

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Розробка системи прогнозування часу виконання
машинного коду з використанням статистичного
аналізу та машинного навчання»**

Виконав студент групи ІПЗм-24-1
ШЕКЕТА Олександр Олексійович

Керівник роботи:
ЛОБАНЧИКОВА Надія Миколаївна

Рецензент:
БАКАЛЮК Тетяна Анатоліївна

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«18» вересня 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ШЕКЕТА Олександр Олексійович**

Керівник роботи: **ЛОБАНЧИКОВА Надія Миколаївна**

Тема роботи: **«Розробка системи прогнозування часу виконання машинного коду з використанням статистичного аналізу та машинного навчання»**,

затверджена Наказом закладу вищої освіти від **«18» вересня 2025 р. № 491/с**

Вихідні дані для роботи: **аналіз процесу виконання машинного коду; застосування методів статистичного аналізу та машинного навчання для оцінювання та прогнозування його часових характеристик і продуктивно-критичних ділянок.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Лобанчикова Н.М.	22.09.2025р.	22.09.2025р.
2	Лобанчикова Н.М.	01.10.2025р.	01.10.2025р.
3	Лобанчикова Н.М.	03.11.2025р.	03.11.2025р.

РЕФЕРАТ

Метою кваліфікаційної роботи є розробка системи прогнозування часу виконання машинного коду з використанням статистичного аналізу та машинного навчання.

Об'єктом дослідження є процес виконання машинного коду в сучасних обчислювальних системах.

Предметом дослідження є методи та алгоритми статистичного аналізу й машинного навчання, застосовані для прогнозування часу виконання та виявлення продуктивно-критичних ділянок машинного коду на основі його статичного аналізу.

Теоретична значущість роботи полягає у дослідженні сучасних підходів до оцінювання продуктивності програмного забезпечення, аналізі можливостей застосування методів машинного навчання для аналізу машинного коду, а також у формалізації представлення машинного коду у вигляді текстових ознак для задач прогнозування.

Практична цінність роботи полягає у розробці програмного прототипу системи прогнозування часу виконання машинного коду, який без його фактичного виконання забезпечує підготовку даних, навчання моделі машинного навчання та прогнозування результатів і може використовуватися як допоміжний інструмент статичного аналізу для раннього виявлення продуктивно-критичних ділянок і підвищення ефективності оптимізації програмного забезпечення.

Магістерська кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел, із 30 найменувань (4 сторінки), 3-х додатків (8 сторінок), 20 ілюстрацій по тексту, 4 лістингів коду. Повний обсяг роботи становить 77 сторінок, з них 59 сторінок основного тексту.

КЛЮЧОВІ СЛОВА: МАШИННИЙ КОД, ПРОФІЛЮВАННЯ, МАШИННЕ НАВЧАННЯ, ПРОДУКТИВНІСТЬ, ПРОГНОЗУВАННЯ.

ABSTRACT

The purpose of the qualification thesis is to develop a software system for predicting the execution time of machine code using statistical analysis and machine learning methods.

The object of the research is the process of machine code execution in modern computing systems.

The subject of the research is statistical analysis methods and machine learning algorithms applied to execution time prediction and identification of performance-critical sections of machine code based on its static analysis.

The theoretical significance of the work lies in the study of modern approaches to software performance evaluation, the analysis of the applicability of machine learning methods to machine code analysis, as well as the formalization of machine code representation in the form of textual features for prediction tasks.

The practical value of the work consists in the development of a software prototype for a machine code execution time prediction system that provides data preparation, machine learning model training, and prediction for new code fragments without the need for their actual execution. The proposed approach can be used as an auxiliary static analysis tool for early detection of potentially performance-critical code sections and for improving the efficiency of the software optimization process.

The master's qualification thesis consists of an introduction, three chapters, conclusions, a list of references containing 30 sources (4 pages), 3 appendices (8 pages), 20 illustrations, and 4 code listings. The total volume of the thesis is 77 pages, including 59 pages of the main text.

KEY WORDS: MACHINE CODE, PROFILING, MACHINE LEARNING, PERFORMANCE, PREDICTION.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1. СУЧАСНІ ПІДХОДИ ДО ОЦІНЮВАННЯ ПРОДУКТИВНОСТІ КОДУ ТА МЕТОДИ ЇХ АВТОМАТИЗАЦІЇ	10
1.1 ПРОБЛЕМАТИКА АНАЛІЗУ ТА ПРОГНОЗУВАННЯ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	10
1.2 АНАЛІЗ АНАЛОГІВ ПРОГРАМНОГО ПРОДУКТУ	15
1.3 ВИБІР ПРОГРАМНИХ ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ	23
1.4 ОГЛЯД НАЯВНИХ МЕТОДІВ ТА АЛГОРИТМІВ ПРОГНОЗУВАННЯ ДАНИХ.....	25
Висновки до першого розділу	29
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ ПРОГНОЗУВАННЯ ПРОДУКТИВНО-КРИТИЧНИХ ДІЛЯНОК МАШИННОГО КОДУ	33
2.1. ЗАГАЛЬНА АРХІТЕКТУРА СИСТЕМИ ТА СТРУКТУРА МОДЕЛІ	33
2.2. ЗБІР АПАРАТНИХ ЛІЧИЛЬНИКІВ ЗАСОБАМИ INSTRUMENTS ТА XTRACE.....	35
2.3. ФОРМАЛІЗАЦІЯ ОЗНАК І ПОБУДОВА МОДЕЛІ КЛАСИФІКАЦІЇ	38
2.4. ПРОГРАМНА РЕАЛІЗАЦІЯ КОНВЕЄРА ТА СЦЕНАРІЇ ЗАСТОСУВАННЯ	40
2.5. ОЦІНЮВАННЯ АДЕКВАТНОСТІ ТА ТОЧНОСТІ МОДЕЛІ	46
Висновки до другого розділу	48
РОЗДІЛ 3. ОПИС ПРОГРАМНОГО ПРОДУКТУ	52
3.1 Постановка задачі.....	52
3.2 Сценарій використання	56
Висновки до третього розділу	60
ВИСНОВКИ.....	64
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	66
ДОДАТКИ	70

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI	– (англ. artificial intelligence) – штучний інтелект
CPU	– (англ. central processing unit) – центральний процесор
ML	– (англ. machine learning) – машинне навчання
SVM	– (англ. support vector machine) – метод опорних векторів

ВСТУП

Актуальність теми. Сучасні програмні системи характеризуються зростанням складності та підвищеними вимогами до продуктивності, що істотно ускладнює процес їх оптимізації. У практиці розробки програмного забезпечення основним інструментом аналізу продуктивності залишається динамічне профілювання, яке потребує багаторазових запусків програм, формування спеціалізованих сценаріїв навантаження та подальшого ручного аналізу результатів. Такий підхід є ресурсомістким, залежним від середовища виконання і часто дозволяє виявити продуктивно-критичні ділянки лише на пізніх етапах життєвого циклу програмного забезпечення.

Особливо актуальною ця проблема є для низькорівневого, зокрема машинного, коду, де навіть незначні відмінності в послідовності інструкцій можуть призводити до суттєвих втрат продуктивності через особливості роботи процесора, кеш-пам'яті та інших апаратних компонентів. У зв'язку з цим перспективним напрямом є застосування методів машинного навчання для попереднього прогнозування характеристик виконання коду без необхідності його фактичного запуску.

Ідея такого підходу полягає у навчанні моделі на основі статистично оброблених даних профілювання, здатної за структурою машинного коду оцінювати ймовірність наявності продуктивно-критичних ділянок. Хоча формально тема роботи орієнтована на прогнозування часу виконання машинного коду, фактичний акцент дослідження зроблено на задачі виявлення потенційних «вузьких місць» продуктивності. Розробка інструменту, який дозволяє завчасно звернути увагу розробника на проблемні фрагменти коду, є актуальною як з наукової, так і з практичної точки зору.

Мета роботи – розробка системи прогнозування часу виконання машинного коду з використанням статистичного аналізу та машинного навчання.

Для досягнення поставленої мети необхідно вирішити такі основні завдання:

1. Проаналізувати сучасні підходи до оцінювання продуктивності програмного забезпечення.
2. Дослідити можливості застосування методів машинного навчання для аналізу машинного коду.
3. Обґрунтувати вибір інструментів профілювання та методів формування навчальної вибірки.
4. Розробити модель класифікації машинного коду на основі його текстового представлення.
5. Реалізувати прототип програмної системи, здатної прогнозувати продуктивно-критичні ділянки для нових фрагментів коду.

Об'єктом дослідження є процес виконання машинного коду в сучасних обчислювальних системах.

Предметом дослідження є методи та алгоритми машинного навчання, застосовані для виявлення продуктивно-критичних ділянок машинного коду.

Наукова новизна роботи полягає у застосуванні методів машинного навчання до задачі виявлення продуктивно-критичних ділянок машинного коду на основі аналізу послідовностей машинних інструкцій без необхідності динамічного виконання програм. Особливістю запропонованого підходу є інтерпретація машинного коду як текстового представлення з подальшим формуванням вектору ознак за допомогою TF-IDF n-грам та використання класифікаційної моделі для прогнозування потенційних «вузьких місць» продуктивності, що дозволяє зосередити увагу не на точному числовому оцінюванні часу виконання, а на ідентифікації критичних фрагментів коду на ранніх етапах розробки.

Практична цінність роботи полягає у розробці програмного прототипу системи статичного аналізу машинного коду, яка дозволяє прогнозувати наявність продуктивно-критичних ділянок без необхідності динамічного

виконання програм. Запропонований підхід може бути використаний у процесі розробки та оптимізації програмного забезпечення для попереднього виявлення потенційних «вузьких місць» продуктивності, що сприяє скороченню витрат часу та ресурсів на профілювання і підвищенню ефективності оптимізації коду.

РОЗДІЛ 1. СУЧАСНІ ПІДХОДИ ДО ОЦІНЮВАННЯ ПРОДУКТИВНОСТІ КОДУ ТА МЕТОДИ ЇХ АВТОМАТИЗАЦІЇ

1.1 Проблематика аналізу та прогнозування продуктивності програмного забезпечення

З розвитком індустрії програмного забезпечення продуктивність стає однією з ключових нефункціональних характеристик, що визначає конкурентоспроможність, економічну ефективність і практичну придатність цифрових продуктів. Користувацькі очікування щодо миттєвого відгуку інтерфейсів, стабільних затримок сервісів та передбачуваної поведінки систем під навантаженням зумовлюють необхідність системної роботи з продуктивністю на всіх етапах життєвого циклу. При цьому реальні програмні системи майже завжди містять фрагменти, які непропорційно впливають на загальну швидкодію та споживання ресурсів, тобто утворюють продуктивно-критичні ділянки, що виступають «вузькими місцями» [1]. У сучасній літературі такі ділянки часто розглядаються як «гарячі точки» (hotspots) – функції або блоки, які домінують у профілі виконання та визначають верхню межу оптимізації системи. Зокрема, у роботі [1] показано, що навіть без запуску програми можна статистично ідентифікувати потенційні hotspots шляхом статичного аналізу структури програми (наприклад, графів викликів) із застосуванням графових нейронних мереж.

Традиційним способом виявлення таких ділянок є динамічне профілювання. Воно передбачає виконання програми з вимірюванням часу, збором статистики подій або аналізом апаратних лічильників, після чого інженер оптимізує найповільніші ділянки на основі профілю [2]. Однак динамічні вимірювання мають низку обмежень, які знижують їх ефективність у реальних процесах розробки. По-перше, для профілювання потрібні репрезентативні сценарії навантаження та дані, що часто відсутні на ранніх етапах або складні у відтворенні. По-друге, інструментування та семплювання

створюють накладні витрати й можуть змінювати часову поведінку програми, а отже ускладнювати інтерпретацію результатів. По-третє, профілювання є залежним від апаратної та програмної конфігурації, тому результати можуть суттєво змінюватися при перенесенні на інші платформи або після оновлень середовища виконання [2]. Додатково слід враховувати, що навіть коли профіль коректно вказує на кандидата для оптимізації, внесені зміни не гарантують покращення: емпірично показано, що рефакторинг може як пришвидшувати, так і сповільнювати виконання, причому ефект не завжди передбачуваний без повторних вимірювань [2]. Отже, профілювання лишається необхідним, але дорогим і циклічним процесом, який важко масштабувати на ранніх стадіях розробки.

Особливо гостро ці проблеми проявляються для низькорівневого, зокрема машинного, коду. На рівні машинних інструкцій ефективність виконання значною мірою визначається мікроархітектурними явищами (кеш-пам'ять, прогнозування переходів, робота конвеєра, затримки доступу до пам'яті, конфлікти в підсистемі вибірки інструкцій тощо). Навіть невелика модифікація послідовності інструкцій або розміщення даних може призводити до різких змін кількості кеш-промахів, частоти хибних передбачень і загальної кількості активних циклів. Це ускладнює ручний пошук причин деградації продуктивності та підвищує вартість повторних вимірювань, особливо коли оптимізація виконується ітеративно й зачіпає багато варіантів коду.

У такому контексті практично значущою стає постановка задачі раннього виявлення потенційних продуктивно-критичних ділянок ще до фактичного запуску програми, тобто на основі аналізу самого коду. З методологічної точки зору йдеться про використання статичних характеристик (показників, що можуть бути отримані без виконання програми), щоб визначити, які фрагменти з високою ймовірністю проявляться як «вузькі місця» під навантаженням [3].

Для автоматизації статичного прогнозування доцільним є застосування методів машинного навчання. Алгоритми ML здатні навчатися на історичних даних профілювання, спостерігати приховані закономірності між характеристиками коду та проявами низької продуктивності й надалі переносити ці закономірності на нові фрагменти коду [4, 5, 6]. Це особливо важливо для сучасного ПЗ із високою складністю та великою кількістю конфігурацій: огляди й емпіричні дослідження показують, що класичні емпіричні методи доцільно підсилювати глибоким навчанням, моделями з урахуванням невизначеності та стратегіями поетапного навчання (divide-and-learn) [4, 5, 6]. Водночас важливо розуміти, що не всі проблеми продуктивності є «очевидними» на рівні простих правил: наприклад, нетривіальні патерни (вкладені цикли, рекурсивні виклики, особливості структури керування) можуть системно корелювати з уповільненням і бути придатними для завчасного виявлення в межах задач прогнозування складності виконання [3].

Ключовим кроком у такому підході є формування інформативного подання коду у вигляді ознак. У літературі відомі підходи, що використовують спеціалізовані performance-метрики вихідного коду (наприклад, характеристики циклів, використання операцій введення-виведення всередині циклів, синхронізацію тощо), які покращують прогнозування дефектів продуктивності [7]. Водночас для машинного коду інженерія ознак на рівні вихідних конструкцій є менш прямолінійною, оскільки необхідно враховувати специфіку інструкційної архітектури й ефекти компілятора. Тому в межах даного дослідження як основу подання машинного коду обрано «текстову» інтерпретацію послідовностей інструкцій. Доцільність розгляду коду як послідовності символів/токенів, придатної для автоматичного аналізу, демонструється у роботах, де програмний код трактують як текстовий об'єкт для виявлення структурних патернів і аномалій [8].

Для побудови ознак застосовується TF-IDF-подання (частота терміна – обернена частота документів) [9]: послідовності інструкцій або їх фрагменти

інтерпретуються як «терміни», а їх статистична важливість оцінюється в контексті всієї вибірки. Метод TF-IDF є класичним для інформаційного пошуку та векторної моделі подання текстів. У нашій задачі рідкісні, але показові комбінації інструкцій отримують вищу вагу, що дозволяє статистично «підсвітити» нетипові патерни машинних команд, потенційно пов'язані з апаратними затримками.

На основі сформованих ознак будується модель класифікації, здатна визначати, чи має фрагмент коду ознаки продуктивно-критичності та, за потреби, до якого типу проблем (наприклад, кеш-затримок, розгалужень або простоїв конвеєра) він ближчий. У межах роботи підхід має відповідати кільком ключовим вимогам: бути практично придатним і недорогим у застосуванні, тобто забезпечувати швидке отримання прогнозу без необхідності виконання програми; демонструвати стійкість за умов обмеженого обсягу даних, що є типовим для прототипних рішень; а також залишатися інтерпретованим для розробника, щоб інструмент не лише сигналізував про наявність ризику, а й підказував напрям оптимізації та формував довіру до наданих рекомендацій.

Порівняння основних етапів класичного профілювання та підходу зі статичним прогнозуванням продуктивності, а також схема перетворення машинного коду у векторне подання наведені на рис. 1.1 та 1.2 відповідно. У традиційному підході аналіз продуктивності базується на динамічному виконанні програми: після компіляції здійснюється запуск програми, далі проводиться профілювання зі збором часових і апаратних характеристик, після чого виконується аналіз результатів. Такий процес є ресурсомістким, залежним від середовища виконання та вимагає наявності репрезентативних сценаріїв навантаження. Запропонований підхід зі статичним прогнозуванням також починається з компіляції, однак замість запуску програми застосовується дизасемблювання бінарного коду. Отримана послідовність інструкцій інтерпретується як текст, розбивається на токени / n-грами та

перетворюється у TF-IDF-вектор, який відображає статистичну значущість локальних патернів машинних інструкцій. Сформоване векторне подання подається на вхід класифікатора (ML-моделі), що дозволяє виконати прогнозування наявності або типу продуктивно-критичних ділянок. Завершальним етапом, як і в класичному підході, є аналіз результатів, проте він ґрунтується не на фактичних вимірюваннях, а на статистичному прогнозі, що забезпечує можливість раннього аналізу продуктивності без виконання програми.

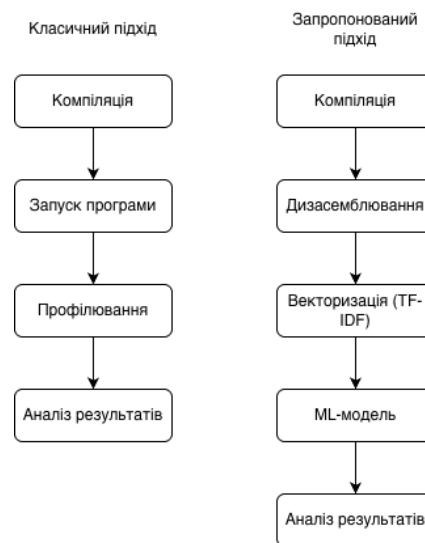


Рис. 1.1 – Порівняння традиційного профілювання та підходу зі статичним прогнозуванням продуктивності



Рис. 1.2 – Схема перетворення машинного коду у векторне подання для задач машинного навчання

Таким чином, актуальність дослідження визначається потребою у скороченні витрат на профілювання й оптимізацію, а також можливістю раннього виявлення потенційних «вузьких місць» на основі статичного аналізу машинних інструкцій із використанням методів статистичного аналізу та машинного навчання [3].

1.2 Аналіз аналогів програмного продукту

Аналіз аналогів програмного продукту доцільно проводити з урахуванням того, які саме задачі вирішує кожен інструмент, на якому рівні абстракції він функціонує та які результати формує для користувача. Для систем, орієнтованих на оцінювання продуктивності, порівняння зазвичай базується на характеристиках використовуваних джерел даних, рівні деталізації аналізу, вимогах до апаратного й програмного середовища, а також на формі подання результатів – від числових оцінок до локалізації проблемних ділянок коду. Не менш важливими є питання інтерпретованості результатів і можливості інтеграції інструмента в існуючі інженерні процеси, зокрема в CI, code review або системи статичного аналізу. При цьому слід зважати на практичний статус відповідних підходів: незважаючи на активний розвиток методів машинного навчання, класичне профілювання все ще залишається домінуючим в інженерній практиці, тоді як багато ML-рішень мають переважно дослідницький або вузькоспеціалізований характер [10].

Перший клас аналогів формують аналітичні моделі продуктивності, що спираються на теоретичні припущення щодо архітектури та алгоритмічної інтенсивності. До них належать моделі сімейства Roofline, які дозволяють оцінити, чи є обчислення обмеженими пропускнуою здатністю пам'яті або обчислювальними ресурсами. Принцип роботи Roofline-моделі полягає у перетворенні характеристик алгоритму та цільової архітектури у показник обчислювальної інтенсивності, після чого визначається теоретична межа досяжної продуктивності (див. рис. 1.3) [11]. Перевагою цього підходу є

наочність і можливість швидкої попередньої діагностики на рівні «режиму» роботи алгоритму. Обмеженням є необхідність експертного калібрування й те, що Roofline-моделі зазвичай не пояснюють локальні мікроархітектурні ефекти на рівні інструкцій і не дають прямої відповіді, які саме рядки коду створюють «вузьке місце» [11].



Рис. 1.3 – Узагальнена схема застосування Roofline-моделі для оцінювання теоретичної межі продуктивності алгоритму

Другий клас аналогів становлять статичні та напівстатичні підходи, що будують прогноз на основі ознак коду, контексту виконання або спрощених моделей. Для задач оцінювання часу виконання та енергоспоживання використовують підходи WCET/energy estimation, які останніми роками доповнюються ML-методами. Наприклад, WORTEX поєднує оцінювання часу та енергії з explainable ML-компонентами. Система виконує статичний аналіз машинного коду, після чого застосовує ML-модель для прогнозування найгіршого часу виконання та енергоспоживання програми (див. рис. 1.4) [12].



Рис. 1.4 – Схема статичного оцінювання найгіршого часу виконання та енергоспоживання програм у системі WORTEX

CAWET розвиває контекстно-орієнтовані оцінки, використовуючи моделі, здатні враховувати контекст базових блоків. Код програми подається у вигляді послідовності базових блоків, для яких формується контекстне представлення та застосовується трансформерна модель для оцінювання найгіршого часу виконання (див. рис. 1.5) [13]. Такі підходи демонструють практичну цінність раннього прогнозування, однак часто мають сильну прив’язку до домену, апаратної платформи та методики збирання даних, що ускладнює їх перенесення між різними проєктами.



Рис. 1.5 – Узагальнена схема контекстно-орієнтованого оцінювання найгіршого часу виконання в підході CAWET

Окремий напрям аналогів пов'язаний із прогнозуванням алгоритмічної складності й дефектів часу виконання. Роботи, присвячені runtime complexity prediction, демонструють, що статичні ознаки коду можуть бути достатніми. Вихідний код перетворюється у набір статичних ознак, які подаються на вхід ML-класифікатора для визначення потенційно проблемної алгоритмічної складності (див. рис. 1.6) [3].



Рис. 1.6 – Схема прогнозування алгоритмічної складності та потенційних дефектів часу виконання на основі статичних ознак коду

Прикладні системи для виявлення дефектів часу виконання (надмірної алгоритмічної складності) на основі ML показують, що автоматичний підхід може перевищувати можливості традиційного статичного аналізу та досягати високих показників точності (порядку 87%) [14, 15]. Важливо, що ці рішення зазвичай формулюють проблему як класифікацію або ранжування ризикових фрагментів, а не як детальне пояснення апаратних причин, тому їх корисність у задачах мікроархітектурної оптимізації може бути обмеженою.

Ще одна група аналогів охоплює прогнозування дефектів продуктивності на основі метрик коду та історичних даних. Підхід із використанням performance-метрик дозволяє підвищувати точність прогнозування performance bugs. Статичні метрики коду разом з історичними

даними подаються на вхід ансамблевої ML-моделі, яка прогнозує ймовірність виникнення дефектів продуктивності (див. рис. 1.7) [7].



Рис. 1.7 – Схема прогнозування дефектів продуктивності на основі статичних метрик коду та історичних даних

Практичний інтерес до таких інструментів підтверджується індустріальними кейсами: наприклад, демонструвалося використання ансамблевих моделей (Random Forest, XGBoost) для передбачення «вузьких місць» за статичними метриками та вимірюваннями продуктивності у версіях коду [16]. Подібні задачі розглядаються і в академічній площині як прогнозування впливу змін коду (комітів) на продуктивність [17]. У такій постановці рішення потенційно можуть бути інтегровані в CI/CD: модель сигналізує, що конкретна зміна має високий ризик викликати деградацію продуктивності, фокусуючи перевірки на «підозрілих» оновленнях [17]. Також досліджується прогнозування нестабільності бенчмарків на основі статичних ознак, де Random Forest демонструє конкурентні показники (AUC 0.79–0.90) [10]. Разом із тим, зростання складності моделей часто призводить до зниження прозорості рішень, що може ускладнювати практичну інтерпретацію результатів у команді розробки [10].

Окремо слід виділити аналоги, що працюють із більш «структурним» або «низькорівневим» поданням коду. Так, Hotspot Spy будує граф викликів та застосовує графові нейронні мережі. На основі статично побудованого графа

викликів GNN-модель визначає функції, які з високою ймовірністю стануть продуктивно-критичними (див. рис. 1.8) [1].

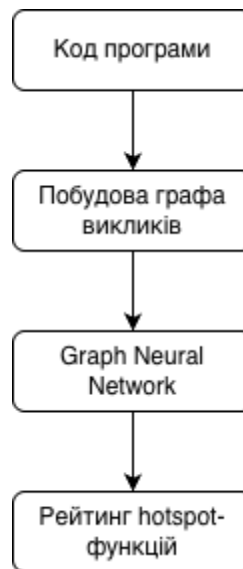


Рис. 1.8 – Схема статичної ідентифікації продуктивно-критичних функцій із використанням графових нейронних мереж (Hotspot Spy)

Інша лінія робіт орієнтована на прогнозування часу виконання після трансформацій коду: MovePerf поєднує глибоке навчання з механізмами взаємодії ознак. Модель аналізує зміни у структурі коду після рефакторингу та прогнозує їх вплив на час виконання програми (див. рис. 1.9) [18].



Рис. 1.9 – Схема прогнозування впливу рефакторингу на час виконання програми в підході MovePerf

Також активно досліджуються трансформерні моделі для прогнозування часу виконання. Послідовності базових блоків подаються на вхід трансформерної моделі, яка прогнозує час виконання кожного блоку з урахуванням контексту (див. рис. 1.10) [19].

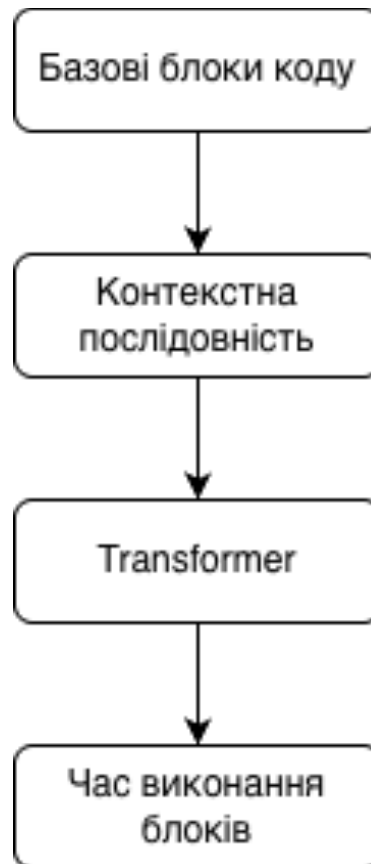


Рис. 1.10 – Схема контекстно-залежного прогнозування часу виконання базових блоків із використанням трансформерної моделі

Ці підходи підтверджують потенціал високоточного прогнозування, але, як правило, потребують значних датасетів і складнішої інфраструктури навчання [19].

Окремий практично важливий клас аналогів – гібридні системи, що поєднують інструментальні/симуляційні вимірювання з ML. Наприклад, у фреймворку PrETi запропоновано поєднати симуляцію з навчанням моделі.

Проміжне подання коду аналізується на рівні LLVM, після чого сформовані ознаки використовуються ML-моделлю для прогнозування часу виконання (див. рис. 1.11) [20].



Рис. 1.11 – Узагальнена схема гібридного прогнозування часу виконання програм на основі аналізу LLVM та машинного навчання (PrETi)

Такий підхід демонструє, що комбінація «низькорівневого» представлення коду та ML може давати конкурентну точність і водночас зменшувати потребу в дорогому повномасштабному профілюванні.

Паралельно розвиваються ML-підходи для ширшого спектра задач програмної інженерії (виявлення багів, дефектів, уразливостей, прогнозування властивостей тестування тощо), які формують методологічну базу для автоматичного аналізу коду [14]. Огляди щодо автоматичного виявлення вразливостей і дефектів підкреслюють важливість коректної постановки задачі, репрезентативності датасетів та метрик оцінювання якості моделей [21]. Хоча ці роботи не завжди націлені саме на продуктивність, вони корисні для обґрунтування підходів до побудови ознак і практик валідації, що є релевантним і для задач прогнозування bottleneck-ів.

Узагальнюючи аналіз, можна зробити висновок, що більшість існуючих рішень орієнтуються або на динамічні вимірювання, або на прогнозування на рівні вихідного коду та високорівневих метрик. Потреба у прогнозуванні продуктивно-критичних ділянок саме на рівні машинних інструкцій із прив'язкою до типів мікроархітектурних «вузьких місць» формує нішу для розроблюваного прототипу, який поєднує статистичне ознакове подання машинного коду (TF-IDF) з ML-класифікацією та може бути використаний як додатковий інструмент раннього аналізу поряд із традиційним профілюванням.

1.3 Вибір програмних засобів для реалізації

Вибір програмних засобів визначався сукупністю вимог, що висувалися до прототипу системи. Ключовими серед них були можливість швидкої розробки та ітеративного експериментування з різними ознаками й моделями, наявність відтворюваного конвеєра підготовки даних, доступ до інструментів компіляції та дизасемблювання, а також автоматизований збір профільних даних, які могли використовуватися як мітки для навчання моделей машинного навчання. Відповідно, обраний набір інструментів мав забезпечувати підтримку всіх етапів процесу – від генерації бінарного коду та отримання асемблерного подання до збирання профілю виконання і формування датасету для подальшого аналізу.

Основною мовою реалізації обрано Python. Це обґрунтовується наявністю розвиненої екосистеми для машинного навчання, обробки даних і автоматизації експериментів. Для ML-компонентів використовується бібліотека `scikit-learn` [22], яка надає готові засоби для векторизації тексту, навчання класифікаторів, побудови конвеєрів, перехресної валідації та оцінювання метрик якості. Такий вибір дозволяє уніфіковано порівнювати класичні моделі (наприклад, SVM, Random Forest, логістична регресія) на спільних даних і швидко змінювати конфігурації експериментів без значних

витрат на інфраструктуру [22]. З практичної точки зору це критично для прототипування: швидке тестування гіпотез щодо ознак і моделей дозволяє раніше перейти від «ідеї» до емпірично перевіреного конвеєра.

Для отримання репрезентативного машинного коду потрібний керований компіляторний ланцюжок. Компіляція тестових програм здійснюється компілятором clang [23], що забезпечує відтворюваність генерації бінарних файлів і можливість контролю параметрів компіляції. Дизасемблювання виконується утилітою llvm-objdump [24], яка формує текстове представлення асемблера з адресами інструкцій і придатна для подальшого парсингу. З практичної точки зору саме пара «clang + llvm-objdump» дозволяє побудувати стабільний конвеєр: вихідний C-код перетворюється на бінарний артефакт, а потім – на стандартизований текстовий список інструкцій, що використовується як вхід для формування контекстних вікон і текстових ознак. Окрім відтворюваності, така зв'язка важлива тим, що дає змогу працювати на рівні інструкцій і досліджувати, як конкретні патерни машинного коду пов'язані з апаратними подіями.

Ключовою вимогою для навчання є наявність цільових міток, що відображають прояви продуктивності. Для їх отримання застосовується інструментарій macOS: Xcode Instruments як середовище профілювання та збору апаратних лічильників [25] і консольна утиліта xctrace для автоматизованого запису та експорту трас без графічного інтерфейсу [26]. Поєднання Instruments і xctrace дозволяє, з одного боку, налаштувати необхідні події та параметри семплювання, а з іншого – запускати профілювання в пакетному режимі зі сценаріїв Python, що є критичним для формування навчального набору з великої кількості тестів. Додатково врахування змін у поведінці та сумісності інструментів профілювання (зокрема, за релізними нотатками) знижує ризик «технічної нестабільності» конвеєра під час експериментів [26].

Для подання машинного коду у вигляді ознак використовується підхід на основі TF-IDF, який широко застосовується в задачах текстової класифікації. На практиці це реалізується як конвеєр нормалізації асемблерних рядків (зменшення впливу адрес і констант) та перетворення тексту у розріджені вектори. Для верифікації практичних аспектів формули/реалізації TF-IDF та доступних функцій обчислення також можуть використовуватися довідкові онлайн-ресурси, що документують стандартні реалізації метрики [9]. У результаті формується високорозмірний розріджений простір ознак, який є зручним для експериментів з лінійними та іншими класичними моделями в середовищі scikit-learn [22].

Таким чином, обрана комбінація засобів (Python + scikit-learn + clang/llvm-objdump + Xcode Instruments/xctrace) забезпечує покриття повного циклу прототипу: автоматизоване формування даних, перетворення машинного коду у форму, придатну для ML, та подальше навчання і застосування моделі в сценаріях статичного аналізу.

1.4 Огляд наявних методів та алгоритмів прогнозування даних

У задачах прогнозування характеристик виконання програмного коду та пошуку потенційних «вузьких місць» використовується різноманітний набір підходів, які відрізняються як способом формування ознак, так і типом застосовуваних моделей. Для впорядкованого огляду ці підходи доцільно групувати за їх концептуальною основою: від простих евристик і правил, через методи, що спираються на статичні метрики коду або його лексичні подання, до більш складних структурних моделей, побудованих на графах, а також сучасних рішень на основі глибокого навчання й трансформерних архітектур. Окрему групу становлять методи, які додатково враховують конфігураційну змінність програм і невизначеність отримуваних прогнозів.

Евристичні підходи та правила, які реалізуються в інструментах статичного аналізу та/або вбудовуються в IDE, зазвичай ефективні для

«очевидних» патернів (наприклад, надмірна вкладеність циклів або потенційно блокуючі виклики). Однак такі підходи обмежені набором наперед відомих шаблонів і, як правило, гірше працюють із прихованими або нетривіальними причинами деградації продуктивності. Саме тому сучасні дослідження підкреслюють перспективність поєднання статичного аналізу з ML, який може навчатися на даних і виявляти складні закономірності, недоступні простим правилам [4, 5, 6].

Підходи на основі статичних метрик та класичних ML-моделей є одними з найпоширеніших у практичних постановках, оскільки вони забезпечують прозорість ознак і порівняно невисокі вимоги до обсягу даних. Зокрема, показано, що введення спеціалізованих performance-метрик коду може підвищувати точність прогнозування дефектів продуктивності [7]. Як базові лінії часто використовують інтерпретовані моделі на кшталт логістичної регресії, яка розглядається як стабільний еталон для порівняння [27]. Для підвищення точності застосовують ансамблеві методи (Random Forest, градієнтний бустинг), здатні моделювати нелінійні залежності. Наприклад, Random Forest демонстрував високі показники в задачі прогнозування нестабільності програмних бенчмарків на основі статичних ознак (AUC 0.79–0.90) [10]. Водночас у прикладних сценаріях важливо враховувати, що зростання складності моделі може знижувати інтерпретованість і ускладнювати пояснення причин прогнозу для розробника [10]. Це узгоджується і з емпіричними спостереженнями, що навіть корисні з точки зору якості моделі прогнози потребують коректної інтерпретації та повторної валідації на вимірюваннях, зважаючи на потенційно непередбачуваний ефект змін коду на швидкодію [2].

Окремий клас задач стосується виявлення дефектів часу виконання та проблемної алгоритмічної складності, які є частою причиною деградації продуктивності. Для таких постановок ML-підходи показують високу ефективність: продемонстровано, що комбінування моделей дозволяє

досягати близько 87% точності у класифікації фрагментів із потенційними проблемами складності [14]. Подібні результати узгоджуються з роботами щодо runtime complexity prediction [3] та ширшими спостереженнями про застосування методів глибокого навчання в задачах прогнозування дефектів програмного забезпечення [15]. На рівні методології ці роботи також підкреслюють значущість якості датасетів і дизайну експериментів, що узгоджується з рекомендаціями оглядових праць з автоматичного виявлення дефектів і вразливостей [21].

Для аналізу машинного або асемблерного коду особливо релевантні підходи, які трактують код як текст і використовують n-грамні або TF-IDF-подання. Такий спосіб дає змогу формалізувати локальні патерни інструкцій і застосувати до них алгоритми текстової класифікації. TF-IDF як складова векторної моделі подання текстів має класичне теоретичне обґрунтування, а практичні реалізації та формули наведено в прикладних довідкових ресурсах [9]. У задачах, де ознаки є високорозмірними та розрідженими, лінійні моделі часто забезпечують сильну базову якість при помірній вартості навчання та застосування. З практичної точки зору це важливо для прототипів і інструментів статичного аналізу, які мають працювати швидко та бути відносно стабільними при обмеженій кількості даних. Наявність успішних кейсів застосування «текстового» трактування коду підтверджує доцільність такого подання для задач аналізу якості реалізації демонструється й у роботах із виявлення аномалій/дефектів коду [8].

Для локалізації «гарячих точок» продуктивності активно застосовуються графові нейронні мережі: Hotspot Spy використовує статичний аналіз на графі програми для ідентифікації функцій, які обмежують продуктивність [1]. Такі моделі потенційно дають вищу якість за рахунок урахування структури програми, але потребують складнішого конвеєра підготовки даних і, як правило, більшого обсягу навчальної вибірки.

Подальший прогрес у глибокому навчанні сформував напрям моделей, які зменшують потребу в ручній інженерії ознак і автоматично вилучають складні закономірності з коду. Прикладом є MovePerf, що поєднує глибоку нейронну мережу та факторизаційні машини для прогнозування часу виконання після рефакторингу й демонструє середню похибку менше 8% [18]. Сучасний напрям – трансформерні архітектури для контекстно-залежного прогнозування часу виконання базових блоків коду та подібних задач [19]. Перевага трансформерів полягає у здатності моделювати довгі залежності та контекст, однак практичне застосування таких моделей часто потребує суттєвих обчислювальних ресурсів і масштабніших датасетів, щоб забезпечити стабільні результати [19]. Аналогічні ідеї контекстності та поєднання з оцінками часу/енергії простежуються й у WCET-напрямі (зокрема, у контекстно-орієнтованих підходах) [13] та explainable ML-рішеннях [12].

Окремо слід виділити підходи, що враховують конфігураційну змінність і невизначеність прогнозів. Для висококонфігурованих систем розвиваються огляди, таксономії та методи навчання моделей продуктивності з урахуванням простору конфігурацій [4], а також стратегії «divide-and-learn» для покращення якості прогнозування продуктивності [6]. Для коректнішої роботи з невизначеністю прогнозів застосовуються Bayesian neural networks, які надають не лише точкову оцінку, а й оцінку довіри до прогнозу [5]. Для практичних інструментів статичного аналізу це важливо, оскільки дозволяє не лише позначати підозрілі фрагменти, а й ранжувати їх за ступенем впевненості моделі.

З прикладної точки зору важливим є й те, як прогнозування вбудовується в інженерний процес. Індустріальні кейси демонструють використання ансамблевих моделей для передбачення performance-ризиків на основі статичних метрик і історичних вимірювань [16], а академічні роботи пропонують підходи до класифікації комітів за ризиком внесення регресій

продуктивності з потенціалом інтеграції в CI/CD [17]. Додатково перспективними є гібридні системи раннього прогнозування, що поєднують низькорівневий аналіз/симуляцію з ML (наприклад, на рівні LLVM), демонструючи практичну здійсненість такого підходу [20].

Підсумовуючи огляд, можна зазначити, що вибір алгоритму має враховувати баланс між точністю, інтерпретованістю та вартістю застосування. В умовах обмеженого обсягу даних і потреби у швидкому прогнозуванні на розріджених «текстових» ознаках машинного коду раціонально починати з інтерпретованих лінійних моделей як базового рішення, тоді як глибокі архітектури [18, 19] і графові підходи [1] доцільно розглядати як наступний крок після накопичення достатньої кількості даних та підтвердження практичної корисності прототипу.

Висновки до першого розділу

У першому розділі магістерської кваліфікаційної роботи виконано ґрунтовний аналіз сучасних підходів до оцінювання продуктивності програмного забезпечення, а також розглянуто методи автоматизації виявлення продуктивно-критичних ділянок коду. Основну увагу приділено як класичним інженерним підходам, що використовуються у промисловій практиці, так і сучасним науковим дослідженням, спрямованим на застосування методів машинного навчання для аналізу та прогнозування характеристик виконання програм.

У ході аналізу встановлено, що продуктивність програмного забезпечення є однією з ключових нефункціональних характеристик, яка безпосередньо впливає на споживчі властивості програмних систем, їх масштабованість, стабільність і конкурентоспроможність. Зростання складності сучасних програмних продуктів, поява багатоядерних архітектур і гетерогенних обчислювальних платформ призводять до того, що традиційні методи оптимізації та інтуїтивний підхід до пошуку «вузьких місць» дедалі

частіше виявляються недостатніми. Це зумовлює потребу у системних та автоматизованих засобах аналізу продуктивності.

Проаналізовано класичні методи динамічного профілювання, які на сьогодні залишаються основним інструментом аналізу швидкодії програмного забезпечення. Показано, що профілювання дозволяє отримувати точні емпіричні дані про час виконання функцій, частоту викликів, завантаження процесора та використання апаратних ресурсів. Водночас виявлено низку принципових обмежень такого підходу. До них належать висока вартість проведення експериментів, необхідність підготовки репрезентативних сценаріїв навантаження, залежність результатів від конкретної апаратної та програмної конфігурації, а також складність застосування профілювання на ранніх етапах розробки, коли програмний продукт ще не є стабільним або повністю функціональним.

Особливої актуальності ці обмеження набувають у контексті аналізу продуктивності низькорівневого, зокрема машинного, коду. У першому розділі показано, що на рівні машинних інструкцій продуктивність значною мірою визначається мікроархітектурними особливостями процесора, такими як робота кеш-пам'яті, механізми прогнозування переходів, організація конвеєра виконання інструкцій та затримки доступу до пам'яті. Навіть незначні зміни у послідовності інструкцій або розміщенні даних можуть призводити до істотних змін часу виконання, що ускладнює ручний аналіз і робить процес оптимізації ітеративним та дорогим.

У результаті аналізу наукових публікацій зроблено висновок, що перспективним напрямом розвитку інструментів аналізу продуктивності є перехід від виключно динамічних методів до статичних або напівстатичних підходів, здатних прогнозувати потенційні продуктивно-критичні ділянки коду без фактичного виконання програми. У розділі розглянуто різні класи таких підходів, зокрема аналітичні моделі продуктивності, методи оцінювання

найгіршого часу виконання, а також рішення, що ґрунтуються на статистичному аналізі та машинному навчанні.

Окрему увагу приділено методам машинного навчання як інструменту автоматизації аналізу продуктивності. Показано, що ML-моделі здатні навчатися на історичних даних профілювання та виявляти приховані закономірності між структурою програмного коду та проявами низької продуктивності. Такі закономірності не завжди можуть бути формалізовані у вигляді простих правил або евристик, що обмежує ефективність традиційних статичних аналізаторів. У цьому контексті машинне навчання розглядається як засіб доповнення класичних підходів, а не їх повна заміна.

У першому розділі також проаналізовано існуючі підходи до формування ознак для задач прогнозування продуктивності. Встановлено, що значна частина робіт орієнтована на використання високорівневих статичних метрик вихідного коду або структурних подань програм, таких як графи викликів. Водночас для задач аналізу машинного коду такі підходи мають обмежену застосовність через втрату інформації про конкретні інструкції та їх взаємодію з апаратною архітектурою. Це зумовлює необхідність пошуку альтернативних способів представлення машинного коду, придатних для автоматизованого аналізу.

На основі огляду літератури обґрунтовано доцільність трактування машинного коду як текстового об'єкта з подальшим застосуванням методів обробки текстових даних. Показано, що використання символьних n-грам і TF-IDF-подання дозволяє формалізувати локальні патерни машинних інструкцій та використовувати добре відомі алгоритми текстової класифікації. Такий підхід має низку переваг, зокрема відносну простоту реалізації, універсальність і можливість застосування при обмеженому обсязі навчальних даних.

У результаті аналізу сучасних методів і алгоритмів прогнозування сформульовано основні вимоги до системи, що розробляється в межах даної

магістерської роботи. До таких вимог належать можливість статичного аналізу без виконання програми, здатність працювати з машинним кодом на рівні інструкцій, помірні обчислювальні витрати, інтерпретованість результатів для розробника та придатність до використання на ранніх етапах життєвого циклу програмного забезпечення.

Таким чином, у першому розділі сформовано теоретичну та методологічну основу дослідження, визначено актуальність обраного напрямку та обґрунтовано вибір підходу до прогнозування продуктивності машинного коду з використанням статистичного аналізу та машинного навчання. Отримані висновки стали підґрунтям для проєктування архітектури системи, вибору методів збору даних, формування ознак і алгоритмів класифікації, які реалізовано та досліджено у наступних розділах роботи.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ ПРОГНОЗУВАННЯ ПРОДУКТИВНО-КРИТИЧНИХ ДІЛЯНОК МАШИННОГО КОДУ

2.1. Загальна архітектура системи та структура моделі

Загальна архітектура розробленої системи включає кілька взаємопов'язаних підсистем. По-перше, це підсистема збору профільних даних на основі Xcode Instruments та утиліти xtrace, яка забезпечує вимірювання апаратних лічильників під час виконання тестових програм. По-друге, підсистема спрощення профілів і побудови «карти» bottleneck-адрес перетворює детальний XML-профіль на компактне подання у вигляді відповідності між адресами інструкцій та домінуючим типом апаратного «вузького місця». По-третє, підсистема синхронізації профілю з дизасемблерним кодом та формування навчальної вибірки забезпечує відображення цих адрес на конкретні інструкції та побудову контекстних вікон. По-четверте, підсистема побудови символічних ознак і навчання класифікаційної моделі реалізує перетворення послідовностей інструкцій у вектори ознак і навчає модель машинного навчання. Нарешті, підсистема застосування моделі до нового коду забезпечує аналіз довільного C-файлу та формування звітів, у тому числі з HTML-візуалізацією «вузьких місць».

На вході системи використовується набір C-програм, які розміщуються в каталозі tests. Для кожної такої програми реалізовано однаковий конвеєр. Спочатку вихідний файл компілюється у бінарний файл з використанням компілятора clang [23]; результати компіляції зберігаються в каталозі binaries. Далі отриманий бінарний файл дизасемблюється за допомогою llvm-objdump [24], унаслідок чого формується текстова асемблерна презентація в каталозі assemblies. Після цього бінарний файл профілюється інструментом CPU Counters через xtrace із використанням попередньо підготовленого шаблону Instruments; результати профілювання у вигляді .trace-файлів накопичуються в

каталозі traces. Наступним етапом іде експорт трас у формат XML і спрощення їх до компактного JSON-подання з виділеними bottleneck-адресами, що зберігаються у каталогах profiles та profiles_simplified. На основі спрощених профілів та асемблерного коду формується навчальна вибірка у вигляді контекстних вікон інструкцій з відповідними мітками типу bottleneck; сукупність цих даних записується у файл training_data.json. Потім на основі цього файлу відбувається навчання символічної TF-IDF + Linear SVM-моделі, артефакти якої (модель та метадані) зберігаються в каталозі model_artifacts. Нарешті, навчену модель застосовують до нового фрагмента коду за допомогою скрипта predict_bottlenecks.py, який генерує консольний звіт та, за потреби, HTML-сторінку з підсвіченими «вузькими місцями».

Як показано на рис. 2.1, каталоги tests, binaries, assemblies, traces, profiles, profiles_simplified та model_artifacts, а також сценарії generate_training_data.py, train_bottleneck_model.py і predict_bottlenecks.py утворюють єдину структуру проєкту, у якій кожен елемент відповідає певному кроку конвеєра від збору даних до застосування моделі.

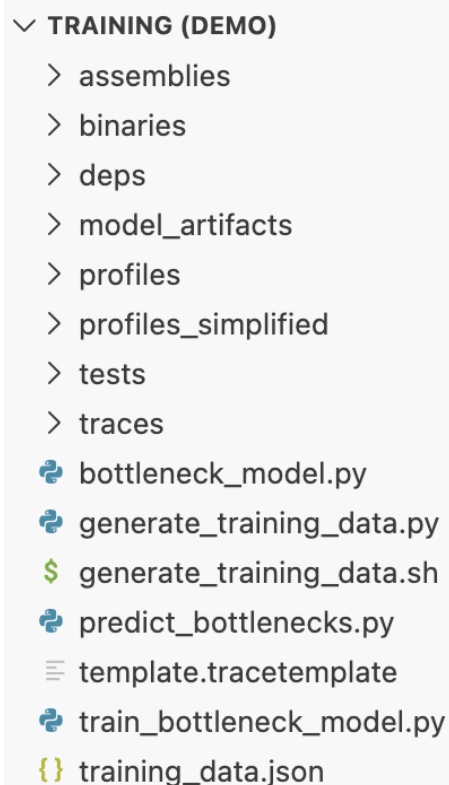


Рис. 2.1 – Структура каталогу проєкту з основними підсистемами

Таким чином, запропонована архітектура програмного комплексу реалізує повний цикл побудови моделі: від формування набору даних на основі реальних вимірювань до інтеграції навченої класифікаційної моделі в інструмент підтримки розробника. Іншими словами, система охоплює всі стадії – від збору та підготовки даних до практичного використання моделі для аналізу коду – в єдиному автоматизованому конвеєрі. Подібний наскрізний підхід реалізовано також у системі PrETi [20], що поєднує симуляцію виконання програм на рівні LLVM з методами ML для передбачення часу виконання на ранніх етапах розробки. Це підтверджує ефективність інтеграції статичного аналізу та машинного навчання для задач прогнозування продуктивності.

2.2. Збір апаратних лічильників засобами Instruments та xctrace

Ключовим етапом побудови моделі є отримання достовірних міток, які характеризують тип продуктивного «вузького місця» для конкретних адрес машинного коду. Для цього в роботі використано апаратні лічильники, що збираються інструментом CPU Counters у складі Xcode Instruments, і консольну утиліту xctrace, яка дозволяє запускати профілювання без залучення графічного інтерфейсу.

У межах магістерської роботи було створено користувацький шаблон Instruments у файлі `template.tracetemplate`. У цьому шаблоні як інструмент обрано CPU Counters, режим роботи встановлено Manual, вибірка здійснюється за подіями (Sample By = Events), базовою подією для семплювання є `CORE_ACTIVE_CYCLES`, а інтервал вибірки Sample Every дорівнює 1000. Додатково у шаблоні задано набір подій, за якими збираються апаратні лічильники: `L1D_CACHE_MISS_LD`, `L1D_CACHE_MISS_ST`, `BRANCH_MISPRED_NONSPEC`, `L1I_CACHE_MISS_DEMAND` та `MAP_STALL_DISPATCH`. Ці події відповідають промахам кешу даних

першого рівня при читанні та записі, помилковим передбаченням переходів, промахам кешу інструкцій і простоям конвеєра на етапі диспетчеризації.

Конфігурація шаблону дозволяє вважати, що для кожної адреси, яка часто трапляється у вибірках, можна визначити домінуючий тип апаратного bottleneck-а. На рис. 2.2 показано приклад налаштованого шаблону CPU Counters у середовищі Instruments: видно, що обрано режим Manual, семплювання за подіями CORE_ACTIVE_CYCLE та зазначено перелік апаратних лічильників, які необхідно збирати.

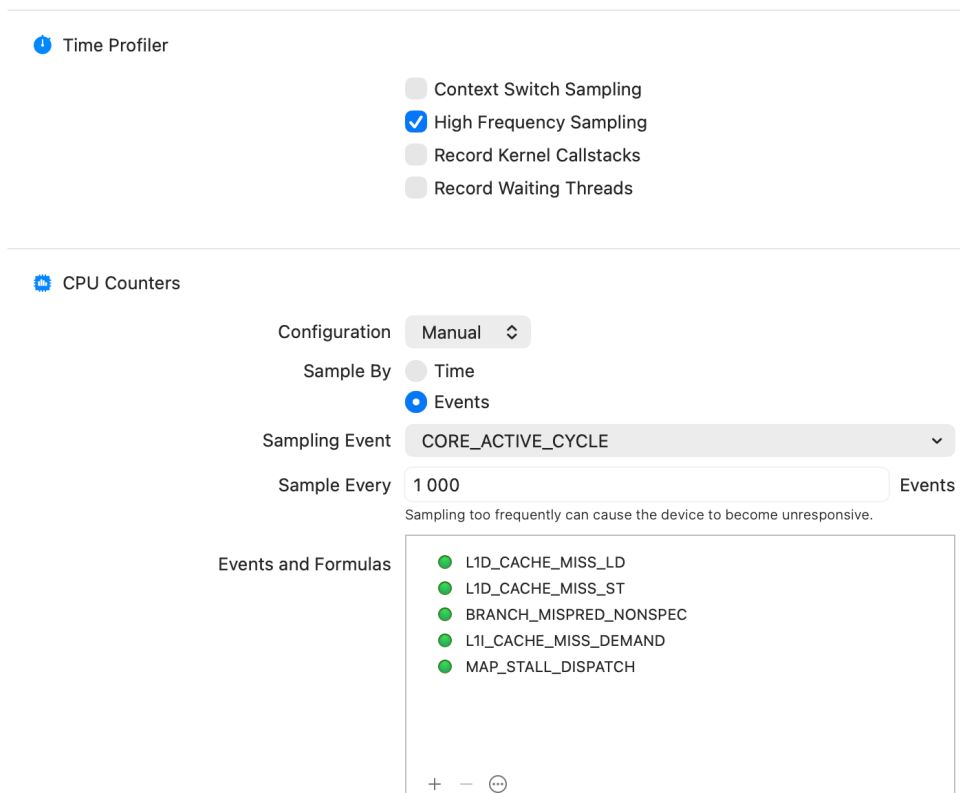


Рис. 2.2 – Вікно Xcode Instruments з відкритим шаблоном CPU Counters

Автоматизований запуск xctrace реалізовано в модулі `deps/trace_ops.py`. Ключовий фрагмент коду наведено у лістингу 2.1.

Лістинг 2.1 – Виклик `xctrace record` з використанням шаблону Instruments

```
def record_trace(binary: Path, xctrace: str, verbose: bool) -> Path:
    if not TEMPLATE.exists():
        sys.exit(f"Trace template not found: {TEMPLATE}")

    TRACES_DIR.mkdir(parents=True, exist_ok=True)
    output = TRACES_DIR / f"{binary.name}.trace"
```

```

if output.exists():
    output.unlink()

cmd = [
    xctrace,
    "record",
    "--template",
    str(TEMPLATE),
    "--output",
    str(output),
    "--launch",
    "--",
    str(binary),
]
run_command(cmd, verbose=verbose)
return output

```

Ця функція перевіряє наявність файлу-шаблону `template.tracetemplate`, формує шлях до вихідного `.trace`-файлу, будує команду `xctrace record`, яка запускає профілювання для переданого бінарного файлу, і виконує її за допомогою допоміжної функції `run_command`. У результаті для кожного бінарного файлу в каталозі `traces` створюється відповідний файл трасування.

Після завершення запису трас наступним кроком є експорт результатів у XML-формат за допомогою команди `xctrace export` з використанням XPath-виразів, що обирають потрібні таблиці з даними про події. Отримані XML-файли містять детальну інформацію про стек викликів, значення апаратних лічильників для кожної вибірки та шляхи до вихідних файлів. На рис. 2.3 наведено фрагмент XML-файлу профілю, у якому видно елементи `pmc-events` з переліком числових значень лічильників для вибраної адреси.

```

<row>
  <sample-time ref="11305" />
  <thread ref="2" />
  <process ref="4" />
  <core ref="123" />
  <thread-state ref="8" />
  <backtrace id="19096">
    <frame id="19097" name="main" addr="0x102e004b5">
      <binary ref="19087" />
      <source line="7">
        <path ref="19088" />
      </source>
    </frame>
    <frame ref="19089" />
  </backtrace>
  <pmc-event ref="13289" />
  <pmc-events id="19098" fmt="(29), (60), (14), (8), (277), (29), (60), (14), (8), (277)">29
    60 14 8 277 29 60 14 8 277</pmc-events>
</row>

```

Рис. 2.3 – Фрагмент XML-файлу профілю з елементами `pmc-events`, що містять значення апаратних лічильників для конкретних вибірок

Оскільки сировинний XML-формат є громіздким, у модулі `deps/profile.py` реалізовано алгоритм спрощення профілю, зосереджений у функції `simplify_profile`. Ця функція будує словники відповідностей між ідентифікаторами і шляхами до вихідних файлів та бінарників, знаходить рядки профілю, що відносяться до конкретного тестового файлу з каталогу `tests`, визначає адресу функції `main`, накопичує значення лічильників для кожної адреси та для кожної такої адреси вибирає подію з максимальним значенням як тип `bottleneck-a`. На виході формується компактна структура у форматі JSON, яка містить адресу `main` і словник `bottlenecks`; саме ця структура використовуються на наступних етапах для формування навчальної вибірки.

2.3. Формалізація ознак і побудова моделі класифікації

Після того, як для кожної адреси машинного коду визначено домінуючий тип `bottleneck-a`, необхідно побудувати навчальну вибірку у вигляді пар «контекст інструкцій – клас». Для цього в роботі реалізовано послідовність кроків. Спочатку парситься асемблерний файл у каталозі `assemblies`, визначається межа функції `main` та формується впорядкований список адрес усіх інструкцій цієї функції. Далі для кожного індексу i у списку адрес формується вікно контексту розміру $2k+1$ інструкцій, де k – розмір контексту; у реалізованій моделі k дорівнює 2, отже кожне вікно містить п'ять інструкцій: дві попередні, поточну та дві наступні. Якщо на початку чи в кінці функції бракує інструкцій, вікно симетрично доповнюється порожніми рядками.

Для кожного такого вікна визначається мітка класу. Якщо адреса поточної інструкції присутня у словнику `bottlenecks`, отриманому з профілю, вікну приписується відповідний тип події (наприклад, `L1D_CACHE_MISS_LD` або `BRANCH_MISPRED_NONSPEC`). Якщо адреса не позначена як проблемна, вікну приписується мітка `NONE`, яка означає відсутність домінуючого `bottleneck-a`. Сформований таким чином набір

прикладів зберігається у файлі `training_data.json`; кожен елемент містить список рядків `instructions` та рядок `type`.

Таким чином, середовище навчання можна описати як множину:

$$D = \{(x_i, y_i)\}_{i=1}^N,$$

де x_i – текстове представлення вікна інструкцій, а y_i належить до множини класів і відповідає типу апаратного bottleneck-а для центру вікна або відсутності такого у випадку класу NONE.

Наступним кроком є побудова текстового представлення для кожного вікна. У модулі `bottleneck_model.py` реалізовано функцію `normalize_instruction`, яка виконує нормалізацію окремої інструкції, та функцію `render_window`, яка послідовно застосовує нормалізацію до кожної інструкції у вікні й об'єднує їх в один рядок. Нормалізація уніфікує регістри (`x0`, `9` тощо замінюються на узагальнений маркер `xreg`, `w0`, `w1` тощо – на `wreg`), перетворює конкретні числові константи у маркери `#imm` або `0ximm`, приводить інструкції до нижнього регістру та видаляє зайві пробіли. У результаті модель фокусується не на конкретних номерах регістрів чи точних значеннях імедіатів, а на структурі та типах операцій.

Для векторизації текстових представлень використовується символний TF-IDF на n -грамах довжиною від чотирьох до семи символів. Формально кожному тексту x_i ставиться у відповідність вектор:

$$\varphi(x_i) = (tf_idf(g_1, x_i), \dots, tf_idf(g_M, x_i)),$$

де g_j є j -тою символною n -грамою із словника, а M – розмір цього словника. Значення `tf_idf` відображають, наскільки характерною є та чи інша n -грама для даного прикладу в контексті всього навчального набору: рідкісні, але інформативні фрагменти інструкцій отримують вищу вагу.

Як базовий алгоритм класифікації обрано лінійну опорну векторну машину з функцією втрат `hinge`, реалізовану у вигляді класу `LinearSVC` з балансуванням класів. Такий вибір обґрунтовується здатністю лінійних SVM працювати з високорозмірними розрідженими векторами, типовими для

TF-IDF-представлень, а також практичною ефективністю на задачах текстової класифікації та відносною інтерпретованістю (можливістю аналізувати ваги ознак для кожного класу).

Навчання моделі здійснюється у скрипті `train_bottleneck_model.py`. На першому кроці завантажується файл `training_data.json`, формуються списки текстів та міток і перевіряється узгодженість розміру контексту між даними та налаштуваннями моделі. Далі створюється конвеєр, який складається з двох основних компонентів: `TfidfVectorizer` [28], що перетворює текстові вікна в розріджені вектори ознак, та `LinearSVC` [29], який навчається розрізняти класи на основі цих ознак. За потреби виконується перехресна валідація для оцінки макро-F1-міри; після цього модель навчається на всій вибірці, обчислюється класифікаційний звіт з `precision`, `recall` і F1-мірою для кожного класу, і, нарешті, модель разом із метаданими зберігається до каталогу `model_artifacts`.

2.4. Програмна реалізація конвеєра та сценарії застосування

Для того щоб побудована модель була придатною до використання у реальному процесі розробки, у межах магістерської роботи реалізовано набір сценаріїв і модулів, які автоматизують усі ключові етапи конвеєра. До основних елементів програмної реалізації належать скрипт `generate_training_data.py` для формування навчальних даних, пакет `deps` з модулями для компіляції, пошуку тестів, профілювання, обробки профілів та формування вибірки, скрипт `train_bottleneck_model.py` для навчання моделі та скрипт `predict_bottlenecks.py` для застосування моделі до нового коду та побудови HTML-звіту.

Конвеєр формування навчальних даних у `generate_training_data.py` можна описати на основі фрагмента коду, поданого в лістингу 2.2.

Лістинг 2.2 – Основний цикл формування `training_data.json`

```
def main() -> None:
    args = parse_args()
    ensure_tools_available(args.clang, args.xctrace, shlex.split(args.objdump)[0])
```

```

tests = resolve_tests(args.tests_override)
if not tests:
    sys.exit("No tests found to process.")

if not TEMPLATE.exists():
    sys.exit(f"Trace template not found: {TEMPLATE}")

for src in tests:
    print(f"Processing {src.name}")
    binary = compile_source(src, args.clang, args.verbose)
    asm_path = emit_assembly(binary, args.objdump, args.verbose)
    print(f"  binary:      {binary}")
    print(f"  assembly:    {asm_path}")

    trace_path = record_trace(binary, args.xctrace, args.verbose)
    profile_path = export_trace(trace_path, args.xctrace, args.verbose)
    print(f"  trace:       {trace_path}")
    print(f"  profile:     {profile_path}")

    summary = simplify_profile(profile_path)
    if summary:
        simplified_dir = ROOT / "profiles_simplified"
        simplified_dir.mkdir(parents=True, exist_ok=True)
        out_path = simplified_dir / f"{profile_path.stem}.json"
        with out_path.open("w", encoding="utf-8") as f:
            json.dump(summary, f, indent=2, sort_keys=True)
        print(f"  simplified: {out_path}")

combined = build_training_data(args.context)
write_training_data(combined)
print(f"Wrote combined training data with {len(combined)} samples")

```

Цей сценарій автоматично знаходить усі C-файли в каталозі tests, для кожного з них послідовно виконує компіляцію у бінарний файл, вивантаження дизасемблерного коду, профілювання з використанням шаблону Instruments, експорт трас у формат XML та спрощення профілів до JSON-представлення. Після обробки всіх тестів функції build_training_data та write_training_data формують єдиний файл training_data.json, який містить вибірку контекстних вікон інструкцій із відповідними мітками типу bottleneck.

Окремий сценарій train_bottleneck_model.py реалізує навчання моделі на сформованих даних. Основна логіка зосереджена у функції main, де послідовно виконуються завантаження вибірки, побудова конвеєра «ознаки → класифікатор», опційна перехресна перевірка, остаточне навчання,

обчислення метрик та збереження моделі. Узагальнений фрагмент коду наведено в лістингу 2.3.

Лістинг 2.3 – Основні кроки навчання моделі у `train_bottleneck_model.py`

```
texts, labels, context_size = load_training_samples(training_path,
context=args.context)

pipeline = make_pipeline()
if not args.skip_cv:
    scores = cross_val_score(pipeline, texts, labels, cv=5, scoring="f1_macro")
    print(f"5-fold F1_macro mean={scores.mean():.3f}, std={scores.std():.3f}")

pipeline.fit(texts, labels)
preds = pipeline.predict(texts)
class_order = list(pipeline.named_steps["classifier"].classes_)
report = classification_report(labels, preds, labels=class_order,
target_names=class_order, digits=3)
print(report)

save_model(pipeline, context_size, model_path=model_out)
print(f"Saved model to {model_out} (context={context_size})")
```

На початку викликається функція `load_training_samples`, яка читає `training_data.json`, перевіряє розмір контексту та формує списки текстових представлень вікон інструкцій (`texts`) і відповідних міток типу `bottleneck` (`labels`). Далі функція `make_pipeline` створює конвеєр із двох основних блоків: `TfidfVectorizer`, що перетворює тексти у символічні TF-IDF n-грамні вектори, та `LinearSVC` з балансуванням класів, який виконує багатокласову класифікацію. Якщо прапорець `--skip-cv` не задано, виконується п'ятикратна перехресна перевірка з обчисленням макро-F1-міри (`f1_macro`), що дає первинну оцінку здатності моделі до узагальнення. Після цього конвеєр навчається на всій вибірці, обчислюється класифікаційний звіт (`classification_report` [30]) з показниками `precision`, `recall` і F1-міри для кожного класу, а потім модель разом із метаданими зберігається у файл `bottleneck_model.joblib`. На рис. 2.4 показано типовий фрагмент консольного виводу скрипта `train_bottleneck_model.py` із числовими значеннями цих метрик.

```
[Oleksandrs-MacBook-Air:training (demo) oleksandrsheketa$ python3 ./train_bottleneck_model.py
5-fold F1_macro mean=0.218, std=0.035
precision    recall  f1-score   support

BRANCH_MISPRED_NONSPEC      0.276    0.730    0.400        37
  L1D_CACHE_MISS_LD        0.228    0.660    0.339        47
  L1D_CACHE_MISS_ST        0.273    0.818    0.409        33
  L1I_CACHE_MISS_DEMAND     0.204    1.000    0.339        10
  MAP_STALL_DISPATCH        0.476    0.540    0.506       291
    NONE                    0.946    0.825    0.881      2301

      accuracy                   0.791      2719
    macro avg                   0.400    0.762    0.479      2719
    weighted avg                0.863    0.791    0.817      2719

Saved model to model_artifacts/bottleneck_model.joblib (context=2)
```

Рис. 2.4 – Консольний вивід скрипта `train_bottleneck_model.py` з класифікаційним звітом

Застосування моделі до нового коду реалізовано у скрипті `predict_bottlenecks.py`. Його інтерфейс дозволяє передати шлях до С-файлу, а також опційні параметри, зокрема шлях до моделі, спосіб дизасемблювання, кількість інструкцій, що слід показати під час демонстрації, та шлях до HTML-звіту. На високому рівні логіка роботи полягає в тому, що створюється тимчасовий каталог для компіляції і дизасемблювання, вхідний С-файл компілюється у бінарний файл, бінарний файл дизасемблюється у асемблерний код, асемблерний файл парситься, будуються вікна інструкцій і подаються на вхід моделі, а отримані прогнози використовуються для формування як текстового, так і HTML-звіту.

Фрагмент коду, що виконує прогноз і будує HTML-звіту, наведено в лістингу 2.4.

Лістинг 2.4 – Застосування моделі та генерація HTML-звіту

```
counts, label_order, preds = predict(asm_texts, Path(args.model))

print(f"Analyzed {len(instructions)} instructions from {asm_path}")
print("Predicted counts per bottleneck:")
for label in label_order or sorted(counts):
    print(f"  {label}: {counts.get(label, 0)}")

if args.show > 0:
    limit = min(args.show, len(instructions))
    print("\nSample predictions:")
    for idx in range(limit):
```

```

        label = preds[idx]
        print(f"[{label}] {instructions[idx][1]}")

    if args.keep_temp:
        print(f"Temporary files kept in {tmpdir}")
    if args.html_out:
        addr_map = symbolize_addresses(binary, [addr for addr, _ in
instructions])
        asm_entries: list[dict] = []
        source_labels: dict[int, str] = {}
        for (addr, text), label in zip(instructions, preds):
            src_file, src_line = addr_map.get(addr, (None, None))
            asm_entries.append(
                {
                    "address": f"0x{addr}" if not addr.startswith("0x") else
addr,
                    "text": text,
                    "label": label,
                    "src_file": src_file,
                    "src_line": src_line,
                }
            )
            if src_line is not None and label != "NONE":
                source_labels[src_line] = label
        source_lines = source.read_text().splitlines()
        html_out = Path(args.html_out)
        build_html(source, asm_entries, source_lines, html_out, source_labels)
        print(f"Wrote HTML report to {html_out}")

```

У цьому фрагменті викликається функція `predict`, яка застосовує навчену модель до списку асемблерних інструкцій і повертає розподіл прогнозів по класах та список міток для кожної інструкції. Далі будується мапа відповідності між адресами машинного коду та рядками вихідного C-файлу за допомогою утиліти `atos`, формується список асемблерних записів з полями адреси, тексту інструкції, прогнозованої мітки та відповідного рядка C-коду, а також створюється словник `source_labels`, де ключами є номери рядків вихідного файлу, а значеннями – типи `bottleneck`-ів. Функція `build_html`, що використовує ці структури, генерує HTML-документ з двома колонками, в якому асемблер та C-код відображаються синхронно, а проблемні ділянки підсвічуються. На рис. 2.5 показано приклад консольного запуску `predict_bottlenecks.py` з виводом кількості інструкцій кожного типу `bottleneck`-

а, а на рис. 2.6 – приклад HTML-звіту з підсвіченими рядками вихідного коду та асемблерними інструкціями.

```
[Oleksandrs-MacBook-Air:training (demo) oleksandrsheketa$ python3 ./predict_bottlenecks.py tes]
ts/1.c --html-out ~/Desktop/1.html
Analyzed 76 instructions from /tmp/bottleneck_test_uvi6gwbx/1.s
Predicted counts per bottleneck:
  BRANCH_MISPRED_NONSPEC: 1
  L1D_CACHE_MISS_LD: 4
  L1D_CACHE_MISS_ST: 6
  L1I_CACHE_MISS_DEMAND: 3
  MAP_STALL_DISPATCH: 5
  NONE: 57
Wrote HTML report to /Users/oleksandrsheketa/Desktop/1.html
```

Рис. 2.5 – Приклад консольного запуску predict_bottlenecks.py з виводом кількості інструкцій кожного типу bottleneck-а для аналізованого файлу

Assembly	Source (3.c)
0x100000460 sub sp, sp, #0x40	1 // Program 3 - Branch that is almost always taken (predictable)
0x100000464 stp x29, x30, [sp, #0x30]	2 #include <stdio.h>
0x100000468 add x29, sp, #0x30	3 #include <stdlib.h>
0x10000046c mov w8, #0x86a0 ; =34464	4
0x100000470 movk w8, #0x1, lsl #16	5 int main(void) {
0x100000474 str w8, [sp, #0x8]	6 const int N = 100000;
0x100000478 stur wzr, [x29, #-0x4]	7 int *a = malloc(N * sizeof(int));
0x10000047c stur w8, [x29, #-0x8]	8 if (!a) ret L1D_CACHE_MISS_ST
0x100000480 mov x0, #0x1a80 ; =6784	9
0x100000484 movk x0, #0x6, lsl #16	10 for (int i = 0; i < N; ++i) {
0x100000488 bl 0x1000005a4 <_printf+0x1000005a4>	11 a[i] = i;
0x10000048c stur x0, [x29, #-0x10]	12 }
0x100000490 ldur x8, [x29, #-0x10]	13
0x100000494 cbnz x8, 0x1000004a8 <_main+0x48>	14 long long sum = 0;
0x100000498 b 0x10000049c <_main+0x3c>	15 for (int i = 0; i < N; ++i) {
0x10000049c mov w8, #0x1 ; =1	16 if (i < N - 1) { // always true except last iteration
0x1000004a0 stur w8, [x29, #-0x4]	17 sum += a[i];
0x1000004a4 b 0x100000594 <_main+0x134>	18 } else {
0x1000004a8 stur wzr, [x29, #-0x14]	19 sum -= a[i];
0x1000004ac b 0x1000004b0 <_main+0x50>	20 }
0x1000004b0 ldur w9, [sp, #0x8]	21 }
0x1000004b4 ldur w8, [x29, #-0x14]	22
0x1000004b8 subs w8, w8, w9	23 printf("%lld\n", sum);
0x1000004bc b.ge 0x1000004e8 <_main+0x88>	24 free(a);
0x1000004c0 b 0x1000004c4 <_main+0x64>	25 return 0;
0x1000004c4 ldur w8, [x29, #-0x14]	26 }
0x1000004c8 ldur x9, [x29, #-0x10]	
0x1000004cc ldursw x10, [x29, #-0x14]	
0x1000004d0 str w8, [x9, x10, lsl #2]	
0x1000004d4 b 0x1000004d8 <_main+0x78>	
0x1000004d8 ldur w8, [x29, #-0x14]	
0x1000004dc add w8, w8, #0x1	
0x1000004e0 stur w8, [x29, #-0x14]	
0x1000004e4 b 0x1000004b0 <_main+0x50>	

Рис. 2.6 – Фрагмент HTML-звіту, згенерованого predict_bottlenecks.py: ліворуч – асемблерний код з адресами і мітками bottleneck-ів, праворуч – вихідний C-код із підсвіченими рядками, що відповідають проблемним інструкціям

Загалом описаний набір сценаріїв і модулів формує цілісний та відтворюваний інструментальний ланцюг, який охоплює весь життєвий цикл

моделі – від збору й підготовки даних до навчання та практичного застосування результатів аналізу. Така архітектура забезпечує чітке розділення відповідальностей між етапами, спрощує експериментування з параметрами (наприклад, розміром контекстного вікна чи конфігурацією класифікатора) та дозволяє інтегрувати запропонований підхід у реальний процес оптимізації програмного коду. Автоматизація ключових кроків мінімізує ручне втручання та знижує ризик помилок, а наявність як текстового, так і візуального (HTML) представлення результатів робить систему зручною не лише для дослідницьких експериментів, а й для практичного використання розробниками під час аналізу продуктивності та пошуку вузьких місць у коді.

2.5. Оцінювання адекватності та точності моделі

Оцінка адекватності та точності запропонованої моделі є обов'язковою складовою магістерської роботи. У даному дослідженні якість класифікатора оцінювалася на сформованому наборі `training_data.json`, який містить 2719 навчальних прикладів, кожен із яких є контекстним вікном інструкцій із відповідною міткою типу `bottleneck-a`. Класифікація здійснюється на шість класів: `NONE` (відсутність домінуючого `bottleneck-y`), `BRANCH_MISPRED_NONSPEC`, `L1D_CACHE_MISS_LD`, `L1D_CACHE_MISS_ST`, `L1I_CACHE_MISS_DEMAND` та `MAP_STALL_DISPATCH`.

Після навчання моделі на всій вибірці було побудовано класифікаційний звіт, який містить значення `precision`, `recall` та `F1-score` для кожного класу окремо. Для оцінки загальної якості моделі використовувалися три основні метрики: загальна асигурація як частка правильно класифікованих прикладів, макроусереднена `F1`-міра як середнє значення `F1` по всіх класах без урахування їх частоти та зважена `F1`-міра, у якій внесок кожного класу пропорційний кількості прикладів цього класу. Значення цих метрик отримуються з класифікаційного звіту, який виводить скрипт

train_bottleneck_model.py після навчання моделі; типовий фрагмент такого виводу показано на рис. 2.4.

За результатами типового запуску скрипта `python3 train_bottleneck_model.py --training-data training_data.json --model-out model_artifacts/bottleneck_model.joblib` отримано загальну точність (accuracy), що дорівнює 0,791, макроусереднену F1-міру, близьку до 0,479, і зважену F1-міру, близьку 0,817. Для окремих класів F1-міра набуває таких орієнтовних значень: клас NONE демонструє F1-міру приблизно 0,881 при високій точності та повноті, що свідчить про здатність моделі надійно відокремлювати «нормальні» інструкції від усіх інших; клас MAP_STALL_DISPATCH має F1-міру порядку 0,506, що можна вважати прийнятною якістю наявного датасету; класи BRANCH_MISPRED_NONSPEC, L1D_CACHE_MISS_LD, L1D_CACHE_MISS_ST та L1I_CACHE_MISS_DEMAND демонструють F1-міру на рівні приблизно 0,34–0,41, що свідчить про середню якість розрізнення тонших типів bottleneck-ів на обмеженій вибірці.

Отримані результати свідчать, що модель досить добре навчається відокремлювати «звичайні» інструкції (клас NONE) від інструкцій, де спостерігається певний bottleneck, помірно добре розрізняє типи bottleneck-ів, пов'язаних із простоям конвеєра (MAP_STALL_DISPATCH), але має обмежену здатність однозначно відносити приклади до конкретних типів промахів кешу або помилкових передбачень переходів. Серед основних причин такого обмеження можна назвати перекося розподілу класів, коли прикладів класів NONE і MAP_STALL_DISPATCH суттєво більше, ніж інших класів; відносно невеликий обсяг навчальної вибірки, сформованої на основі обмеженого набору тестових програм; а також використання символічного представлення, яке враховує лише локальний контекст інструкцій, але не бере до уваги обсяг оброблюваних даних, параметри вхідних наборів чи особливості структури програми на вищому рівні.

З точки зору адекватності моделі для задачі магістерської роботи такий рівень точності можна вважати прийнятним для прототипу. Модель підтверджує гіпотезу, згідно з якою локальні патерни інструкцій, особливо для класу NONE та типових bottleneck-ів, містять достатньо інформації для статистичного прогнозування. Використання символічної TF-IDF + Linear SVM-моделі виявилось життєздатним базовим підходом, а подальше покращення якості можна очікувати після розширення навчального набору та переходу до складніших моделей, зокрема нейронних, як це було окреслено в першому розділі.

З практичної точки зору модель уже на стадії прототипу може допомагати розробнику швидко звузити коло підозрілих ділянок коду, вказати, який тип апаратного bottleneck-а найімовірніше проявиться при виконанні, і використовуватися як доповнення до класичних профілювальних інструментів на етапі статичного аналізу. Візуальне подання результатів у HTML-звіті, про яке йшлося в попередньому підрозділі, полегшує інтерпретацію рішень моделі й робить інструмент зручним для повсякденного використання розробниками.

Висновки до другого розділу

У другому розділі магістерської кваліфікаційної роботи виконано проєктування та практичну реалізацію системи прогнозування продуктивно-критичних ділянок машинного коду на основі методів статистичного аналізу та машинного навчання. Основною метою даного розділу було не лише теоретично обґрунтувати вибраний підхід, а й продемонструвати можливість його реалізації у вигляді працездатного програмного прототипу, придатного для застосування в реальних інженерних сценаріях.

У процесі роботи розроблено цілісну архітектуру програмного комплексу, що охоплює повний цикл обробки даних — від підготовки вихідних програм і збору профільної інформації до навчання моделі

машинного навчання та її подальшого застосування. Такий наскрізний підхід дозволяє забезпечити відтворюваність результатів, знизити обсяг ручної роботи та мінімізувати ризик помилок на окремих етапах аналізу. Архітектура системи передбачає чітке розділення відповідальностей між модулями, що спрощує як експериментування з параметрами, так і подальший розвиток програмного продукту.

У розділі детально описано процес автоматизованого збору апаратних лічильників продуктивності з використанням Xcode Instruments та утиліти xtrase. Обґрунтовано вибір конкретних подій, які відображають типові мікроархітектурні причини деградації продуктивності, зокрема кеш-промахи, неправильні передбачення переходів та простої конвеєра процесора. Показано, що використання апаратних лічильників дозволяє отримати більш глибоке уявлення про причини втрат продуктивності порівняно з простими часовими вимірюваннями.

Важливим результатом другого розділу є розробка методу спрощення та агрегації профільних даних. Оскільки сирі профілі трасування мають великий обсяг і складну структуру, у роботі запропоновано алгоритм перетворення детальних XML-профілів у компактне JSON-подання, яке містить лише найбільш значущу інформацію — відповідність між адресами машинних інструкцій і домінуючим типом апаратного bottleneck-а. Такий підхід дозволив без втрати суттєвих характеристик підготувати дані для подальшого використання в задачах машинного навчання.

Окрему увагу у другому розділі приділено формуванню навчальної вибірки. Запропоновано підхід, за якого кожен приклад представляє собою контекстне вікно машинних інструкцій фіксованого розміру з відповідною міткою класу. Показано, що використання локального контексту дозволяє врахувати оточення інструкції, що є важливим для коректного прогнозування продуктивно-критичних ділянок. Для зменшення розмірності та шуму в даних реалізовано процедури нормалізації асемблерних інструкцій, які

абстрагуються від конкретних номерів регістрів і значень констант, зосереджуючи увагу моделі на структурі операцій.

У межах розділу обґрунтовано вибір символного TF-IDF-подання машинного коду як базового способу формування ознак. Такий підхід дозволяє інтерпретувати послідовності машинних інструкцій як текстові об'єкти та використовувати добре відомі методи обробки текстових даних. Застосування символних n-грам забезпечує фіксацію локальних патернів інструкцій, які можуть бути пов'язані з конкретними типами апаратних bottleneck-ів. Обраний спосіб подання є універсальним і не залежить від конкретної архітектури на рівні вихідного коду, що є важливою перевагою для задач аналізу машинного коду.

Для побудови моделі класифікації використано лінійну опорну векторну машину з балансуванням класів. Показано, що цей алгоритм добре пристосований до роботи з високорозмірними розрідженими ознаками, характерними для TF-IDF-подання, та забезпечує прийнятний компроміс між точністю прогнозування, швидкістю навчання і інтерпретованістю результатів. Реалізована модель здатна виконувати багатокласову класифікацію та прогнозувати як наявність продуктивно-критичних ділянок, так і їх домінуючий тип.

У другому розділі також проведено оцінювання адекватності та точності побудованої моделі. На основі експериментальних результатів встановлено, що модель демонструє високу здатність відокремлювати інструкції без ознак bottleneck-ів від потенційно проблемних, а також помірну точність класифікації окремих типів апаратних «вузьких місць». Виявлені обмеження пояснюються дисбалансом класів, обмеженим обсягом навчальної вибірки та використанням лише локального контексту без урахування глобальної структури програми. Водночас отримані результати є прийнятними для прототипу та підтверджують коректність обраної методології.

З практичної точки зору у другому розділі доведено можливість реалізації системи статичного прогнозування продуктивності у вигляді інструментального програмного комплексу. Реалізований конвеєр автоматизує ключові етапи аналізу, зменшує навантаження на розробника та створює передумови для інтеграції такого підходу у реальні процеси оптимізації програмного забезпечення. Наявність візуального представлення результатів додатково підвищує практичну цінність системи та спрощує інтерпретацію прогнозів.

Таким чином, у другому розділі магістерської роботи доведено практичну реалізованість запропонованого підходу до прогнозування продуктивно-критичних ділянок машинного коду, сформовано технічну основу програмного продукту та отримано експериментальні результати, які підтверджують доцільність використання методів машинного навчання для задач статичного аналізу продуктивності.

РОЗДІЛ 3. ОПИС ПРОГРАМНОГО ПРОДУКТУ

3.1 Постановка задачі

У цьому розділі описується програмний продукт, який реалізує практичне застосування побудованої моделі до аналізу нових фрагментів коду. Центральним елементом є сценарій `predict_bottlenecks.py`, що входить до складу програмного комплексу й відповідає за компіляцію переданого С-файлу, витяг асемблерних інструкцій, застосування навченого класифікатора та формування звіту з результатами. Саме цей сценарій дозволяє інтегрувати результати дослідження в робочий процес розробника, перетворюючи теоретичну модель на прикладний інструмент статичного аналізу.

Розроблюваний програмний продукт позиціонується як легкий консольний інструмент для розробників системного та високопродуктивного програмного забезпечення, який може бути інтегрований у наявний процес розробки без суттєвих змін інфраструктури. Його головне призначення – забезпечити швидкий статичний аналіз окремих фрагментів С-коду та виділити ті ділянки, що з найбільшою ймовірністю спричинятимуть проблеми продуктивності під час виконання. На відміну від традиційного профілювання, що потребує запуску повноцінних тестових навантажень, запропонований інструмент працює лише з вихідним кодом та результатами компіляції, що робить його зручним на ранніх етапах життєвого циклу програмного забезпечення.

Логіка роботи `predict_bottlenecks.py` організована як послідовність кроків. На першому етапі сценарій аналізує аргументи командного рядка: позиційний параметр задає шлях до С-файлу, який потрібно проаналізувати, а додаткові іменовані параметри дозволяють вказати розташування моделі, використаний компілятор, спосіб дизасемблювання, а також налаштувати режим роботи (наприклад, зберегти тимчасові файли чи вивести приклади

прогнозів у консоль). Після розбору аргументів скрипт перевіряє наявність необхідних інструментів у системі: компілятора `clang`, інструмента дизасемблювання `llvm-objdump` (зазвичай викликається через `xcrun llvm-objdump`), а у випадку генерації HTML-звіту також утиліти `atos`, яка використовується для зіставлення адрес машинного коду з рядками вихідного файлу. Далі створюється тимчасовий каталог, у якому відбуваються всі операції з компіляцією та дизасемблюванням, щоб не засмічувати основну структуру проєкту.

На наступному етапі вихідний C-файл компілюється за допомогою компілятора `clang` з використанням фіксованого набору ключів. Сценарій викликає внутрішню функцію `compile_to_tmp`, яка формує команду `clang` із параметрами, що викликають агресивну оптимізацію й зберігають достатньо налагоджувальної інформації (зокрема, використовується режим без опускання покажчика кадрового регістра). На виході отримується бінарний файл, що розміщується у тимчасовому каталозі. Потім викликається функція `dump_assembly`, яка запускає `llvm-objdump` із параметрами дизасемблювання в текстовий формат; отриманий асемблерний файл також зберігається у тимчасовому каталозі й виступає джерелом списку інструкцій для подальшого аналізу.

Після успішного компілювання та дизасемблювання сценарій переходить до обробки асемблерного файлу. Використовується функція `parse_assembly`, яка проходить по рядках файлу, виділяє адреси інструкцій і текст власне машинних команд, відфільтровує службові мітки та порожні рядки й формує впорядкований список пар «адреса – інструкція». Цей список є низькорівневим представленням коду, на основі якого формуються контекстні вікна для моделі. Саме на цьому етапі виконується попередня очистка тексту асемблера, наприклад, видалення байтових послідовностей, якщо дизасемблер вивів їх разом з інструкціями.

Після отримання списку інструкцій сценарій завантажує навчену модель. Для цього використовується функція `load_model` з модуля `bottleneck_model.py`, яка відкриває файл `bottleneck_model.joblib` (якщо користувач не вказав інший шлях), завантажує конвеєр `TfidfVectorizer` + `LinearSVC` і метадані, включно з розміром контексту. Далі функція `predict` формує для кожної інструкції вікно потрібного розміру, перетворює його в текстове представлення, застосовує модель та отримує прогнозований клас `bottleneck`-а для кожної інструкції. Одночасно формується й агрегований підрахунок кількості інструкцій кожного класу; ці дані використовуються для короткого текстового резюме в консолі.

Окрім консольного виводу, важливою частиною функціональності сценарію є побудова HTML-звіту. Якщо користувач передає параметр `--html-out`, `predict_bottlenecks.py` додатково виконує зіставлення адрес машинного коду з рядками вихідного C-файлу за допомогою утиліти `atos`; ця можливість дозволяє позначати не лише конкретні асемблерні інструкції, але й відповідні їм рядки початкового коду. На основі цієї інформації формується структура даних для HTML-шаблону, яка включає текстовий вміст вихідного файлу, список асемблерних інструкцій із адресами, прогнозовані класи `bottleneck`-ів для кожної інструкції, а також відповідність між рядками C-коду та типами «вузьких місць». Потім викликається функція `build_html`, яка генерує готову HTML-сторінку та зберігає її у файл за шляхом, вказаним у параметрі `--html-out`.

Узагальнюючи описані кроки, можна сказати, що сценарій `predict_bottlenecks.py` послідовно розбирає аргументи та перевіряє доступність зовнішніх інструментів, компілює та дизасемблює вихідний C-файл, виділяє з асемблера впорядкований список інструкцій, будує для них контекстні вікна, застосовує до цих вікон навчену модель для визначення типів `bottleneck`-ів і, зрештою, формує як текстовий, так і, за потреби, HTML-звіт для подальшого аналізу розробником.

Окремої уваги заслуговують параметри командного рядка, які дозволяють гнучко налаштовувати поведінку сценарію. Позиційний параметр задає шлях до С-файлу, що аналізується; якщо файл не знайдено, сценарій завершить роботу з повідомленням про помилку. Опція `--model` дозволяє вказати шлях до файлу моделі, якщо користувач не бажає використовувати значення за замовчуванням. За допомогою опцій `--clang` та `--objdump` можна задати альтернативні команди компіляції та дизасемблювання, що дає змогу легко адаптувати сценарій до специфіки конкретного середовища. Опція `--keep-temp` дозволяє зберегти тимчасовий каталог із бінарними та асемблерними файлами для подальшого аналізу; у звичайному режимі цей каталог автоматично видаляється. Прапорець `--verbose` вмикає детальний вивід команд, що виконуються, а параметр `--show` задає кількість прогнозованих інструкцій, які слід додатково відобразити в консолі з мітками класів. Нарешті, параметр `--html-out` вказує шлях до HTML-файлу, який буде згенеровано для візуального аналізу результатів; якщо цей параметр не задано, побудова HTML-звіту не виконується, і користувач отримує лише текстову інформацію в консолі.

На рис. 3.1 наведено приклад запуску `predict_bottlenecks.py` у терміналі: видно команду з переданим шляхом до С-файлу, коротке резюме у вигляді кількості інструкцій кожного класу, а також кілька прикладів інструкцій з їх прогнозованими `bottleneck`-мітками.

```
[Oleksandrs-MacBook-Air:training (demo) oleksandrsheketa$ python3 ./predict_bottlenecks.py tests/1.c --html-out ~/Desktop/1_out.html]
Analyzed 76 instructions from /tmp/bottleneck_test_s0rp0gn/_1.s
Predicted counts per bottleneck:
  BRANCH_MISPRED_NONSPEC: 1
  L1D_CACHE_MISS_LD: 4
  L1D_CACHE_MISS_ST: 6
  L1I_CACHE_MISS_DEMAND: 3
  MAP_STALL_DISPATCH: 5
  NONE: 57
Wrote HTML report to /Users/oleksandrsheketa/Desktop/1_out.html
```

Рис. 3.1 – Приклад запуску `predict_bottlenecks.py` у терміналі: вхідний С-файл, підрахунок інструкцій кожного класу та фрагмент списку інструкцій з прогнозами типів `bottleneck`-ів.

Додатково слід зазначити, що архітектура сценарію `predict_bottlenecks.py` спроектована з урахуванням подальшого розширення функціональності. Модульна структура з чітко відокремленими етапами компіляції, дизасемблювання, парсингу асемблеру та застосування моделі дозволяє без суттєвих змін інтегрувати нові типи класифікаторів, альтернативні формати вхідних даних або додаткові способи візуалізації результатів. Зокрема, у майбутньому можливе розширення підтримки інших мов програмування на основі LLVM, додавання нових класів «вузьких місць» або інтеграція інструмента в автоматизовані конвеєри CI/CD для регулярного контролю потенційних проблем продуктивності на ранніх етапах розробки.

3.2 Сценарій використання

Другою важливою складовою описуваного програмного продукту є демонстрація існуючого інтерфейсу для аналізу конкретних C-файлів. Практичне використання сценарію `predict_bottlenecks.py` складається з двох основних етапів: запуску аналізу для вибраного файлу та перегляду HTML-звіту з інтерпретацією результатів. У цьому підрозділі розглядається типовий сценарій, який відображає, як розробник може використовувати інструмент у процесі роботи з продуктивно чутливим кодом.

Перед безпосереднім запуском аналізу передбачається, що користувач виконав початкове налаштування середовища: встановив необхідні компілятор та інструменти дизасемблювання, завантажив або навчив модель і переконався, що вона доступна за шляхом, вказаним у параметрі `--model`. Типовим користувачем системи є розробник, який добре орієнтується в низькорівневих аспектах виконання програм і здатен інтерпретувати типи `bottleneck-ів`, проте сама взаємодія з інструментом не вимагає глибоких знань машинного навчання: усі складні операції приховані за простим інтерфейсом командного рядка.

Для демонстрації можна взяти умовний C-файл, який містить, наприклад, один або кілька вкладених циклів із обробкою масивів, а також розгалуження з нетривіальними умовами. Розробник відкриває термінал, переходить до каталогу проєкту та виконує команду на кшталт: `predict_bottlenecks.py source.c --html-out report.html`. Після цього сценарій послідовно компілює файл, дизасемблює отриманий бінарник, формує вікна інструкцій, застосовує модель і, нарешті, створює HTML-файл із назвою `report.html`. У консолі відображається інформація про кількість проаналізованих інструкцій і кількість інструкцій кожного класу; на цьому етапі користувач уже може зробити попередній висновок про те, наскільки «насиченим» є код потенційними bottleneck-ами.

Після формування HTML-звіту користувач відкриває файл `report.html` у браузері. На рис. 3.2 наведено фрагмент HTML-звіту для типового тестового файлу, де чітко видно відповідність між проблемними інструкціями та рядками вихідного коду.

Assembly	Source (1.c)
0x100000460 sub sp, sp, #0x40	1 // Program 1 - Sequential sum over an array (good locality, no branch)
0x100000464 stp x29, x30, [sp, #0x30]	2 #include <stdio.h>
0x100000468 add x29, sp, #0x30	3 #include <stdlib.h>
0x10000046c mov w8, #0x86a0 ; =34464	4
0x100000470 movk w8, #0x1, lsl #16	5 int main(void) {
0x100000474 str w8, [sp, #0x8]	6 const int N = 100000;
0x100000478 stur wzr, [x29, #-0x4]	7 int *a = malloc(N * sizeof(int));
0x10000047c stur w8, [x29, #-0x8]	8 if (!a) return 1;
0x100000480 mov x0, #0x1a80 ; =6784	9
0x100000484 movk x0, #0x6, lsl #16	10 for (int i = 0; i < N; ++i) {
0x100000488 bl 0x10000056c <_printf+0x10000056c>	11 a[i] = i;
0x10000048c stur x0, [x29, #-0x10]	12 }
0x100000490 ldur x8, [x29, #-0x10]	13
0x100000494 cbnz x8, 0x1000004a8 <_main+0x48>	14 long long sum = 0;
0x100000498 b 0x10000049c <_main+0x3c>	15 for (int i = 0; i < N; ++i) {
0x10000049c mov w8, #0x1 ; =1	16 sum += a[i]; // linear, branch-free
0x1000004a0 stur w8, [x29, #-0x4]	17 }
0x1000004a4 b 0x10000055c <_main+0xfc>	18
0x1000004a8 stur wzr, [x29, #-0x14]	19 printf("%lld\n", sum);
0x1000004ac b 0x1000004b0 <_main+0x50>	20 free(a);
0x1000004b0 ldr w9, [sp, #0x8]	21 return 0;
0x1000004b4 ldur w8, [x29, #-0x14]	22 }
0x1000004b8 subs w8, w8, w9	
0x1000004bc b.ge 0x1000004e8 <_main+0x88>	
0x1000004c0 b 0x1000004c4 <_main+0x64>	
0x1000004c4 ldur w8, [x29, #-0x14]	
0x1000004c8 ldur x9, [x29, #-0x10]	
0x1000004cc ldursw x10, [x29, #-0x14]	
0x1000004d0 str w8, [x9, x10, lsl #2]	
0x1000004d4 b 0x1000004d8 <_main+0x78>	
0x1000004d8 ldur w8, [x29, #-0x14]	
0x1000004dc add w8, w8, #0x1	
0x1000004e0 stur w8, [x29, #-0x14]	
0x1000004e4 h 0x1000004b0 <_main+0x50>	

Рис. 3.2 – Вигляд HTML-звіту, згенерованого `predict_bottlenecks.py`: ліворуч – список асемблерних інструкцій з адресами і підсвіченими bottleneck-рядками, праворуч – вихідний C-код із нумерацією рядків і відповідними підсвітками.

Сторінка складається з двох основних вертикальних панелей. У лівій панелі відображається асемблерний код: кожен рядок містить адресу інструкції та її текстове представлення, а для інструкцій, які модель віднесла до певного типу bottleneck-а, фон підсвічується. У правій панелі показано вихідний С-код із нумерацією рядків; рядки, до яких належать підсвічені асемблерні інструкції, також мають виділення фоном. Як правило, один і той самий тип bottleneck-а (наприклад, `L1D_CACHE_MISS_LD`) позначається однаковим стилем підсвічування для полегшення візуального сприйняття.

Інтерфейс HTML-звіту підтримує інтерактивну роботу з даними. При наведенні курсора миші на рядок асемблерного коду у лівій панелі відповідні рядки С-коду у правій панелі автоматично підсвічуються, що дозволяє швидко зрозуміти, на який саме фрагмент вихідного тексту припадає певна інструкція. Якщо асемблерний рядок має bottleneck-мітку, при наведенні на нього з'являється невелика підказка (tooltip), що містить назву типу bottleneck-а, наприклад `L1D_CACHE_MISS_LD` або `BRANCH_MISPRED_NONSPEC`. Аналогічно, коли користувач наводить курсор на рядок С-коду, у лівій панелі підсвічуються всі асемблерні інструкції, що пов'язані з цим рядком, а якщо серед них є bottleneck-інструкції, для них також відображається підказка з типом «вузького місця». Така двостороння навігація значно полегшує аналіз ситуацій, коли одна й та сама ділянка вихідного коду компілюється у кілька різних інструкцій з різними характеристиками.

На рис. 3.3 показано приклад взаємодії користувача з HTML-звітом: на екранному зображенні видно, як наведення на асемблерну інструкцію спричиняє підсвічування відповідних рядків С-коду та появу підказки з типом bottleneck-а. Це ілюструє інтуїтивний характер інтерфейсу, який дозволяє досить легко перейти від статистичних значень апаратних лічильників до конкретних фрагментів вихідного коду.

0x100000474	slr	w0, l>μ, #0x0J	0	const int N = 100000;
0x100000478	stur	wzr, [x29, #-0x4]	7	int *a = malloc(N * sizeof(int));
0x10000047c	stur	w8, [x29, #-0x8]	8	L1D_CACHE_MISS_ST urn 1;
0x100000480	mov	x0, #0x1a80 ; =6784	9	
0x100000484	movk	x0, #0x6, lsl #16	10	for (int i = 0; i < N; ++i) {
0x100000488	bl	0x10000056c <_printf+0x10000056c>	11	a[i] = i;
0x10000048c	stur	x0, [x29, #-0x10]	12	}
0x100000490	ldur	x8, [x29, #-0x10]	13	
0x100000494	cbnz	x8, 0x1000004a8 <_main+0x48>	14	long long sum = 0;
0x100000498	b	0x10000049c <_main+0x3c>	15	for (int i = 0; i < N; ++i) {
0x10000049c	mov	w8, #0x1 ; =1	16	sum += a[i]; // linear, branch-

Рис. 3.3 – Демонстрація інтерактивної роботи з HTML-звітом:

наведення курсора на асемблерний рядок підсвічує пов’язані рядки С-коду та показує підказку з типом bottleneck-а

Сценарії використання інструменту можуть відрізнятися залежно від етапу життєвого циклу проєкту. На ранніх етапах розробки аналіз окремих функцій або модулів дозволяє виявити невдалі алгоритмічні рішення ще до інтеграції їх в основний код. На етапі оптимізації вже наявної системи розробник може порівнювати HTML-звіти до та після внесення змін, відстежуючи, як перерозподіляються bottleneck-інструкції та чи зникають підозрілі ділянки. У рамках командної роботи інструмент може використовуватися як доповнення до процедури код-рев’ю, коли для кожного складного зміненого модуля автоматично генерується окремий звіт.

Важливою частиною сценарію використання є інтерпретація отриманих результатів. Після перегляду HTML-звіту розробник може локалізувати ділянки С-коду, що відповідають інструкціям із найагресивнішими типами bottleneck-ів (наприклад, пов’язаними з кеш-промахами або неправильним передбаченням переходів), і прийняти рішення щодо можливої оптимізації: спростити логіку розгалужень, змінити структуру даних, розбити великі цикли на декілька дрібніших тощо. Таким чином, інструмент не лише виявляє потенційні проблеми, а й підказує напрямки для подальшого вдосконалення реалізації.

Окрім базового сценарію використання, коли розробник аналізує один конкретний С-файл, інструмент може бути інтегрований у ширший процес розробки. Наприклад, під час рев’ю коду можна періодично запускати

`predict_bottlenecks.py` для нових або змінених модулів і накопичувати HTML-звіти в окремому каталозі, що дозволить відстежувати, як зміни у коді впливають на прогнозовану продуктивність. Хоча у межах магістерської роботи та виробничої практики основна увага приділялася саме реалізації й демонстрації базового сценарію, архітектура сценарію та формат HTML-звіту залишають достатній запас для подальшої інтеграції з іншими інструментами й процесами.

У підсумку сценарій `predict_bottlenecks.py` та його HTML-інтерфейс забезпечують практичне застосування розробленої моделі в реальних умовах. Вони дозволяють не лише отримувати числову оцінку кількості потенційних bottleneck-інструкцій, але й швидко знаходити відповідні ділянки вихідного коду, візуально порівнювати різні фрагменти та приймати обґрунтовані рішення щодо подальшої оптимізації.

Висновки до третього розділу

У третьому розділі магістерської кваліфікаційної роботи наведено детальний опис програмного продукту, який реалізує практичне застосування розробленої у попередніх розділах системи прогнозування продуктивно-критичних ділянок машинного коду. Основною метою даного розділу було продемонструвати, яким чином теоретичні положення, методи та алгоритми, обґрунтовані раніше, втілюються у вигляді завершеного програмного інструменту, придатного для використання у реальному процесі розробки програмного забезпечення.

У межах розділу сформульовано постановку задачі програмного продукту та визначено його функціональне призначення. Показано, що розроблений інструмент орієнтований на статичний аналіз окремих фрагментів С-коду з метою раннього виявлення потенційних проблем продуктивності без необхідності фактичного виконання програм. Такий підхід є особливо корисним на ранніх етапах життєвого циклу програмного

забезпечення, коли відсутні стабільні тестові сценарії або проведення повноцінного профілювання є недоцільним з точки зору витрат часу і ресурсів.

У третьому розділі детально описано логіку роботи програмного сценарію `predict_bottlenecks.py`, який є ключовим компонентом взаємодії користувача з системою. Показано, що сценарій реалізує повний цикл аналізу в автоматизованому режимі, починаючи з обробки аргументів командного рядка і перевірки наявності зовнішніх інструментів та завершуючи формуванням прогнозів і генерацією звітів. Така організація дозволяє мінімізувати участь користувача у внутрішніх процесах аналізу та зосередитися на інтерпретації результатів.

Особливу увагу приділено опису процесу компіляції та дизасемблювання вхідного C-файлу. Показано, що використання стандартного компіляторного ланцюжка на основі `clang` і `llvm-objdump` забезпечує відтворюваність результатів та дозволяє отримувати уніфіковане асемблерне подання машинного коду. Формування впорядкованого списку машинних інструкцій є ключовим етапом, оскільки саме на його основі будуються контекстні вікна, що подаються на вхід навченій моделі машинного навчання.

У розділі описано механізм застосування навченої моделі до нового коду. Показано, що для кожної інструкції формується контекстне вікно фіксованого розміру, яке перетворюється у текстове представлення та використовується для прогнозування типу потенційного `bottleneck`-а. Окрім індивідуальних прогнозів для кожної інструкції, система формує агреговану статистику, що дозволяє отримати загальне уявлення про «насиченість» аналізованого фрагмента коду продуктивно-критичними ділянками.

Важливим практичним результатом третього розділу є реалізація HTML-інтерфейсу візуалізації результатів аналізу. Показано, що поєднання асемблерного коду з відповідними рядками вихідного C-коду та підсвічуванням прогнозованих `bottleneck`-ів значно спрощує інтерпретацію

результатів і знижує поріг входження для користувача. Інтерактивні можливості HTML-звіту, зокрема синхронне підсвічування пов'язаних рядків асемблера та вихідного коду, дозволяють швидко локалізувати проблемні ділянки та зрозуміти їхній контекст у програмі.

У третьому розділі також розглянуто типові сценарії використання програмного продукту у процесі розробки програмного забезпечення. Показано, що інструмент може бути застосований як на етапі початкового проєктування алгоритмів, так і під час оптимізації вже наявного коду або проведення код-рев'ю. Можливість швидкого аналізу окремих модулів або функцій дозволяє розробнику заздалегідь зосередити увагу на найбільш підозрілих фрагментах і зменшити обсяг ручної роботи під час подальшого профілювання.

З практичної точки зору розроблений програмний продукт є легким консольним інструментом, який не потребує складної інсталяції або спеціалізованої інфраструктури. Його модульна архітектура та чітке розділення етапів обробки даних створюють передумови для подальшого розвитку системи, зокрема шляхом додавання нових типів bottleneck-ів, підтримки інших мов програмування або інтеграції з автоматизованими конвеєрами CI/CD.

Узагальнюючи результати третього розділу, можна зробити висновок, що розроблений програмний продукт є функціонально завершеним прототипом системи статичного аналізу продуктивності машинного коду. Він наочно демонструє можливість практичного застосування методів машинного навчання для прогнозування характеристик виконання програм та створює зручний інструмент для підтримки рішень щодо оптимізації коду.

Таким чином, у третьому розділі доведено, що результати теоретичних і експериментальних досліджень, отримані у межах магістерської роботи, можуть бути успішно реалізовані у вигляді прикладного програмного продукту. Розроблений інструмент підтверджує практичну цінність

запропонованого підходу та може розглядатися як основа для подальших досліджень і впровадження у реальних проєктах.

ВИСНОВКИ

У даній магістерській роботі розроблено програмну систему прогнозування часу виконання машинного коду з використанням методів статистичного аналізу та машинного навчання. Основний акцент дослідження зроблено на задачі раннього виявлення потенційних продуктивно-критичних ділянок машинного коду без необхідності його фактичного виконання, що є актуальним для сучасних високопродуктивних та системних програмних рішень.

У ході виконання роботи проаналізовано сучасні підходи до оцінювання продуктивності програмного забезпечення, зокрема класичні методи динамічного профілювання, а також їх обмеження, пов'язані з високими часовими витратами, залежністю від середовища виконання та складністю застосування на ранніх етапах розробки. На основі огляду наукових джерел обґрунтовано доцільність використання методів машинного навчання для статичного аналізу машинного коду та прогнозування його часових характеристик і потенційних «вузьких місць».

Запропоновано підхід до представлення машинного коду у вигляді текстових ознак шляхом використання символічного TF-IDF-подання, що дозволяє формалізувати локальні патерни машинних інструкцій і застосовувати алгоритми текстової класифікації. На основі зібраних профільних даних сформовано навчальну вибірку та побудовано багатокласову модель класифікації з використанням лінійної опорної векторної машини, здатну прогнозувати наявність і тип апаратних продуктивних bottleneck-ів.

У рамках практичної частини роботи реалізовано програмний прототип, який охоплює повний цикл обробки даних: автоматизований збір апаратних лічильників продуктивності, синхронізацію профілів з дизасемблерним кодом, формування навчальної вибірки, навчання моделі машинного навчання та її

застосування до аналізу нових фрагментів коду. Проведено експериментальну оцінку адекватності та точності моделі, результати якої підтвердили можливість статистичного прогнозування продуктивно-критичних ділянок на основі локального контексту машинних інструкцій.

Розроблений програмний продукт забезпечує зручний інтерфейс взаємодії з користувачем у вигляді консольного інструменту та HTML-звіту з візуалізацією результатів аналізу. Це дозволяє швидко локалізувати потенційно проблемні ділянки вихідного коду та використовувати отримані прогнози як допоміжну інформацію під час прийняття рішень щодо оптимізації. Таким чином, система може ефективно доповнювати традиційні інструменти профілювання, особливо на ранніх етапах життєвого циклу програмного забезпечення.

Отримані результати підтверджують життєздатність запропонованого підходу та його практичну цінність як прототипу інструменту статичного аналізу продуктивності. Подальші дослідження можуть бути спрямовані на розширення та балансування навчальної вибірки, врахування додаткових контекстних і структурних характеристик коду, а також застосування більш складних моделей машинного навчання, зокрема нейронних і графових, з метою підвищення точності та узагальнювальної здатності системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hotspy: identifying performance hotspot with graph neural network based static analysis / D. Huo та ін. CCF transactions on high performance computing. 2025. Т. 7, № 3. С. 260–274. URL: <https://doi.org/10.1007/s42514-025-00220-7> (дата звернення: 19.12.2025).
2. How software refactoring impacts execution time / L. Traini та ін. ACM transactions on software engineering and methodology. 2022. Т. 31, № 2. С. 1–23. URL: <https://doi.org/10.1145/3485136> (дата звернення: 19.12.2025).
3. Learning based methods for code runtime complexity prediction / J. Sikka та ін. Advances in information retrieval (ECIR 2020). 2020. Т. 12035. С. 313–325. URL: https://doi.org/10.1007/978-3-030-45439-5_21 (дата звернення: 19.12.2025).
4. Gong J., Chen T. Deep configuration performance learning: a systematic survey and taxonomy. ACM transactions on software engineering and methodology. 2024. С. 1–62. URL: <https://doi.org/10.1145/3702986> (дата звернення: 19.12.2025).
5. Ha H., Fan Z., Zhang H. Uncertainty-Aware performance prediction for highly configurable software systems via bayesian neural networks. ArXiv preprint. 2022. С. 1–13. URL: <https://doi.org/10.48550/arXiv.2212.13359> (дата звернення: 19.12.2025).
6. Gong J., Chen T. Predicting Software Performance with Divide-and-Learn. Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE 2023). 2023. С. 1–13. URL: <https://doi.org/10.1145/3611643.3616334> (дата звернення: 19.12.2025).
7. Enhancing performance bug prediction using performance code metrics / G. Zhao та ін. Proceedings of the IEEE/ACM 21st international conference on

mining software repositories (MSR 2024). 2024. С. 50–62. URL: <https://doi.org/10.1145/3643991.3644920> (дата звернення: 19.12.2025).

8. DeleSmell: code smell detection based on deep learning and latent semantic analysis / Y. Zhang та ін. Knowledge-Based systems. 2022. Т. 255. С. 109737. URL: <https://doi.org/10.1016/j.knosys.2022.109737> (дата звернення: 19.12.2025).

9. Google. BigQuery text analysis functions – TF_IDF. Google Cloud. URL: https://cloud.google.com/bigquery/docs/reference/standard-sql/text-analysis-functions#tf_idf (дата звернення: 19.12.2025).

10. Laaber C., Basmaci M., Salza P. Predicting unstable software benchmarks using static source code features. Empirical software engineering. 2021. Т. 26, № 6. С. 1–53. URL: <https://doi.org/10.1007/s10664-021-09996-y> (дата звернення: 19.12.2025).

11. Ding N., Awan M., Williams S. Instruction Roofline: An insightful visual performance model for GPUs. Concurrency and computation: practice and experience. 2021. С. 1–19. URL: <https://doi.org/10.1002/cpe.6591> (дата звернення: 19.12.2025).

12. Reymond H., Puaut I., Amalou A. N. WORTEX: worst-case execution time and energy estimation in low-power microprocessors using explainable ML. Proceedings of the 22nd international workshop on worst-case execution time analysis (WCET 2024). 2024. Т. 121. С. 1–14. URL: <https://doi.org/10.4230/OASIcs.WCET.2024.1> (дата звернення: 19.12.2025).

13. Amalou A. N., Fromont E., Puaut I. CAWET: context-aware worst-case execution time estimation using transformers. Proceedings of the 35th euromicro conference on real-time systems (ECRTS 2023). 2023. Т. 262. С. 1–20. URL: <https://doi.org/10.4230/LIPIcs.ECRTS.2023.7> (дата звернення: 19.12.2025).

14. Maigari Z. S., Folorunso O. A. Predicting time complexity defects in software systems using machine learning techniques. International journal of

advances in engineering and management. 2025. Т. 7, № 7. С. 212–216. URL: <https://doi.org/10.35629/5252-0707212216> (дата звернення: 19.12.2025).

15. On the use of deep learning in software defect prediction / G. Giray та ін. Journal of systems and software. 2023. Т. 195. С. 1–26. URL: <https://doi.org/10.1016/j.jss.2022.111537> (дата звернення: 19.12.2025).

16. Jahandideh Y. Advanced static analysis for predicting performance bugs (1st offering) - JLR challenge #4 by: younes jahandideh. University of Windsor. URL: <https://www.uwindsor.ca/science/computerscience/416419/advanced-static-analysis-predicting-performance-bugs-1st-offering-jlr-challenge-4-younes> (дата звернення: 19.12.2025).

17. Learning to characterize performance regression introducing code changes / D. ALShoaibi та ін. Proceedings of the ACM conference. 2022. С. 1590–1597. URL: <https://doi.org/10.1145/3477314.3507150> (дата звернення: 19.12.2025).

18. Predicting a program's execution time after move method refactoring based on deep learning and feature interaction / Y. Yu та ін. Applied sciences. 2025. Т. 15, № 8. С. 1–26. URL: <https://doi.org/10.3390/app15084270> (дата звернення: 19.12.2025).

19. Amalou A. N., Fromont E., Puaut I. Fast and accurate context-aware basic block timing prediction using transformers. Proceedings of the 33rd ACM SIGPLAN international conference on compiler construction (CC 2024). 2024. С. 227–237. URL: <https://doi.org/10.1145/3640537.3641572> (дата звернення: 19.12.2025).

20. PrETi: predicting execution time in early stage with LLVM and machine learning / R. Xu та ін. ArXiv preprint. 2025. С. 1–11. URL: <https://doi.org/10.48550/arXiv.2503.13679> (дата звернення: 19.12.2025).

21. Shen Z., Chen S. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. Security and

- communication networks. 2020. Т. 2020. С. 1–16. URL: <https://doi.org/10.1155/2020/8858010> (дата звернення: 19.12.2025).
22. scikit-learn. Getting started. scikit-learn. URL: https://scikit-learn.org/stable/getting_started.html (дата звернення: 19.12.2025).
23. LLVM Project. Welcome to Clang's documentation!. Clang C Language Family Frontend for LLVM. URL: <https://clang.llvm.org/docs/> (дата звернення: 19.12.2025).
24. LLVM Project. Llvm-objdump - LLVM's object file dumper. The LLVM Compiler Infrastructure Project. URL: <https://llvm.org/docs/CommandGuide/llvm-objdump.html> (дата звернення: 19.12.2025).
25. Apple. Performance and metrics. Apple Developer Documentation. URL: <https://developer.apple.com/documentation/xcode/performance-and-metrics> (дата звернення: 19.12.2025).
26. Apple. Xcode 14 release notes. Apple Developer Documentation. URL: <https://developer.apple.com/documentation/xcode-release-notes/xcode-14-release-notes> (дата звернення: 19.12.2025).
27. Logistic regression / Y. Geng та ін. Practical machine learning illustrated with KNIME. 2024. С. 99–132. URL: https://doi.org/10.1007/978-981-97-3954-7_4 (дата звернення: 19.12.2025).
28. scikit-learn. TfidfVectorizer. scikit-learn. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html (дата звернення: 19.12.2025).
29. scikit-learn. LinearSVC. scikit-learn. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html> (дата звернення: 19.12.2025).
30. scikit-learn. Classification_report. scikit-learn. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html (дата звернення: 19.12.2025).

ДОДАТКИ

Код для генерації даних для тренування (generate_training_data.py):

```
#!/usr/bin/env python3
import argparse
import json
import shlex
import sys

from deps.paths import ROOT, TEMPLATE
from deps.utils import ensure_tools_available
from deps.tests import resolve_tests
from deps.build import compile_source, emit_assembly
from deps.trace_ops import record_trace, export_trace
from deps.profile import simplify_profile
from deps.training_data import build_training_data, write_training_data

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Compile tests, dump assembly, record xctrace runs, and export
profiles."
    )
    parser.add_argument(
        "--tests-override",
        help="Comma-separated list of test basenames to process (e.g. 1,5,10).
Defaults to every *.c in tests/.",
    )
    parser.add_argument(
        "--clang",
        default="clang",
        help="clang executable to use (default: clang on PATH).",
    )
    parser.add_argument(
        "--xctrace",
        default="xctrace",
        help="xctrace executable to use (default: xctrace on PATH).",
    )
    parser.add_argument(
        "--objdump",
        default="xcrun llvm-objdump",
        help="llvm-objdump invocation (default: xcrun llvm-objdump).",
    )
    parser.add_argument(
        "--context",
        type=int,
        default=2,
        help="Number of instructions before/after bottleneck to include (default:
2).",
    )
    parser.add_argument(
        "--verbose",
        action="store_true",
```

```

        help="Print commands before running them.",
    )
    return parser.parse_args()

def main() -> None:
    args = parse_args()
    ensure_tools_available(args.clang, args.xctrace, shlex.split(args.objdump)[0])

    tests = resolve_tests(args.tests_override)
    if not tests:
        sys.exit("No tests found to process.")

    if not TEMPLATE.exists():
        sys.exit(f"Trace template not found: {TEMPLATE}")

    for src in tests:
        print(f"Processing {src.name}")
        binary = compile_source(src, args.clang, args.verbose)
        asm_path = emit_assembly(binary, args.objdump, args.verbose)
        print(f"  binary:      {binary}")
        print(f"  assembly:    {asm_path}")

        trace_path = record_trace(binary, args.xctrace, args.verbose)
        profile_path = export_trace(trace_path, args.xctrace, args.verbose)
        print(f"  trace:       {trace_path}")
        print(f"  profile:     {profile_path}")

        summary = simplify_profile(profile_path)
        if summary:
            simplified_dir = ROOT / "profiles_simplified"
            simplified_dir.mkdir(parents=True, exist_ok=True)
            out_path = simplified_dir / f"{profile_path.stem}.json"
            with out_path.open("w", encoding="utf-8") as f:
                json.dump(summary, f, indent=2, sort_keys=True)
            print(f"  simplified: {out_path}")

        combined = build_training_data(args.context)
        write_training_data(combined)
        print(f"Wrote combined training data with {len(combined)} samples")

if __name__ == "__main__":
    main()

```

Код для тренування моделі (train_bottleneck_model.py):

```
#!/usr/bin/env python3
import argparse
from pathlib import Path

from sklearn.metrics import classification_report
from sklearn.model_selection import cross_val_score

from bottleneck_model import load_training_samples, make_pipeline, save_model

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="Train a bottleneck type classifier.")
    parser.add_argument(
        "--training-data",
        default="training_data.json",
        help="Path to training_data.json (default: training_data.json).",
    )
    parser.add_argument(
        "--model-out",
        default="model_artifacts/bottleneck_model.joblib",
        help="Where to write the trained model (default: model_artifacts/bottleneck_model.joblib).",
    )
    parser.add_argument(
        "--context",
        type=int,
        help="Context size to enforce. Defaults to the size inferred from the dataset.",
    )
    parser.add_argument(
        "--skip-cv",
        action="store_true",
        help="Skip cross-validation scoring.",
    )
    return parser.parse_args()

def main() -> None:
    args = parse_args()
    training_path = Path(args.training_data)
    model_out = Path(args.model_out)

    texts, labels, context_size = load_training_samples(training_path,
context=args.context)

    pipeline = make_pipeline()
    if not args.skip_cv:
        scores = cross_val_score(pipeline, texts, labels, cv=5, scoring="f1_macro")
        print(f"5-fold F1_macro mean={scores.mean():.3f}, std={scores.std():.3f}")
```

```
pipeline.fit(texts, labels)
preds = pipeline.predict(texts)
class_order = list(pipeline.named_steps["classifier"].classes_)
report = classification_report(labels, preds, labels=class_order,
target_names=class_order, digits=3)
print(report)

save_model(pipeline, context_size, model_path=model_out)
print(f"Saved model to {model_out} (context={context_size})")

if __name__ == "__main__":
    main()
```

Код реалізації моделі (bottleneck_model.py):

```

"""
Helpers for training and running a simple bottleneck detector on ARM64 assembly.

The model is intentionally lightweight: we normalize registers/immediates to
reduce vocabulary noise, featurize with character n-grams, and train a linear
classifier. Everything is kept in one place so training and inference share the
same preprocessing.
"""

from __future__ import annotations

import json
import re
from pathlib import Path
from typing import Iterable, Sequence, Tuple

import joblib
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC

# Default context matches current dataset (5 instructions -> context 2).
DEFAULT_CONTEXT_SIZE = 2
MODEL_PATH = Path("model_artifacts") / "bottleneck_model.joblib"
METADATA_PATH = MODEL_PATH.with_suffix(".meta.json")

def normalize_instruction(inst: str) -> str:
    """Canonicalize an instruction to reduce sparsity."""
    inst = inst.strip().lower()
    inst = re.sub(r"x\d+", "xreg", inst)
    inst = re.sub(r"w\d+", "wreg", inst)
    inst = re.sub(r"#0x[0-9a-f]+", "#imm", inst)
    inst = re.sub(r"#-?\d+", "#imm", inst)
    inst = re.sub(r"0x[0-9a-f]+", "0ximm", inst)
    inst = re.sub(r"\s+", " ", inst)
    return inst

def window_instructions(instructions: Sequence[str], context: int =
DEFAULT_CONTEXT_SIZE) -> list[list[str]]:
    """Return sliding windows with padding to match the training context."""
    windows: list[list[str]] = []
    size = 2 * context + 1
    for idx in range(len(instructions)):
        window = []
        for offset in range(idx - context, idx + context + 1):
            if 0 <= offset < len(instructions):
                window.append(instructions[offset])
            else:
                window.append("")

```

```

        # enforce expected length in case context is tampered with
        if len(window) < size:
            window.extend([""] * (size - len(window)))
        elif len(window) > size:
            window = window[:size]
        windows.append(window)
    return windows

def render_window(window: Sequence[str]) -> str:
    """Join and normalize a context window into model input text."""
    return " | ".join(normalize_instruction(item) for item in window)

def load_training_samples(path: Path, context: int | None = None) -> Tuple[list[str],
list[str], int]:
    """Load training_data.json into texts/labels."""
    raw = json.loads(path.read_text())
    if not raw:
        raise ValueError(f"No data found in {path}")
    inferred_context = (len(raw[0]["instructions"]) - 1) // 2
    context_size = context if context is not None else inferred_context
    if context_size != inferred_context:
        raise ValueError(
            f"Context mismatch: dataset has context {inferred_context}, requested "
            f"{context_size}"
        )

    texts: list[str] = []
    labels: list[str] = []
    for sample in raw:
        window = sample["instructions"]
        if len(window) != 2 * context_size + 1:
            raise ValueError(f"Unexpected window size {len(window)} in sample "
{sample}")
        texts.append(render_window(window))
        labels.append(sample.get("type") or "NONE")
    return texts, labels, context_size

def make_pipeline() -> Pipeline:
    """Build the text featurizer + classifier."""
    vectorizer = TfidfVectorizer(analyzer="char", ngram_range=(4, 7), lowercase=False)
    classifier = LinearSVC(class_weight="balanced")
    return Pipeline([("vectorizer", vectorizer), ("classifier", classifier)])

def train_pipeline(texts: Iterable[str], labels: Sequence[str]) -> Pipeline:
    pipeline = make_pipeline()
    pipeline.fit(list(texts), labels)
    return pipeline

def save_model(pipeline: Pipeline, context_size: int, *, model_path: Path =
MODEL_PATH) -> None:
    model_path.parent.mkdir(parents=True, exist_ok=True)
    joblib.dump(pipeline, model_path)

```

```

classes = list(pipeline.named_steps["classifier"].classes_)
metadata = {
    "context_size": context_size,
    "vectorizer": "char-level TF-IDF ngrams (4,7), normalized
registers/immediates",
    "classifier": "LinearSVC class_weight=balanced",
    "labels": classes,
}
meta_path = model_path.with_suffix(".meta.json")
meta_path.write_text(json.dumps(metadata, indent=2))

def load_model(model_path: Path = MODEL_PATH) -> Tuple[Pipeline, int, list[str]]:
    pipeline: Pipeline = joblib.load(model_path)
    meta_path = model_path.with_suffix(".meta.json")
    labels: list[str] = []
    if meta_path.exists():
        metadata = json.loads(meta_path.read_text())
        context_size = int(metadata.get("context_size", DEFAULT_CONTEXT_SIZE))
        labels = list(metadata.get("labels", []))
    else:
        context_size = DEFAULT_CONTEXT_SIZE
    if not labels and hasattr(pipeline.named_steps["classifier"], "classes_"):
        labels = list(pipeline.named_steps["classifier"].classes_)
    return pipeline, context_size, labels

```