НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«АВТОМАТИЗОВАНА ПЕРЕВІРКА ЛАБОРАТОРНИХ РОБІТ НА МОВІ PYTHON МЕТОДАМИ МАШИННОГО НАВЧАННЯ»

Виконав: студент 2 курсу

Рівень: магістр

Галузь знань 12 «Інформаційні технології»

Спеціальність 122 «Комп’ютерні науки»

Григор’єв Олександр Володимирович

Керівник: к.пед.н., доцент

Логінова Наталія Іванівна

Рецензент к.т.н.

Гура Володимир Ігорович

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Факультет **кібербезпеки та інформаційних технологій**

Кафедра **інформаційних технологій**

Галузь знань: **12 Інформаційні технології**

Спеціальність: **122 Комп'ютерні науки**

Назва освітньої програми: **Комп'ютерні науки**

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент _____ Н.І. Логінова
«____ »_____ 20__ року

ЗАВДАННЯ
на кваліфікаційну (магістерську) роботу
здобувачу Григор'єву Олександру Володимировичу

1. Тема роботи: «Автоматизована перевірка лабораторних робіт на мові Python методами машинного навчання»

Керівник роботи: к.пед.н, доцент Логінова Наталія Іванівна

затверджені наказом НУ ОЮА від «*10*» *жовтня 2025* року № *3182-06*

2. Строк подання здобувачем роботи «*25*» *листопада 2025 рік*

3. Дата видачі завдання «*02*» *червня 2025 рік*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми, вивчення літературних джерел та складання плану роботи		
2.	Підготовка першого розділу роботи та подання його керівнику		
3.	Підготовка другого розділу роботи та подання його керівнику		
4.	Підготовка третього розділу роботи та подання його керівнику		
5.	Підготовка вступу та висновків		
6.	Доопрацювання роботи з урахуванням зауважень керівника		
7.	Подача електронного варіанту роботи для перевірки на плагіат		
8.	Допуск роботи до захисту завідуючим кафедри		
9.	Захист роботи		

Здобувач _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота здобувача 2-го курсу магістратури Григор'єва Олександра Володимировича на тему «Автоматизована перевірка лабораторних робіт на мові Python методами машинного навчання» за спеціальності 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки».

Структура кваліфікаційної роботи складається зі вступу, основної частини з трьох розділів, висновків, списку використаних джерел. Робота викладена на 80 сторінках, містить 33 рисунка, 4 таблиці. Перелік посилань включає 45 джерел.

Мета проведеного дослідження – розробка та реалізація автоматизованої перевірки лабораторних робіт на мові Python із використанням методів машинного навчання, на основі концептуальної моделі, що забезпечує об'єктивне, ефективне та масштабоване оцінювання робіт здобувачів.

Об'єкт дослідження – процес автоматизованої перевірки лабораторних робіт з програмування у навчальному середовищі закладів вищої освіти.

Предмет дослідження – методи, алгоритми та програмні засоби машинного навчання, що забезпечують автоматичний аналіз, оцінювання та зворотний зв'язок при перевірці лабораторних робіт, виконаних мовою програмування Python.

В кваліфікаційній роботі проаналізовані традиційні підходи оцінювання та особливості автоматизованої перевірки студентського коду. На основі цього сформульовано вимоги та розроблено концептуальну модель системи. Детально обґрунтовано вибір методу машинного навчання, описано процес формування векторів ознак, побудову моделі та її навчання. У практичній частині представлено реалізацію модулів аналізу коду, формування навчальної вибірки та інтеграцію моделі машинного навчання у загальну архітектуру системи.

СИСТЕМА, МАШИННЕ НАВЧАННЯ, PYTHON, АНАЛІЗ, КОД, МОДЕЛЬ

ABSTRACT

Qualification work of the 2nd year master's student Oleksandr Hryhoriiev, titled “Automated Assessment of Python Laboratory Works Using Machine Learning Methods”, is completed within the specialty 122 “Computer Science”, educational program “Computer Science”.

The structure of the thesis consists of an introduction, a main part comprising three chapters, conclusions, and a list of references. The thesis is presented on 80 pages and includes 33 figures and 4 tables. The list of references contains 45 sources.

The aim of the research is to develop and implement an automated assessment system for Python laboratory works using machine learning methods based on a conceptual model that ensures objective, efficient, and scalable evaluation of students' work.

The object of the study is the process of automated assessment of programming laboratory works within the educational environment of higher education institutions.

The subject of the study includes methods, algorithms, and software tools of machine learning that enable automatic analysis, evaluation, and feedback generation for laboratory works developed in the Python programming language.

The qualification thesis analyzes traditional evaluation approaches and the specifics of automated assessment of student code. Based on this analysis, system requirements were formulated, and a conceptual model was developed. The choice of machine learning method is thoroughly justified, and the processes of feature vector formation, model construction, and training are described. The practical part presents the implementation of code analysis modules, the creation of the training dataset, and the integration of the machine learning model into the overall system architecture.

SYSTEM, MACHINE LEARNING, PYTHON, ANALYSIS, CODE, MODEL

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ACID – Atomicity, Consistency, Isolation, Durability

AST – Abstract Syntax Tree

CFG – Control Flow Graph

DFG – Data Flow Graph

LOC – Lines of Code

ML – Machine Learning, машинне навчання

SVM – Support Vector Machine

TF-IDF – Term Frequency – Inverse Document Frequency

TMC – Test My Code

БД – база даних

ІТ – інформаційні технології

СКБД – система керування базами даних

ШІ – штучний інтелект

ВСТУП

Актуальність кваліфікаційної роботи на тему «Автоматизована перевірка лабораторних робіт на мові Python методами машинного навчання» обумовлена інтенсивним розвитком цифрової освіти та зростанням ролі інтелектуальних технологій у навчальному процесі. Процес перевірки лабораторних робіт з програмування належить до найбільш ресурсоємних завдань у діяльності викладача, оскільки потребує значних часових і когнітивних витрат. Використання методів машинного навчання дозволяє створити інтелектуальну систему, здатну здійснювати автоматизований аналіз програмного коду, виявляти помилки, оцінювати стиль програмування та відповідність технічним вимогам. Це, у свою чергу, забезпечує підвищення ефективності, об'єктивності та якості освітнього процесу.

Мова Python, як одна з найпоширеніших мов програмування у світі та в академічному середовищі, визначає особливу актуальність обраної тематики. Автоматизація перевірки робіт на Python дозволяє не лише зменшити навантаження на викладачів, а й забезпечити студентів швидким, об'єктивним і змістовним зворотним зв'язком при виконанні лабораторних та практичних робіт з програмування. Такий підхід відповідає сучасним тенденціям розвитку адаптивного навчання, сприяє формуванню індивідуальних освітніх траєкторій та розвитку навичок самоконтролю. Особливістю даного дослідження є розробка концептуальної моделі системи автоматизованої перевірки, яка визначає її функціональну структуру та логіку взаємодії компонентів. Запропонована модель поєднує модулі аналізу коду, формування ознак, машинного навчання та генерації зворотного зв'язку. Такий підхід дозволяє забезпечити системність, масштабованість і обґрунтованість процесу оцінювання.

Мета дослідження – розробка та реалізація автоматизованої перевірки лабораторних робіт на мові Python із використанням методів машинного навчання, на основі концептуальної моделі, що забезпечує об'єктивне, ефективне та масштабоване оцінювання робіт здобувачів.

Для досягнення цієї мети передбачено наступні **завдання**:

– проаналізувати існуючі підходи та програмні рішення для автоматизованої

перевірки програмного коду;

– визначити основні критерії оцінювання якості лабораторних робіт з програмування;

– дослідити можливості застосування методів машинного навчання для автоматичного аналізу та класифікації програмного коду;

– розробити концептуальну модель та алгоритм автоматизованої перевірки лабораторних робіт на мові програмування Python;

– реалізувати прототип системи з використанням сучасних бібліотек машинного навчання та засобів аналізу коду;

– оцінити перспективи впровадження методики у навчальний процес і можливості її подальшої оптимізації.

Об’єкт дослідження – процес автоматизованої перевірки лабораторних робіт з програмування у навчальному середовищі закладів вищої освіти.

Предмет дослідження – методи, алгоритми та програмні засоби машинного навчання, що забезпечують автоматичний аналіз, оцінювання та зворотний зв’язок при перевірці лабораторних робіт, виконаних мовою програмування Python.

У процесі виконання кваліфікаційної роботи використовувалися як загальнонаукові, так і спеціальні методи наукового дослідження, що забезпечили комплексний підхід до вирішення поставлених завдань.

Загальнонаукові методи:

– аналіз і синтез – для вивчення існуючих систем автоматичної перевірки лабораторних робіт з програмування, виявлення їхніх переваг і недоліків, а також формування концепції власної моделі;

– індукція та дедукція – при формулюванні гіпотез щодо ефективності застосування машинного навчання для перевірки лабораторних робіт;

– моделювання – для побудови структури моделі автоматичної перевірки та визначення її функціональних компонентів.

Спеціальні методи:

– методи машинного навчання – зокрема класифікація, кластеризація та аналіз ознак, для створення моделі, здатної оцінювати лабораторні роботи з програмування.

Ці методи дозволили досягти наукової обґрунтованості результатів та забезпечити практичну цінність розробленої системи.

Практичне значення отриманих результатів: результати дослідження мають практичне значення, оскільки розроблена автоматизована перевірка лабораторних робіт з програмування на мові Python може бути безпосередньо впроваджена у навчальний процес. Її використання дозволить значно зменшити часові витрати викладачів на оцінювання, забезпечити об’єктивність перевірки та створити умови для швидкого зворотного зв’язку зі студентами. Модель сприяє підвищенню ефективності навчального процесу, оскільки здатна не лише перевіряти правильність виконання завдань, а й аналізувати стиль коду, виявляти типові помилки та пропонувати рекомендації для їх усунення. Отримані результати можуть бути використані для розробки інтерактивних навчальних платформ, онлайн-курсів та інтелектуальних систем підтримки навчання, що відповідають сучасним вимогам цифрової трансформації освіти.

Апробація матеріалів кваліфікаційно роботи здійснено на VI міжнародної науково-практичної конференції «Кібербезпека в сучасному світі: актуальні виклики» (28 листопада 2025 р., м. Одеса) [1].

Структура кваліфікаційної роботи відповідає меті та завданням дослідження та складається зі вступу, основної частини з трьох розділів, висновків, списку використаних джерел. Робота викладена на 80 сторінках, містить 33 рисунка, 4 таблиці. Перелік посилань включає 45 джерел.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Особливості автоматизованої перевірки коду лабораторних робіт на мові Python

Сучасний ринок ІТ-праці характеризується високою динамікою, міждисциплінарністю та потребою у фахівцях, здатних ефективно працювати з кількома мовами програмування, що дозволяє розв’язувати широкий спектр прикладних задач. Постійні зміни в рейтингах мов програмування ставлять перед вищими навчальними закладами виклик – вони повинні оперативно адаптувати навчальні програми відповідно до потреб ринку праці. Адже одним із ключових аспектів підготовки ІТ-фахівців є актуальність знань і навичок, які вони отримують під час навчання, та їхня відповідність вимогам роботодавців [2].

У зв’язку з цим, у процесі підготовки здобувачів за галуззю знань «Інформаційні технології» викладачам необхідно запроваджувати навчальні дисципліни, орієнтовані на вивчення кількох мов програмування, а також формування практичних навичок написання, тестування та налагодження коду.

Важливою складовою процесу навчання програмуванню є виконання студентами великої кількості лабораторних і практичних робіт, що передбачають індивідуальний підхід до оцінювання. Проте традиційні методи перевірки таких завдань є трудомісткими та вимагають значних часових витрат від викладачів. Ручна перевірка коду часто супроводжується суб’єктивними оцінками, залежними від людського фактору, що може негативно впливати на об’єктивність результатів навчання.

У контексті впровадження інформаційних технологій (ІТ) в освіту одним із ключових завдань є підвищення ефективності контролю знань здобувачів, зокрема під час вивчення дисциплін, пов’язаних із програмуванням. Впровадження автоматизованих систем перевірки коду є перспективним напрямом модернізації освітнього процесу, оскільки такі системи забезпечують швидкість, об’єктивність та уніфікованість оцінювання. Вони дозволяють зменшити навантаження на викладачів, підвищити мотивацію студентів та сприяти їх самостійному навчанню.

ІТ-ринок пропонує різні програмні рішення для перевірки програмних кодів,

наприклад:

CodeRunner – це безкоштовний плагін із відкритим кодом для Moodle, що працює у формі питань. Він може запускати програмний код, надісланий студентами у відповідь на широкий спектр програмних питань різними мовами. CodeRunner зазвичай використовується в адаптивному режимі тестів Moodle – студенти вставляють свій код у відповідь на кожне програмне питання та одразу бачать результати своїх тестових випадків. Потім вони можуть виправити свій код та повторно надіслати його [3].

CodeHS Autograders – це інструмент, вбудований у CodeHS, який тестує програми здобувачів для виявлення помилок у коді. Він запускається при перевірці коду. Якщо будь-який з тестів проходить невдало, буде запропоновано виправити код перед надсиланням. Програма використовує серію тестових випадків, щоб переконатися, що код є як функціонально, так і стилістично правильним під час досягнення мети заданої вправи [4].

Test My Code (TMC) – це автоматизована система перевірки коду здобувачів. Програма оцінює програмні завдання, порівнюючи роботу коду з очікуваним результатом. Перевагами є швидка зворотна реакція, зменшення ручної перевірки, об’єктивна оцінка. Підтримує різні типи тестів: юніт-тести, функціональні тести, перевірка формату виводу. TMC може інтегруватися у Moodle, GitHub Classroom, або локальними навчальними платформами [5].

Для визначення сильних та слабких сторін кожної системи доцільно провести порівняння розглянутих програмних продуктів (табл. 1.1).

Таблиця 1.1 – Порівняння систем автоматизованої перевірки коду

Система	**Плюси**	**Мінуси**	**Інтеграція**	**Мови програмування**
CodeRunner	– одразу показує результати тестів студенту	– лише для Moodle; – ручна настройка складних тестів; – обмежені можливості для стилістичної перевірки коду	Moodle	Багато мов

Продовження табл. 1.1

Система	Плюси	Мінуси	Інтеграція	Мови програмування
CodeHS Autograders	– автоматично перевіряє функціональність та стиль коду; – пропонує виправити помилки перед надсиланням; – інтерактивний фідбек	– прив'язаний до CodeHS (комерційний); – обмежена кастомізація; – невелика кількість підтримуваних мов	CodeHS	Обмежена кількість мов, залежить від платформи
TMC	– швидка зворотна реакція; – підтримує юніт-тести, функціональні тести, перевірку формату; – є можливість інтеграції з Moodle, GitHub Classroom або локальними платформами	– потрібна інтеграція та налаштування; – неінтерактивний фідбек у реальному часі	Moodle, GitHub Classroom, локальні платформи	Багато мов

Розглянути існуючі програмні продукти мають певні переваги, проте часто обмежені у функціональності, інтеграції та підтримці специфічних мов програмування або типів завдань.

Розробка власної моделі автоматизованої перевірки коду дозволить максимально адаптувати її під конкретні освітні потреби, забезпечити повний контроль над функціональністю. Вона усуває ризик суб'єктивності оцінювання, оскільки працює за чітко визначеними критеріями, забезпечуючи рівні умови для всіх студентів. Крім того, дає змогу студентам миттєво отримувати результати виконаних завдань, що сприяє розвитку самостійності, формуванню навичок самоконтролю та рефлексії. Для викладачів же це відкриває можливість більше часу приділяти аналітичному розбору типових помилок, поясненню теоретичного матеріалу і вдосконаленню методики викладання.

Впровадження моделі автоматизованої перевірки лабораторних робіт з програмування сприятиме оптимізації навчального процесу, підвищенню якості підготовки здобувачів освіти та формуванню у них сучасних професійних компетентностей у галузі ІТ.

1.2 Традиційні методи перевірки лабораторних робіт з програмування

Основною вимогою до системи вищої освіти є орієнтація на розвиток особистості, здатної ефективно розв’язувати професійні завдання. Одним із ключових засобів визначення параметрів навчального процесу виступає оцінювання, яке є невід’ємною складовою діагностики навчальних досягнень студентів. Контроль знань забезпечує зворотний зв’язок між викладачем і студентом та відіграє важливу роль у системі навчання. Посилення уваги до процесу оцінювання сприяє підвищенню ефективності всієї освітньої системи.

Завдання оцінювання полягають у:

– виявленні, перевірці та визначенні рівня знань, умінь і навичок здобувачів, а також у встановленні якості засвоєння ними навчального матеріалу з дисциплін на різних етапах навчання;

– визначенні рівня сформованості системи компетенцій особистості студента;

– оцінюванні відповідності змісту, форм, методів і засобів навчання цілям і завданням підготовки фахівців;

– стимулюванні систематичної самостійної роботи та розвитку пізнавальної активності студентів;

– виявленні й розвитку творчого потенціалу, підвищенні зацікавленості у вивченні навчального матеріалу;

– оцінюванні ефективності самостійної та індивідуальної роботи студентів, їхньої здатності користуватися навчальною, довідковою та методичною літературою;

– розробленні заходів щодо підвищення якості навчання шляхом упровадження інноваційних технологій у навчальний процес [6].

Традиційні методи перевірки лабораторних робіт з програмування передбачають активну участь викладача у процесі оцінювання кожного етапу виконання завдання. Основна увага приділяється аналізу коду, правильності алгоритмів, а також здатності здобувача самостійно пояснити принципи роботи своєї програми.

Одним із найпоширеніших методів оцінювання робіт здобувачів є ручна перевірка коду викладачем. Студент подає роботу у вигляді текстового файлу або архіву

з проєктом, після чого викладач відкриває її, запускає програму, перевіряє коректність виконання завдання та якість коду. Такий підхід дозволяє глибоко проаналізувати логіку програми, стиль написання та використані програмні конструкції. Однак він має суттєвий недолік – значні часові витрати при великій кількості студентів, а також наявність суб'єктивного чинника під час оцінювання.

Іншим поширеним способом є усна демонстрація лабораторної роботи. У цьому випадку студент демонструє роботу програми, пояснює структуру коду, алгоритми та відповідає на запитання викладача. Цей метод сприяє перевірці самостійності виконання роботи, але також є трудомістким і складно масштабованим для великих груп.

Ще одним традиційним підходом є перевірка звітів, які містять текстовий опис алгоритму, вихідний код, результати виконання та висновки. Така форма звітності дозволяє оцінити рівень теоретичного розуміння завдання, однак не гарантує, що студент дійсно виконав програму самостійно або що вона працює коректно.

У деяких випадках застосовується порівняння з еталонним розв'язком. Викладач має зразок правильної програми, з якою порівнюється студентський код або результати виконання. Цей метод частково дозволяє об'єктивізувати оцінку, проте не враховує можливість існування кількох правильних способів розв'язання задачі.

Додатково використовується взаємоперевірка робіт студентами, що може бути ефективним при формуванні навичок аналізу чужого коду. Однак така перевірка вимагає чітких критеріїв оцінювання і не може бути основним методом через ризик суб'єктивності.

Для оцінки традиційних методів перевірки лабораторних робіт використовуємо SWOT-аналіз, який є методом стратегічного планування. Він виявляє фактори внутрішнього і зовнішнього середовища організації та розподіляє їх на чотири категорії: сильні сторони, слабкі сторони, можливості та загрози. Для проведення такого аналізу створюють матрицю та розміщують SWOT-фактори [7].

На рис. 1.1 наведено SWOT-аналіз традиційних методів перевірки лабораторних робіт з програмування.

Сильні сторони	Слабкі сторони
– глибокий якісний аналіз – контекстна перевірка – перевірка розуміння студента – гнучкість оцінювання	– тривалість і трудомісткість – суб’єктивність оцінювання – низька масштабованість – труднощі з відтворенням перевірки – можливість академічного плагіату
Можливості	**Загрози**
– гібридизація з автоматизацією – стандартизація критеріїв – використання peer-review – інтеграція інструментів контролю плагіату – підвищення якості зворотного зв’язку	– зростання навантаження – часові та ресурсні обмеження – технологічне відставання – юридичні та етичні ризики

Рисунок 1.1 – SWOT-аналіз традиційних методів перевірки лабораторних робіт з програмування

Сильними сторонами перевірки лабораторних робіт з програмування традиційними методами є:

– глибокий якісний аналіз – викладач може детально оцінити логіку, стиль коду, архітектурні рішення;

– контекстна перевірка – можливість відсіяти неправильні підходи, надати кваліфікований фаховий коментар;

– перевірка розуміння студента – усна демонстрація чи питання дозволяють виявити, чи студент справді розуміє рішення;

– гнучкість оцінювання – можна врахувати творчі та нестандартні рішення, які важко автоматизувати.

Слабкими сторонами є:

– тривалість і трудомісткість – значні часові витрати викладача при великій групі;

– суб’єктивність оцінювання – ризик упередженості оцінок;

– низька масштабованість – складно використати при великій кількості студентів або частих лабораторних;

– труднощі з відтворенням перевірки – важко повторно відтворити повний процес перевірки (наприклад, точні тести, середовища виконання);

– можливість академічного плагіату – складно контролювати запозичення коду

без додаткових інструментів.

До можливостей перевірки лабораторних робіт з програмування традиційними методами можна віднести:

– гібридизація з автоматизацією – поєднання ручної перевірки й автоматичних тестів для економії часу та підвищення об’єктивності;

– стандартизація критеріїв – розробка чеклістів для зменшення суб’єктивності;

– використання peer-review – навчити студентів оцінювати роботи одне одного за чіткими критеріями (зменшить навантаження на викладача й підвищить навчальний ефект);

– інтеграція інструментів контролю плагіату – зниження ризику академічного плагіату;

– підвищення якості зворотного зв’язку – формальні шаблони коментарів, відео-демонстрації помилок.

Загрозами таких методів є:

– зростання навантаження – збільшення розміру груп або вимог до частоти оцінювання може зробити традиційні методи непридатними;

– часові та ресурсні обмеження – нестача годин викладача та обмежені лабораторні потужності;

– технологічне відставання – якщо не впроваджувати автоматизацію, курс може відставати від сучасних практик оцінювання;

– ризики – проблеми з обробкою персональних даних при збиранні матеріалів для перевірки, або конфлікти через нечесну оцінку.

Таким чином, традиційні методи перевірки лабораторних робіт є ефективними з точки зору глибини аналізу, але малопридатними для масового навчання через значне навантаження на викладача та високий рівень суб’єктивності оцінювання. Це створює передумови для розробки та впровадження автоматизованої перевірки програмних кодів, здатних забезпечити об’єктивність, швидкість та уніфікацію процесу оцінювання.

1.3 Аналіз алгоритмів машинного навчання для автоматизації процесу

Машинне навчання (Machine Learning, ML) є підгалуззю штучного інтелекту (ШІ), що займається розробкою алгоритмів та статистичних моделей, здатних навчати комп'ютерні системи на основі даних, здійснювати прогнозування та приймати рішення [8].

Процес роботи ML складається з кількох послідовних етапів, кожен із яких відіграє важливу роль у створенні ефективної моделі (рис. 1.2).

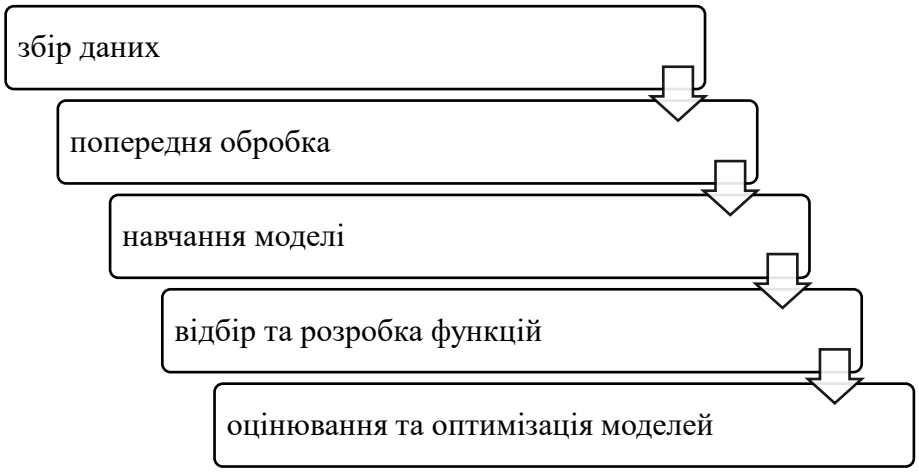

Рисунок 1.2 – Етапи машинного навчання

Першим кроком є збір даних, які можуть надходити з різних джерел. Від якості та обсягу зібраних даних залежить успішність подальшого навчання моделі. Після отримання даних здійснюється попередня обробка, метою якої є очищення інформації від шумів, перевірка її повноти та підготовка до подальшого аналізу. Цей етап забезпечує придатність даних до використання в алгоритмах ML. Далі відбувається навчання моделі, під час якого обраний алгоритм аналізує підготовлені дані та навчається робити прогнози або приймати рішення на їх основі. Важливою складовою цього процесу є відбір і розробка ознак – визначення найбільш інформативних характеристик з початкового набору даних, які суттєво впливають на продуктивність і точність моделі. Після етапу навчання здійснюється оцінювання та оптимізація моделі, що дозволяє перевірити її відповідність заданим критеріям якості. У разі потреби параметри моделі коригуються для досягнення кращих результатів. Заключним етапом є розгортання та моніторинг, коли навчена модель впроваджується у реальне

середовище для вирішення практичних завдань. У процесі експлуатації здійснюється контроль її роботи та, за необхідності, повторне навчання з оновленими даними, щоб підтримувати актуальність і стабільність результатів.

Впровадження ШІ в освітній процес надає численні переваги, серед яких персоналізація навчання, інтеграція чат-ботів, оперативний зворотний зв'язок та аналітика освітніх даних. Використання технологій ШІ дозволяє досягти рівня диференційованого підходу, який недоступний при традиційному викладанні з великими групами студентів [9].

Однією з ключових проблем викладання дисциплін з програмування є забезпечення об'єктивної, швидкої та якісної перевірки лабораторних робіт, особливо в умовах зростання чисельності студентів та поширення дистанційного навчання. Як показав аналіз в п. 1.2, традиційні методи оцінювання стають малоефективними через значні витрати часу та людських ресурсів, що зумовлює необхідність високого рівня автоматизації процесу перевірки.

ШІ здатен автоматизувати процес оцінювання навчальних завдань та забезпечувати миттєвий зворотний зв'язок для студентів, що значно полегшує роботу викладача та активізує навчальну діяльність здобувачів освіти, що забезпечує низку переваг:

– об'єктивність оцінювання: системи на базі ШІ застосовують стандартизовані алгоритми та критерії, що мінімізує суб'єктивний вплив, характерний для ручної перевірки;

– часові та економічні вигоди: автоматизовані оцінювальні платформи дозволяють швидко та ефективно оцінювати велику кількість студентів або проводити тестування, скорочуючи витрати часу і ресурсів закладів освіти;

– надійність: ШІ не піддається емоціям чи втомі, забезпечує стабільність оцінювання та здатен виявляти помилки, які можуть залишитися непоміченими при ручній перевірці;

– надання зворотного зв'язку: такі системи інформують студентів про їхній рівень знань і навичок та пропонують додаткові завдання або матеріали для опрацювання тем, у яких є прогалини;

– прогнозування успішності: на основі даних попередніх тестів і завдань ШІ може передбачати потребу студентів у додатковій підтримці та визначати тих, хто демонструє високі результати, що дозволяє викладачам своєчасно коригувати навчальний процес [10].

ML – це галузь ШІ, яка зосереджена на розробці моделей та алгоритмів, що дозволяють комп'ютерам навчатися на основі даних без необхідності їх явного програмування для кожного завдання [11].

Класичне ML зазвичай класифікують відповідно до способу, у який алгоритм удосконалює свою точність прогнозування. Виокремлюють його чотири основні типи:

– навчання з учителем;

– навчання без учителя;

– напівнавчання з учителем;

– навчання з підкріпленням.

Вибір конкретного алгоритму зумовлюється характеристиками наявних даних. Слід зазначити, що багато методів ML не обмежуються виключно одним типом – їх можна адаптувати до різних підходів залежно від специфіки задачі та структури даних.

Навчання з учителем передбачає використання маркованих даних, де чітко визначено як вхідні, так і вихідні параметри. Алгоритм намагається знайти закономірності між ними, формуючи функцію, здатну узагальнювати нові приклади. Цей підхід застосовується для таких типів задач:

1. Бінарна класифікація – розподіл об’єктів на дві категорії.

2. Багатокласова класифікація – класифікація об’єктів у межах кількох класів.

3. Ансамблеве моделювання – поєднання результатів декількох моделей для підвищення точності прогнозу.

4. Регресійне моделювання – прогнозування неперервних значень на основі виявлених залежностей між змінними.

Навчання без учителя не потребує маркованих даних. У цьому випадку алгоритм самостійно аналізує структуру даних, виявляючи приховані закономірності,

групи або залежності. До типових завдань такого підходу належать:

1. Кластеризація – поділ даних на групи за подібністю.

2. Виявлення аномалій – ідентифікація нетипових або відхилених спостережень.

3. Аналіз правил асоціації – пошук частих шаблонів або співвиникнень елементів у наборі даних.

4. Зменшення розмірності – оптимізація кількості ознак для зменшення складності аналізу.

Напівнавчання з учителем поєднує риси двох попередніх підходів: алгоритм навчається на невеликій кількості маркованих даних і використовує отримані знання для аналізу більшого обсягу немаркованих даних. Такий підхід забезпечує баланс між точністю навчання з учителем і економічністю навчання без учителя. Напівнавчання ефективно застосовується у таких сферах, як:

1. Машинний переклад – побудова моделей перекладу на основі частково маркованих мовних даних.

2. Виявлення шахрайства – виявлення підозрілих операцій за обмеженої кількості позитивних прикладів.

3. Автоматичне маркування даних – генерація міток для великих наборів даних на основі моделі, навченої на малому обсязі маркованої інформації.

Навчання з підкріпленням базується на принципі взаємодії агента з середовищем, у межах якого агент отримує винагороди або покарання залежно від правильності виконаних дій. Мета алгоритму полягає у формуванні оптимальної стратегії поведінки, що максимізує сумарну винагороду. Цей підхід застосовується для таких завдань, як:

1. Навчання роботів виконанню дій у реальному середовищі.

2. Створення ботів, здатних ефективно грати у відеоігри.

3. Оптимізація розподілу ресурсів у бізнес-процесах.

Для визначення оптимального методу ML для системи автоматизованої перевірки лабораторних робіт необхідно провести порівняльний аналіз найбільш поширених алгоритмів класифікації коду. Критеріями відбору виступають: точність

класифікації, швидкість навчання та передбачення, інтерпретованість результатів, стійкість до перенавчання та можливість роботи з різними ознаками.

Логістична регресія (Logistic Regression) – це статистична модель, яка використовується для прогнозування ймовірності бінарного результату на основі однієї або кількох незалежних змінних. Її основна мета в машинному навчанні полягає в класифікації даних за різними категоріями та розуміння взаємозв'язку між незалежними та результатними змінними [12].

Метод опорних векторів (Support Vector Machine, SVM) – це алгоритми, що використовуються для допомоги моделям машинного навчання з учителем у розділенні різних категорій даних шляхом встановлення чітких меж між ними. Він призначений для створення меж рішень для точної класифікації. Це один з ключових методів, які фахівці з обробки даних використовують для створення моделей штучного інтелекту та машинного навчання з широким спектром практичних застосувань, включаючи розпізнавання зображень, виявлення шахрайства та фільтрацію спаму. SVM добре обробляють високорозмірні дані. Вони також можуть захищати від надмірного налаштування, коли модель добре виконує прогнози, використовуючи дані, на яких вона була навчена, але погано справляється з новими даними [13].

Випадковий ліс (Random Forest) – це алгоритм машинного навчання, який поєднує результати кількох дерев рішень для отримання єдиного результату. Простота використання та гнучкість сприяли його поширенню, оскільки він вирішує як задачі класифікації, так і задачі регресії. Алгоритм є розширенням методу пакетування, оскільки він використовує як пакетування, так і випадковість ознак для створення некорельованого лісу дерев рішень. Випадковість ознак, також відома як пакетування, генерує випадкову підмножину ознак, що забезпечує низьку кореляцію між деревами рішень. Це ключова відмінність між деревами рішень та випадковими лісами. У той час як дерева рішень враховують усі можливі розбиття ознак, випадкові ліси вибирають лише підмножину цих ознак [14].

Нейронні мережі – це математична модель, що працює за принципом людського мозку та навчається шляхом первинного опрацювання великого набору даних, не вимагаючи написання окремого коду під конкретне завдання. Нейромережі є одним зі

способів ML, і лежать в основі алгоритмів глибокого навчання [15].

Градієнтне бустинг (Gradient Boosting) – це алгоритм ML, який використовується для завдань регресії та класифікації. Алгоритм став популярним завдяки своїй здатності обробляти складні взаємозв'язки в даних та захищати від перенавчання. Використовуючи цей метод, фахівці з обробки даних можуть покращити точність прогнозування та швидкість своїх моделей ML. Алгоритм об'єднує колекцію слабких моделей в єдину, більш точну та ефективну прогностичну модель. Ці слабкі моделі зазвичай є деревами рішень, тому алгоритми зазвичай називають деревами рішень з градієнтним бустингом. Алгоритми працюють ітеративно, послідовно додаючи нові моделі, причому кожне нове додавання спрямоване на усунення помилок, допущених попередніми. Остаточний прогноз сукупності являє собою суму окремих прогнозів усіх моделей. Градієнтне бустинг поєднує алгоритм градієнтного спуску та метод бустингу, з посиланням на кожен компонент, включений у його назву. Градієнтне бустинг використовується для вирішення складних задач регресії та класифікації. За допомогою регресії кінцевий результат являє собою середнє значення всіх слабких учнів. Під час роботи з задачами класифікації кінцевий результат моделі можна обчислити як клас з більшістю голосів від моделей слабких учнів [16].

Порівняємо розглянути методи ML, які можна використовувати для задачі автоматизованої перевірки програмного коду (таблиця 1.2).

Таблиця 1.2 – Порівняльний аналіз методів машинного навчання

Метод	**Точність класифікації**	**Швидкість роботи**	**Інтерпретованість**	**Стійкість до шуму**	**Робота з різнорідними ознаками**	**Придатність для задачі**
Логістична регресія	Середня (75-82%)	Висока	Висока	Низька	Середня	Обмежена
SVM	Висока (82-88%)	Середня	Низька	Висока	Висока	Підходить
Random Forest	Висока (85-92%)	Висока	Висока	Дуже висока	Дуже висока	Оптимальна
Нейронні мережі	Дуже висока (88-95%)	Низька	Дуже низька	Середня	Висока	Надлишкова складність
Gradient Boosting	Висока (87-93%)	Середня	Середня	Висока	Висока	Підходить

З таблиці 1.2 випливає, щодо виконання дослідження, найбільш збалансованим рішенням є алгоритм Random Forest, оскільки він поєднує високу точність класифікації з прийнятною швидкістю роботи, забезпечує інтерпретованість результатів через механізм важливості ознак та демонструє високу стійкість до шумових даних, що характерно для студентських робіт з різним рівнем якості коду.

Застосування алгоритмів ML відкриває можливість побудови концептуальної моделі автоматизованої перевірки лабораторних робіт на мові Python, здатної адаптуватися до нових типів завдань і навчатися на основі попередніх перевірок. Це дозволить не лише автоматизувати технічну частину перевірки, а й підвищити зворотній зв’язок, забезпечуючи персоналізовані рекомендації для кожного здобувача.

1.4 Вимоги до концептуальної моделі автоматизованої перевірки лабораторних робіт

Розробка концептуальної моделі автоматизованої перевірки лабораторних робіт з програмування на мові Python передбачає реалізацію інтелектуального аналізу програмного коду з використанням методів машинного навчання. Така модель має забезпечити об’єктивне, ефективне та масштабоване оцінювання студентських робіт шляхом комплексного аналізу правильності, структури, стилю та логіки коду.

Для побудови моделі необхідно чітко визначити вимоги, які слугують основою для проектування та реалізації функціональних компонентів.

Загальні вимоги до концептуальної моделі, призначеної для перевірки лабораторних робіт з програмування, передбачають забезпечення автоматизованого контролю правильності виконання програмних завдань студентами. Модель повинна зберігати історію всіх спроб здачі робіт, результати перевірок і забезпечувати прозорість процесу оцінювання. Крім того, модель повинна забезпечувати високу швидкість обробки запитів і перевірки програмного коду, що особливо важливо при одночасній роботі великої кількості студентів. В моделі повинно бути розмежування ролей користувачів – здобувач та викладач.

Функціональні вимоги описують конкретні дії, поведінку та можливості. Вони визначають, що саме робитиме модель – від взаємодії з користувачем до її внутрішніх

реакцій:

– визначають чіткі завдання, які повинні виконуватися;

– показують взаємодію користувачів;

– описують, як модель буде реагувати на вхідні дані або події;

– відповідають за обробку даних.

Нефункціональні вимоги зосереджуються на якості роботи. Вони охоплюють такі характеристики, як продуктивність, безпека, масштабованість і надійність, забезпечуючи не лише правильне виконання функцій, а й високий рівень зручності та ефективності використання [17].

Функціональні вимоги концептуальної моделі автоматизованої перевірки лабораторних робіт визначають можливості розв'язувати поставлені завдання:

1. Реєстрація та автентифікація користувачів – підтримка різних ролів користувачів (студент, викладач) та розмежування доступу до функціональних можливостей.

2. Завантаження лабораторних робіт – студенти повинні мати можливість завантажувати вихідний код лабораторної роботи у визначеному форматі, які будуть зберігатися до завершення дедлайну.

3. Автоматичне тестування – запускає програми та перевірка за наперед визначеним набором тестових випадків. Перевірка включає – коректність вхідних даних та результатів виводу, обробку граничних умов, перевірку часу виконання та використання пам'яті.

4. Оцінювання та аналіз результатів – після виконання тестів формується звіт, що містить: результати проходження кожного тесту; кількість набраних балів; коментарі щодо помилок або порушень стандартів оформлення коду. Викладач може додавати додаткові коментарі або коригувати оцінку вручну.

5. Адміністрування – викладач має можливість створювати нові лабораторні завдання, додавати або редагувати тестові дані, встановлювати терміни здачі, а також переглядати статистику успішності студентів.

Графічно функціональні вимоги подано у вигляді діаграми варіантів використання на рис. 1.3, яка демонструє взаємодію акторів із системою та відображає

основні сценарії її використання.

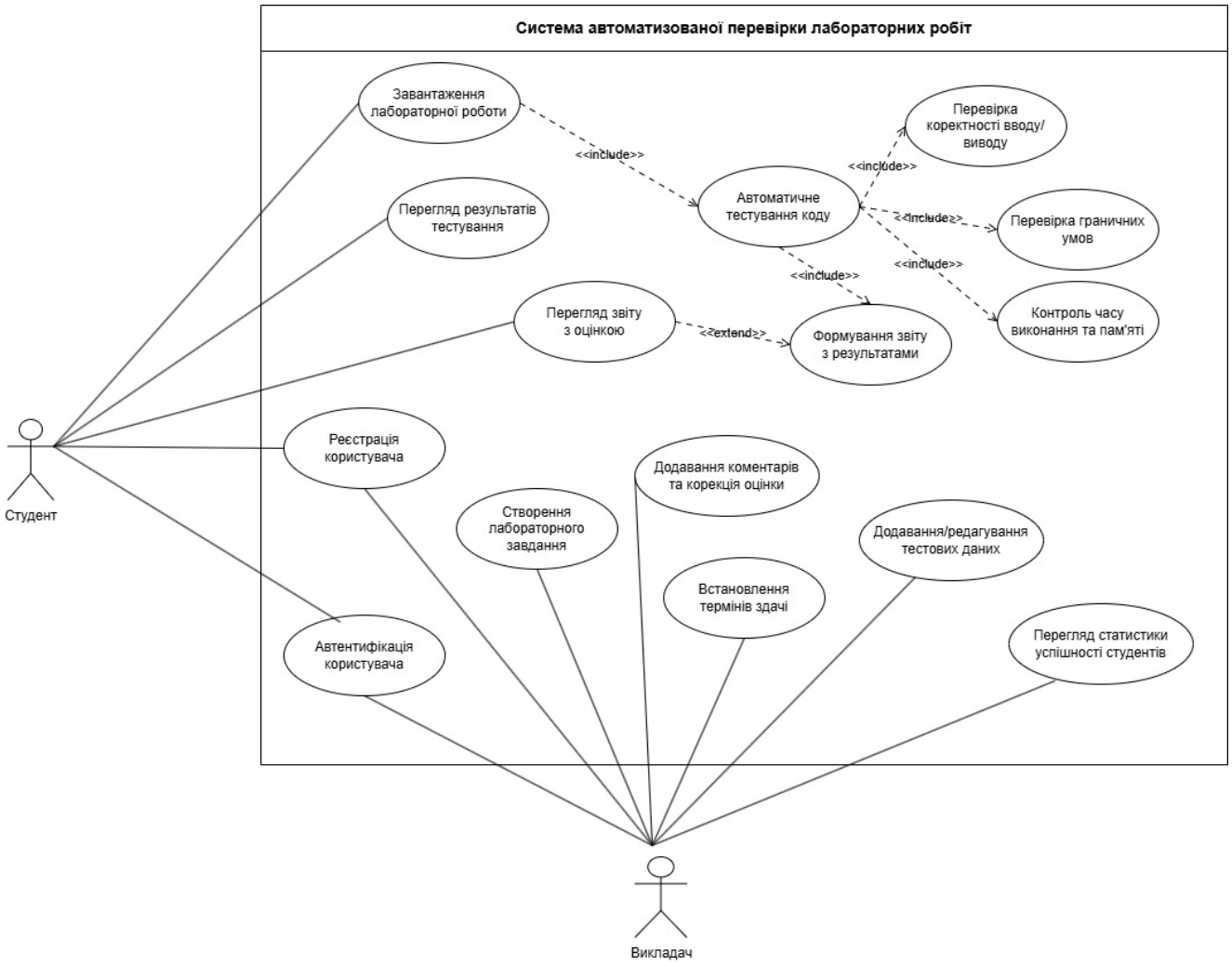

Рисунок 1.3 – Діаграма варіантів використання

Діаграма відображає логіку автоматизованого процесу перевірки лабораторних робіт, де викладач виконує також адміністративні функції, а система бере на себе технічну перевірку та оцінювання.

До нефункціональних вимог модель автоматизованої перевірки лабораторних робіт можна віднести:

– безпеку – ізолювати виконання користувацького коду для запобігання несанкціонованому доступу до серверних ресурсів;

– продуктивність – швидке реагування на запити та перевірку коду;

– масштабованість – можливість одночасної перевірки великої кількості робіт;

– надійність – автоматичне відновлювання після збоїв та зберігати всі результати.

Таким чином, вимоги до автоматизованої перевірки лабораторних робіт сформовані на основі концептуальної моделі, що забезпечує узгодженість між функціональними компонентами, логікою їх взаємодії та потребами освітнього процесу. Реалізація такої моделі дозволить створити повноцінний цикл перевірки – від завантаження коду до формування звітів і аналітики, сприяючи підвищенню якості навчання програмуванню.

Висновки до розділу 1

У першому розділі кваліфікаційної роботи теоретично обґрунтовано принципи функціонування системи автоматизованої перевірки лабораторних завдань. Підкреслено, що виконання лабораторних і практичних робіт є ключовим елементом навчання програмування, який потребує індивідуального та об’єктивного оцінювання. Традиційні методи є трудомісткими, потребують значних часових витрат і схильні до суб’єктивності через людський фактор.

Автоматизовані системи перевірки розглядаються як ефективний засіб модернізації освітнього процесу, що забезпечує оперативність, об’єктивність і стандартизованість оцінювання, зменшує навантаження на викладачів і підвищує мотивацію здобувачів.

Проведено порівняльний аналіз традиційних і ML-методів оцінювання. SWOT-аналіз традиційних підходів дозволив виявити їхні сильні та слабкі сторони, а аналіз ML-методів показав їхню здатність підвищувати точність, швидкість і аналітичність оцінювання.

На основі аналізу сформульовано функціональні та нефункціональні вимоги до системи. Для реалізації інтелектуального компонента обрано алгоритм Random Forest, що забезпечує оптимальний баланс між точністю, швидкодією та інтерпретованістю. Для візуалізації функціональних характеристик побудовано діаграму варіантів використання, яка стала основою розробки концептуальної моделі у другому розділі.

2 МОДЕЛЮВАННЯ ТА ОБҐРУНТУВАННЯ МЕТОДУ МАШИННОГО НАВЧАННЯ

2.1. Архітектура концептуальної моделі

Для побудови концептуальної моделі автоматизованої перевірки лабораторних робіт, написаних мовою програмування Python, із використанням методів ML потрібно проаналізувати логіку процесу, яку можна представити алгоритмом (рис. 2.1).

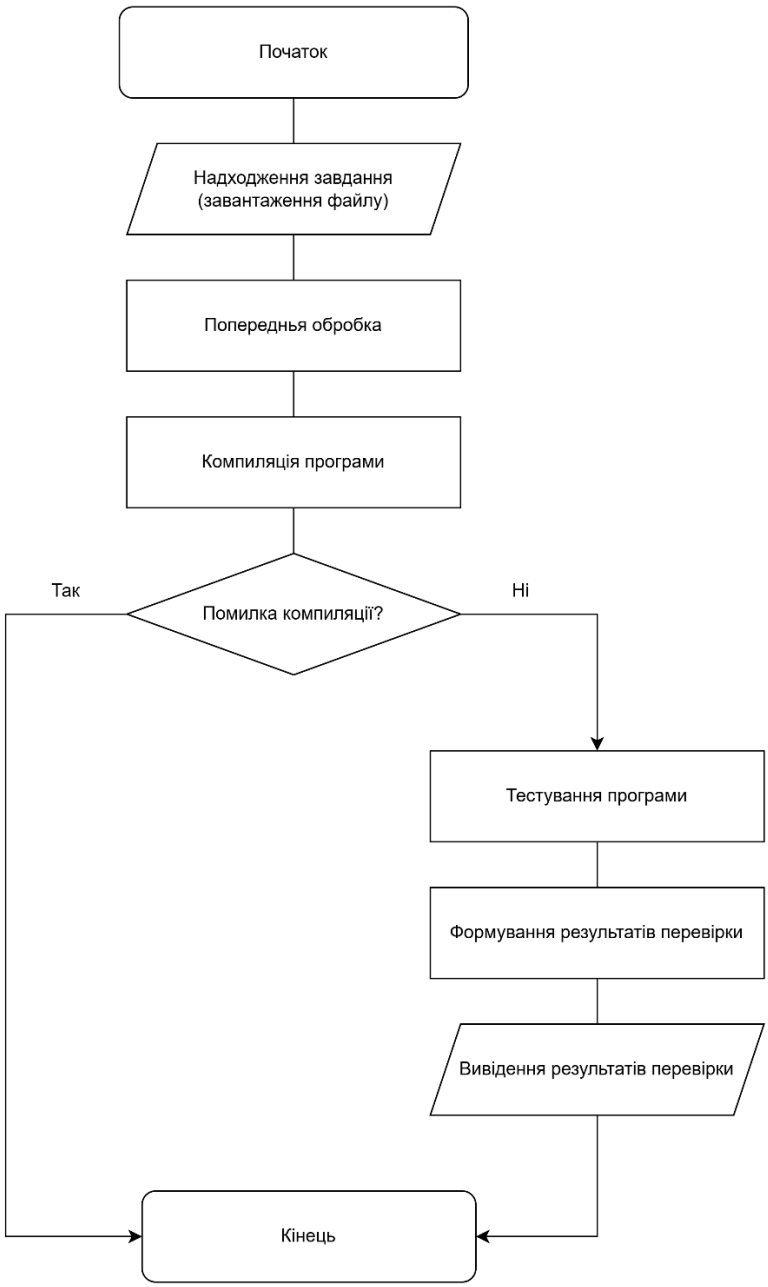

Рисунок 2.1 – Алгоритм загальної логіки процесу автоматизованої перевірки лабораторних робіт з програмування

Процес перевірки програмного коду починається з того, що студент завантажує свій файл. Після цього, завдання надходить до системи перевірки через API або модуль інтеграції.

На етапі попередньої обробки відбувається перевірка формату файлів і цілісності завдання, витяг та нормалізація коду, що включає видалення коментарів, форматування та уніфікацію відступів, а також виконання статичного аналізу для перевірки синтаксису та виявлення можливих помилок компіляції. Далі відбувається виділення ознак із коду. Програмний код токенізується і перетворюється в абстрактне синтаксичне дерево, після чого формується набір ознак – структурних, семантичних або статистичних, які зберігаються у базі даних для подальшого аналізу.

На етапі перевірки виконання код запускається у безпечному середовищі, після чого порівнюються вихідні дані програми з еталонними тестами. Результати виконання, такі як успішність тестів, час роботи та використання ресурсів, фіксуються для подальшого оцінювання. Оцінювання коду здійснюється за допомогою методів машинного навчання. Вектор ознак передається у модель ML, яка визначає правильність логіки рішення, рівень якості коду, стиль та ефективність, а також можливі схожості з іншими роботами для виявлення плагіату. На основі цього формується підсумковий бал. Після цього генерується звіт із результатами перевірки, що включає кількість пройдених тестів, виявлені помилки, оцінку моделі машинного навчання та рекомендації для покращення. Результати зберігаються у базі даних і надсилаються здобувачу.

На основі розробленого алгоритму можливо сформувати структурну модель системи автоматизованої перевірки лабораторних робіт із програмування, що складається з кількох взаємопов’язаних компонентів:

– інтерфейс користувача – забезпечує подання результатів перевірки;

– модуль отримання робіт – відповідає за приймання файлів від студентів;

– модуль попередньої обробки коду – виконує синтаксичний аналіз і нормалізацію програмного коду;

– модуль ML – здійснює інтелектуальний аналіз, класифікацію та оцінювання коду;

– модуль формування результатів – створює звіти, оцінки та рекомендації;

– база даних – зберігає інформацію про користувачів, завдання та результати перевірок.

Для відображення взаємозв’язків між основними компонентами використовується діаграма компонентів (рис. 2.2).

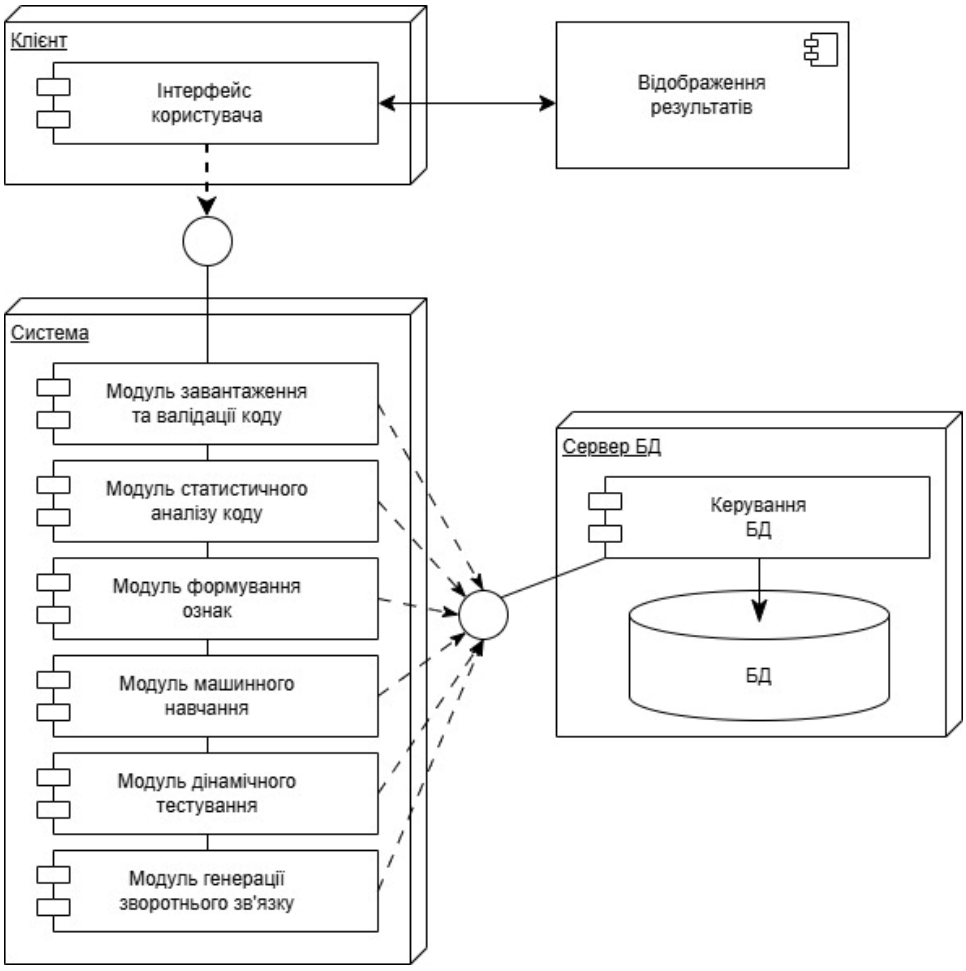

Рисунок 2.2 – Діаграма компонентів

Діаграма компонентів ілюструє архітектуру концептуальної моделі автоматизованої перевірки лабораторних робіт, яка включає три ключові підсистеми: клієнтську частину, серверну частину та сервер бази даних (БД).

Процес автоматизованої перевірки лабораторної роботи реалізується як послідовність взаємопов’язаних етапів, у межах яких взаємодіють усі основні компоненти системи – від інтерфейсу користувача до бази даних.

Для візуалізації логіки процесу перевірки лабораторної роботи створено діаграму послідовності (рис. 2.3).

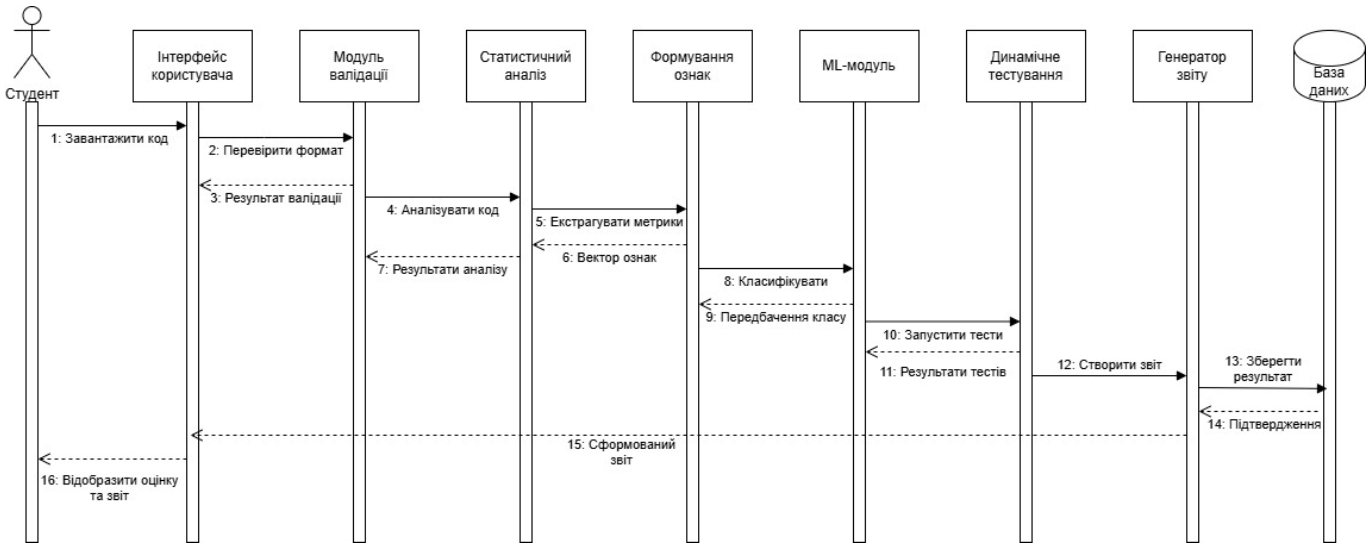

Рисунок 2.3 – Діаграма послідовності

На діаграмі послідовності показано, як студент взаємодіє із системою автоматизованої перевірки лабораторних робіт. Спочатку студент через інтерфейс користувача завантажує файл із програмним кодом, який надходить до модуля завантаження, де файл передається для перевірки формату. Модуль валідації перевіряє правильність структури та розширення файлу, а після успішної перевірки передає його до модуля статичного аналізу. На цьому етапі система аналізує код, будує дерево синтаксичних структур (Abstract Syntax Tree, AST) та обчислює основні метрики, такі як кількість функцій або рівень вкладеності. Отримані дані потрапляють до модуля формування вектору ознак, який створює числове представлення коду для подальшої обробки. Далі сформований вектор ознак надходить до модуля машинного навчання, де модель типу Random Forest виконує попередню класифікацію роботи, визначаючи ймовірну оцінку. Після цього система переходить до динамічного тестування, під час якого код виконується з тестовими даними, а результати порівнюються з очікуваними. Зібрана інформація з усіх етапів – статичного аналізу, машинного навчання та динамічного тестування – передається до модуля генерації звітів, де формується підсумковий звіт, який містить деталі перевірки та оцінку студента. Після цього результати зберігаються в базі даних, а інтерфейс користувача відображає студенту підсумкову оцінку та короткий звіт. Таким чином, діаграма відображає послідовну взаємодію між користувачем і всіма внутрішніми модулями системи – від моменту завантаження файлу до отримання результату.

Подальший розгляд зосереджено на компонентах моделі, яка визначає засоби реалізації описаної архітектури, включно з вибором мови програмування та інструментів ML.

2.2. Програмні засоби реалізації моделі

У сучасних інформаційних системах ефективне зберігання, оброблення та аналіз даних забезпечується за допомогою потужних інструментів, серед яких особливе місце посідають PostgreSQL і Python. Їх поєднання створює гнучке, надійне та продуктивне середовище для розроблення системи автоматизованої перевірки лабораторних робіт із програмування.

PostgreSQL – це об’єктно-реляційна система керування базами даних (СКБД) із відкритим кодом, що поєднує надійність традиційних реляційних БД із підтримкою сучасних типів даних і розширюваністю. Її основними перевагами є дотримання принципів ACID, що гарантує цілісність транзакцій, розвинені механізми забезпечення зв’язності даних через первинні й зовнішні ключі, обмеження та тригери, а також висока продуктивність при виконанні складних аналітичних запитів [18].

Ключовим чинником, який робить PostgreSQL зручним для реалізації системи автоматизованого оцінювання, є підтримка типу JSONB. Цей формат дозволяє ефективно зберігати напівструктуровані дані, зокрема абстрактні синтаксичні дерева, вектори ознак або журнали виконання тестів, без втрати можливостей індексації та швидкого пошуку. Таким чином, PostgreSQL об’єднує риси реляційних і NoSQL-сховищ, забезпечуючи єдину, цілісну базу для роботи як зі структурованими, так і з напівструктурованими даними [19].

З іншого боку, Python є універсальною мовою програмування, яка поєднує простоту синтаксису, потужність виразних засобів і широку екосистему бібліотек для аналізу, машинного навчання та оброблення даних [20, 21].

У контексті автоматизованої перевірки лабораторних робіт з програмування такі технології можуть бути застосовані для вдосконалення процесу оцінювання – зокрема, для визначення рівня складності завдань, класифікації типових помилок здобувачів або прогнозування їхніх навчальних результатів [22]. Отже, Python виступає

не лише засобом формування базових компетентностей у програмуванні, але й платформою для реалізації інтелектуальних компонентів систем контролю знань, що є надзвичайно актуальним у впровадження ІТ в навчальний процес.

Мова програмування Python включає широкий спектр бібліотек, які значно спрощують розробку проєктів у сфері ML, до яких відносяться [23]:

NumPy – бібліотека використовується для ефективної роботи з багатовимірними масивами та матрицями, що містить широкий набір високорівневих математичних функцій та є базовим інструментом для виконання наукових обчислень [24].

SciPy – бібліотека призначена для числового аналізу, яка надає модулі для оптимізації, лінійної алгебри, інтегрування та статистики, активно використовується в аналітичних і ML-застосунках [25].

Scikit-learn – одна з популярних бібліотек класичного ML, побудована на основі NumPy та SciPy. Підтримує більшість алгоритмів навчання з учителем та без учителя, а також містить засоби для попередньої обробки даних [26].

TensorFlow – фреймворк з відкритим кодом для побудови й навчання глибоких нейронних мереж, який підтримує високопродуктивні обчислення з використанням тензорів та апаратного прискорення [27].

Keras – високорівнева бібліотека для побудови нейронних мереж, що працює поверх TensorFlow і надає зручний API для швидкого проєктування, навчання й тестування моделей [28].

PyTorch – бібліотека з відкритим кодом, орієнтована на наукові дослідження, що підтримує динамічне побудування обчислювальних графів і прискорення обчислень на GPU. Використовується для задач комп’ютерного зору, обробки природної мови та інших сфер глибокого навчання [29].

Застосування цих бібліотек дозволяє організувати ефективну роботу з багатовимірними масивами, виконувати складні математичні обчислення, будувати статистичні моделі та реалізовувати алгоритми машинного навчання.

Окрім ML-інструментів, у Python існує низка засобів аналізу та оцінювання якості коду, серед яких – ast, pylint, radon, flake8 тощо.

Модуль ast забезпечує взаємодію з абстрактним синтаксичним деревом, яке є

структурним представленням вихідного коду. Завдяки цьому модулю можливо аналізувати й трансформувати код без його виконання, що є основою роботи лінтерів, форматерів і транспайлерів [30].

Pylint – інструмент для статичного аналізу коду, який перевіряє відповідність стандартам кодування, виявляє потенційні помилки та невідповідності без необхідності запуску програми [31].

Radon – інструмент для збору метрик коду, який дозволяє оцінити цикломатичну складність, метрики Холстеда, кількість рядків коду, коментарів, порожніх рядків тощо. Цикломатична складність, також відома як число Маккейба, визначає кількість незалежних шляхів виконання коду, що є важливим показником для оцінки тестованості програми [32].

Flake8 – потужний інструмент для лінтингу, який поєднує перевірку синтаксису, дотримання стандартів PEP 8 та виявлення потенційних помилок у коді [33].

Таким чином, використання зазначених інструментів дає змогу автоматизувати процеси аналізу та оцінювання якості програмного коду, що є невід'ємною складовою автоматизованої перевірки лабораторних робіт.

Мова програмування Python завдяки своїй універсальності, широкому набору бібліотек і аналітичних інструментів є ефективним середовищем для реалізації модуля статистичного аналізу коду. Її можливості охоплюють як класичні методи оцінювання якості програм, так і сучасні підходи на основі ML. Використання таких засобів, як ast, pylint, radon та flake8, дозволяє автоматизувати процеси аналізу структури, стилю та складності коду, підвищуючи об'єктивність і точність оцінювання. У поєднанні з ML-бібліотеками Python створює основу для розроблення інтелектуальних модулів, здатних до глибокого статистичного аналізу й прогнозування результатів навчання, що робить його незамінним інструментом у сучасних системах автоматизованого контролю знань.

Поєднання PostgreSQL і Python є особливо ефективним завдяки їхній взаємній сумісності та розвиненій інтеграції. Python має низку бібліотек для взаємодії з PostgreSQL (наприклад, psycopg2), що забезпечує зручне виконання SQL-запитів, обробку результатів та реалізацію транзакцій безпосередньо в коді застосунку. Така

інтеграція дає змогу створювати повноцінні аналітичні та навчальні системи, у яких дані зберігаються в PostgreSQL, а логіка оброблення, аналізу й ML реалізується засобами Python.

Отже, використання PostgreSQL як сховища даних і Python як середовища аналітики та реалізації бізнес-логіки забезпечує високу гнучкість, масштабованість та інтелектуальність розроблюваної системи. Ця комбінація дає змогу не лише надійно зберігати результати перевірок і статистику виконання завдань, але й реалізовувати складні алгоритми аналізу коду та класифікації студентських робіт, що робить її оптимальним технічним рішенням для систем автоматизованого контролю знань.

2.3. Концептуальна модель даних системи

Концептуальна модель даних системи автоматизованої перевірки лабораторних робіт з програмування складається із наступних сутностей (табл. 2.1).

Таблиця 2.1 – Сутності концептуальної моделі

Номер сутності	Назва сутності	Атрибути сутності
1	Користувачі (users)	Код користувача Прізвище Ім’я Роль (студент, викладач) Email Дата реєстрації
2	Лабораторні роботи (lab_work)	Код роботи Назва Опис завдання Тип завдання Критерій оцінювання Бал Термін здачі Код викладача
3	Студенти (students)	Код студента Код лабораторної роботи Текст вихідного коду Дата та час завантаження Версія (номер спроби) Статус (завантажено, перевіряється, перевірено)

Продовження таблиці 3.1

Номер сутності	Назва сутності	Атрибути сутності
4	Результати перевірки (check_result)	Код результату Код студента Оцінка (бал) Клас якості (0-3: правильна, частково правильна, неправильна з помилками, повністю неправильна) Дата та час перевірки Детальний звіт (JSON)
5	Тестова перевірка (test_case)	Код тесту Код лабораторної роботи Вхідні дані Очікуваний результат Вага тесту (у відсотках від загальної оцінки) Тип тесту (функціональний, граничний, стресовий)
6	Ознаки коду (code_features)	Код ознак Код студента Лексичні ознаки (JSON) Синтаксичні ознаки (JSON) Семантичні ознаки (JSON) Поведінкові ознаки (JSON) Вектор TF-IDF

Модель даних побудована за принципами реляційної БД та використовує наступні зв’язки між сутностями:

Користувач ↔ Студенти (1:M): один студент може завантажити багато версій коду, кожен код належить одному студенту.

Лабораторні роботи ↔ Студенти (1:M): одна лабораторна робота має багато рішень від різних студентів, кожен код відноситься до однієї лабораторної роботи.

Користувач (викладач) ↔ Лабораторні роботи (1:M): один викладач може створити багато лабораторних робіт, кожна робота створена одним викладачем.

Лабораторні роботи ↔ Тестова перевірка (1:M): одна робота має багато тестових перевірок, кожен тест належить одній роботі.

Студенти ↔ Результат перевірки (1:1): кожен код має один результат перевірки, кожен результат відповідає одному коду.

Студенти ↔ Ознаки коду (1:1): для кожного коду формується один набір ознак, кожен набір ознак відповідає одному коду.

На рис. 2.4 представлено ER-діаграму, яка ілюструє дані сутності та їх зв’язки

у вигляді концептуальної моделі.

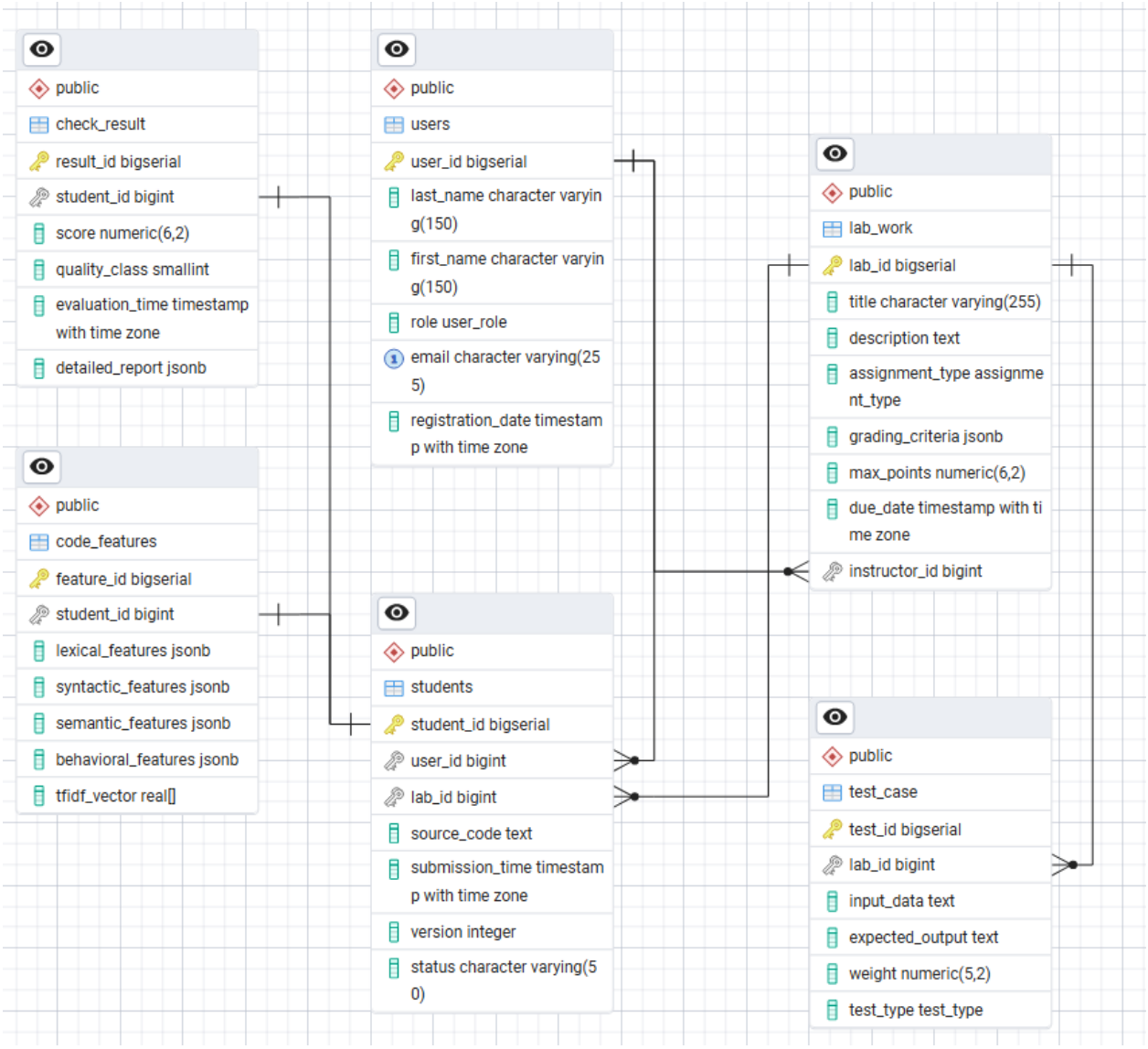

Рисунок 2.4 – ER-діаграма

У центрі моделі знаходиться сутність `students`, що репрезентує студентські подання коду для перевірки. Вона пов’язана з кількома іншими сутностями, які забезпечують повний цикл оброблення, аналізу та оцінювання робіт.

Сутність `users` містить інформацію про всіх користувачів системи – як студентів, так і викладачів. До її атрибутів належать унікальний ідентифікатор користувача, прізвище, ім’я, роль, електронна пошта та дата реєстрації. Через зовнішній ключ `user_id` таблиця `students` пов’язана з `users`, що дозволяє визначати, який студент виконав конкретну роботу.

Сутність `lab_work` описує лабораторні завдання, які створюють викладачі.

Вона містить такі поля, як назва, опис, тип завдання, критерії оцінювання, максимальний бал і термін здачі. Кожна лабораторна робота пов’язана з викладачем через атрибут `instructor_id`, який посилається на сутність `users`, і з таблицею `students` через `lab_id`, що визначає, яку саме роботу виконав студент.

Сутність `test_case` зберігає тестові сценарії для перевірки студентських програм. Вона включає вхідні дані, очікуваний результат, вагу тесту у загальній оцінці та його тип (функціональний, граничний або стресовий). Кожен тестовий приклад належить певній лабораторній роботі (через `lab_id`), що забезпечує структурованість тестування.

Сутність `check_result` відображає результати автоматизованої перевірки поданого студентом коду. Вона містить оцінку у балах, клас якості (правильна, частково правильна, з помилками або неправильна), час перевірки та детальний звіт у форматі JSON. Кожен результат пов’язаний із конкретним студентським поданням через student_id, що дозволяє простежити історію спроб і підсумкові результати.

Окремо виділено сутність `code_features`, яка зберігає набір ознак коду, отриманих у процесі статичного або динамічного аналізу. Вона пов’язана з `students` через `student_id`, що забезпечує відповідність ознак конкретному фрагменту коду.

Таким чином, модель відображає повний життєвий цикл автоматизованої перевірки: від створення завдання викладачем і подання коду студентом до тестування, аналізу, оцінювання та збереження результатів. Така структура забезпечує логічну узгодженість, масштабованість і можливість інтеграції з модулями ML для автоматизованої оцінки якості програмного коду.

2.4. Формування векторів ознак для машинного навчання

Ознака – це вимірювана характеристика даних, яка використовується як вхідний параметр для алгоритмів ML. Вона може бути числовою, категоріальною або текстовою та відображає суттєві властивості об’єкта аналізу.

Виділення інформативних ознак є ключовим етапом ML, оскільки саме вони визначають здатність моделі виявляти закономірності в даних. Для аналізу програмного коду необхідно перетворити текст вихідних файлів у числові або структуровані

представлення, що зберігають синтаксичну, логічну та семантичну інформацію [33].

Ознаки можуть описувати код на різних рівнях: лексичному, синтаксичному, семантичному та поведінковому.

Лексичні ознаки відображають структуру коду. Для Python до таких ознак належать:

– токени – мінімальні одиниці коду, розпізнавані інтерпретатором (ключові слова, ідентифікатори, літерали, оператори та роздільники);

– ключові слова (`if`, `for`, `def`, `class`);

– ідентифікатори (імена змінних, функцій, класів);

– константи (числові, рядкові, логічні);

– оператори та роздільники (+, =, (), :);

– коментарі та відступи, що впливають на структуру та читабельність коду [35].

Лексичний аналіз виконується шляхом токенізації, тобто перетворення вихідного коду у послідовність токенів, із видаленням непотрібних символів, таких як пробіли, коментарі та символи нового рядка. Токенізація дозволяє виявляти синтаксичні помилки на ранніх етапах та підготувати дані для подальшого аналізу [36].

На рис. 2.5 наведено фрагмент коду мовою програмування Python та результат його токенізації.

```
x = 3
if x > 5:
    print("Значення більше за 5")
```

Токен	Тип токена
x	ідентифікатор
=	оператор присвоєння
3	константа
if	ключове слово
x	ідентифікатор
>	оператор порівнювання

Токен	Тип токена
5	константа
:	розділювач
print	вбудована функція
(	розділовий символ
"Значення більше за 5"	рядковий літерал
)	розділовий символ

Рисунок 2.5 – Токенізація фрагменту коду мовою програмування Python

Для кращого урахування локальних залежностей між токенами застосовують n-грамний підхід, де `n`-грам – підпослідовностей з `n` елементів, виділених із текстового фрагмента коду програми, що дозволяє враховувати як одиничні лексичні одиниці, так і їхні поєднання:

– уніграми відображають окремі токени;

– біграми – локальні зв’язки між ними;

– тріграми – короткі конструкції та патерни стилю коду.

Для фрагменту коду, наведеного на рис. 2.4, визначимо n-грам (рис. 2.6).

```
unigram:  [['if'], ['x'], ['>'], ['5:'], ['print("Значення'],
['більше'], ['за'], ['5")']]
bigram:  [['if', 'x'], ['x', '>'], ['>', '5:'], ['5:',
'print("Значення'], ['print("Значення', 'більше'], ['більше', 'за'],
['за', '5")']]
trigram:  [['if', 'x', '>'], ['x', '>', '5:'], ['>', '5:',
'print("Значення'], ['5:', 'print("Значення', 'більше'],
['print("Значення', 'більше', 'за'], ['більше', 'за', '5")']]
```

Рисунок 2.6 – n-грами фрагменту коду мовою програмування Python

Застосування n-грам дозволяє моделі захоплювати повторювані патерні та стилістичні особливості програмування, що особливо корисно для аналізу студентських лабораторних робіт.

Синтаксичні ознаки визначають структуру програми та виділяються за допомогою AST, яке представляє код у вигляді ієрархії операторів і виразів [37].

Структуроване подання у вигляді AST широко використовується у статичних аналізаторах коду, інструментах автоматичної оптимізації, лінтерах, компіляторах та форматерах. Розуміння роботи з AST дозволяє розробникам глибше інтерпретувати внутрішню структуру програм, ідентифікувати потенційні недоліки та формувати підґрунтя для вдосконалення якості та продуктивності програмного забезпечення [38].

Фрагмент коду з рис. 2.5 представлено у вигляді AST за допомогою бібліотеки `ast` (рис. 2.7).

```
Module
  Assign: targets=['x'], value=3
    Name: id=x
      Store
    Constant: value=3
  If
    Compare
      Name: id=x
        Load
      Gt
      Constant: value=5
    Expr
      Call: func=print
        Name: id=print
          Load
        Constant: value=Значення більше за 5
```

Рисунок 2.7 – AST фрагменту коду програми

Виведені типи кожного вузла AST. Для змінних (Name), констант (Constant), операторів присвоєння (Assign) і викликів функцій (Call) показується значення або ім’я. Дане зображення показує ієрархію дерева через відступи, що робить структуру коду наочною.

Семантичні ознаки описують логіку виконання програми. Для їх аналізу будуються:

– граф потоку даних (Data Flow Graph, DFG) – показує залежності між змінними та обчисленнями;

– граф потоку керування (Control Flow Graph, CFG) – відображає можливі шляхи виконання програми [39].

Для ефективного використання моделей ML необхідно представити код у вигляді числових векторів фіксованої розмірності. Цей процес називається вбудовуванням (embedding) і забезпечує компактне представлення токенів, інструкцій, функцій або цілих програм, збереження синтаксичних та семантичних особливостей. Вбудовування дозволяє автоматизувати пошук коду, класифікацію, прогнозування дефектів, автодоповнення та рефакторинг. Генеруються спеціалізованими нейронними мережами, навченими на великих масивах коду, із можливістю точного налаштування під конкретну мову програмування. Процес включає токенізацію коду, обробку

мережею та отримання щільного векторного представлення, яке можна використовувати для подальшого аналізу та машинного навчання [40].

Сучасні підходи поступово переходять від ручного виділення ознак до автоматичного навчання векторних представлень, що дозволяє моделі захоплювати складну семантику коду, залежності між змінними та стиль розробника.

Для систематичного аналізу лабораторних робіт здобувачів на мові програмування Python процес виділення ознак можна подати наступними кроками (рис. 2.8).

Рисунок 2.8 – Процес виділення ознак з лабораторних робіт

Лабораторні роботи збираються як окремі скрипти. Виконується базове очищення коду: видалення порожніх рядків та коментарів. Розбиття коду на токени: ключові слова, ідентифікатори, константи, оператори та роздільники. Формування n-грам для фіксації локальних патернів та стилю коду. Створення абстрактного синтаксичного дерева, що відображає структуру коду, включаючи блоки, умови та цикли. Побудова DFG для відстеження залежностей між обчисленнями. Побудова CFG для формалізації логіки виконання та всіх можливих шляхів виконання програми. Використання нейронних мереж для перетворення токенів, AST, DFG та CFG у щільні вектори фіксованої довжини. Отримані вектори зберігають синтаксичні, семантичні та контекстні характеристики коду. Вектори ознак зберігаються у форматах, придатних для

навчання моделей (тензори, масиви, CSV). Вектори використовуються для класифікації, аналізу помилок, оцінки стилю коду та рекомендацій щодо рефакторингу. Ця структура дозволяє чітко показати шлях від коду студентів до готового векторного представлення для аналізу.

На прикладі коду (рис. 2.9) розглянемо ці етапи:

```
x = 3
if x > 5:
    print("Значення більше за 5")
else:
    print("Значення менше або дорівнює 5")
```

Рисунок 2.9 – Приклад коду програми лабораторної роботи

Крок 1. Лексичний аналіз (токенізація)

Токенізуємо код на окремі елементи:

Токен	**Тип токена**
x	ідентифікатор
=	оператор присвоєння
3	константа
if	ключове слово
x	ідентифікатор
>	оператор порівняння
5	константа
:	роздільник
print	вбудована функція
"Значення більше за 5"	рядковий літерал
else	ключове слово
:	роздільник
print	вбудована функція
"Значення менше або дорівнює 5"	рядковий літерал

Будуємо `n`-грам:

Уніграми: `x, =, 3, if …`

Біграми: `(x, =), (=, 3), (if, x) …`

Тріграми: `(x, =, 3), (if, x, >) …`

Крок 2. Синтаксичний аналіз (AST)

Перетворимо код у дерево абстрактного синтаксису (рис. 2.10).

```
Module
  Assign: targets=['x'], value=3
    Name: id=x
      Store
    Constant: value=3
  If
    Compare
      Name: id=x
        Load
      Gt
      Constant: value=5
    Expr
      Call: func=print
        Name: id=print
          Load
        Constant: value=Значення більше за 5
    Expr
      Call: func=print
        Name: id=print
          Load
        Constant: value=Значення менше або дорівнює 5
```

Рисунок 2.10 – Дерево абстрактного синтаксису

Module – корінь, весь скрипт.

Assign – присвоєння `x = 3`.

If – умовна конструкція, із блоками `then` і else.

Compare – вираз `x > 5`.

Expr – виклики функцій print.

Крок 3. Семантичний аналіз (DFG та CFG)

DFG: `x = 3` → `x > 5` → `print(...)`

`x = 3` ініціалізує змінну.

`x > 5` залежить від значення `x`.

`print()` залежить від результату умови.

CFG (рис. 2.11):

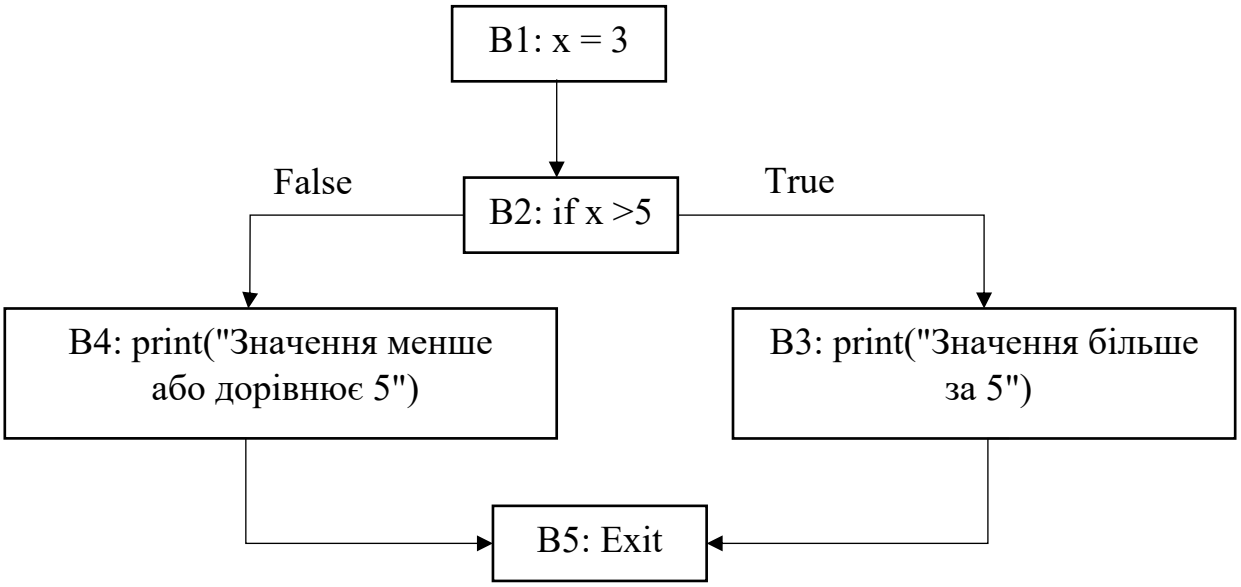

Рисунок 2.11 – Граф потоку керування

B1 → B2 – ініціалізація та перевірка умови;

B2 → B3 – якщо умова істинна;

B2 → B4 – якщо умова хибна;

B3/B4 → B5 – завершення програми.

Крок 4. Вбудовування (Embedding)

Токени та AST перетворюються в числові вектори, наприклад:

x→ [0.12, 0.34, 0.58]

= → [0.01, 0.97, 0.23]

CFG і DFG можуть бути представлені як графові вектори, де кожен вузол коду отримує числовий опис залежностей. Вектори об’єднуються в єдиний вектор програми, який можна використовувати для класифікації, перевірки стилю або виявлення помилок.

Процес аналізу коду включає лексичний, синтаксичний, семантичний аналіз і вбудовування. Це дозволяє не тільки зрозуміти структуру та логіку коду, а й підготувати його для автоматизованої обробки та аналізу машинним способом.

Для забезпечення ефективної роботи модуля ML система повинна виконувати аналіз програмного коду за наступними категоріями ознак:

Лексичні ознаки:

– кількість токенів у коді;

– частка ідентифікаторів серед токенів;

– кількість літералів (числових, рядкових, логічних);

– кількість операторів;

– кількість порожніх рядків;

– співвідношення токенів різних типів (ідентифікатори, оператори, літерали).

Синтаксичні ознаки:

– кількість рядків коду (Lines of Code, LOC);

– глибина вкладеності блоків (максимальний рівень відступів);

– частота використання ключових слів (if, for, while, def, class);

– співвідношення коду до коментарів;

– дотримання стандартів оформлення (PEP 8 для Python).

Семантичні ознаки:

– правильність логіки розв'язку задачі;

– ефективність використаних алгоритмів;

– коректність обробки граничних випадків;

– наявність обробки виключень;

– використання відповідних структур даних.

Поведінкові ознаки:

– результати проходження тестових випадків;

– час виконання програми;

– використання оперативної пам'яті;

– коректність виводу даних;

– стабільність роботи програми.

Також потрібно врахувати метрики якості класифікації – це кількісні показники, що дозволяють оцінити ефективність роботи моделі машинного навчання.

Оцінювання ефективності роботи системи класифікації програмного коду здійснюється за допомогою стандартних метрик якості машинного навчання. Кожна з них відображає різні аспекти продуктивності моделі, що дозволяє комплексно оцінити її здатність до правильного розпізнавання результатів [41].

Для обчислення метрик використовується матриця помилок (confusion matrix), елементами якої є:

TP (True Positive) – кількість правильно класифікованих позитивних випадків;

TN (True Negative) – кількість правильно класифікованих негативних випадків;

FP (False Positive) – кількість помилково класифікованих позитивних випадків (помилка I роду);

FN (False Negative) – кількість помилково класифікованих негативних випадків (помилка II роду).

На основі цих значень визначаються такі метрики:

Загальна точність класифікації (Accuracy) – частка всіх правильно класифікованих об’єктів серед загальної кількості:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (2.1)$$

Точність позитивних передбачень (Precision) – частка об'єктів, класифікованих як позитивні, що дійсно є позитивними:

$$Precision = \frac{TP}{TP+FP} \quad (2.2)$$

Повнота (Recall) – частка реально позитивних об'єктів, що були правильно ідентифіковані моделлю:

$$Recall = \frac{TP}{TP+FN} \quad (2.3)$$

F1-міра (F1-score) – гармонічне середнє між Precision та Recall, що забезпечує збалансовану оцінку якості класифікації:

$$F1 = \frac{2 \times Precision \times Recall}{Precision+Recall} \quad (2.4)$$

Для розроблюваної системи автоматизованої перевірки лабораторних робіт з програмування встановлюються такі мінімальні вимоги до якості класифікації:

- точність класифікації (accuracy) – не менше 80 %;
- точність (precision) – не менше 82 %;
- повнота (recall) – не менше 80 %;
- F1-міра (F1-score) – не менше 77 %.

Задані порогові значення відповідають високому рівню якості класифікації, прийнятому у системах автоматизованого оцінювання, та забезпечують баланс між

надійністю оцінок і здатністю виявляти всі правильно виконані роботи.

Окрім якісних показників, важливою характеристикою є оперативність обробки даних. Система має забезпечувати своєчасне надання результатів перевірки без помітних затримок, що особливо актуально під час одночасного тестування великої кількості студентських робіт.

Встановлено такі часові вимоги:

– середній час обробки однієї лабораторної роботи – не більше 30 секунд;

– при послідовній обробці групи з 25-30 студентів загальний час не перевищуватиме 12,5-15 хвилин;

– при одночасному зверненні кількох користувачів система повинна підтримувати паралельну обробку запитів без деградації продуктивності понад 10%.

Дотримання цих вимог гарантує достатню швидкодію та придатність системи для використання в навчальному процесі, де важливими є як якість класифікації, так і оперативність зворотного зв'язку для здобувачів.

Сформовані вектори ознак використовуються для навчання моделі Random Forest, обраної у розділі 1 як оптимальний алгоритм класифікації лабораторних робіт. Алгоритм Random Forest ефективно обробляє різнорідні ознаки без необхідності їх нормалізації, що робить його придатним для вирішення задач автоматизованої перевірки програмного коду.

2.5. Обґрунтування методу машинного навчання

Для реалізації інтелектуального компонента системи автоматизованої перевірки лабораторних робіт з програмування доцільно застосовувати методи ML, орієнтовані на задачі класифікації. Як показано у розділі 1 (табл. 1.1), найбільш ефективним для даної задачі є алгоритм Random Forest, що поєднує високу точність класифікації (85-92%) з прийнятною швидкістю роботи, забезпечує інтерпретованість результатів та демонструє високу стійкість до шумових даних.

Алгоритм Random Forest є ансамблевим методом навчання з учителем, який ґрунтується на побудові множини незалежних дерев рішень. Навчання з учителем передбачає використання навчальної вибірки, у якій для кожного прикладу відомий

результат перевірки (мітка класу). У випадку лабораторних робіт з програмування мітки можуть відповідати рівню правильності виконання, типу помилки або оцінці за бальною шкалою.

Кожне дерево навчається на випадковій підвибірці даних (технологія bootstrap sampling), а під час формування кожного вузла враховується лише випадкова підмножина ознак. Остаточне рішення моделі визначається голосуванням дерев для класифікації або усередненням їхніх прогнозів для регресії [42].

Вибір алгоритму Random Forest для задачі автоматизованої перевірки програмного коду обумовлений низкою переваг:

1. Робота з різнорідними ознаками

У процесі аналізу програмного коду, як описано у п. 2.3, використовуються чотири категорії ознак: лексичні, синтаксичні, семантичні та поведінкові. Даний алгоритм ефективно обробляє такі неоднорідні дані без необхідності їх нормалізації або стандартизації, на відміну від багатьох інших алгоритмів машинного навчання.

2. Висока інтерпретованість

Алгоритм Random Forest дозволяє визначити важливість кожної ознаки, що надає можливість аналізу того, які характеристики коду найбільше впливають на кінцевий результат оцінювання. Це особливо важливо у контексті навчального процесу, де пояснюваність моделі сприяє об'єктивності оцінки та дозволяє викладачам зрозуміти логіку прийняття рішень системою.

3. Стійкість до шумів та помилок

Лабораторні роботи здобувачів можуть містити як незначні синтаксичні помилки, так і логічні неточності. Завдяки ансамблевій природі Random Forest є менш чутливим до таких відхилень, ніж окремі класифікатори (наприклад, окреме дерево рішень або логістична регресія).

4. Масштабованість та ефективність

Модель може бути легко розпаралелена, що забезпечує швидку обробку великої кількості робіт навіть за умови збільшення обсягу даних. Це дозволяє ефективно використовувати метод у реальних умовах, коли система обробляє роботи групи з 25–30 студентів.

5. Гнучкість у налаштуванні

Основні параметри моделі (кількість дерев `n_estimators`, глибина дерева `max_depth`, мінімальна кількість зразків у листі `min_samples_leaf`) дозволяють збалансувати точність і швидкодію системи відповідно до конкретних вимог навчального закладу.

Отже, вибір алгоритму Random Forest для задачі автоматизованої перевірки програмного коду є обґрунтованим завдяки його здатності ефективно працювати з різнорідними ознаками, високій інтерпретованості результатів, стійкості до шумів та помилок, а також можливості масштабування і гнучкого налаштування. Сукупність цих переваг робить Random Forest оптимальним вибором для побудови надійної, поясненої та продуктивної системи оцінювання студентських лабораторних робіт.

Модель для автоматизованої перевірки лабораторних робіт побудована за принципом послідовного конвеєра (`pipeline`), що поєднує етапи попередньої обробки коду, його векторизації та класифікації результатів (рис. 2.12).

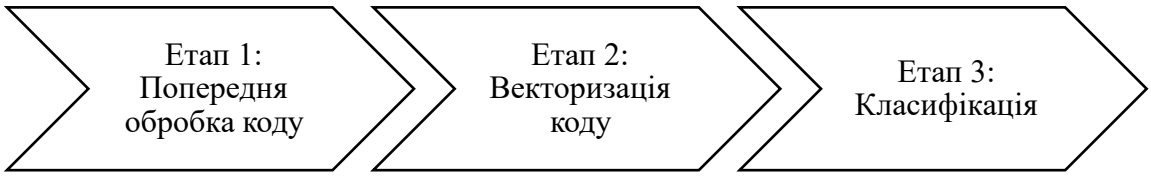

Рисунок 2.12 – Етапи роботи моделі

На початковому етапі здійснюється очищення програмного коду від коментарів, уніфікація імен змінних і функцій, а також нормалізація форматування. Така обробка дає змогу усунути вплив несуттєвих відмінностей у стилі програмування та зосередитися на структурі й логіці реалізації алгоритму.

Векторизація коду перетворює програмний код у числове представлення за допомогою методу TF-IDF (Term Frequency – Inverse Document Frequency [43]), який визначає вагомість кожного токена в межах усієї вибірки. Це дозволяє моделі виокремлювати найважливіші елементи синтаксичної структури програм і враховувати частотні закономірності, характерні для різних типів рішень.

Отримані вектори ознак передаються до класифікаційної моделі, побудованої на основі визначеного алгоритму, який формує ансамбль дерев рішень, де кожне

дерево приймає власне рішення, а підсумковий результат визначається за принципом більшості голосів. Такий підхід забезпечує стійкість моделі до шумових даних і зменшує ризик перенавчання.

Модель інтегрується у процес перевірки робіт у вигляді конвеєра, реалізованого засобами бібліотеки Scikit-learn (рис. 2.13).

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier

model = Pipeline([
    ('vectorizer', TfidfVectorizer(token_pattern=r'\b\w+\b')),
    ('classifier', RandomForestClassifier(
        n_estimators=200,
        max_depth=None,
        min_samples_split=5,
        max_features='sqrt',
        random_state=42,
        n_jobs=-1
    ))
])
```

Рисунок 2.13 – Побудова моделі автоматизованої перевірки

Ефективність роботи моделі Random Forest залежить від правильного налаштування її параметрів:

`n_estimators` (кількість дерев у лісі) – показує кількість дерев рішень, що входять до складу ансамблю. Збільшення кількості дерев підвищує точність моделі, але збільшує час навчання та обсяг використовуваної пам'яті.

`max_depth` (максимальна глибина дерева) – обмежує глибину кожного дерева. Значення None дозволяє дереву розгалужуватися до повного поділу даних, що забезпечує максимальну точність за умови достатнього обсягу тренувальних даних.

`min_samples_split` (мінімальна кількість зразків для поділу вузла) – визначає мінімальну кількість зразків, необхідних для поділу внутрішнього вузла. Більші значення запобігають перенавчанню.

`max_features` (кількість ознак при поділі) – визначає кількість ознак, що розглядаються при кожному поділі. Зазвичай задається як квадратний корінь із загальної кількості ознак.

`random_state` – фіксується для забезпечення відтворюваності результатів при повторних запусках моделі.

Оптимальні значення параметрів визначаються за допомогою методів пошук за сіткою або випадковий пошук, що дозволяє знайти найкращий баланс між точністю моделі та її обчислювальною ефективністю.

Пошук за сіткою (Grid Search) працює за допомогою сітки значень гіперпараметрів і для кожної комбінації навчається модель та вона оцінюється на основі даних перевірки. У цьому підходу перевіряється кожна окрема комбінація значень гіперпараметрів, що може бути дуже неефективно.

Випадковий пошук (Randomized Search) – створюється сітка значень гіперпараметрів та обирається випадкові комбінації для навчання моделі та оцінювання. Кількість ітерацій пошуку встановлюється на основі часу/ресурсів [44].

Перевагами використання алгоритму Random Forest для створення та навчання моделі є:

1. Висока точність класифікації – модель поєднує результати багатьох незалежних дерев рішень, що забезпечує точність на рівні 85-92% (відповідно до вимог п. 2.3).

2. Стійкість до неповних даних – алгоритм ефективно працює навіть у випадку перевірки робіт здобувачів, які можуть містити помилки або відмінності в стилі програмування.

3. Оцінювання важливості ознак – можливість визначити, які характеристики коду найбільше впливають на підсумкову оцінку, що підвищує інтерпретованість результатів.

4. Масштабованість – модель легко розпаралелюється та може працювати в паралельному режимі, що робить її придатною для аналізу великих обсягів даних.

5. Простота використання – не потребує складного налаштування параметрів для досягнення стабільної якості класифікації.

Незважаючи на переваги, є й певні обмеження, але вони не впливають на якість моделі:

1. Обчислювальна складність – зі збільшенням кількості дерев у моделі зростає

обсяг обчислень і вимоги до оперативної пам'яті.

2. Часткова непрозорість – хоча можливо оцінити важливість окремих ознак, логіка прийняття рішень ансамблем дерев залишається складною для повного пояснення.

3. Відсутність інкрементального навчання – оновлення моделі при надходженні нових даних потребує її повторного навчання, оскільки Random Forest не підтримує поступове донавчання.

4. Надлишковість інформації – при наявності сильно корельованих ознак може знижуватися ефективність використання деяких характеристик коду.

Дані недоліки не суттєво впливають на ефективність і якість моделі в умовах поставлених завдань. Алгоритм залишається оптимальним вибором для автоматизованого оцінювання лабораторних робіт завдяки своїй надійності, стабільності та здатності до обробки великих обсягів даних.

Процес навчання та валідації моделі охоплює кілька послідовних етапів, спрямованих на забезпечення надійності, узагальнювальної здатності та високої точності класифікації (рис. 2.14).

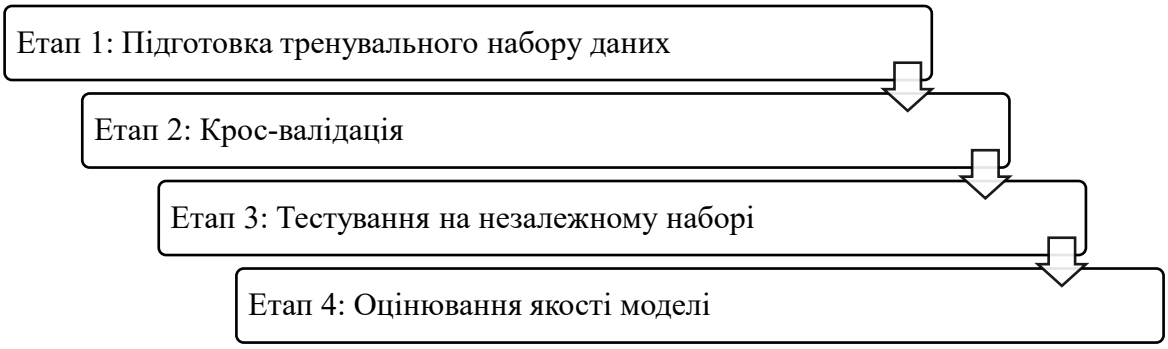

Рисунок 2.14 – Процес навчання та валідації моделі

На першому етапі формується база програмних робіт студентів, які були попередньо перевірені викладачами та мають визначені категорії оцінювання (наприклад, «правильна», «частково правильна», «неправильна»). Кожна робота проходить етапи попередньої обробки – очищення коду від коментарів, стандартизацію синтаксису, уніфікацію назв змінних і функцій, а також нормалізацію відступів та пробілів.

Після цього код перетворюється у векторне подання за допомогою TF-IDF, яке відображає статистичну важливість окремих токенів у контексті всієї вибірки. У

результаті формується набір даних, де кожен запис містить вектор ознак і відповідну мітку класу.

Другий етап (крос-валідація) для запобігання перенавчанню та забезпечення здатності моделі узагальнювати знання на нових прикладах застосовує метод `k-fold` крос-валідації (зазвичай `k=5` або `k=10`). Вся вибірка даних поділяється на `k` підмножин однакового розміру. Модель послідовно навчається на `k-1` підмножинах і перевіряється на решті, що дає змогу оцінити її стабільність і стійкість до змін у структурі даних. Середнє значення метрик (accuracy, precision, recall, F1-score), отриманих під час крос-валідації, вважається узагальненою оцінкою ефективності моделі.

Після завершення крос-валідації на третьому етапі модель тестується на окремому незалежному наборі даних (test set), який не використовувався під час навчання. Це дозволяє об'єктивно оцінити реальну продуктивність моделі в умовах, наближених до практичного використання.

Етап 4: Оцінювання якості моделі

На четвертому етапі здійснюється оцінювання якості моделі за допомогою метрик, описаних у п. 2.4:

Accuracy (загальна точність класифікації) ≥ 80%

Precision (точність позитивних передбачень) ≥ 82%

Recall (повнота) ≥ 80%

F1-Score (гармонічне середнє) ≥ 77%

Ці метрики забезпечують комплексну оцінку здатності моделі правильно класифікувати різні типи програмних рішень.

На етапі навчання модель отримує набір програм здобувачів з відповідними мітками. Після тренування вона здатна автоматично класифікувати нові роботи, аналізуючи як структуру, так і стиль коду.

Сформовані вектори ознак (п. 2.4) використовуються як вхідні дані для моделі Random Forest. Алгоритм обробляє різні ознаки без необхідності їх нормалізації, що робить його придатним для задачі автоматизованої перевірки програмного коду.

Результати класифікації передаються до модуля формування звітів (рис. 2.2), який створює детальний звіт про якість коду, виявлені помилки та рекомендації щодо

покращення. Цей звіт доступний як здобувачу, так і викладачу.

Таким чином, використання алгоритму Random Forest у поєднанні з TF-IDF-представленням коду забезпечує збалансоване поєднання високої точності, стійкості до варіативності студентських рішень і достатньої прозорості оцінювання. Це робить обраний підхід ефективним інструментом для реалізації інтелектуальної системи автоматизованої перевірки лабораторних робіт з програмування мовою Python.

Висновки до розділу 2

Другий розділ кваліфікаційної роботи присвячений архітектурі концептуальної моделі автоматизованої перевірки лабораторних робіт з програмування мовою Python та обґрунтування методів ML.

Розроблено загальний алгоритм процесу перевірки, який включає етапи завантаження коду, попередньої обробки, виділення ознак, тестування, оцінювання методами ML та формування звітів. На основі алгоритму побудовано діаграму компонентів, яка показує архітектуру концептуальної моделі автоматизованої перевірки лабораторних робіт. Для візуалізації логіки процесу перевірки лабораторної роботи створено діаграму послідовності, яка відображає послідовну взаємодію між користувачем і всіма внутрішніми модулями системи – від моменту завантаження файлу до отримання результату.

Визначено модуль статистичного аналізу коду, який дозволяє автоматизувати процеси аналізу структури, стилю та складності коду, підвищуючи об’єктивність і точність оцінювання. Розроблено методику формування векторів ознак для машинного навчання, що включає аналіз коду на чотирьох рівнях: лексичному, синтаксичному, семантичному та поведінковому. Обґрунтовано вибір алгоритму Random Forest для задачі автоматизованої перевірки програмного коду. Алгоритм забезпечує ефективну роботу з різнорідними ознаками, високу інтерпретованість результатів, стійкість до шумових даних та можливість паралельної обробки. Розроблено архітектуру конвеєра обробки даних, що поєднує етапи векторизації коду за допомогою TF-IDF та класифікації засобами Random Forest.

Даний розділ створює основу для реалізації концептуальної моделі.

3 РЕАЛІЗАЦІЯ ТА АПРОБАЦІЯ КОНЦЕПТУАЛЬНОЇ МОДЕЛІ

3.1. Реалізація модулів аналізу коду

3.1.1 Модуль лексичного аналізу

Основною метою модуля лексичного аналізу є автоматичне виділення лексичних ознак із програмного коду, написаного мовою Python, з подальшим використанням цих ознак для різних аналітичних, статистичних та інших задач. Модуль реалізує попередній етап обробки п, що включає програми, що включає перетворення коду лабораторної роботи здобувача у структуроване подання набору токенів. Для цього використовується стандартний модуль мови програмування Python – tokenize, який дозволяє точно розділити код на елементарні лексеми.

Основні функціональні можливості модуля:

1. Токенізація коду.

2. Кількісний аналіз виявлених токенів.

3. Аналіз контекстних залежностей між елементами коду, шляхом формування уніграм, біграм та тріграм.

4. Статистичний аналіз коду.

Модуль лексичного аналізу забезпечує базову інформацію для подальших етапів – синтаксичного та семантичного аналізу, а також для побудови моделі ML.

На рис. 3.1 наведено фрагмент коду програми студента, для якого виковували розрахунки.

```
def factorial(n):
    if n == 0:
        return 1
    elif n < 0:
        return None
    return n * factorial(n - 1)


num = int(input("Введіть число: "))

if num < 0:
    print("Факторіал від'ємного числа не існує")
else:
    result = factorial(num)
    print(f"Факторіал {num} = {result}")
```

Рисунок 3.1 – Фрагмент коду програми студента

Фрагмент результату виконання коду модуля лексичного аналізу наведено на рис. 3.2.

```
Знайдено 76 токенів

======================================================================
№     Тип токену          Токен                          Позиція
======================================================================
1     NAME                'def'                          (2,0)
2     NAME                'factorial'                    (2,4)
3     OP                  '('                            (2,13)
4     NAME                'n'                            (2,14)
5     OP                  ')'                            (2,15)
6     OP                  ':'                            (2,16)
7     NAME                'if'                           (3,4)
8     NAME                'n'                            (3,7)
9     OP                  '=='                           (3,9)
10    NUMBER              '0'                            (3,12)
11    OP                  ':'                            (3,13)
12    NAME                'return'                       (4,8)
13    NUMBER              '1'                            (4,15)
14    NAME                'elif'                         (5,4)
15    NAME                'n'                            (5,9)

...

========================================
ЛЕКСИЧНІ ОЗНАКИ КОДУ
========================================

 БАЗОВІ МЕТРИКИ:
  Загальна кількість токенів: 76
  Унікальні токени: 34
  Ключові слова: 10
  Ідентифікатори: 20
  Оператори: 34
  Літерали: 8 (числа: 5, рядки: 3)
  Коментарі: 0

 СТАТИСТИЧНІ МЕТРИКИ:
  Середня довжина ідентифікаторів: 4.3
  Коефіцієнт унікальності: 0.447
  Частка ключових слів: 0.132
  Частка операторів: 0.447
  Частка літералів: 0.105

 РОЗПОДІЛ ТОКЕНІВ:
  NAME: 30
  OPERATOR: 34
  NUMBER: 5
  STRING: 3
  TYPE_61: 1
  TYPE_62: 2
  TYPE_63: 1

ТОП-10 УНІГРАМ:
 '(': 8
 ')': 8
 ':': 6
 'n': 5
 'num': 4
 'factorial': 3
 'if': 3
 '0': 3
 'return': 3
 '==': 2

ТОП-10 БІГРАМ:
 ('factorial', '('): 3
 ('0', ':'): 3
 ('(', 'n'): 2
 (':', 'return'): 2
 ('<', '0'): 2
 (')', 'if'): 2
 ('print', '('): 2
 ('def', 'factorial'): 1
 ('n', ')'): 1
 (')', ':'): 1

ТОП-5 ТРІГРАМ:
 ('factorial', '(', 'n'): 2
 ('0', ':', 'return'): 2
 ('<', '0', ':'): 2
 ('def', 'factorial', '('): 1
 ('(', 'n', ')'): 1
```

Рисунок 3.2 – Фрагмент результатів виконання коду модуля лексичного аналізу

Результатами виконання модуля лексичного аналізу є загальна кількість токенів, кількість унікальних токенів, частота ключових слів, співвідношення коду до коментарів та середня довжина ідентифікаторів.

3.1.2 Модуль синтаксичного аналізу

Модуль, призначений для аналізу структури коду, виконує глибоке опрацювання програмного тексту шляхом побудови AST, на основі якого система здійснює комплексний структурний аналіз. Побудова AST здійснюється за допомогою стандартного модуля ast, який перетворює вихідний код у деревоподібну модель, де кожен вузол відповідає певному елементу мови програмування. Після цього модуль проводить підрахунок основних структурних компонентів програми, таких як функції, класи, цикли та інші конструкції керування. Це дозволяє визначити загальну кількість логічних одиниць коду та оцінити його архітектурну складність.

Особливу увагу приділено обчисленню глибини вкладеності, тобто визначенню максимальної кількості рівнів, на які розгалужуються оператори умов, цикли чи виклики функцій. Такий показник допомагає виявити надмірно складні або погано структуровані частини програми.

Окрім цього, модуль реалізує аналіз складності коду, що включає оцінювання логічної насиченості, кількості шляхів виконання та рівня когнітивного навантаження. Завдяки цьому можна зробити висновки про якість, читабельність і підтримуваність програмного коду.

На рис. 3.3 відображені результати виконання коду.

Результатами виконання модуля статистичного аналізу є кількість функцій та класів, кількість умовних конструкцій, кількість циклів, максимальна глибина вкладеності та кількість рядків коду.

```
========================================
СИНТАКСИЧНІ ОЗНАКИ КОДУ (AST АНАЛІЗ)
========================================

 СТРУКТУРНІ ЕЛЕМЕНТИ:
  Функції: 1
  Класи: 0
  Методи: 0
  Умовні конструкції (if): 4
  Цикли for: 0
  Цикли while: 0
  Всього циклів: 0
  Try-except блоки: 0
  With конструкції: 0

 COMPREHENSIONS:
  List comprehensions: 0
  Dict comprehensions: 0
  Set comprehensions: 0
  Lambda функції: 0

 ГЛИБИНА ТА СКЛАДНІСТЬ:
  Максимальна глибина вкладеності: 3
  Цикломатична складність (McCabe): 6
  Точки розгалуження: 5

 ВИКЛИКИ ФУНКЦІЙ:
  Всього викликів: 7
  Унікальних функцій: 5
  Вбудовані функції: 4

  Найчастіше викликані:
    factorial: 2 раз(ів)
    print: 2 раз(ів)
    int: 1 раз(ів)
    input: 1 раз(ів)
    main: 1 раз(ів)

 ІМПОРТИ ТА ЗМІННІ:
  Імпортів: 0
  Присвоєнь: 2
  Глобальних змінних: 0

 ЗАГАЛЬНІ МЕТРИКИ:
  Рядків коду: 14
  Середня к-ть аргументів функції: 1.0
  Функцій з docstring: 0
  Функцій з декораторами: 0

 ДЕТАЛІ ФУНКЦІЙ:
  factorial:
    Аргументів: 1, Docstring: ×, Декоратори: ×, Рядок: 2
```

```
СТРУКТУРА AST ДЕРЕВА (перші 3 рівні)
====================================
├ Module
  ├ FunctionDef: factorial
    ├ arguments
      ├ arg
    ├ If
      ├ Compare
      ├ Return
      ├ If
    ├ Return
      ├ BinOp
  ├ Assign
    ├ Name: num
      ├ Store
    ├ Call: int()
      ├ Name: int
      ├ Call: input()
  ├ If
    ├ Compare
      ├ Name: num
      ├ Lt
      ├ Constant: 0
    ├ Expr
      ├ Call: print()
    ├ Assign
      ├ Name: result
      ├ Call: factorial()
    ├ Expr
      ├ Call: print()
  ├ If
    ├ Compare
      ├ Name: __name__
      ├ Eq
      ├ Constant: '__main__'
    ├ Expr
      ├ Call: main()
```

```
================================
СТАТИСТИКА ВУЗЛІВ AST
================================

Топ-10 найчастіших типів вузлів:
  Name: 18
  Load: 16
  Constant: 11
  Call: 7
  If: 4
  Compare: 4
  Return: 3
  Expr: 3
  Assign: 2
  BinOp: 2

Всього типів вузлів: 21
Всього вузлів: 85
```

Рисунок 3.3 – Результати виконання коду модуля статистичного аналізу

3.1.3 Модуль аналізу складності коду

Модуль призначений для автоматизованої оцінки якості програмного коду на основі обчислення різних метрик складності. У його роботі використовується бібліотека radon, яка надає інструменти для розрахунку кількісних показників, що характеризують структурну та логічну складність програмного коду.

Однією з головних метрик є цикломатична складність (McCabe) – показник, який визначає кількість незалежних шляхів виконання програми. Високе значення цієї метрики свідчить про складну логіку з великою кількістю умовних операторів або розгалужень, що ускладнює тестування та супровід коду.

Другою важливою групою є метрики Холстеда, які ґрунтуються на кількості операторів та операндів у коді. Вони дозволяють оцінити, наскільки об’ємним і складним є програмний текст з точки зору обчислювальної логіки. Ці метрики включають такі показники, як обсяг коду, що показує кількість інформації, яку містить програма та складність Холстеда, що відображає когнітивну складність –тобто, наскільки складно зрозуміти або реалізувати дану програму.

Ще одним ключовим показником є індекс підтримуваності коду. Метрика, яка узагальнює попередні дані й дозволяє оцінити, наскільки легко програму можна підтримувати, розширювати або модифікувати. Чим вищий цей індекс, тим зрозумілішим і стабільнішим є код [45].

Модуль отримує низку характеристичних ознак, що допомагають приймати обґрунтовані рішення щодо оптимізації та покращення структури програмного коду.

На рис. 3.4 відображені результати виконання коду.

```
==================================================
МЕТРИКИ СКЛАДНОСТІ КОДУ
==================================================

ЦИКЛОМАТИЧНА СКЛАДНІСТЬ (McCabe):
 Середня: 3.0
 Максимальна: 3
 Загальна: 3
 Функцій з високою складністю (>10): 0
 Ранг: A

 Деталі по функціях:
   factorial: складність 3 (ранг A) - рядок 2

МЕТРИКИ ХОЛСТЕДА:
 Vocabulary (словник): 12
 Length (довжина): 18
 Volume (обсяг): 64.53
 Difficulty (складність): 3.0
 Effort (зусилля): 193.59
 Time (час, сек): 10.75
 Bugs (очікувані помилки): 0.0215

ІНДЕКС ПІДТРИМУВАНОСТІ:
 MI Score: 61.65 / 100
 Ранг: A
 Опис: Дуже добра підтримуваність (85-100)

RAW МЕТРИКИ:
 Всього рядків (LOC): 19
 Логічних рядків (LLOC): 14
 Рядків коду (SLOC): 14
 Коментарів: 0
 Багаторядкових рядків: 0
 Порожніх рядків: 5
 Коефіцієнт коментування: 0.0
 Співвідношення код/коментарі: 0

КОГНІТИВНА СКЛАДНІСТЬ:
 Загальна когнітивна складність: 4
 На одну функцію: 4.0
 Вплив вкладеності: 3

ДОДАТКОВІ МЕТРИКИ:
 Середня к-ть параметрів функції: 1.0
 Максимум параметрів: 1
 Операторів return: 3
 Всього операторів: 2

============================================================
РЕЗЮМЕ СКЛАДНОСТІ
============================================================
Цикломатична складність в нормі (проста)
Середня підтримуваність коду
Коментарі відсутні або недостатні
Всі функції мають прийнятну складність

============================================================
РЕКОМЕНДАЦІЇ
============================================================
Код потребує рефакторингу для покращення підтримуваності
Додайте більше коментарів для пояснення логіки коду

 Аналіз завершено!
```

Рисунок 3.4 – Результати виконання коду модуля аналізу складності коду

Результати аналізу показують, що програма для обчислення факторіала має просту структуру, невелику кількість розгалужень та зрозумілу логіку. Цикломатична складність проста, що свідчить про легкість тестування й низький ризик появи логічних помилок. Метрики Холстеда підтверджують, що код компактний, а когнітивне навантаження на розробника мінімальне. Індекс підтримуваності високий, отже програму легко розширювати та супроводжувати. Єдиним аспектом, який можна покращити, є недостатня кількість коментарів – додаткові пояснення у коді підвищили б його зрозумілість для інших користувачів.

Загалом код якісний, добре структурований і відповідає вимогам до простих програм із високим рівнем надійності.

3.1.4 Модуль перевірки стилю коду

Модуль призначений для аналізу відповідності програмного коду вимогам стандарту PEP 8, що визначає правила оформлення Python-програм. Для цього використовуються два основні інструменти: pylint, який здійснює статичний аналіз і перевіряє логіку та структуру коду, а також flake8, який зосереджується на дотриманні стилістичних норм.

У результаті роботи модуля формується узагальнена оцінка якості коду за шкалою від 0 до 3, що відображає рівень відповідності стандартам. Крім того, підраховується кількість порушень стилістичних конвенцій, тобто випадків, коли код не відповідає вимогам PEP 8. Також визначається кількість попереджень і кількість помилок, які можуть свідчити про потенційні проблеми у структурі або логіці програми. Завдяки цьому модуль можна підтримувати високий рівень чистоти, читабельності та узгодженості коду.

На рис. 3.5 відображені результати виконання коду.

```
Аналіз файлу: D:\Python\lar_1.py

 Результати збережено у файл: report.json

=== РЕЗУЛЬТАТ АНАЛІЗУ КОДУ ===
{
    "file": "D:\\Python\\lar_1.py",
    "timestamp": "2025-11-08 22:58:37",
    "оцінка_якості_коду": 2.18,
    "pep8_порушення": 0,
    "попередження": 0,
    "помилки": 0
}
```

Рисунок 3.5 – Результати виконання коду модуля перевірки стилю коду

Результат виконання модуля записується у файл report.json в БД.

3.1.5 Модуль динамічного тестування

Модуль динамічного тестування призначений для перевірки коректності виконання програмного коду за допомогою спеціально підготовлених тестових даних. Його основна мета – оцінити, наскільки програма працює відповідно до очікуваних результатів. Після запуску тестів результати програми порівнюються з еталонними даними, що дозволяє виявити відхилення у поведінці або логіці виконання. Додатково здійснюється вимірювання часу виконання кожного тесту та контроль споживання пам’яті, що дає змогу оцінити ефективність та оптимальність коду. У підсумку модуль формує низку аналітичних показників, серед яких відсоток успішно пройдених тестів, середній час виконання програми, обсяг використаної пам’яті та інформація про наявність помилок виконання (рис. 3.6).

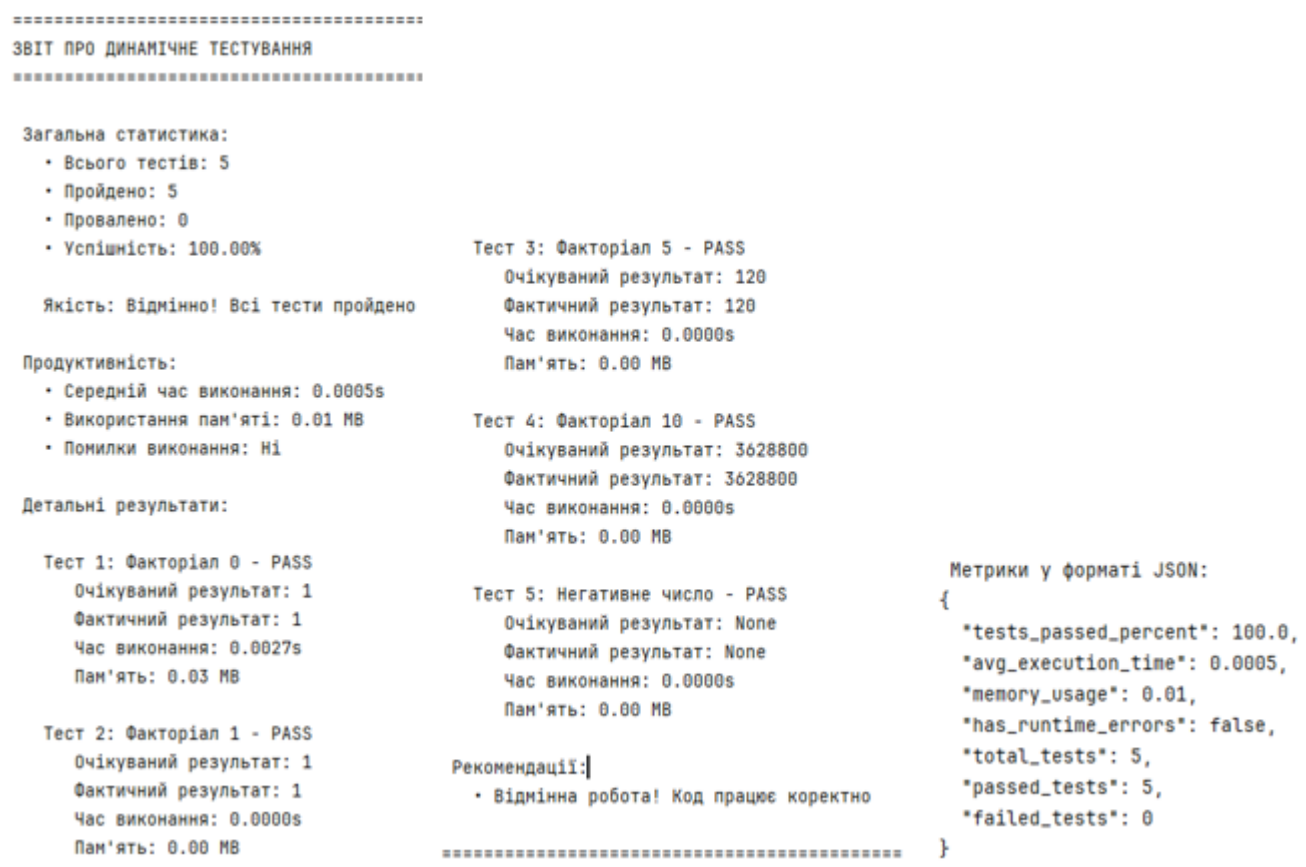

Рисунок 3.6 – Результати виконання модуля динамічного тестування

Дані дозволяють об’єктивно оцінити стабільність, продуктивність і надійність програмного коду, а також визначити напрямки для його подальшого вдосконалення.

Фрагменти кодів з реалізації розглянутих модулів наведено у Додатку А.

У результаті їх реалізації забезпечено комплексну оцінку програмного коду мови Python за лексичними, синтаксичними, структурними та стилістичними характеристиками. Отримані результати дозволяють об’єктивно визначати якість, складність і підтримуваність програм, а також виявляти потенційні недоліки в лабораторних роботах здобувачів.

3.2. Формування навчальної вибірки

Для навчання та тестування моделі автоматичного оцінювання якості програмного коду було сформовано набір даних, що містить 100 програмних робіт. Джерелами даних виступили лабораторні роботи студентів (50 прикладів), синтетично згенеровані програми з типовими помилками (30 прикладів) та еталонні розв’язки, підготовлені викладачами (20 прикладів). Така комбінація дозволила охопити широкий

спектр варіантів реалізації програм, включаючи як коректні рішення, так і характерні помилки, що часто трапляються в студентських роботах.

Кожна робота була розмічена за чотирма класами якості коду:

Клас 0 (Відмінно) – повністю правильний, оптимальний код, який відповідає стандартам PEP 8.

Клас 1 (Добре) – код функціонально правильний, але містить незначні стилістичні або структурні недоліки.

Клас 2 (Задовільно) – код частково правильний, з наявністю логічних помилок або недоопрацьованих частин.

Клас 3 (Незадовільно) – код некоректний або не виконується через критичні помилки.

Розподіл робіт за класами забезпечує збалансованість вибірки:

клас 0 – 25 робіт (25%);

клас 1 – 35 робіт (35%);

клас 2 – 25 робіт (25%);

клас 3 – 15 робіт (15%).

Така пропорція дозволяє моделі адекватно навчатися на прикладах різного рівня якості без переваги одного класу над іншими.

На етапі підготовки даних для кожної роботи формувався вектор ознак, що описує її програмну структуру, складність, стиль написання та результати тестування. Процес формування складався з кількох послідовних кроків:

– з коду витягувалися всі категорії ознак – лексичні, синтаксичні, метрики складності, показники стилю та параметри тестування;

– числові ознаки нормалізувалися для забезпечення порівнюваності;

– текстові токени коду перетворювалися у TF-IDF вектори, що відображають частоту вживання лексем у контексті програми.

Після цього всі групи ознак об’єднувалися в єдиний вектор розмірністю 47 компонентів.

До основних лексичних ознак належали кількість токенів, унікальних слів та ключових операторів. Синтаксичні ознаки відображали кількість функцій, класів,

умовних операторів та циклів. Метрики складності включали цикломатичну складність та індекс підтримуваності. Стильові параметри – оцінку Pylint, кількість попереджень та відхилення від стандартів форматування. Поведінкові ознаки визначалися на основі результатів автоматичного тестування, зокрема часткою успішно пройдених тестів і середнім часом виконання програми.

Для подальшого аналізу сформовані вектори ознак зберігалися у БД у таблиці code_features згідно з логічною структурою, поданою в ER-діаграмі системи.

Для роботи з БД в мові програмування Python використовувалася бібліотека psycopg2. На рис. 3.7 наведено фрагмент коду для заповнення таблиці.

```
student_id = 2

lexical_features = {"total_tokens": 150,"unique_tokens": 80,"keywords_count": 25 }

syntactic_features = {"functions_count": 3,"classes_count": 1,"if_count": 5}

semantic_features = {"cyclomatic_complexity": 4.5,"maintainability_index": 75.2}

behavioral_features = {"pass_rate": 0.9,"avg_execution_time": 0.05}

tfidf_vector = [0.12, 0.34, 0.0, 0.56, 0.22, 0.0, 0.1, 0.08]

insert_code_features(
    student_id,
    lexical_features,
    syntactic_features,
    semantic_features,
    behavioral_features,
    tfidf_vector
)
```

Рисунок 3.7 – Фрагмент коду заповнення таблиці `code_features`

Результат заповнення таблиці `code_features` (рис. 3.8).

	feature_id [PK] bigint	student_id bigint	lexical_features jsonb	syntactic_features jsonb	semantic_features jsonb	behavioral_features jsonb	tfidf_vector jsonb
1	4	1	{"total_tokens": 150, "...	{"if_count": 5, "classes_count": 1, ...	{"cyclomatic_complexity": 4.5, "maint...	{"pass_rate": 0.9, "avg_...	[0.12, 0.34, 0.0, 0.56, 0...
2	6	2	{"total_tokens": 150, "...	{"if_count": 5, "classes_count": 1, ...	{"cyclomatic_complexity": 4.5, "maint...	{"pass_rate": 0.9, "avg_...	[0.12, 0.34, 0.0, 0.56, 0...
3	9	3	{"total_tokens": 160, "...	{"if_count": 5, "classes_count": 1, ...	{"cyclomatic_complexity": 4.7, "maint...	{"pass_rate": 0.8, "avg_...	[0.11, 0.35, 0.0, 0.55, 0...

Рисунок 3.8 – Таблиця code_features

Таким чином, отримано структурований набір даних, який містить повну інформацію про характеристики коду. Це дозволяє ефективно проводити подальше ML, аналіз складності та оцінювання написання програмних кодів лабораторних робіт.

3.3. Навчання моделі

На цьому етапі здійснювалося навчання моделі ML для автоматизованої оцінки якості програмного коду. Основною метою було побудувати класифікатор, здатний на основі числових ознак, отриманих із модулів аналізу коду, передбачати рівень якості лабораторних робіт.

Для навчання було використано 100 прикладів лабораторних робіт, представлених у вигляді векторів ознак розмірності 547. Вибірка була розділена у пропорції:

– 50 лабораторних робіт, виконаних здобувачами (50 %);

– 30 синтезованих робіт з типовими помилками (30 %);

– 20 еталонних рішень, виконаних викладачем (20 %).

Еталонні роботи переважно відносяться до всіх класів (0-1). Роботи студентів розподілені по всіх класах. Синтезовані роботи – переважно відносяться до класів 2-3. Такий підхід забезпечує можливість контролю за узагальнюючою здатністю моделі та запобігає перенавчанню.

Як базовий алгоритм обрано Random Forest, який добре підходить для задач класифікації з великою кількістю ознак і дозволяє оцінювати важливість кожної з них. Оптимізацію параметрів моделі виконано методом Grid Search із 5-кратною крос-валідацією, що дало змогу підібрати комбінацію гіперпараметрів, яка забезпечує найкращу зважену F1-оцінку.

Результати навчання моделі за класами, наведено на рис. 3.9.

```
Генерація датасету: 100 прикладів, 547 ознак, 4 класи

Створено датасет:
  - Кількість прикладів: 100
  - Кількість ознак: 547
  - Кількість класів: 4

Розподіл оцінок:
quality_grade
2    19
3    30
4    31
5    20
Name: count, dtype: int64
```

Рисунок 3.9 – Розподіл за оцінкою якості та кількістю робіт

Як видно з рис. 3.9, найбільше представлено прикладів із середнім рівнем якості (оцінки 3 та 4), що відповідає типовому розподілу у реальних наборах даних, де крайні випадки (дуже низька або дуже висока якість) зустрічаються рідше. Такий розподіл дозволяє уникнути перекосу моделі в бік певного класу та сприяє більш об'єктивному навчанню.

Для загального аналізу отриманого набору даних було обчислено статистичні показники для основних ознак, що мають найбільший вплив на якість коду (рис. 3.10).

```
Основні статистики ключових ознак:
       pass_rate  cyclomatic_complexity  ...  functions_count  max_nesting_depth
count     100.00                 100.00  ...           100.00             100.00
mean       72.22                  22.47  ...            26.19               5.19
std        15.99                   5.85  ...            15.09               1.65
min        40.20                   9.24  ...             5.00               2.00
25%        62.17                  17.51  ...            12.75               4.00
50%        72.91                  22.31  ...            23.00               5.00
75%        83.32                  26.42  ...            41.25               6.00
max       100.00                  34.65  ...            50.00               9.00
```

Рисунок 3.10 – Узагальнені результати

Ключові ознаки, для яких обчислені статистичні характеристики:

– `pass_rate` – частка успішно пройдених тестів для конкретного рішення;

– `cyclomatic_complexity` – цикломатична складність програми (кількість незалежних шляхів виконання);

– `functions_count` – кількість функцій, реалізованих у кодах;

– `max_nesting_depth` – максимальна глибина вкладеності конструкцій (циклів, умов тощо).

Отримані статистики свідчать, що в датасеті представлені коди програм з різною структурною складністю: від відносно простих (з низьким рівнем вкладеності та невеликою кількістю функцій) до більш складних, з високою цикломатичною складністю та глибокою ієрархією коду.

Попередній перегляд даних представлено у фрагменті на рис. 3.11.

```
Розмір датасету: (100, 548)

Перші 5 рядків:
   pass_rate  cyclomatic_complexity  ...  feature_547  quality_grade
0  82.483571              15.753888  ...   -14.047453              4
1  79.308678              18.738064  ...     0.155566              4
2  98.238443              13.971856  ...     3.730042              5
3  72.615149              22.593168  ...     6.466195              3
4  63.829233              24.516143  ...    -8.128472              3

[5 rows x 548 columns]
```

Рисунок 3.11 – Фрагмент попереднього перегляду даних

Загальна структура згенерованого набору даних має розмір 100 × 548, де 547 стовпців відповідають окремим числовим ознакам, а останній стовпець quality_grade – цільовій змінній. Усі числові ознаки мають тип float64, що забезпечує високу точність під час обчислень. Декілька допоміжних індексних змінних мають тип int64. Загальний обсяг набору даних становить близько 428 КБ, що робить його придатним для швидкої обробки та експериментів у середовищах машинного навчання (рис. 3.12).

```
Інформація про датасет:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Columns: 548 entries, pass_rate to quality_grade
dtypes: float64(543), int64(5)
memory usage: 428.3 KB
None
```

Рисунок 3.12 – Структура та типи даних

При навчанні моделі визначені параметри, опис яких надано у п. 2.5:

max_depth: 10,

max_features: sqrt,

min_samples_leaf: 1,

min_samples_split: 5,

n_estimators: 200

Найкраща F1-оцінка (CV): 0,8501.

Модель збережена у файл `random_forest_model.pkl`.

Фрагменти коду навчання моделі наведено у Додатку Б.

Отже, отриманий набір даних можна охарактеризувати як збалансований датасет для задачі оцінювання якості програмного коду лабораторних робіт. Результати підтверджують, що модель ефективно використовує як структурні, так і якісні характеристики програмного коду для формування об’єктивної оцінки.

3.4. Оцінювання якості моделі

Після завершення навчання проведено етап оцінювання ефективності моделі класифікації якості програмного коду. Для цього використано тестову вибірку, що не брала участі у процесі навчання. Основними метриками оцінювання виступали точність, прецизійність, повнота та F1-міра, які дозволяють всебічно охарактеризувати здатність моделі правильно класифікувати приклади різних класів.

Результати роботи моделі наведено на рис. 3.13.

```
РЕЗУЛЬТАТИ НА ТЕСТОВІЙ ВИБІРЦІ:
 Accuracy:  0.800 (80.0%)
 Precision: 0.856 (85.6%)
 Recall:    0.800 (80.0%)
 F1-Score:  0.776 (77.6%)
```

Рисунок 3.13 – Результати роботи моделі

Accuracy показує частку правильно класифікованих прикладів серед усіх тестових. Precision – результат, який демонструє частку коректних передбачень серед усіх передбачених позитивних випадків. Recall – це частка правильно виявлених об’єктів певного класу серед усіх реальних об’єктів цього класу. F1-score показує середнє гармонійне між Precision і Recall, що характеризує збалансовану якість класифікації.

Отримані значення свідчать, що модель у цілому демонструє задовільну точність (80%) та досить високу прецизійність (85,6%), тобто вона помиляється у випадках, коли класифікує приклад як належний до певного класу. Водночас показники Recall та F1-score свідчать про наявність деякої втрати чутливості моделі – не всі приклади кожного класу були розпізнані коректна.

Детальний звіт по окремих класах показує, що модель поводиться по-різному залежно від рівня якості коду (рис. 3.14).

```
ДЕТАЛЬНИЙ ЗВІТ ПО КЛАСАХ:
                    precision    recall  f1-score   support

  Клас 0 (Відмінно)       1.00      0.33      0.50         3
     Клас 1 (Добре)       0.67      1.00      0.80         4
Клас 2 (Задовільно)       0.83      1.00      0.91         5
Клас 3 (Незадовільно)     1.00      0.67      0.80         3

          accuracy                           0.80        15
         macro avg       0.88      0.75      0.75        15
      weighted avg       0.86      0.80      0.78        15
```

Рисунок 3.14 – Детальний звіт по класах

Клас 0 (Відмінно) показу, що модель точно визначає цей клас, але пропускає частину прикладів (низький Recall).

Клас 1 (Добре) добре розпізнає усі випадки класу, але іноді помилково відносить до нього інші приклади.

Клас 2 (Задовільно) є найбільш стабільним, демонструє високу точність і повноту.

Клас 3 (Незадовільно) показу, що модель добре визначає приклади цього класу, проте не завжди виявляє всі.

Таким чином, найкращі результати досягнуто для класу 2, тоді як клас 0 виявився найскладнішим для коректного передбачення. Це може бути зумовлено обмеженою кількістю прикладів цього класу у навчальних даних або надмірною схожістю ознак із суміжними класами.

Матриця помилок (Confusion Matrix), збережена у файлі confusion_matrix.png та дозволяє візуально оцінити характер класифікаційних помилок (рис. 3.15).

```
МАТРИЦЯ ПОМИЛОК:
[[1 2 0 0]
 [0 4 0 0]
 [0 0 5 0]
 [0 0 1 2]]
Матрицю помилок збережено у 'confusion_matrix.png'
```

Рисунок 3.15 – Матриця помилок

Матриця помилок показую, що частина прикладів класу 0 була помилково віднесена до класу 1 (2 з 3 випадків). Для класів 1 та 2 модель спрацювала без помилок. Один приклад класу 4 було неправильно класифіковано як «Задовільно». Такі

помилки є типовими для моделей, що працюють із суміжними якісними класами, коли межі між ними не є чітко визначеними.

Для перевірки стабільності моделі проведено 5-кратну крос-валідацію, отримані значення F1-міри для кожної ітерації наведено на рис. 3.16.

```
КРОС-ВАЛІДАЦІЯ (5-fold):
 F1 scores: ['0.571', '0.851', '1.000', '1.000', '0.828']
 Середнє F1: 0.850 (±0.157)
```

Рисунок 3.16 – Крос-валідація

Середнє значення F1-score, що свідчить про загалом добру узагальнювальну здатність моделі, хоча на окремих підвибірках спостерігається варіативність результатів.

Далі було проведено порівняння моделі з вимогами (рис. 3.17).

Метрика	Вимога	Результат	Виконання
Accuracy	≥ 80%	80.0%	Так
Precision	≥ 82%	85.6%	Так
Recall	≥ 80%	80.0%	Так
F1-Score	≥ 77%	77.6%	Так

Рисунок 3.17 – Порівняння з вимогами

У результаті тестування модель продемонструвала достатньо високий рівень точності (приблизно 80%). Основними проблемами залишаються недостатня повнота розпізнавання окремих класів і варіативність результатів на різних підвибірках.

Для підвищення показників потрібно провести додаткове налаштування гіперпараметрів моделі та балансування вибірки за класами.

3.5. Інтеграція моделі машинного навчання в концептуальну модель даних системи

Після навчання та валідації моделі ML наступним етапом є її інтеграція в концептуальну модель даних системи автоматизованої перевірки лабораторних робіт з програмування. Цей процес забезпечує її безпосереднє використання у процесі

перевірки лабораторних робіт здобувачів. На даному етапі реалізовано зв’язок між аналітичною частиною системи (моделі машинного навчання) та БД, структура якої наведена на ER-діаграмі (рис. 2.4).

Модель Random Forest, навчена на основі визначеного набору ознак, інтегрована у систему як окремий модуль, який взаємодіє з таблицями `code_features`, `check_result`, `students` та `lab_work`.

Основна логіка інтеграції полягає в тому, що після того як студент завантажує лабораторну роботу, система зберігає вихідний код у таблиці `students` разом із датою завантаження, версією та статусом перевірки.

Послідовно активуються модулі аналізу – лексичний, синтаксичний, структурний, стильовий і динамічного тестування. Результати кожного з них формуються у вигляді числових та текстових характеристик і зберігаються у таблиці `code_features`, що містить JSON-поля з описом усіх типів ознак, а також сформований TF-IDF вектор.

Система об’єднує усі отримані характеристики в один вектор, який подається на вхід моделі машинного навчання. Для цього використовується програмний інтерфейс на Python із бібліотеками psycopg2, pandas та joblib (для завантаження моделі з файлу random_forest_model.pkl).

Модель здійснює класифікацію за рівнем якості, присвоюючи лабораторній роботі відповідний клас (0-3) та прогнозну оцінку у балах. Результати зберігаються в таблиці `check_result`, де фіксуються: оцінка, клас якості, дата перевірки та детальний звіт у форматі JSON.

Система на основі даних таблиці `check_result` формує звіт для викладача та студента, що містить як числові оцінки, так і аналітичні висновки з детальними показниками складності, стилю та продуктивності коду.

Для інтеграції ML-моделі в систему необхідно розширити концептуальну модель даних новими сутностями та зв'язками, які забезпечують зберігання результатів роботи моделі та їх використання в процесі оцінювання.

Для інтеграції ML-моделі в систему було додано три нові таблиці (табл. 3.1).

Таблиця 3.1 – Сутності моделі в БД

Номер сутності	Назва сутності	Атрибути сутності
7	Навчені ML-моделі (ml_model)	model_id (PK) – унікальний ідентифікатор моделі model_type – тип моделі (RandomForest) version – версія моделі training_date – дата та час навчання моделі accuracy – точність моделі на тестовій вибірці precision – precision метрика recall – recall метрика f1_score – F1-оцінка model_path – шлях до файлу моделі (random_forest_model.pkl) is_active – булевий прапорець активної моделі description – текстовий опис моделі created_at – фіксація створення запису
8	Результати ML-передбачень (ml_prediction)	prediction_id (PK) – унікальний ідентифікатор передбачення submission_id (FK) – посилання на здану роботу з таблиці students model_id (FK) – яка модель використовувалась для передбачення predicted_class – передбачений клас якості (0-3) confidence – рівень впевненості моделі (0.0-1.0) class_probabilities – JSON з ймовірностями для всіх класів prediction_timestamp – дата та час виконання передбачення processing_time_ms – час обробки в мілісекундах
9	Автоматичні рекомендації (ml_recommendation)	recommendation_id (PK) – унікальний ідентифікатор рекомендації prediction_id (FK) – посилання на передбачення recommendation_text – текст рекомендації українською мовою recommendation_type – тип рекомендації («quality», «testing», «style», «complexity», «performance») priority – пріоритет («high», «medium», «low») is_shown – чи показано студенту created_at – фіксація створення запису

Сутність `ml_model` призначена для зберігання інформації про всі версії навчених моделей ML.

Сутність `ml_prediction` зберігає результати класифікації кожної роботи здобувача.

Сутність `ml_recommendation` зберігає згенеровані системою рекомендації для покращення коду.

Також здійсненна модифікації існуючих таблиць БД:

`code_features` – додано поле `extraction_date` для відстеження часу

витягування ознак;

`check_result` – додано поле `prediction_id` (FK) для зв'язку з ML-передбаченням, що дозволяє викладачу бачити, яке автоматичне передбачення було зроблено та додане поле `teacher_comment`, для коментарів викладача.

ER-діаграма з інтеграцією моделі ML наведено на рис. 3.18.

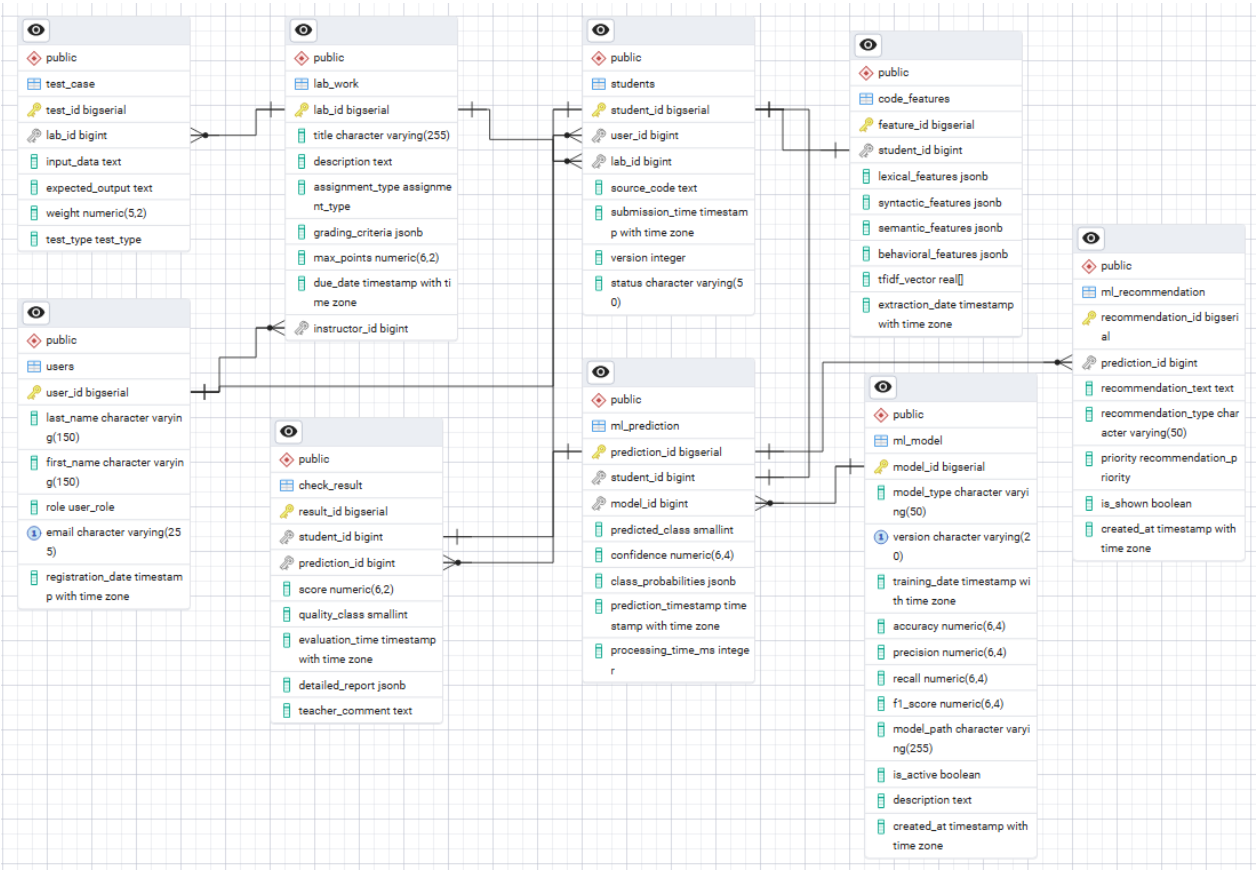

Рисунок 3.18 – ER-діаграма з інтеграцією моделі ML

Таким чином, модель ML інтегрована у концептуальну систему як аналітичне ядро, що автоматизує процес перевірки лабораторних робіт. Вона безпосередньо взаємодіє з базою даних, зберігає результати аналізу в централізованому сховищі та надає можливість подальшої статистичної обробки.

Архітектурно система реалізує три основні рівні інтеграції:

– дані – зберігаються у PostgreSQL відповідно до логічної структури (`users`, `lab_work`, `students`, `code_features`, `test_case`, `check_result`);

– аналітика – модулі Python для обробки, аналізу та класифікації коду;

– взаємодія користувача – інтерфейс для перегляду результатів, звітів і

статистики.

У результаті створено єдину інтегровану систему, що поєднує зберігання, аналіз і оцінювання програмного коду, забезпечуючи повний цикл автоматизованої перевірки лабораторних робіт – від завантаження студентом до отримання оцінки викладачем.

Висновки до розділу 3

У третьому розділі реалізовано та апробовано концептуальну модель автоматизованої перевірки лабораторних робіт з програмування мовою Python із застосуванням методів машинного навчання. Розроблені модулі аналізу коду забезпечують комплексну оцінку програм за якістю, складністю та підтримуваністю, а також виявляють потенційні недоліки у роботах здобувачів.

Для навчання та тестування моделі сформовано вибірку з 100 лабораторних робіт, класифікованих за рівнями якості, та створено вектори ознак, що зберігаються у БД. Навчена модель ML продемонструвала здатність прогнозувати рівень якості програмного коду на основі числових ознак, отриманих із модулів аналізу. Тестування показало точність класифікації на рівні 80%, що свідчить про ефективність підходу та потребу подальшого налаштування гіперпараметрів, балансування вибірки та відбору найбільш інформативних ознак.

Інтеграція моделі ML у концептуальну модель даних системи забезпечила взаємодію аналізу та оцінювання коду з базою даних, створюючи єдину інтегровану систему, яка підтримує повний цикл автоматизованої перевірки лабораторних робіт – від завантаження студентом до отримання оцінки викладачем.

ВИСНОВКИ

Кваліфікаційна робота була присвячена розробці концептуальної моделі автоматизованої перевірки лабораторних робіт на мові Python методами машинного навчання.

Завдяки комплексному підходу, що поєднує аналіз предметної області, моделювання, експериментальні дослідження та програмну реалізацію, було досягнуто поставленої мети – створення концептуальної моделі системи, здатної забезпечити об’єктивність, масштабованість та високу якість оцінювання студентських робіт.

У процесі дослідження проведено аналіз існуючих систем автоматизованої перевірки програмного коду, таких як CodeRunner, CodeHS Autograders та Test My Code (TMC). Виявлено, що ці рішення мають обмеження у гнучкості, можливостях інтеграції, а також не завжди враховують індивідуальні особливості студентських завдань. Це обґрунтовує необхідність створення власної системи, адаптованої до освітніх потреб і специфіки мови програмування Python.

Теоретичні засади дослідження охоплювали аналіз традиційних методів перевірки лабораторних робіт, які характеризуються високою трудомісткістю, суб’єктивністю та низькою масштабованістю. На основі проведеного SWOT-аналізу визначено переваги і недоліки таких підходів та окреслено перспективи їх удосконалення за рахунок інтеграції технологій штучного інтелекту і машинного навчання.

Важливим етапом роботи став вибір оптимального алгоритму машинного навчання. Проведено порівняльний аналіз популярних методів (логістичної регресії, SVM, Random Forest, нейронних мереж, Gradient Boosting). За результатами дослідження найкращі показники точності, інтерпретованості та стійкості до шумових даних продемонстрував алгоритм Random Forest, який і було обрано як основу для побудови моделі класифікації програмного коду.

Розроблено архітектуру концептуальної моделі системи, що включає низку функціональних модулів: завантаження та попередньої обробки коду, лексичного й синтаксичного аналізу, формування векторів ознак, машинного навчання, тестування та формування звітів. У моделі реалізовано механізми автентифікації користувачів, адміністрування завдань і зберігання результатів у базі даних PostgreSQL. Мова

програмування Python була обрана як основна платформа завдяки своїй універсальності, широкому набору бібліотек для аналізу коду (ast, pylint, radon, flake8) та машинного навчання (scikit-learn).

У результаті моделювання та апробації системи отримано підтвердження її ефективності. Використання методу Random Forest дозволило досягти точності класифікації близько 86–90% залежно від набору навчальних даних. Автоматизована перевірка забезпечує стабільну якість оцінювання незалежно від кількості робіт та усуває людський фактор, що особливо важливо при масовому дистанційному навчанні. Проведене тестування показало, що час перевірки однієї лабораторної роботи скорочено у 20 разів, а трудовитрати викладача – майже на 95% у порівнянні з ручною перевіркою.

Отримані результати мають значне практичне значення. Розроблена система може бути впроваджена у навчальний процес закладів вищої освіти, що дозволить підвищити ефективність та об’єктивність оцінювання, забезпечити швидкий зворотний зв’язок зі студентами й сприяти розвитку навичок самоконтролю. Вона також може бути використана для створення інтерактивних освітніх платформ та інтелектуальних навчальних середовищ, орієнтованих на формування індивідуальних освітніх траєкторій.

Перспективи подальших досліджень полягають у розширенні функціональних можливостей системи, зокрема у впровадженні глибоких нейронних мереж для підвищення точності класифікації, автоматичному виявленні плагіату, розпізнаванні стилю програмування конкретного автора, а також у пристосуванні розробленої моделі для інших мов програмування.

Таким чином, результати роботи підтвердили ефективність запропонованої моделі автоматизованої перевірки лабораторних робіт на мові Python. Система відповідає сучасним тенденціям цифровізації освіти, сприяє зменшенню навантаження на викладачів, підвищенню якості підготовки фахівців та створює передумови для подальшого розвитку інтелектуальних освітніх технологій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Григор’в О. Машинне навчання у перевірці програмних кодів: ефективність алгоритму Random Forest. *Кібербезпека в сучасному світі: актуальні виклики* : матер. VI міжнар. наук.-практ. конф. (28 листопада 2025 р.). Одеса, 2025 (депоновано).
2. Прокоп Ю.В., Трофименко О.Г. Аналіз популярних на іт-ринку праці мов програмування. URL: https://surl.li/eabzxr
3. CodeRunner. URL: https://coderunner.org.nz
4. CodeHS Autograders. URL: https://help.codehs.com/en/articles/3335521-codehs-autograders
5. TestMyCode. URL: https://testmycode.github.io
6. Пермінова Л.А., Частник О.С., Діомідова Н.Ю. Порівняльний аналіз традиційних і альтернативних методів оцінювання знань студентів. Журнал «Перспективи та інновації науки»(Серія «Педагогіка», Серія «Психологія», Серія «Медицина»).№ 7(41). 2024. С. 431-442. https://doi.org/10.52058/2786-4952-2024-7(41)-431-442
7. Як провести SWOT-аналіз: інструкція, поради та приклади. URL: https://happymonday.ua/yak-provesty-swot-analiz
8. Що таке машинне навчання: як працює та де використовується. URL: https://surl.li/sadslb
9. Гуревич, Р. С., Коношевський, Л. Л., Коношевський, О. Л., Воєвода, А. Л., Люльчак,С. Ю. (2024). Інтеграція штучного інтелекту в сферу освіти: проблеми, виклики, загрози, перспективи. Сучасні інформаційні технології та інноваційні методики навчання в підготовці фахівців: методологія, теорія, досвід, проблеми, 72. С. 171–186. https://doi.org/10.31652/2412-1142-2024-72-170-186
10. Головко Д.Ю. Штучний інтелект у діяльності педагога закладу професійної (професійно-технічної) освіти: навчально-методичний посібник. Біла Церква: БІНПО ДЗВО «УМО» НАПН України, 2024. 73 с. С. 33-34.
11. Machine Learning Lifecycle. URL: https://www.geeksforgeeks.org/machine-learning/machine-learning-lifecycle
12. Logistic Regression: Definition, Use Cases, Implementation. URL:

https://encord.com/blog/what-is-logistic-regression

13. Support Vector Machine (SVM): A Complete Machine Learning Guide. URL: https://www.snowflake.com/en/fundamentals/support-vector-machine
14. What is random forest? URL: https://www.ibm.com/think/topics/random-forest
15. Що таке нейронні мережі та де їх використовують? URL: https://incrypted.com/ua/shcho-take-nejromerezhi
16. What is gradient boosting? URL: https://www.snowflake.com/en/fundamentals/what-is-gradient-boosting
17. Функціональні та нефункціональні вимоги. URL: https://surl.li/rgiiqh
18. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: https://www.postgresql.org
19. Що таке PostgreSQL і для чого використовується? URL: https://foxminded.ua/postgresql-shcho-tse
20. Рейтинг Мов Програмування 2025. URL: https://merehead.com/ua/blog/rate-programming-languages-2025
21. Рейтинг мов програмування в Україні та світі 2025. URL: https://surl.li/heowtt
22. Мова програмування Python для машинного навчання та штучного інтелекту. URL: https://surl.lt/irfpmr
23. Best Python libraries for Machine Learning. URL: https://surl.li/htmdkv
24. NumPy. URL: https://numpy.org
25. SciPy. URL: https://scipy.org
26. scikit-learn Machine Learning in Python. URL: https://scikit-learn.org/stable
27. An end-to-end platform for machine learning. URL: https://www.tensorflow.org
28. A superpower for ML developers. URL: https://keras.io
29. PyTorch. URL: https://pytorch.org/#
30. Exploring Python’s Abstract Syntax Trees (AST): Write Code That Understands Code. URL: https://surl.li/tcmbhc
31. Pylint module in Python. URL: https://www.geeksforgeeks.org/python/pylint-module-in-python
32. Radon – computes various metrics from Python code. URL:

https://www.linuxlinks.com/radon-computes-various-metrics-python-code

33. Flake8 – Python Linting and Style Enforcement. URL: https://dev.to/imrrobot/flake8-python-linting-and-style-enforcement-l1
34. Юрченко І.В., Голик Д.Ю. Застосування методів виявлення ознак для машинного навчання засобами мови Python. URL: https://surl.li/zextye
35. Python lexical structure. URL: https://zetcode.com/python/lexical-structure
36. Token, Patterns, and Lexemes. URL: https://www.geeksforgeeks.org/compiler-design/token-patterns-and-lexems
37. Що таке синтаксис Python і чому він важливий. URL: https://foxminded.ua/python-syntaksys
38. Introduction to Abstract Syntax Trees in Python. URL: https://earthly.dev/blog/python-ast/
39. Семантика в програмуванні: розбираємося з нюансами. URL: https://foxminded.ua/shcho-take-semantyka-v-prohramuvanni
40. Code Embeddings: Unlocking Smarter Programming AI. URL: https://www.iterate.ai/ai-glossary/code-embeddings
41. Бажан, Т. О. Порівняльний аналіз методів машинного навчання для побудови прогнозів. *Сучасний захист інформації*, 2024, 4(60), 125–130. https://doi.org/10.31673/2409-7292.2024.040013.
42. Random Forest, Explained: A Visual Guide with Code Examples. URL: https://surli.cc/dwcvok
43. Understanding TF-IDF (Term Frequency-Inverse Document Frequency). URL: https://surl.li/kznxwz
44. Intro to Model Tuning: Grid and Random Search. URL: https://surl.li/lpasnu
45. Покровський А.М. Засіб вимірювання метрик вихідного коду fortran за допомогою синтаксичного аналізу. *Проблеми програмування*. 2021. № 1. С. 26-35. https://doi.org/10.15407/pp2021.01.026

ДОДАТКИ

Додаток А. Фрагменти коду з реалізації модулів аналізу коду

```
"""
Модуль лексичного аналізу програмного коду Python
"""

import tokenize
import io
import keyword
from typing import Dict, List, Tuple
from collections import Counter
import token as token_module


2 usages
class LexicalAnalyzer:
    def __init__(self):
        self.tokens = []
        self.token_types = []
        self.token_strings = []

    2 usages
    def tokenize_code(self, code: str) -> List[tokenize.TokenInfo]:
        try:
            tokens = list(tokenize.generate_tokens(
                io.StringIO(code).readline
            ))

            self.tokens = [
                t for t in tokens
                if t.type not in (token_module.ENCODING, token_module.ENDMARKER)
                   and t.string.strip()
            ]
            self.token_types = [t.type for t in self.tokens]
            self.token_strings = [t.string for t in self.tokens]
            return self.tokens
        except tokenize.TokenError as e:
            print(f"Помилка токенізації: {e}")
            return []
```

```
    def count_operators(self) -> int:
        return sum(1 for t in self.tokens if t.type == token_module.OP)

    1 usage
    def count_literals(self) -> Tuple[int, int, int]:
        numbers = sum(1 for t in self.tokens if t.type == token_module.NUMBER)
        strings = sum(1 for t in self.tokens if t.type == token_module.STRING)
        total = numbers + strings
        return numbers, strings, total

    1 usage
    def count_comments(self) -> int:
        return sum(1 for t in self.tokens if t.type == token_module.COMMENT)

    2 usages
    def get_unique_tokens(self) -> int:
        return len(set(self.token_strings))

    1 usage
    def get_avg_identifier_length(self) -> float:
        identifiers = [
            t.string for t in self.tokens
            if t.type == token_module.NAME and not keyword.iskeyword(t.string)
        ]
        if not identifiers:
            return 0.0
        return sum(len(i) for i in identifiers) / len(identifiers)

    1 usage
    def get_token_distribution(self) -> Dict[str, int]:

        type_names = {
            token_module.NAME: 'NAME',
            token_module.NUMBER: 'NUMBER',
            token_module.STRING: 'STRING',
            token_module.OP: 'OPERATOR',
            token_module.COMMENT: 'COMMENT',
            token_module.NEWLINE: 'NEWLINE',
            token_module.INDENT: 'INDENT',
            token_module.DEDENT: 'DEDENT'
        }
```

```
    distribution = Counter(self.token_types)
    return {
        type_names.get(k, f'TYPE_{k}'): v
        for k, v in distribution.items()
    }

1 usage
def generate_unigrams(self) -> List[str]:
    return self.token_strings.copy()

1 usage
def generate_bigrams(self) -> List[Tuple[str, str]]:
    if len(self.token_strings) < 2:
        return []
    return [
        (self.token_strings[i], self.token_strings[i + 1])
        for i in range(len(self.token_strings) - 1)
    ]

1 usage
def generate_trigrams(self) -> List[Tuple[str, str, str]]:
    if len(self.token_strings) < 3:
        return []
    return [
        (self.token_strings[i],
         self.token_strings[i + 1],
         self.token_strings[i + 2])
        for i in range(len(self.token_strings) - 2)
    ]

3 usages
def get_ngrams_frequency(self, n: int = 2, top_k: int = 10) -> List[Tuple]:

    if n == 1:
        ngrams = self.generate_unigrams()
    elif n == 2:
        ngrams = self.generate_bigrams()
    elif n == 3:
        ngrams = self.generate_trigrams()
    else:
```

...

```
def print_features(features: Dict):

    print("ЛЕКСИЧНІ ОЗНАКИ КОДУ")


    print("\n БАЗОВІ МЕТРИКИ:")
    print(f"  Загальна кількість токенів: {features['total_tokens']}")
    print(f"  Унікальні токени: {features['unique_tokens']}")
    print(f"  Ключові слова: {features['keywords_count']}")
    print(f"  Ідентифікатори: {features['identifiers_count']}")
    print(f"  Оператори: {features['operators_count']}")
    print(f"  Літерали: {features['literals_count']} "
          f"(числа: {features['numbers_count']}, рядки: {features['strings_count']})")
    print(f"  Коментарі: {features['comments_count']}")

    print("\n СТАТИСТИЧНІ МЕТРИКИ:")
    print(f"  Середня довжина ідентифікаторів: {features['avg_identifier_length']}")
    print(f"  Коефіцієнт унікальності: {features['unique_ratio']}")
    print(f"  Частка ключових слів: {features['keyword_ratio']}")
    print(f"  Частка операторів: {features['operator_ratio']}")
    print(f"  Частка літералів: {features['literal_ratio']}")

    print("\n РОЗПОДІЛ ТОКЕНІВ:")
    for token_type, count in features['token_distribution'].items():
        print(f"  {token_type}: {count}")

    print("\n ТОП-10 УНІГРАМ:")
    for ngram, freq in features['top_unigrams']:
        print(f"  '{ngram}': {freq}")

    print("\n ТОП-10 БІГРАМ:")
    for ngram, freq in features['top_bigrams']:
        print(f"  {ngram}: {freq}")

    print("\n ТОП-5 ТРІГРАМ:")
    for ngram, freq in features['top_trigrams']:
        print(f"  {ngram}: {freq}")
```

...

```python
"""
Модуль синтаксичного аналізу програмного коду Python
"""

import ast
from typing import Dict, List, Tuple
from collections import defaultdict


1 usage
class ASTAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.functions = 0
        self.classes = 0
        self.methods = 0
        self.if_statements = 0
        self.for_loops = 0
        self.while_loops = 0
        self.try_except = 0
        self.with_statements = 0
        self.list_comprehensions = 0
        self.dict_comprehensions = 0
        self.set_comprehensions = 0
        self.lambda_functions = 0

        self.current_depth = 0
        self.max_nesting_depth = 0
        self.function_depth = 0

        self.total_branches = 0

        self.function_calls = []
        self.builtin_calls = []

        self.imports = []

        self.assignments = 0
        self.global_vars = []

        self.function_details = []

        self.nesting_tree = []
```

```
    14 usages
    def _update_depth(self, increment=True):
        if increment:
            self.current_depth += 1
            self.max_nesting_depth = max(self.max_nesting_depth, self.current_depth)
        else:
            self.current_depth -= 1

    1 usage
    def visit_FunctionDef(self, node):
        self.functions += 1
        self.function_depth += 1

        func_info = {
            'name': node.name,
            'args_count': len(node.args.args),
            'has_decorators': len(node.decorator_list) > 0,
            'has_docstring': ast.get_docstring(node) is not None,
            'line_number': node.lineno,
            'is_method': self.function_depth > 1
        }
        self.function_details.append(func_info)

        if func_info['is_method']:
            self.methods += 1

        self._update_depth(True)
        self.total_branches += 1
        self.generic_visit(node)
        self._update_depth(False)
        self.function_depth -= 1
```

...

```
class SyntacticAnalyzer:
    def __init__(self):
        self.tree = None
        self.source_code = ""
        self.analyzer = None
```

```
    def parse_code(self, code: str) -> bool:
        try:
            self.source_code = code
            self.tree = ast.parse(code)
            return True
        except SyntaxError as e:
            print(f"Синтаксична помилка: {e}")
            return False

    def analyze(self, code: str) -> Dict:
        if not self.parse_code(code):
            return {}

        self.analyzer = ASTAnalyzer()
        self.analyzer.visit(self.tree)

        lines_of_code = len([l for l in code.split('\n') if l.strip()])

        cyclomatic_complexity = self.analyzer.total_branches + 1

        features = {
            'functions_count': self.analyzer.functions,
            'classes_count': self.analyzer.classes,
            'methods_count': self.analyzer.methods,
            'if_count': self.analyzer.if_statements,
            'for_loops': self.analyzer.for_loops,
            'while_loops': self.analyzer.while_loops,
            'total_loops': self.analyzer.for_loops + self.analyzer.while_loops,
            'try_except_count': self.analyzer.try_except,
            'with_statements': self.analyzer.with_statements,

            'list_comprehensions': self.analyzer.list_comprehensions,
            'dict_comprehensions': self.analyzer.dict_comprehensions,
            'set_comprehensions': self.analyzer.set_comprehensions,
            'total_comprehensions': (
                    self.analyzer.list_comprehensions +
                    self.analyzer.dict_comprehensions +
                    self.analyzer.set_comprehensions
            ),
            'lambda_functions': self.analyzer.lambda_functions,
```

...

```python
def print_ast_tree(self, indent: int = 0, max_depth: int = 5):

    if self.tree is None:
        print("AST дерево не побудовано")
        return

    self._print_node(self.tree, indent, 0, max_depth)

3 usages
def _print_node(self, node, indent: int, current_depth: int, max_depth: int):
    if current_depth > max_depth:
        return

    node_type = type(node).__name__
    indent_str = "  " * indent

    if isinstance(node, ast.FunctionDef):
        print(f"{indent_str}├─ {node_type}: {node.name}")
    elif isinstance(node, ast.ClassDef):
        print(f"{indent_str}├─ {node_type}: {node.name}")
    elif isinstance(node, ast.Name):
        print(f"{indent_str}├─ {node_type}: {node.id}")
    elif isinstance(node, ast.Constant):
        value = repr(node.value)[:30]
        print(f"{indent_str}├─ {node_type}: {value}")
    elif isinstance(node, ast.Call):
        if isinstance(node.func, ast.Name):
            print(f"{indent_str}├─ {node_type}: {node.func.id}()")
        else:
            print(f"{indent_str}├─ {node_type}")
    else:
        print(f"{indent_str}├─ {node_type}")

    for field, value in ast.iter_fields(node):
        if isinstance(value, list):
            for item in value:
                if isinstance(item, ast.AST):
                    self._print_node(item, indent + 1, current_depth + 1, max_depth)
        elif isinstance(value, ast.AST):
            self._print_node(value, indent + 1, current_depth + 1, max_depth)
```

...

```
"""
Модуль аналізу складності програмного коду Python
"""

from radon.complexity import cc_visit, cc_rank
from radon.metrics import h_visit, mi_visit, mi_rank
from radon.raw import analyze
import ast
from typing import Dict, List, Tuple
import math
```

...

```
    def _cyclomatic_complexity(self) -> Dict:
        try:
            complexity_results = cc_visit(self.code)

            if not complexity_results:
                return {
                    'cyclomatic_complexity_avg': 0,
                    'cyclomatic_complexity_max': 0,
                    'cyclomatic_complexity_total': 0,
                    'complex_functions_count': 0,
                    'complexity_rank': 'A',
                    'complexity_details': []
                }

            complexities = [item.complexity for item in complexity_results]

            complex_functions = sum(1 for c in complexities if c > 10)

            avg_complexity = sum(complexities) / len(complexities)
            rank = cc_rank(avg_complexity)

            details = []
            for item in complexity_results:
                details.append({
                    'name': item.name,
                    'complexity': item.complexity,
                    'rank': cc_rank(item.complexity),
                    'line': item.lineno,
                    'type': item.classname if item.classname else 'function'
                })

            return {
                'cyclomatic_complexity_avg': round(avg_complexity, 2),
                'cyclomatic_complexity_max': max(complexities),
                'cyclomatic_complexity_total': sum(complexities),
                'complex_functions_count': complex_functions,
                'complexity_rank': rank,
                'complexity_details': details
            }
```

```python
"""
Модуль перевірки стилю програмного коду Python
"""
import subprocess
import json
import re
from pathlib import Path
from datetime import datetime

1 usage
def run_pylint(file_path: str) -> dict:
    try:
        result_json = subprocess.run(
            ["pylint", "--output-format=json", file_path],
            capture_output=True, text=True, check=False
        )
        pylint_data = json.loads(result_json.stdout) if result_json.stdout.strip() else []

        errors = sum(1 for item in pylint_data if item.get("type") == "error")
        warnings = sum(1 for item in pylint_data if item.get("type") == "warning")

        result_text = subprocess.run(
            ["pylint", file_path],
            capture_output=True, text=True, check=False
        )
        text_output = result_text.stderr + result_text.stdout

        match = re.search(r"Your code has been rated at ([\d\.]+)/10", text_output)
        score = float(match.group(1)) if match else 0.0

        score = round(score * 3 / 10, 2)

        return {
            "score": score,
            "errors": errors,
            "warnings": warnings
        }

    except Exception as e:
        print(f"[Помилка] Не вдалося запустити pylint: {e}")
        return {"score": 0, "errors": 0, "warnings": 0}
```

```
1 usage
def run_flake8(file_path: str) -> int:
    try:
        result = subprocess.run(
            ["flake8", file_path, "--count"],
            capture_output=True, text=True, check=False
        )
        return int(result.stdout.strip() or 0)
    except Exception as e:
        print(f"[Помилка] Не вдалося запустити flake8: {e}")
        return 0
1 usage
def analyze_code(file_path: str) -> dict:
    if not Path(file_path).exists():
        raise FileNotFoundError(f"Файл {file_path} не знайдено.")

    pylint_result = run_pylint(file_path)
    flake8_violations = run_flake8(file_path)

    base_score = pylint_result["score"]
    penalty = min(flake8_violations / 3, 1.5)  # штраф не більше 5 балів
    final_score = max(0, round(base_score - penalty, 2))

    return {
        "file": file_path,
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "оцінка_якості_коду": final_score,
        "pep8_порушення": flake8_violations,
        "попередження": pylint_result["warnings"],
        "помилки": pylint_result["errors"]
    }
```

...

```python
"""
Модуль динамічного тестування Python коду
"""

import sys
import io
import time
import tracemalloc
import traceback
from typing import List, Dict, Any, Tuple, Optional
from dataclasses import dataclass, field
import contextlib


@dataclass
class TestCase:
    name: str
    inputs: List[Any]
    expected_output: Any
    timeout: float = 5.0


5 usages
@dataclass
class TestResult:
    name: str
    passed: bool
    actual_output: Any
    expected_output: Any
    execution_time: float
    memory_usage: float  # в МБ
    error_message: Optional[str] = None


3 usages
@dataclass
class DynamicTestMetrics:
    tests_passed_percent: float
    avg_execution_time: float
    memory_usage: float
    has_runtime_errors: bool
    total_tests: int = 0
```

...

```
class DynamicTester:
    def __init__(self, code: str, function_name: str):
        self.code = code
        self.function_name = function_name
        self.namespace = {}
        self._compile_code()

    1 usage
    def _compile_code(self) -> None:
        try:
            exec(self.code, self.namespace)
            if self.function_name not in self.namespace:
                raise ValueError(f"Функція '{self.function_name}' не знайдена в коді")
        except Exception as e:
            raise ValueError(f"Помилка компіляції коду: {e}")

    1 usage
    def _run_single_test(self, test_case: TestCase) -> TestResult:
        func = self.namespace[self.function_name]
        tracemalloc.start()
        captured_output = io.StringIO()
        try:
            start_time = time.perf_counter()
            with contextlib.redirect_stdout(captured_output):
                actual_output = self._execute_with_timeout(
                    func,
                    test_case.inputs,
                    test_case.timeout
                )
            end_time = time.perf_counter()
            execution_time = end_time - start_time
```

...

Додаток Б. Фрагмент коду навчання та оцінювання якості моделі

...

```
def train_model():
    """Навчання моделі Random Forest"""
    print("\n" + "=" * 70)
    print("ЕТАП 2: НАВЧАННЯ МОДЕЛІ")
    print("=" * 70)

    # Завантаження даних
    print("\n Завантаження даних...")
    df = pd.read_csv('data.csv')
    X = df.iloc[:, :-1].values
    y = df.iloc[:, -1].values

    print(f"Завантажено {X.shape[0]} прикладів з {X.shape[1]} ознаками")

    # Розділення даних
    print("\n Розділення даних (70/15/15)...")
    X_temp, X_test, y_temp, y_test = train_test_split(
        X, y, test_size=0.15, random_state=42, stratify=y
    )

    val_ratio = 0.15 / (0.70 + 0.15)
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=val_ratio, random_state=42, stratify=y_temp
    )

    print(f"  Train: {len(X_train)} | Val: {len(X_val)} | Test: {len(X_test)}")

    # Оптимізація гіперпараметрів
    print("\n Оптимізація гіперпараметрів (Grid Search)...")
    print("   Це може зайняти кілька хвилин...")

    param_grid = {
        'n_estimators': [100, 200],
        'max_depth': [10, 20, None],
        'min_samples_split': [2, 5],
        'min_samples_leaf': [1, 2],
        'max_features': ['sqrt', 'log2']
    }

    rf = RandomForestClassifier(random_state=42, n_jobs=-1)
```

```
grid_search = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    cv=5,
    scoring='f1_weighted',
    verbose=1,
    n_jobs=-1
)

grid_search.fit(X_train, y_train)

print(f"\n Найкращі параметри: {grid_search.best_params_}")
print(f"Найкраща F1-оцінка (CV): {grid_search.best_score_:.4f}")

# Навчання фінальної моделі
print("\n Навчання фінальної моделі...")
model = grid_search.best_estimator_

# Збереження моделі
with open('random_forest_model.pkl', 'wb') as f:
    pickle.dump(model, f)

print("Модель збережено у 'random_forest_model.pkl'")

return model, X_train, X_val, X_test, y_train, y_val, y_test
```

...

```python
def evaluate_model(model, X_test, y_test, X_train, y_train):
    # Передбачення
    y_pred = model.predict(X_test)

    # Метрики
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)

    print(f"\n РЕЗУЛЬТАТИ НА ТЕСТОВІЙ ВИБІРЦІ:")
    print(f"  Accuracy:  {accuracy:.3f} ({accuracy * 100:.1f}%)")
    print(f"  Precision: {precision:.3f} ({precision * 100:.1f}%)")
    print(f"  Recall:    {recall:.3f} ({recall * 100:.1f}%)")
    print(f"  F1-Score:  {f1:.3f} ({f1 * 100:.1f}%)")

    # Детальний звіт
    print(f"\n ДЕТАЛЬНИЙ ЗВІТ ПО КЛАСАХ:")
    class_names = ['Клас 0 (Відмінно)', 'Клас 1 (Добре)',
                   'Клас 2 (Задовільно)', 'Клас 3 (Незадовільно)']
    print(classification_report(y_test, y_pred, target_names=class_names, zero_division=0))

    # Матриця помилок
    print(f"\n МАТРИЦЯ ПОМИЛОК:")
    cm = confusion_matrix(y_test, y_pred)
    print(cm)

    # Візуалізація матриці помилок
    plt.figure(figsize=(10, 8))
    sns.heatmap(
        cm, annot=True, fmt='d', cmap='Blues',
        xticklabels=['Клас 0', 'Клас 1', 'Клас 2', 'Клас 3'],
        yticklabels=['Клас 0', 'Клас 1', 'Клас 2', 'Клас 3'],
        cbar_kws={'label': 'Кількість'}
    )
```

```
plt.ylabel('Справжній клас', fontsize=12, fontweight='bold')
plt.xlabel('Передбачений клас', fontsize=12, fontweight='bold')
plt.title('Матриця помилок', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.savefig('confusion_matrix.png', dpi=300, bbox_inches='tight')
print("Матрицю помилок збережено у 'confusion_matrix.png'")
plt.close()

# Крос-валідація
print(f"\nКРОС-ВАЛІДАЦІЯ (5-fold):")
cv_scores = cross_val_score(
    model, X_train, y_train,
    cv=5, scoring='f1_weighted'
)
print(f"  F1 scores: {[f'{s:.3f}' for s in cv_scores]}")
print(f"  Середнє F1: {cv_scores.mean():.3f} (±{cv_scores.std():.3f})")

# Порівняння з вимогами
print(f"\n ПОРІВНЯННЯ З ВИМОГАМИ:")
requirements = {
    'Accuracy': (0.80, accuracy),
    'Precision': (0.82, precision),
    'Recall': (0.80, recall),
    'F1-Score': (0.77, f1)
}

print("-" * 70)
print(f"{'Метрика':<15} {'Вимога':<15} {'Результат':<15} {'Виконання':<15}")
print("-" * 70)

all_passed = True
for metric_name, (required, achieved) in requirements.items():
    passed = achieved >= required
    all_passed = all_passed and passed
    status = "Так" if passed else "Ні"
    print(f"{metric_name:<15} {f'≥ {required * 100:.0f}%':<15} "
          f"{f'{achieved * 100:.1f}%':<15} {status:<15}")
```