створенні наступного покоління, є відбором назви туру. Під час турнірного відбору невелика кількість осіб вибирається рівномірно випадковим чином, і з них вибирається один таким чином, що ймовірність того, що він буде найкращим, є найвищою, а ймовірність для кожного, хто стоїть нижче в рейтингу, є експоненціально нижчою. Це гарантує, що сильніші люди з більшою ймовірністю передадуть свій генетичний матеріал новому поколінню. Модель побудована з нової функції та попередньо фіксованого набору функцій. Це означає, що набір функцій збільшується ітераційно, причому наступна функція є фокусом пошуку, як описано в наступному розділі. Відповідність функції визначається шляхом додавання її до раніше виправлених функцій і обчислення прискорення, яке можна отримати за допомогою моделі машинного навчання на основі цих функцій замість евристики розгортання GCC. Функція, яка покращує прискорення, краща за іншу, яка цього не робить.

Підсистема машинного навчання на рисунку 3.1 - це частина системи, яка забезпечує зворотний зв’язок із компонентом пошуку щодо якості функції. Як згадувалося раніше, система підтримує список хороших базових функцій, який спочатку є порожнім. Він постійно шукає найкращу наступну функцію для додавання до базових функцій, ітеративно створюючи список хороших функцій.

Система зупиняється, якщо їй не вдалося додати нову функцію, яка покращує результати. Остаточним результатом роботи системи буде список хороших характеристик у кінці пошуку. На рисунку 3.2 показано деталі процесу

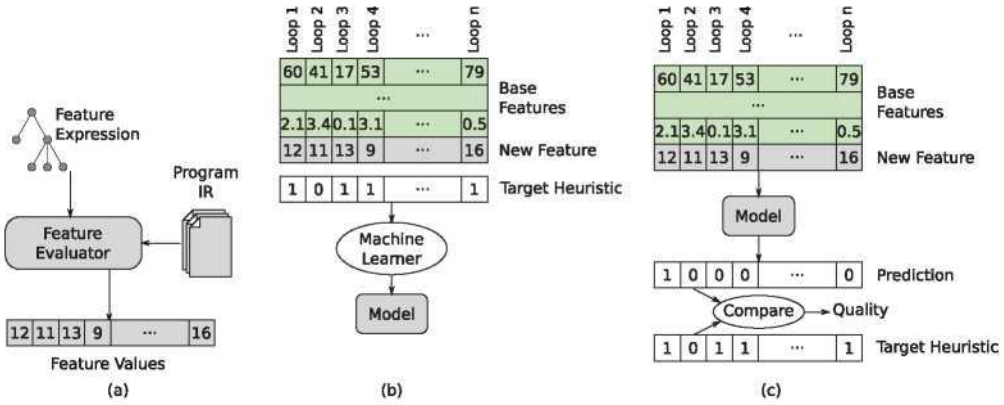

Рисунок 3.2 - Машинне навчання, що використовується для визначення якості функції.

Щоб оцінити якість нової функції, ми спочатку обчислюємо значення функції для всіх програм, як показано на рисунку 3.2 (a). Програмними даними можуть бути цикли IR для кількох тестів, а характеристикою може бути глибина циклу, помножена на кількість базових блоків. Система оцінювання обчислює вираз ознаки для кожної даної програми, створюючи вектор результуючих значень.

Потім ми поєднуємо цю функцію з попередніми базовими функціями та просимо алгоритм машинного навчання вивчити модель, яка може передбачити цільове евристичне значення (показано на рисунку 3.2 (b)). Можна використовувати будь-який інструмент машинного навчання. Однак, оскільки інструмент повинен вивчати модель кожного разу, коли нам потрібно обчислити придатність функції, він повинен бути швидким; ми можемо обчислити придатність кількох мільйонів функцій під час нашого процесу пошуку.

Панель (a) демонструє п’ятикратну перехресну перевірку. Набір даних розділений на п'ять наборів. Для кожного з п’яти згорток робиться прогноз для одного з наборів за допомогою моделі, навченої на інших чотирьох наборах. П’ять прогнозів зібрано для створення повного прогнозу. Панель (b) показує два

рівні перехресної перевірки, на кожному з яких проводиться один тестовий набір. Зовнішній рівень використовується для оцінки техніки, а внутрішній – для оцінки особливості.

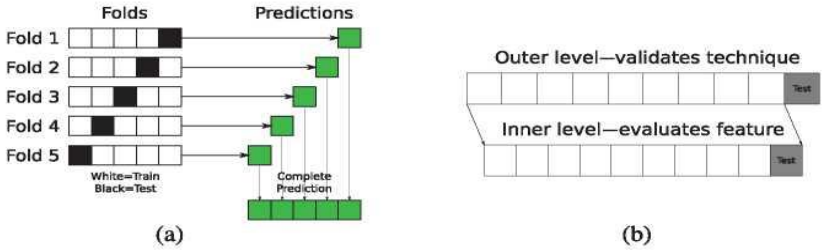

Рисунок 3.3 - Перехресна перевірка.

Деякі найсучасніші інструменти машинного навчання, такі як Support Vector Machines (SVM), працюють дуже повільно, іноді на два порядки порівняно з більш простими інструментами, такими як дерева рішень. Отже, у цій роботі ми використовуємо дерева рішень C4.5 [9], які є значно швидшими. Однак, оскільки, як і багато інших пошукових проблем, це можна тривіально розпаралелювати, час виконання повільніших інструментів може стати терпимим. Нарешті, ми перевіряємо якість моделі та повідомляємо про це компоненту пошуку (показано на малюнку 7(c)). У цій роботі ми використовували прискорення прогнозу як придатність моделі для функції. Серед інших можливостей можна було б використати точність передбачення, але ми виявили, що в цьому випадку система може зосередитися на підвищенні точності невеликих циклів за рахунок тих, які домінують у часі виконання. Покращення прискорення набагато ближче до мети оптимізації компілятора.

Перехресна перевірка . Перехресна перевірка використовується в машинному навчанні, щоб уникнути переобладнання навчального набору. Переобладнання означає, що прогностична модель може бути дуже хорошою для точок у навчальному наборі, але поганою для інших точок. Типовий сценарій перехресної перевірки поділяє вхідні дані на декілька наборів (скажімо, 10). Один набір залишають поза увагою та називають тестовим набором, решта (зазвичай дев’ять десятих усіх даних) називають навчальним набором. Модель

навчається на навчальному наборі, і вона передбачає значення для тестового набору. Це повторюється для кожного розділу, так що кожен розділ є тестовим набором один раз, і передбачення створюються для всіх вхідних даних. Загальний прогноз (хоча і складений з різних моделей) потім оцінюється на якість. Цей процес показано на малюнку 8(a). У нас є два рівні перехресної перевірки, показані на малюнку 8(b) - внутрішній рівень, який використовується під час пошуку, який оцінює, наскільки добре працюють окремі набори функцій, і зовнішній рівень, який оцінює, наскільки успішно завершилася вся процедура. Зовнішній рівень є більш типовим і використовується для визначення того, наскільки добре працює генератор функцій. Цей рівень означає, що дані програми розділені, забезпечуючи навчальний набір для всього пошуку функцій, як на рисунку 7(a)-(c), і повні набори функцій створюються для кожного розділу. Моделі, отримані з цих наборів, використовуються для прогнозування значень для тестових наборів, а весь прогноз використовується для оцінки ефективності методу. Це нічим не відрізняється від більшості експериментів з машинним навчанням.

З іншого боку, внутрішній рівень використовується під час пошуку функцій, щоб процес пошуку міг оцінити, наскільки добре працює кожен набір функцій. Крок, показаний на малюнку 7(b), фактично проходить перехресну перевірку (ми використовуємо 10-кратну перехресну перевірку, тому створюється 10 моделей для 10 пар навчального набору/тестового набору). Разом об’єднані кілька моделей створюють прогноз на малюнку 7(c) для кількох наборів тестів. Внутрішній рівень жодного разу не побачить тестовий набір зовнішнього рівня. Навчальні та тестові набори внутрішнього рівня походять лише з навчального набору зовнішнього рівня.

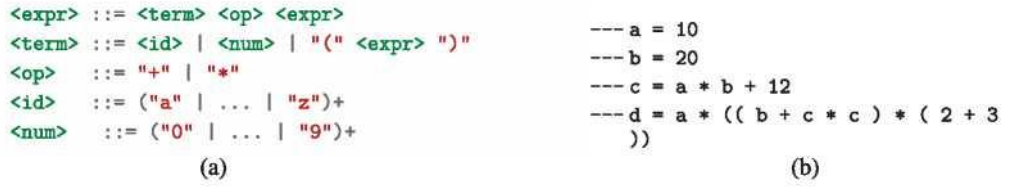

Рисунок 3.4 Граматика простої мови (a) і приклади висловлювань з неї

Скупість . На практиці система використовує інформацію на додаток до метрики якості, описаної раніше. Трапляється, що вирази ознак, отримані еволюційним пошуком, можуть швидко стати дуже довгими. Дві функції можуть мати однакові результати, але мати різну довжину (наприклад, якщо одна функція є loop.depth, більш складною функцією без додаткової сили прогнозування є loop.depth+(1+2) xloop.depth. Щоб вирішити цю проблему, ми беремо на озброєння добре відому методологію GP re war ding parsimony. Якщо цільова функція, обчислена інструментом машинного навчання для двох функцій, дає однакове значення, тоді ми визначаємо, що вираз ознаки, який є коротшим, є кращим.

3.2 Визначення характерного простору

Наведено приклад іграшки, щоб показати, як працює процес. Мова іграшок допускає лише набори операторів присвоєння; ліворуч від кожного буде назва змінної; у правій частині буде вираз, що містить змінні, постійні цілі числа, оператори '+' і '*' і круглі дужки. Форма Бекуса Наура (BNF) для простої мови та приклади висловлювань показані на рисунку 3.4. Експерт-компілятор, думаючи про функції, які потрібно обчислити над виразами, швидше за все, розробить прості функції, такі як кількість операторів у виразі або глибина виразу. Він запровадить і перевірить ці функції та, якщо пощастить, виявить, що вони дещо корисні для машинного навчання, але, швидше за все, не працюють так добре, як він сподівався. Тоді він, ймовірно, створить невеликі варіації своїх оригінальних рис. Наприклад, щоб розширити свій підрахунок множинних вузлів, він може подумати, що іншою функцією може бути підрахунок тих множинних вузлів, які мають лівим дочірнім оператором «+», а правий дочірній елемент — константа. Він може додати цю функцію до свого набору та знову спробувати свій інструмент машинного навчання.

Існує необмежена кількість цих функцій, однак, оскільки експерт з компілятора може вибрати подальше вдосконалення своєї нової функції,

вказавши властивості, рекурсивно, для дітей вузла «+», наприклад, обмеживши набір значень для константи або будь-яку кількість складних модифікацій, які тільки можна собі уявити. Експерт повторюватиме цей процес, постійно впроваджуючи та тестуючи нові функції, доки або не буде знайдено подальших удосконалень у моделі машинного навчання, або поки не закінчиться час для експерименту.

```
<feature> ::= "countNodesMatching(" <matches> ")"
<matches> ::= "isConstant" | "isVariable" | "isAnyType"
            | ("isPlus" | "isTimes")
              ("&& leftChildMatches(" <matches> ")")?
              ("&& rightChildMatches(" <matches> ")")?
```

Рисунок 3.5 - Проста граматика ознак.

Речення з граматики - це вирази, які можна оцінити порівняно з висловлюваннями з мови іграшок. Кожна функція підраховує кількість піддерев у операторі мови іграшки, які відповідають шаблону

Використаний тут підхід полягає в автоматизації цього трудомісткого процесу. Оскільки ця система досліджуватиме простір потенційних функцій, вона повинна знати, що це за простір. Простір ознак описується за допомогою граматики ознак, де мова прийнятий граматикою є деякою підмножиною існуючої мови програмування. Кожне речення е з граматики буде вираз, який може обчислити значення функції, коли виконується над внутрішнім представленням поточного розділу коду компілятора. Автор компілятора повинен вибрати, який простір функцій шукати, і розробити граматику функцій для представлення цього простору. Сам процес проектування не автоматизований; автор компілятора може зробити доступними в компіляторі функції, які досліджують абстрактне синтаксичне дерево, графи потоку керування та будь-які інші структури даних. Граматики функцій і граматика мови компілятора не повинні бути пов’язані.

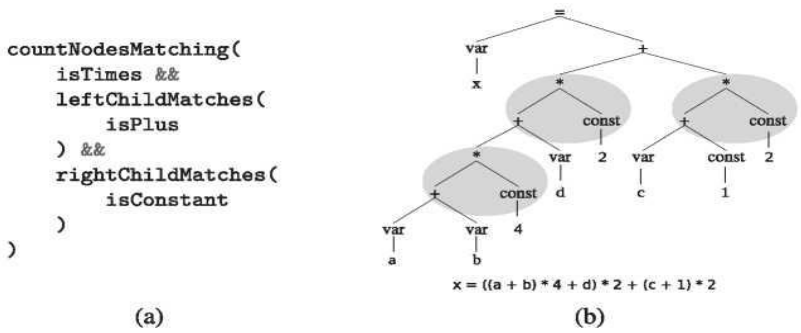

Рисунок 3.6 - Приклад функції з граматики на Рисунок 3.4.

У правій частині малюнка наведено зразок AST з крихітної мови, що показує відповідні підструктури. Таким чином функція оцінюється в три.

На рисунку 3.4 показана проста граматика, яка описує набір таких функцій у стилі псевдокоду. Ці приклади функцій можна обчислити на основі виразів мови іграшок, і в цьому випадку кожна функція підраховує кількість піддерев у виразі, що відповідає шаблону. На рисунку 3.5 показаний приклад функції з цієї граматики та оцінка щодо прикладу фрагмента програми, що показує відповідні піддерева. Застосування цієї конкретної функції до цього фрагмента коду дає значення 3. У наступному розділі описано, як буде обчислюватися функція.

Граматика ознак визначає набір ознак, кожна з яких є реченням цієї граматики. Після того, як ознака була витягнута з граматики, її потрібно буде оцінити за внутрішніми даними компілятора, щоб обчислити значення функції. Граматика здатна створювати речення, які є підмножиною певної мови програмування. Таким чином, інтерпретатор або компілятор для функцій вже існує у формі будь-яких компіляторів або інтерпретаторів для базової мови або якщо їх не існує.

```
1  {int num_vars = 0;}
2  <feature> ::= "int " <defs> ";"
3                "return " <expr> ";"
4  <defs>    ::= <def>
5              | <def> "," <defs>
6  <def>     ::= "i" { print( ++num_vars ) } "=" { print( random( 0, 255 )) } ";"
7  <expr>    ::= <expr> " + " <expr> | <var>
8  <var>     ::= {print("i" + random(1 to num_vars))}
```

Рисунок 3.7 - Граматика ознак із семантичними діями

що узагальнює фрагмент коду з рисунка 3.6. Семантичні дії знаходяться між фігурними дужками.

Базова мова, на якій створюються функції, як правило, є еквівалентом Тьюринга; однак використання чистих CFG може унеможливити вираження всіх типів ознак. У наступному розділі розповідається про те, як обробляти випадки, коли обмеження бути контекстно-вільним є занадто обмежуючим.

CFG підлягають ряду обмежень, які обмежують типи речень, які можна створити. Розглянемо, наприклад, простий приклад, у якому буде визначено кілька змінних, а потім з використанням цих змінних буде створено вираз, як показано на рисунку 3.6. Це не можна виразити за допомогою CFG, оскільки CFG не може передати залучену семантику [8]. Більш реалістичні ситуації виникають під час побудови вкладених циклів або визначення функцій, оскільки таблицю символів змінено, що виходить за межі синтаксичних можливостей CFG. Ситуація має паралелі у світі розбору програм. Там CFG використовується для опису синтаксису мови, а семантика мови вбудована як «семантичні дії» ([6]) у граматику. Під час аналізу ці дії дозволяють запускати довільний код під час процесу аналізу; наприклад, таблиці символів оновлюються та перевіряються. Ця система має подібний механізм, що дозволяє граматикам виробляти більш складні функції. Семантичні дії вбудовані в граматику, і всякий раз, коли продукцію n вибрано, дії виконуються у відповідному місці. Ці семантичні дії є фрагментами довільного коду, які можуть оновлювати стан і друкувати значення. Наприклад, на рисунку 3.7 показана граматика, яка узагальнює фрагмент коду з рисунка 3.6 Перша дія в рядку 1 ініціалізує лічильник змінної нулем; ця дія виконується перед розгортанням будь-яких правил. Наступні дії відбуваються в рядку 6, який визначає іншу змінну та збільшує кількість. Остання дія в рядку 8 друкує назву випадкової змінної з доступних. Семантичні дії можуть бути розміщені на вході або виході з усієї граматики, правил і продукцій, а також усередині продукцій, як показано в прикладі. Ці додаткові можливості потрібно використовувати лише помірно, але вони дозволяють надзвичайно детально та потужно контролювати функції, які можна створити.

3.3 Генерування характеристик з граматик

Тепер, коли існує граматика, що описує простір ознак, з неї можна створити будь-яку кількість ознак. Потрібно просто почати з кореневого правила граматики (яке у випадку граматики на малюнку 10 є правилом <feature>) і розгорнути будь-які нетермінали в ньому. Щоразу, коли є вибір виробництва для розширення, вони вибираються випадковим чином за допомогою колеса рулетки [4], де ймовірність вибору кожного виробництва пропорційна його вазі. Продовжуючи, доки в реченні не залишиться нетерміналів, буде завершена функція. Для прикладу на рисунку 3.5 похідна схема наведена на рисунку 3.7. Перший крок починається з кореневого правила граматики функції - розміщення одного нетермінала як поточного речення. На другому кроці нетермінал замінюється єдиним можливим правилом, залишаючи лише один нетермінал для заміни. На третьому кроці нетермінал <matches> замінюється; є п'ять виробництв, і остання вибирається випадковим чином; нетермінал замінюється значенням виробництва, що дає два нетермінали для заміни. Нарешті, на четвертому кроці, решта <matchs> нетерміналів замінюється.

```
<A> ::= <A> "a"
```

Рисунок 3.8 - Необмежена, рекурсивна граматика.

Єдине речення, яке розпізнає граматика, це необмежена послідовність as. Спроба згенерувати речення з цієї граматики ніколи не закінчиться, тому що завжди буде незамінений нетермінальний <A>.

```
<A> ::= <A><A><A> | "a"
```

Рисунок 3.9 - Імовірнісно рекурсивна граматика.

У будь-якій точці розширення речення всі нетермінали можна замінити терміналами. Однак, оскільки два виробництва вибрано з однаковою

ймовірністю, а перше виробництво генерує таку кількість рекурсивних нетерміналів, швидше за все, відбудеться вибух нетерміналів, і речення ставатиме все довшим і довшим.

У той час як двозначність створює проблеми при розборі речень із граматики, створення речень не вразливе до неї. З іншого боку, створення речень страждає від необмеженої рекурсії. Можливо, найпростішим прикладом цього є граматика на рисунку 3.8, яка, очевидно, створює нескінченний рядок «a». Спроба створити речення з цієї граматики не вдасться. Мова визначення граматики допускає вбудовані дії, подібні до семантичних дій у генераторах синтаксичних аналізаторів, які дозволяють вручну вирішувати такі проблеми, наприклад, накладаючи обмеження на глибину рекурсії. Подібні граматики можуть бути статично ідентифіковані та позначені як такі; однак на практиці ці граматики навряд чи можна побачити. Більш тонкі проблеми з рекурсією викликані імовірнісною причиною. Розглянемо граматику на малюнку 19. Мова, яку вона розпізнає, складається з непарної кількості послідовних літер «a». Однак, якщо дві постановки вибрані випадковим чином, то, ймовірно, утворяться дуже довгі струни. 6 Оскільки рядки містять більше нетерміналів, зростає ймовірність їх зростання при кожному розширенні.

Вибухової довжини речень можна уникнути, змінивши ймовірність різних постановок. Якби у прикладі з рисунка 19 вага першої продукції була менше однієї третини ваги другої продукції, тоді очікувана довжина речення після заміни кожного нетерміналу один раз була б меншою, ніж початкова; розширення речення не вибухне. Визначення відповідних вагових коефіцієнтів для виробництва зазвичай неможливо зробити аналітично. Це пов’язано з наявністю семантичних дій, які вводять довільно складний код у розширювач граматики. Однак, як видно з рисунка 3.10, коли вагові коефіцієнти створюють невибухову граматику, існує відносно невелика чутливість до значень вагових коефіцієнтів. Таким чином, досить легко бути консервативним щодо вагових коефіцієнтів, і граматика від цього не сильно постраждає; метод проб і помилок дає прийнятні результати з дуже невеликими витратами часу чи зусиль.

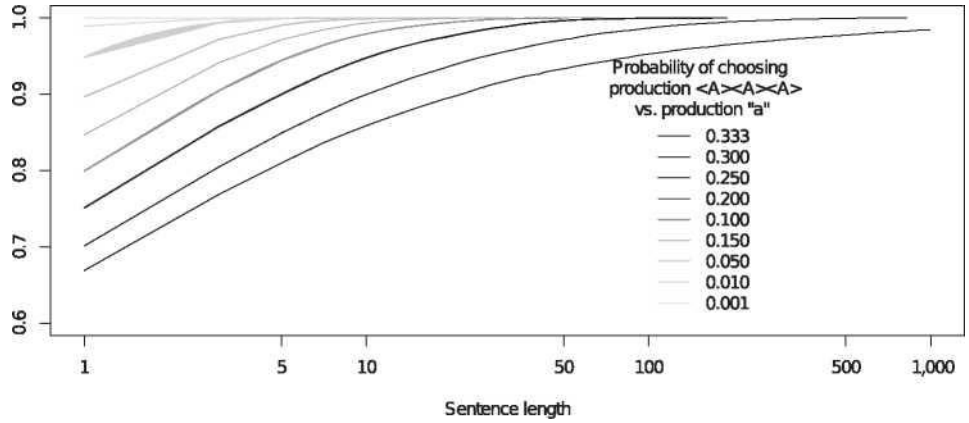

Рисунок 3.10 . Кумулятивна ймовірність отримати речення заданої довжини при розширенні граматики <A> :: = <A><A><A> |

«a» з різною вагою для двох виробництв. Навіть коли вагові коефіцієнти наближаються до початку -безперервного генерування речень (що вперше відбувається, коли ймовірність створення <A><A><A> на третину нижча за ймовірність створення «a»), існує значне упередження в бік коротких речень. Графіки також допомагають пояснити, чому встановлення ваг методом проб і помилок є тривіальним. Консервативне ставлення до ваг не має великого впливу на довжину речення; вони завжди будуть короткими для можливої ваги.

Як описано в попередньому розділі, граматика повинна зважувати продукцію, щоб запобігти невловимим, вибуховим реченням. Наслідком цього є те, що граматична система упереджена до коротких речень. На рисунку 3.10 показано зміщення довжини речення для граматики на мрисунку 3.9. Можна побачити, що утворюються короткі речення, навіть коли виробництво зважено близько до порогу, за яким починається стрімке розширення. Для цієї граматики переважна більшість речень будуть коротшими за 10 символів. Створення граматик, які створюють переважно короткі речення, має деякі недоліки. По суті, уникаючи вибуху, перевага коротким реченням перешкоджає вивченню більшої частини граматики. Часто після генерації кількох тисяч випадкових речень система зазвичай відтворює речення, які вже бачили; дослідження фактично припиняється.

Найпростіший спосіб пошуку в просторі функцій – це випадкова генерація функцій. Тисячі функцій можна створити за лічені секунди, які потім можна оцінити. Однак упередження в бік коротких речень означає, що цей підхід буде практично обмежений невеликою частиною повної мови функцій, прийнятої pCFG. Ранні експерименти показали, що обсяг простору функцій, який можна охопити, був набагато меншим, ніж очікувалося. Можна спробувати використати семантичні дії, щоб змінити упередженість простору ознак, коли пошук починає насичуватися дослідженням коротких функцій. Це, однак, важко організувати, що покладає важкий тягар на автора граматики, щоб додати велику кількість крихкого коду, який безпосередньо не пов’язаний з описом простору функцій. Можна також розставити ваги постановок на користь довгих речень. Це швидко призводить до невиправданих речень у генераторі функцій. Навіть якщо генератор налаштований на вихід, коли речення стає занадто великим, генератор витрачає більшу частину свого часу на створення функцій, які не працюють або які вже бачили раніше. На щастя, проблема зміщення коротких речень вирішується як побічний ефект інших методів пошуку, ніж наївний випадковий підхід. Припустимо, що для початку є якась функція, вибрана з упередженого простору. До нього можна внести невеликі зміни; можливо вкоротити, можливо подовжити. Функція більше не буде обмежена накладеним зміщенням pCFG (зміщення тепер лише інформуватиме, звідки слід починати пошук і його напрямок, не обмежують обсяг цього пошуку). Таким чином, використовуючи еволюційний пошук, зміщення коротких речень більше не є проблемою.

Об’єкти, за якими здійснюється пошук у системі, - це дерева вибору, зроблені під час побудови речення чи функції. Дерево подібне до дерева синтаксичного аналізу, за винятком того, що замість того, щоб містити вузол для кожного нетермінала та терміналу в дереві синтаксичного аналізу, воно кодує вибір між створенням правил та будь-якими іншими необхідними даними, які ведуть до дерева синтаксичного аналізу. Оскільки деякі правила можуть мати лише одиничну продукцію, вони не вимагають вибору, а тому не мають представлення в дереві вибору. Дерева записують усе, що потрібно для

ефективного виклику операторів генетичного пошуку в об’єктах.

Дерево вибору кодує вибір, зроблений під час генерації ознаки або речення з pCFG. Кожне піддерево описує вибір, зроблений для розширення окремого правила та його дітей. Кожен вузол містить випадкові біти, які використовувалися для вибору виробництва, і будь-які випадкові дані, необхідні для семантичних дій цього виробництва. Дерево вибору дуже схоже на дерево синтаксичного аналізу для речення з доданими випадковими бітами, за винятком того, що воно також може містити надлишкову інформацію, яка не використовується при виведенні ознаки. Для цих цілей передбачається, що граматика написана мовою БНФ. Дерева вибору можна використовувати двома додатковими способами; один записує вибір, зроблений під час розширення функції, і після запису його можна використовувати для повторного відтворення цих виборів, щоб відтворити ту саму функцію, що й раніше. Під час запису будь-які випадкові біти запам’ятовуються, а піддерева розмежовуються згідно з правилами та діями граматики. Під час повторного відтворення джерело випадкових бітів замінюється тими, що були записані раніше.

Приклад дерева вибору наведено на рисунку 3.11. Проста граматика показана в першому блоці рисунка 3.11(a). Далі на рисунка 3.11(b) показано можливе похідне речення babb, починаючи з <A>, де на кожному кроці один нетермінал замінюється продукцією. Нарешті, на рисунку 3.11(c) є дерево вибору, яке генерує попереднє виведення. Перше правило <A> має три варіанти, тому генерується випадкове число 219 (для простоти передбачається, що вони мають довжину лише один байт). Це використовується для вибору колеса рулетки, де ймовірність вибору кожного продукту пропорційна його вазі. Є три варіанти для продукції <A>, і оскільки всі продукції мають одиничні ваги, вибір рулетки еквівалентний функції модуля. Випадкове число 219 за модулем 3 дорівнює 0, тому вибрано перше виробництво, <A> ^ <A><A>. Подальші вузли в дереві представляють решту замін нетерміналів. У випадках, коли нетермінали <B> замінені, вибору немає; позначається X.

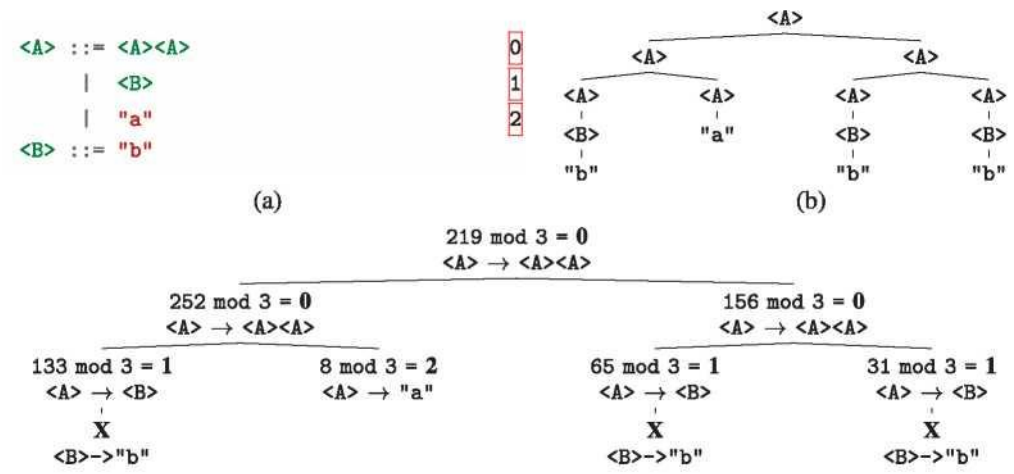

Рисунок 3.11 - Приклад дерева вибору.

Проста граматика показана на (a), де продукція правила для нетермінального < A> пронумерована для ясності. Можливий вивід речення babb наведено на (b). На (c) є дерево вибору для виведення в (b); під час вибору продукції для правила < A> потрібне випадкове число (у цьому прикладі з [25]), а потім продукція вибирається за допомогою вибору на колесі рулетки (еквівалентно mod 3 у цьому випадку), щоб отримати номер продукції. Не всі правила пропонують вибір, тому не потребують випадкових бітів (позначених X).

Система пропонує кілька пошукових операторів, придатних для еволюційного пошуку, підйому на пагорб або інших методів. Кожне з них потребує певного методу, щоб взяти певну кількість перспективних дерев і створити одне або більше нових дерев-кандидатів, які будуть наступними точками, які розглядатимуть пошук. Під час сходження на пагорби одне дерево певним чином змінюється, щоб створити нове дерево. В еволюційному пошуку нове дерево також може успадкувати частини двох дерев. Оператори на вузлах дерева вибору включають:

- Видалення: піддерево буде замінено новим випадковим деревом під час відтворення.

- Змінити випадкові біти: тепер вузол може вибрати інше виробництво; це може призвести до того, що деякі діти будуть відсутні або нерелевантні. Часто

правила мають багато продукцій зі схожими формами (наприклад, правило для бінарних виразів може мати багато продукцій з однаковими двома нетерміналами); зміна випадкових бітів може просто замінити одне таке виробництво іншим. Константи також генеруються з випадкових бітів, тому цей тип мутації може шукати різні константи.

- Перетасування дітей (кілька варіантів): якщо діти походять із непов’язаних нетерміналів, то це може бути еквівалентно видаленню всіх дітей і створенню їх заново (хоча і з деякими заздалегідь заданими випадковими бітами). Якщо нащадки пов’язані, наприклад, нащадки багатьох двійкових операторів, їхні порядки можуть бути змінені; наприклад, лівий і правий діти віднімання можуть бути поміняні місцями.

- Перехресні піддерева: два піддерева з двох дерев вибору міняються місцями. Це дозволяє обмінюватися інформацією між деревами вибору, як це потрібно для методів GP. Існує два основних варіанти цього оператора: перший вибирає будь-які піддерева з двох дерев вибору; другий є більш цільовим - він спочатку розширює обидва дерева, запам’ятовуючи назви правил, до яких відноситься кожен вузол у деревах. Потім він віддає перевагу вибору двох піддерев, які стосуються одного правила, тобто мають однаковий нетермінал. Обидва ці основні варіанти мають параметри, що вказують ймовірність вибору піддерев, ґрунтуючись, наприклад, на глибині або кількості вузлів піддерева.

Оператори можуть змінити дерево вибору так, щоб не було достатньо випадкових бітів для завершення повторного відтворення дерева для створення функції. Наприклад, оператор пошуку мутацій може видалити деяке піддерево або змінити просту продукцію без дочірніх елементів на необхідну

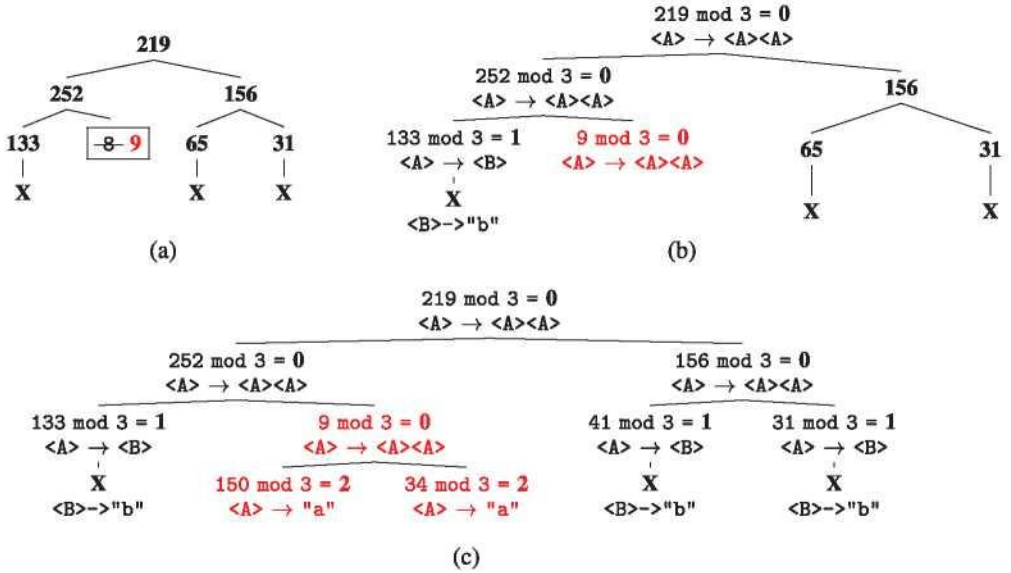

Рисунок 3.12 - Відновлення дерева вибору.

На рисунку (a) дерево вибору з рисунка 3.12 має один мутований вузол (позначено рамкою). Повтор починає зчитувати дерево вибору на рисунку (b) як зазвичай, доки не буде досягнуто мутованого вузла, коли буде вибрано інше виробництво. Нове виробництво потребує більше даних, ніж містить дерево, тому генерується та записується в дерево (c). Після цього залишки дерева залишаються незмінними.

кілька дітей. У цьому розділі описано, як ці дерева ремонтуються, щоб вони завжди були дійсними; жодна пошукова операція, якою б різкою вона не була, не може створити недійсне дерево. Механізм відновлення дерев полягає в тому, що коли під час відтворення потрібні додаткові біти, вони створюються випадковим чином і записуються назад у дерево вибору. Під час відтворення будь-які біти, присутні в дереві в правильних місцях, стають корисними. Однак, якщо в будь-який момент буде відсутня інформація, це спричинить перехід із режиму відтворення на режим запису, доки необхідна інформація не буде створена. Таким чином, читання дерева вибору може змінити дерево як побічний ефект, але в жодному разі дерево не втратить інформації. На малюнку 22(a) показано дерево вибору з рисунка 3.11, яке було мутовано під час генетичного пошуку так, що випадкові біти одного вузла змінилися (позначено прямокутником). Система починає відтворювати дерево (Рисунок 3.12(b)),

розширюючи кореневий нетермінал, роблячи вибір відповідно до даних у дереві. Це відбувається як зазвичай, доки не буде виявлено мутований вузол. У цей момент випадкові біти у вибраному вузлі вибирають інше виробництво, ніж у вихідному дереві; <A> стає <A><A>, а не «a». Тепер, якби дерево було повним, під мутованим вузлом має бути два дочірні вузли, яких немає. Щоб вирішити цю проблему, система починає випадковим чином створювати будь-які вузли, які їй потрібні, створюючи нове піддерево, яке базується на мутованому вузлі (Рисунок 3.12 (c)). Повернувшись після відновлення мутованого вузла, система починає відтворення, як зазвичай, надаючи повну функцію та оновлюючи дерево, щоб тепер воно було повним і запам'ятало нову інформацію, яка була використана для відновлення дерева. Також може статися, що модифікації дерева вибору збільшать кількість інформації в дереві понад необхідну. Це відбувається, якщо біти, які вибирають вузол із багатьма дочірніми елементами, змінюються так, що вузол матиме лише одного дочірнього елемента, або до вузла додаються додаткові біти, які не використовуються для вибраного виробництва. Це не пошкоджує дерево вибору; додаткові дані просто не впливають на функцію, отриману з дерева.

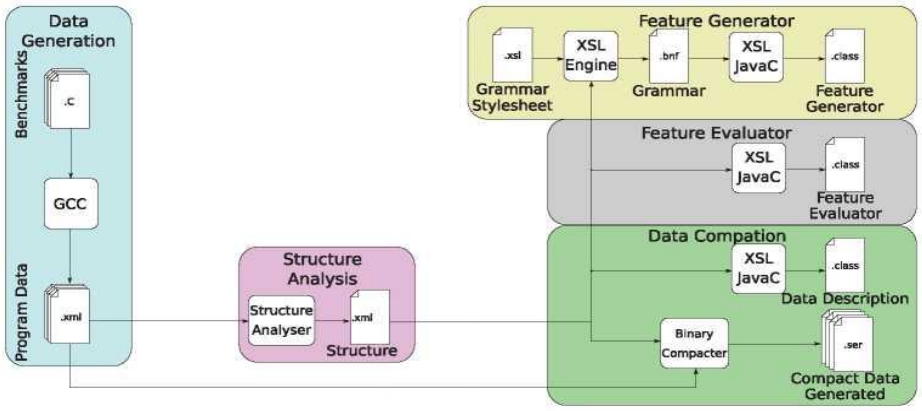

Рисунок 3.13 - Огляд системи створення граматики для циклів RTL у GCC

Система пропонує два режими відтворення дерева вибору; у першому випадку будь-які сторонні дані не видаляються з дерева, а у другому зайві біти

та вузли видаляються з дерева. Тепер, коли стало зрозуміло, що жодні зміни в деревах вибору не можуть пошкодити їх без відновлення, у наступному розділі розглядаються типи операторів пошуку дерева вибору, які надає система.

На першому етапі система витягує внутрішні структури даних GCC для кожного з тестів у файли XML. У RTL інструкції мають алгебраїчну форму з представленням списку списків. Кожен вузол у RTL може мати певну кількість атрибутів. Представлення циклів RTL витягується, доповнюється, щоб включити структуру основних блоків у циклі та інструкції RTL, що містяться в їхніх блоках. Також експортується будь-яка інформація, яку GCC може обчислити в той час, наприклад, приблизні частоти блоків, глибини циклу тощо. Частина даних наведена на рисунку 3.14.

```
<loop name="UTDSP.fft_1024.fft.3" ... insns="28" expected-iter="11">
  <basic-block index="10" ... frequency="9100" loop-depth="3"> ...
    <insn ...> ...
      <set ...>
        <reg ... mode="SF"> <int>112</int> ... </reg>
        <mult ... mode="SF">
          <reg mode="SF"> <int>94</int> ... </reg>
          <reg mode="SF"> <int>84</int> ... </reg>
        </mult>
      </set>
    </insn>
```

Рисунок 3.14 - XML-представлення RTL.

Багато доступних атрибутів і значень видалено для ясності. Такі атрибути, як очікуваний ітер і частота, є припущеннями GCC, які він оцінює за відсутності профілюючої інформації. У прикладі показано частину третього циклу функції fft у тесті UTDSP, fft 1024. Показано одну інструкцію в одному з базових блоків, яка встановлює значення одного регістра до добутку двох інших.

Створені ієрархічні дані відповідають ряду правил відносин. Наприклад, цикли містять базові блоки як дочірні, а вони, у свою чергу, містять інструкції. Ці стосунки -ніколи не порушуються. Було б марнотратно створювати функцію на кшталт «підрахувати кількість базових блоків, які містять три цикли», оскільки вона завжди матиме нульовий результат. Створені граматики

автоматично виводяться на основі структурних правил даних, щоб гарантувати, що такі неможливі функції ніколи не генеруються, що покращує ефективність пошуку. Оскільки правила, яким дотримуються внутрішні структури даних GCC, не можна відразу вивести машиночитаним способом із вихідного коду GCC, застосовано простіший підхід до вивчення файлів даних XML, щоб знайти спостережувану структуру всередині. Файли даних XML повторюються, і збирається номер кожного типу вузла в XML; записується кількість дочірніх елементів кожного типу вузла; і будується гістограма дочірніх типів для кожного дочірнього слота в кожному типі вузла, як і гістограма значень атрибутів для кожного типу вузла. Ці аналізи збираються в «структурний документ» для використання на наступних етапах.

Щоб підвищити продуктивність, файли даних XML, створені, перетворюються в компактний двійковий формат, використовуючи структурний документ як керівництво. У структурному документі оголошуються всі різні типи вузлів, назви їхніх атрибутів і значення, а також дочірні елементи, які може мати кожен вузол. Система спочатку створює клас Java, який представляє кожен тип вузла. Кожен атрибут відображається на поле відповідного типу, а дочірні елементи вузла упаковуються у вигляді масиву. Документ структури використовується як вихідні дані, щоб система знала, які класи Java потрібні. Далі файли даних програми XML зчитуються, і кожен конвертується в об’єкти класів Java і серіалізується у файли. Отримані дані набагато менші, ніж вихідний XML, їх можна завантажити в пам’ять усі одночасно, і набагато швидше обчислювати значення функцій.

Незважаючи на те, що оцінка функцій може бути виконана за допомогою існуючої мови сценаріїв, загальні мови сценаріїв виявилися надто повільними для пошуку. Компіляція також було випробувано функції програм C і Java.

У C спроба створити GCC, а потім програму функцій також виявилася надто повільною. Для Java компіляція могла бути виконана в процесі, але отримані класи після завантаження в пам’ять не мали тенденції збирання сміття, тому зрештою вся пам’ять була вичерпана. Замість цього було створено

спеціальний інтерпретатор функцій. Не було потреби створювати синтаксичний аналізатор для інтерпретатора, оскільки граматика функцій може створювати AST безпосередньо, а не лише рядки.

```
<numeric> ::= <numeric> ( "+" | "-" | "*" | "/" ) <numeric>
            | <value-of-an-attribute>
            | ( "sum" | "min" | "max" | "avg" )
              "(for-each-child-that(" <match> "do" <numeric> "))"
            | "count-children-matching(" <match> ")"
            | "on-child" <random> "do" <numeric>
<match>   ::= <match> ( "or" | "and" | "xor" ) <match>
            | "not(" <match> ")"
            | <numeric> ( "<" | ">" ) <numeric>
            | "is-type(" <node-type> ")"
            | <attribute> "=" <value>
```

Рисунок 3.15 - Спрощена підмножина автоматично створеної граматики.

Інтерпретатор підтримує:

— Отримання типів вузлів.

— Отримання значень атрибутів вузлів, включаючи визначення відсутності атрибутів. - Логічні операції, арифметичні та порівняння.

— Агрегатори (тобто підсумовування, знаходження екстремумів).

—Повторення дітей, нащадків.

— Стандартні операції потоку керування.

— Зіставлення шаблону. Підтримка виконання для оцінки функцій також обчислюється з документу структури. У наступному розділі описано, як створюються функції, які використовують цей інтерпретатор.

3.4 Методика експерименту

Генератор функцій шукає одну функцію за раз. Він надає перевагу функціям, які в поєднанні з функціями, вибраними на попередніх етапах, найбільше покращують продуктивність інструменту машинного навчання. Еволюційний пошук кожної ознаки включав популяцію зі 100 особин. Кожному дозволялося працювати, доки 15 поколінь не приведуть до покращення

найкращих характеристик популяції, або максимум 200 поколінь, залежно від того, що відбудеться раніше. Пошук нових функцій для додавання припинявся, коли було досягнуто 2500 поколінь або коли нам п’ять разів не вдалося знайти покращену функцію.

Ми розділили цикли на 10 груп, залишивши одну групу для тестування, щоб ми могли виконати 10-кратну перехресну перевірку. Цикли, які використовуються для створення функцій і подальшого вивчення моделі, ніколи не використовуються для оцінки моделі. Остаточне оцінювання завжди відбувається на невидимих циклах. Алгоритм машинного навчання, який використовувався для визначення якості функцій, був простим деревом рішень C4.5 [2], обраний за його швидкість. Коли функція була оцінена, ми навчили дерево рішень на восьми з решти дев’яти розділів циклу, які називаються навчальним набором. Потім ми попросили дерево рішень передбачити фактори розгортання для циклів у решті (дев’ятої) частини, що називається внутрішнім набором перевірки . Потім це було використано для визначення прискорення, досягнутого цими факторами розгортання.

Системі знадобилося 2 дні, щоб вивчити найкращий набір функцій і модель для цієї проблеми. Хоча це значний проміжок часу, це одноразова діяльність, яка виконується «на заводі» і може бути легко розпаралелена. Якщо взяти до уваги кількість часу, який потрібен автору компілятора, щоб розробити хорошу евристику, ці витрати насправді невеликі. Теоретично можливо, щоб система створювала надзвичайно дорогі обчислювальні -функції, збільшуючи час компіляції . Система змушує ці оцінки функцій витримувати час очікування, даючи їм щонайбільше 2 секунди для оцінки в усіх циклах (таким чином, з 2778 циклами за 2 секунди жодна функція не може зайняти в середньому більше 0,7 мс для обчислення на цикл). Якщо функція вичерпується, вона відкидається і не може вносити вклад до генофонду. Ми виявили, що тиск на простіші функції означає, що функції рідко закінчуються. Функції, вибрані системою для розгортання, не мають істотного впливу на час виконання GCC.

Щоб визначити, наскільки добре працює наша техніка та інші, ми

спочатку проводимо граничне дослідження. Як описано, ми вичерпно перерахували коефіцієнти розгортання циклу до 15 для кожного циклу, записуючи найкраще налаштування для кожного. Це обмеження було вибрано, оскільки це був найбільший коефіцієнт розгортання, обраний евристикою за замовчуванням для будь-якого циклу. Потім ми запустили кожен контрольний тест із набором найкращих факторів розгортання та зафіксували прискорення.

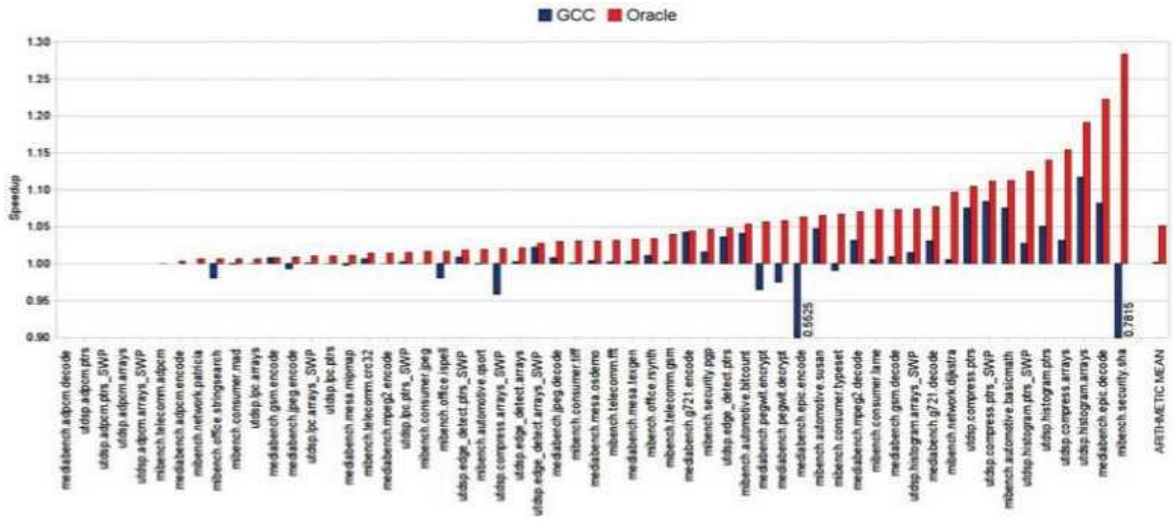

Рисунок 3.16 - Прискорення факторів розгортання, обраних евристикою GCC за замовчуванням, і прискорення найкращого можливого фактора розгортання оракула.

Oracle на рисунку 3.16, показують максимально досяжне прискорення порівняно з відсутністю розгортання для контрольних тестів. Що відразу стає очевидним, так це те, що вплив розгортання циклу різко відрізняється в різних тестах із середнім прискоренням 1,05. Для деяких тестів, таких як adpcm від MediaBench, жоден фактор розгортання не впливає на продуктивність. Проте у випадку security_sha від MiBench потенційне прискорення становить 1,28. Нам хотілося б створити схему, яка здатна використовувати цей потенціал: забезпечувати прискорення, коли вони доступні, і не сповільнювати роботу програми в інших випадках. Якщо ми тепер розглянемо набір стовпчиків на рисунку 3.16, позначений GCC, ми побачимо продуктивність GCC у тому самому наборі тестів. У деяких випадках він здатний досягти прискорення 1,12 на

histogram.arrays від UTDSP, але у випадку security_sha , який має найбільший потенціал для збільшення продуктивності, він забезпечує значне уповільнення на 0,78. Насправді він уповільнює 12 тестів, найгіршим є epic_encode від MiBench, де уповільнення становить 0,55. Це демонструє труднощі, з якими стикаються автори компіляторів у розробці портативної оптимізації, яка забезпечує підвищення продуктивності.

Враховуючи потенційну продуктивність, доступну від розгортання циклу, і низьку продуктивність GCC ¬, ми зараз демонструємо, як наш підхід покращує це. Порівняння з GCC і Oracle . Смужки на рисунку 3.16, позначені як «Наша техніка» , показують прискорення нашого підходу в наборі тестів. У середньому ми можемо досягти 76% від максимально доступного. У тих тестах, де доступне велике потенційне прискорення, наприклад security_sha , ми можемо досягти прискорення 1,21 порівняно з 0,78 у GCC. Насправді, якщо ми зосередимося на тестах, де доступне значне прискорення (прискорення >1,10), ми зможемо досягти 82% від максимуму. Таким чином, ми маємо техніку, яка в середньому забезпечує понад 75% від максимально доступного прискорення. Це досягається повністю автоматично, і це вигідно порівняно з 3%, досягнутими ручною евристичною евристикою GCC.

Перша і найважливіша функція обчислює кількість ітерацій циклу; зрозуміло, що немає сенсу розгортати цикл більше разів, ніж у нього є ітерацій. Решта функцій менш очевидні, і навряд чи їх вибере автор компілятора, демонструючи силу підходу. Функції відображають елементи, які привабливі для інтуїції, але, тим не менш, є складними, і важко точно пояснити, чому деякі елементи присутні. Це артефакт цільової функції, яка намагається, додаючи функцію, знайти найбільш корисну функцію для того, хто навчається, але не докладає жодних зусиль, щоб знайти функції, обґрунтування яких може зрозуміти людина. Для створення числових констант у цих функціях використовувалися семантичні дії.

Ми представили нову техніку для автоматичного генерування хороших функцій для оптимальної компіляції на основі машинного навчання.

Автоматично виводячи граматику функцій із внутрішнього представлення компілятора, ми можемо здійснювати пошук у просторі функцій за допомогою GP. Ми застосували цю загальну техніку для автоматичного вивчення хороших функцій для розгортання циклу в GCC. Наша техніка автоматично знаходить функції, здатні досягти в середньому 76% від максимально доступного прискорення, перевершуючи функції, які раніше створювалися вручну фахівцями з компіляції. Наш підхід зосереджений на представленні циклу RTL. Однак наша система є загальною, і її легко розширити, щоб охопити різні структури даних у будь-якому компіляторі. У майбутніх роботах досліджуватимуться різні простори функцій для нової оптимізації.

3.5 Висновки по розділу

Система поєднує еволюційний пошук ознак із машинним навчанням для автоматичної генерації корисних функцій, що покращують продуктивність компіляції. Оцінка якості функцій здійснюється за допомогою прискорення виконання програм і перехресної перевірки, з урахуванням простоти виразів.

У розділі описано, як граматика ознак використовується для автоматизації створення функцій, що аналізують програмний код, замінюючи трудомісткий ручний процес проєктування таких функцій. Використання семантичних дій дозволяє розширити можливості контекстно-вільних граматик і виражати складні ознаки, недоступні через синтаксис лише CFG.

Генерування характеристик із граматики відбувається шляхом послідовного розгортання нетерміналів випадковим вибором виробництв за їхніми вагами, що створює функцію після заміни всіх нетерміналів. Для запобігання нескінченній рекурсії та вибуховому зростанню речень застосовують вагові коефіцієнти, які зміщують ймовірності вибору, а також використовують структуру дерев вибору, що дозволяє ефективно модифікувати та відтворювати згенеровані функції під час пошуку.

ВИСНОВКИ

Дослідження, проведене в рамках цієї роботи, підкреслює значний потенціал використання машинного навчання для автоматизації процесів генерації коду та його оптимізації, що є ключовим кроком до підвищення ефективності розробки програмного забезпечення. Аналіз моделей представлення мовних структур, зокрема предметно-орієнтованих мовних моделей, граматичних моделей і моделей глибокого навчання для моделювання послідовностей, показав, що ці підходи забезпечують надійну основу для створення семантично коректного та синтаксично правильного коду. Предметно-орієнтовані моделі виявилися ефективними для специфічних доменів, тоді як моделі глибокого навчання, такі як трансформери, продемонстрували високу здатність до узагальнення та обробки складних мовних послідовностей, що є критично важливим для генерації коду.

Розгляд підходів до автоматичної генерації програм виявив, що сучасні технології генерації коду, синтез генераторів із прикладів та вдосконалене представлення мов програмного забезпечення дозволяють значно скоротити час розробки та мінімізувати помилки. Технології, такі як синтез коду з прикладами, дають змогу створювати генератори, які адаптуються до потреб користувачів, тоді як представлення мов програмного забезпечення на основі абстрактних синтаксичних дерев і нейронних мереж сприяють підвищенню якості згенерованого коду. Ці методи, підтримані машинним навчанням, забезпечують не лише автоматизацію, але й можливість оптимізації коду з точки зору продуктивності та читабельності.

Дослідження методів побудови ознак і навчання моделей підкреслило важливість генерації даних, визначення характерного простору та створення характеристик із граматик для ефективного навчання моделей машинного навчання. Генерація даних із використанням синтетичних прикладів і аугментації виявилася дієвим способом підвищення якості моделей, особливо в умовах обмеженої кількості реальних даних. Визначення характерного простору

дозволило ідентифікувати ключові ознаки, що впливають на генерацію коду, тоді як методи генерації характеристик із граматик забезпечили структуроване представлення мовних конструкцій. Експериментальна методика, застосована в роботі, підтвердила, що комбінація цих підходів сприяє створенню моделей, які здатні генерувати оптимізований код із високим рівнем точності та адаптивності.

Узагальнюючи, машинне навчання відкриває нові перспективи для автоматизації генерації коду та його оптимізації, дозволяючи створювати програмне забезпечення швидше, ефективніше та з меншою кількістю помилок. Перспективи подальших досліджень включають інтеграцію квантових обчислень для прискорення обробки великих масивів даних, удосконалення моделей із підтримкою мультимодального навчання для обробки комбінацій тексту, коду та специфікацій, а також розробку адаптивних систем, які можуть динамічно оптимізувати код залежно від апаратних обмежень. Ці напрямки сприятимуть розвитку інтелектуальних систем розробки програмного забезпечення, здатних відповідати викликам сучасної індустрії інформаційних технологій.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Gulwani S., Polozov O., Singh R. *Program Synthesis.* — Foundations and Trends in Programming Languages, 2017. — Vol. 4, No. 1-2. — С. 1–119. — Режим доступу: https://doi.org/10.1561/2500000010

2. Devlin J., Chang M.-W., Lee K., Toutanova K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* — Proceedings of NAACL-HLT, 2019. — С. 4171–4186. — Режим доступу: https://arxiv.org/abs/1810.04805

3. Chen M., Tworek J., Jun H., et al. *Evaluating Large Language Models Trained on Code.* — arXiv preprint, 2021. — 26 с. — Режим доступу: https://arxiv.org/abs/2107.03374

4. Brown T. B., Mann B., Ryder N., et al. *Language Models are Few-Shot Learners.* — Advances in Neural Information Processing Systems, 2020. — Vol. 33. — С. 1877–1901. — Режим доступу: https://papers.nips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

5. Allamanis M., Barr E. T., Devanbu P., Sutton C. *A Survey of Machine Learning for Big Code and Naturalness.* — ACM Computing Surveys, 2018. — Vol. 51, No. 4. — С. 1–37. — Режим доступу: https://doi.org/10.1145/3212695

6. 1. Лавренчук С., Ілюшик Р. Дослідження технології обробки природної мови та машинного навчання при створенні chat-bot засобами Python. КОМП'ЮТЕРНОІНТЕГРОВАНІ ТЕХНОЛОГІЇ: ОСВІТА, НАУКА, ВИРОБНИЦТВО. 2019. № 37. С. 36–42. URL: https://doi.org/10.36910/6775-2524-0560-2019-37-6 (дата звернення: 20.11.2023).

7. Rabinovich M., Stern M., Klein D. *Abstract Syntax Networks for Code Generation and Semantic Parsing.* — Proceedings of the 55th Annual Meeting of the ACL, 2017. — С. 1139–1149. — Режим доступу: https://doi.org/10.18653/v1/P17-1105

8. Alon U., Zilberstein M., Levy O., Yahav E. *Code2vec: Learning Distributed Representations of Code.* — Proceedings of the ACM on Programming Languages,

2019. — Vol. 3, No. POPL. — С. 1–29. — Режим доступу: https://doi.org/10.1145/3290353

9. Vaswani A., Shazeer N., Parmar N., et al. *Attention is All You Need.* — Advances in Neural Information Processing Systems, 2017. — Vol. 30. — С. 5998–6008. — Режим доступу: https://papers.nips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

10. Li Y., Choi D., Chung J., et al. *Competition-Level Code Generation with AlphaCode.* — Science, 2022. — Vol. 378, No. 6624. — С. 1092–1097. — Режим доступу: https://doi.org/10.1126/science.abq1158

11. Наука про дані: машинне навчання та інтелектуальний аналіз даних – Електронний навчальний посібник / В. Б. Мокін, М. В. Дратований – Вінниця : ВНТУ, 2024. – 263 с. режим доступу https://pdf.lib.vntu.edu.ua/books/2024/Mokin_2024_263.pdf

12. 52. Marchenko D., Matvyeyeva K. Theoretical research of the technology of finishing cylinders with antifriction materials. Problems of tribology. 2021. Vol. 100, no. 2. P. 65– 70. URL: https://doi.org/10.31891/2079-1372-2021-100-2-65-70 (date of access: 20.11.2023).

13. Feng Z., Guo D., Tang D., et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages.* — Findings of EMNLP, 2020. — С. 1536–1547. — Режим доступу: https://doi.org/10.18653/v1/2020.findings-emnlp.139

14. Parr T. *The Definitive ANTLR 4 Reference.* — Pragmatic Bookshelf, 2013. — 328 с. — Режим доступу: https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/

15. Aujla GS Data Offloading in 5G-Enabled Software-Defined Vehicular Networks: A Stackelberg Game-Based Approach / GS Aujla // IEEE Commun. Mag.-vol. 55, no. 7.-July 2017.

16. Стефанів А. М. Методи обробки природної мови із використанням інформаційної технології Spark MLlib : master's thesis. 2018. URL:http://elartu.tntu.edu.ua/handle/lib/23766 (дата звернення: 20.11.2023).

17. Raychev V., Vechev M., Yahav E. *Code Completion with Statistical Language Models.* — Proceedings of the 35th ACM SIGPLAN Conference on PLDI, 2014. — С. 419–428. — Режим доступу: https://doi.org/10.1145/2594291.2594321

18. Hindle A., Barr E. T., Su Z., Gabel M., Devanbu P. *On the Naturalness of Software.* — Proceedings of the 34th ICSE, 2012. — С. 837–847. — Режим доступу: https://doi.org/10.1109/ICSE.2012.6227135

19. Austin J., Odena A., Nye M., et al. *Program Synthesis with Large Language Models.* — arXiv preprint, 2021. — 35 с. — Режим доступу: https://arxiv.org/abs/2108.07732

20. Chaudhary R. Network Service Chaining in Fog and Cloud Computing for the 5G Environment: Data Management and Security Challenges / R. Chaudhary, N.Kumar, S. Zeadally // IEEE Communications Magazine.- vol. 55, no. 11.-November, 2017.- P. 114-122.

21. Shin E. C., Polozov O., Song D. *Unsupervised Program Synthesis with Neural Program Synthesis.* — Proceedings of ICLR, 2021. — 12 с. — Режим доступу: https://openreview.net/pdf?id=7tW6MdnG4Z

22. Goodfellow I., Bengio Y., Courville A. *Deep Learning.* — MIT Press, 2016. — 800 с. — Режим доступу: https://www.deeplearningbook.org/

23. Murali V., Chaudhuri S., Jermaine C. *Bayesian Sketch Learning for Program Synthesis.* — Proceedings of ICML, 2018. — С. 3607–3616. — Режим доступу: http://proceedings.mlr.press/v80/murali18a.html

24. Z. Fisches. Neural Self-Supervised Models of Code. Masters thesis, ETH Zurich, 2020.

25. Saxena S. Machine Learning for Compilers and Architecture. 2021.

26. Google Research. MLGO: A Machine Learning Guided Compiler Optimizations Framework. 2021.

27. . A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A Survey on Compiler Autotuning using Machine Learning. CSUR, 51(5), 2018

28. Tufano M., Watson C., Bavota G., et al. *An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation.* — ACM

Transactions on Software Engineering and Methodology, 2020. — Vol. 29, No. 4. — С. 1–29. — Режим доступу: https://doi.org/10.1145/3412376

29. GitHub. *GitHub Copilot Documentation.* — 2023. — Режим доступу: https://docs.github.com/en/copilot

30. OpenAI. *Codex: A Model for Code Generation.* — 2021. — Режим доступу: https://openai.com/research/codex

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи бакалавра: " Використання машинного навчання для автоматизації процесів генерації коду та його оптимізації "

Обсяг пояснювальної записки: 60 аркушів

Дата закінчення дипломної роботи 10 червня 2025р.

Підпис студента _____