Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Дослідження методів розпізнавання українських текстів, згенерованих великими мовними моделями (Large Language Models) / Exploring Methods for Detecting Ukrainian Texts Generated by Large Language Models

Текстова частина до курсової роботи
за спеціальністю „Комп'ютерні науки" 6.050101

Керівник дипломної роботи
Ст. Викладач Кундік К.В.

_____

(підпис)

"____" _____ 2025 р.

Виконав студент Гоголь А. О.

"11" Травня 2025 р.

Київ 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мультимедійних систем,

доц., к.ф.-м.н.

_____О. П. Жежерун

(підпис)

„____”_____2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на дипломну роботу

студенту Гоголю А.О. факультету Інформатики 4 курсу

ТЕМА Дослідження методів розпізнавання українських текстів, згенерованих великими мовними моделями (Large Language Models) / Exploring Methods for Detecting Ukrainian Texts Generated by Large Language Models

Вихідні дані:

- Різні варіації моделей розпізнавання текстів
- Набори даних для навчання і валідації

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Вступ

Розділ 1: Основи мовних моделей, дослідження існуючих методів та інструментів. Вибір методів для імплементації.

Розділ 2: Збір даних.

Розділ 3: Імплементація алгоритмів. Проведення експериментів. Аналіз результатів.

Висновки

Список літератури

Додатки

Дата видачі „___” _____ 2025 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Дослідження методів розпізнавання українських текстів, згенерованих великими мовними моделями (Large Language Models) / Exploring Methods for Detecting Ukrainian Texts Generated by Large Language Models

Календарний план виконання роботи:

Назва етапу курсової роботи	Термін виконання етапу	Примітка
Отримання завдання на курсову роботу.	04.11.2024	
Огляд технічної літератури за темою роботи.	30.01.2025	
Виконати аналіз сучасних методів.	15.02.2025	
Вибір та деталізація основних алгоритмів для адаптація.	30.02.2025	
Програмування обраних алгоритмів.	15.03.2025	
Збір даних для навчання та валідації моделей.	18.04.2025	
Навчання моделей та їх валідація.	21.04.2025	
Аналіз результатів та їх покращення.	30.04.2025	
Написання текстової частини роботи та слайдів презентації.	02.05.2025	

Аналіз отриманих результатів з керівником.	05.05.2025	
Корегування роботи	08.05.2025	
Попередній захист дипломної роботи	26.05.2025	
Захист дипломної роботи	09.06.2025	

Студент **Гоголь А.О.**

Керівник **Кундік К.В.**

“_____”_____

Зміст

Анотація

Дипломна робота присвячена дослідженню та адаптації методів детекції українськомовних текстів, згенерованих великими мовними моделями. Мета роботи – розробка ефективного алгоритму для розпізнавання походження українського тексту: написаного людиною чи згенерованого штучним інтелектом.

У дослідженні проаналізовано сучасні підходи до виявлення згенерованого мовними моделями тексту та здійснено їх адаптацію для української мови. Методи детекції систематизовано на дві категорії: ті, що не потребують попереднього навчання (black-box та white-box варіанти алгоритму DNA-GPT), та ті, що базуються на машинному навчанні. Для експериментальної перевірки сформовано збалансований набір даних українських текстів, що включає як оригінальні людські тексти, так і синтезовані за допомогою мовних моделей. На цьому наборі даних проведено оцінку ефективності реалізованих алгоритмів, включаючи донавчання класифікатора на базі моделі BERT. Валідацію результатів здійснено на текстах, згенерованих двома провідними мовними моделями: GPT-4o-mini та Llama 3:70B.

Ключові слова: великі мовні моделі, машинне навчання, DNA-GPT, Llama, GPT-4o-mini, BERT, детекція AI-тексту

Вступ

З 2020 року, після виходу моделі GPT-3 від OpenAI[1], великі мовні моделі набули величезної популярності завдяки здатності генерувати змістовні тексти з мінімальними вказівками від користувача. Інструменти на кшталт ChatGPT[2], LLaMA[2], Claude[3], Gemini[4] та інші швидко знайшли застосування в журналістиці, освіті, маркетингу, науковій діяльності та багатьох інших сферах. З кожним роком технологія вдосконалювалася, і відрізнити згенерований мовною моделлю текст від людського ставало дедалі складніше. Це породило серйозні виклики для інформаційної безпеки, включаючи ризики поширення дезінформації, проведення маніпулятивних кампаній та порушення академічної доброчесності.

У відповідь на ці загрози почали з'являтися спеціалізовані методи та сервіси для виявлення згенерованого мовними моделями тексту. Станом на 2025 рік функціонують численні платформи – ZeroGPT[5], Copyleaks[6], OriginalityAI[7] та інші – що пропонують послуги з розпізнавання походження тексту. Однак переважна більшість цих рішень орієнтована на англомовний контент і не враховує специфічні морфологічні, синтаксичні та стилістичні особливості української мови. Це суттєво знижує їх ефективність для українськомовних текстів і створює ризики хибнопозитивних результатів, що особливо критично в контексті академічної етики та журналістської відповідальності. Додатковою проблемою є закритість алгоритмів більшості комерційних сервісів – відсутність інформації про методологію, навчальні датасети та результати валідації ускладнює оцінку їх надійності.

Зазначені проблеми визначили мотивацію даного дослідження. Мета роботи – аналіз існуючих методів детекції AI-згенерованого тексту та їх адаптація для ефективної роботи з українськомовним контентом. Для

досягнення цієї мети було проведено систематичне дослідження сучасних підходів до виявлення тексту згенерованого мовними моделями, сформовано репрезентативний набір даних українських текстів, розроблено та імплементовано адаптовані версії алгоритмів детекції, а також здійснено комплексну оцінку їх ефективності. Результати роботи можуть бути застосовані для виявлення порушень академічної доброчесності, протидії дезінформаційним кампаніям та забезпечення достовірності контенту в українськомовному інформаційному просторі.

Розділ 1: Основи мовних моделей, дослідження існуючих методів та інструментів. Вибір методів для імплементації

1.1 Коротка історія мовних моделей

Еволюція мовних моделей пройшла кілька ключових етапів, кожен з яких приніс фундаментальні зміни в підходах до обробки природної мови.

Ранні статистичні моделі базувалися на n-грамних алгоритмах, які передбачали наступне слово, спираючись на обмежений контекст із попередніх n-1 слів. Попри свою простоту та обчислювальну ефективність, ці моделі мали критичні обмеження: нездатність враховувати довгі залежності в тексті та генерація часто незв'язних або граматично некоректних речень.

Прогрес в області глибокого навчання привів до появи рекурентних нейронних мереж[9] та їх удосконалених варіантів – LSTM[10] і GRU[11]. Ці архітектури вирішили проблему згасаючих градієнтів і дозволили моделям зберігати інформацію про довший контекст. Однак послідовна природа обробки даних в рекурентних нейроних мережах унеможливлювала ефективну паралелізацію, що суттєво обмежувало швидкість навчання на великих корпусах.

Фундаментальний прорив відбувся в 2017 році з публікацією роботи "Attention Is All You Need"[12], де було представлено архітектуру Transformer. Ключовою інновацією став механізм самоуваги (self-attention), який дозволив моделі одночасно аналізувати зв'язки між усіма позиціями в

послідовності, забезпечуючи як ефективне захоплення довгих залежностей, так і повну паралелізацію обчислень.

Період 2021-2025 років ознаменувався появою нового покоління мовних моделей: GPT-4 від OpenAI з покращеними мультимодальними можливостями, відкриті моделі LLaMA від Meta, Gemini від Google, Claude від Anthropic та інші. Сучасні дослідження фокусуються на підвищенні енергоефективності, зменшенні галюцинацій, покращенні точності та розробці механізмів контролю за генерацією. Ці моделі створюють нові виклики, включаючи необхідність розробки надійних методів детекції тексту згенерованого мовними моделями.

1.2 Обробка тексту. Токенізація. Ембедінги

Для обробки текстової інформації нейронними мережами необхідно перетворити природну мову на числові представлення. Цей процес включає два ключові етапи: токенізацію та векторизацію через ембедінги.

Токенізація – це процес розбиття тексту на дискретні одиниці названі токенами, які можуть бути словами, частинами слів або символами. Сучасні мовні моделі частіше за все використовують субсловну токенізацію (BPE[13], WordPiece[14], SentencePiece[15]), яка забезпечує оптимальний баланс між розміром словника та здатністю обробляти рідкісні слова. Кожному токену присвоюється унікальний числовий ідентифікатор згідно зі словником моделі.

Ембедінги – це багатовимірні векторні представлення токенів, які кодують їх семантичні та синтаксичні властивості. На відміну від простого one-hot кодування, ембедінги розміщують семантично близькі токени поруч у векторному просторі, що дозволяє моделі виявляти змістові зв'язки між словами.

Основні типи ембедінгів:

1. **Статичні ембедінги** (Word2Vec[16], GloVe[17]) – кожному слову відповідає фіксований вектор незалежно від контексту. Ці методи ефективні для багатьох задач, але не враховують полісемію та контекстуальні варіації значень.
2. **Контекстуалізовані ембедінги** (ELMo, BERT)[18] – векторне представлення токена динамічно змінюється залежно від оточуючого контексту. Це дозволяє розрізняти різні значення багатозначних слів і краще моделювати складні мовні явища.

3. **Сегментні ембедінги** – використовуються в моделях типу BERT для розрізнення різних частин вхідного тексту (наприклад, питання та контексту в задачах QA).

У сучасних мовних моделях ембедінги навчаються разом з усією моделлю методом зворотного поширення помилки, що дозволяє їм адаптуватися під конкретні завдання та захоплювати тонкі семантичні нюанси. Розмірність ембедінгів зазвичай варіюється від 512 до 4096 вимірів залежно від розміру моделі. Саме через високоякісні контекстуалізовані ембедінги сучасні мовні моделі досягають глибокого розуміння природної мови та здатності генерувати семантично коректний текст.

1.3 Архітектура трансформер

Робота "Attention Is All You Need" запропонувала нову іноваційну архітектуру Transformer, що базується на механізмі самоуваги (self-attention). Ця архітектура стала основою для всіх сучасних великих мовних моделей – GPT, BERT, LLaMA та інших. Її масштабованість дозволила створювати моделі з мільярдами параметрів, здатні генерувати текст, що важко відрізнити від людського.

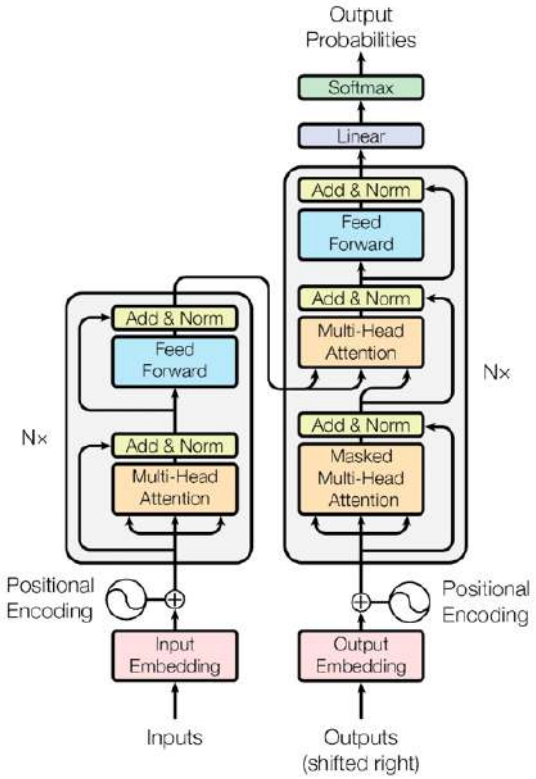

Рисунок 1.1 Схема роботи архітектури трансформер з оригінальної статті[12]

У цій роботі, в подальшому термін “мовна модель” буде означати саме мовну модель на базі архітектури трансформер.

1.4 Ймовірності генерації токенів. Температура

У трансформерних мовних моделях кожен крок генерації ґрунтується на ймовірнісному розподілі наступного токена. Нижче розглядаємо, як формується цей розподіл, яку роль відіграє параметр «температури» τ та як за перплексією оцінюють якість моделі. Всі ці визначення важливо чітко формулювати для подальшого розуміння даної роботи.

На кроці t модель повертає ненормалізований вектор z^t:

$$z^{(t)} = \left(z_1^{(t)}, z_2^{(t)}, \dots, z_V^{(t)}\right) \tag{1.1}$$

де V — розмір словника. Кожен $z_i^{(t)}$ називають логітом: це бал моделі для даного токену зі словника, що приблизно відповідає log p_i+C. Логіти можуть бути будь-якого знака; саме їхні відносні різниці визначають ймовірності після нормалізації.

Щоб далі перетворити логіти на коректний ймовірнісний розподіл, застосовують функцію softmax. Формулу функції наведено нижче:

$$s(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{1.2}$$

де $s(z_i)$ — ймовірність вибору елемента i;

z_i — логіт i-го токена;

n — кількість можливих токенів у словнику.

Отриманий розподіл $s(x_i)$ використовується для вибору наступного токена та обчислення метрик якості моделі. Щоб керувати ступенем випадковості, застосовують «температуру» τ, яка масштабує логіти перед softmax:

$$P_\tau(z_i) = \frac{e^{z_i/\tau}}{\sum_{k=1}^{V} e^{z_k/\tau}} \quad (1.3)$$

де $\tau \in \mathbb{R}^+$ — коефіцієнт, що впливає на "гостроту" розподілу.

Наступні значення τ можна інтерпретувати так:

τ = 1 — зберігає початковий розподіл;

τ < 1 — підсилює контраст, текст стає детермінованішим;

τ > 1 — робить розподіл рівномірнішим, збільшуючи різноманітність.

У практичних системах значення τ добирають емпірично: низькі τ корисні для точних, формальних відповідей, високі — для творчих завдань із багатьма прийнятними варіантами. Цей параметр особливо важливий для детекції: тексти, згенеровані з низькою температурою, легше виявляються алгоритмами детекції через їх вищу передбачуваність. Саме тому в експериментальній частині роботи досліджуються моделі з різними значеннями τ (0.1, 0.5, 0.8) для оцінки впливу температури на ефективність детекції.

1.5 Вибір моделей для дослідження

Для експериментальної частини обрано дві з одних найпопулярніших станом на початок 2025 року сучасних великих мовних моделей:

1. GPT-4o-mini від OpenAI – оптимізована версія моделі GPT-4, розроблена для практичного застосування в реальному часі. Модель забезпечує оптимальний баланс між швидкістю відповіді та якістю генерації, що зробило її стандартом для комерційних чат-ботів, асистентів та API-сервісів. Доступ здійснюється виключно через API OpenAI[19].
2. Llama 3 70B від Meta — потужна відкрита модель з 70 мільярдами параметрів, що стала основою для численних спеціалізованих рішень у дослідницькій спільноті. На відміну від GPT-4o-mini, Llama 3 доступна для локального розгортання та модифікації, що дозволяє глибоку кастомізацію під специфічні завдання. Проте в цій роботі, через обмежені обчислювальні потужності, модель теж використовувалась через API від Llama Cloud[20].

Ці дві моделі представляють різні, але впливові напрямки розвитку мовних моделей. GPT-4o-mini став де-факто стандартом серед комерційних API-рішень завдяки простоті інтеграції та стабільній якості[21]. Водночас Llama 3 70B утвердилася як еталонна модель у відкритій екосистемі, де цінується можливість повного контролю над моделлю та її адаптації під конкретні потреби[22]. Дослідження ефективності детекції текстів, згенерованих цими моделями, має практичну цінність, оскільки охоплює найпоширеніші джерела AI-контенту в сучасному інформаційному просторі.

1.6 Вибір алгоритмів для адаптації

Існує значна кількість методів для визначення походження тексту, тобто чи був він написаний людиною, чи згенерований мовною моделлю. Більшість із цих методів можна умовно розділити на наступні основні категорії:

1. Методи, що потребують навчання: Ці підходи передбачають тренування спеціалізованих моделей на наборах даних, що містять як людські, так і згенеровані тексти.
2. Методи, що не потребують навчання: Ці алгоритми працюють без попереднього етапу навчання на специфічних даних для детекції. Вони, у свою чергу, поділяються на:

   2.1 White-box рішення: Передбачають доступ до внутрішніх параметрів або архітектури мовної моделі, що перевіряється.

   2.2 Black-box рішення: Оперують лише вихідним текстом, не маючи доступу до самої моделі. (Детальніший опис цих підходів наведено у розділі 3).

Цей розподіл продемонстровано на Рисунку 2.1:

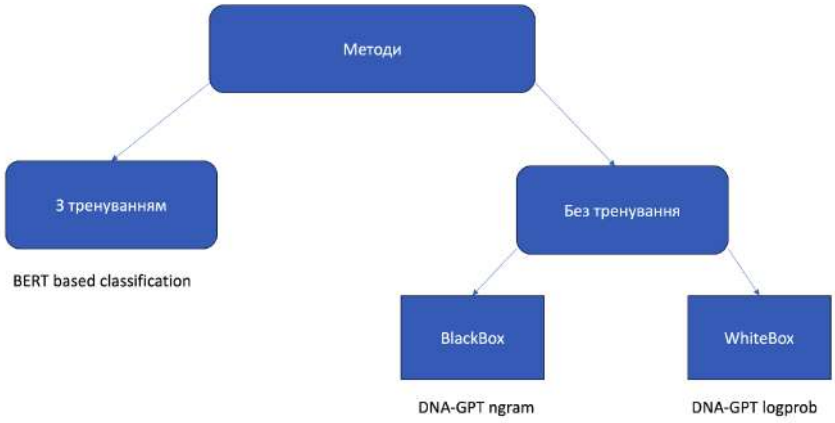

Рисунок 2.1 Ілюстрація різних підходів до вирішення поставленої задачі

Для першої категорії (методи, що потребують навчання) було обрано підхід на основі попередньо навченої моделі BERT. До архітектури BERT додається класифікаційний шар, і вся модель донавчається на підготовленому українськомовному наборі даних. Очікується, що завдяки глибокому контекстному розумінню лексики та синтаксису, BERT демонструватиме високу точність у розрізненні текстів, написаних людиною, та згенерованих штучним інтелектом.

Серед методів без навчання для імплементації обрано алгоритм DNA-GPT[23], який вирізняється універсальністю застосування. Алгоритм підтримує два режими роботи: white-box (з повним доступом до ймовірнісних розподілів моделі-генератора) та black-box (лише з доступом до згенерованого тексту). Ця гнучкість робить DNA-GPT практичним рішенням для різних реальних сценаріїв, де рівень доступу до цільової моделі може варіюватися.

1.7 Вибір інструментів та технологій для реалізації проекту

1.7.1 Мова програмування

Python 3.11[24] — використано як основну мову для реалізації всіх експериментів і скриптів, адже вона має розвинену екосистему бібліотек для машинного навчання.

1.7.2 Хмарні сервіси

OpenAI API — застосовано для отримання відповідей від GPT-4o-mini через HTTPS-запити у процесі порівняння моделей.
Llama Cloud API — використано для викликів Llama 3 70B, забезпечуючи віддалений інференс через HTTPS-запити.

1.7.3 Обробка та аналіз даних

Pandas[25] і NumPy[26] — зчитування, фільтрація та підготовка табличних даних.
NLTK[27] — базовий лінгвістичний препроцесинг (токенізація, видалення стоп-слів, стемінг).
Matplotlib[28] і Seaborn[29] — побудова графіків та візуалізація розподілів ознак.

1.7.4 Токенізація та робота з корпусами

Hugging Face Tokenizers[30] – сегментація тексту на всіх етапах.

Hugging Face Hub[31] — збереження корпусу у форматах JSON і CSV для спільного доступу та версіонування.

1.7.5 Навчання та донавчання моделей

PyTorch[32] — базова платформа для навчання BERT-класифікатора. Transformers від Hugging Face[33] — швидке завантаження архітектур і переднатренованих ваг.

1.8 Основні принципи існуючих алгоритмів

Методи детекції AI-згенерованого тексту можна класифікувати за типом аналізованих ознак на три основні категорії:

1. Статистичний аналіз тексту, що базується на виявленні аномалій у розподілі лінгвістичних характеристик: довжина речень, частотність функціональних слів, розподіл n-грам, лексичне різноманіття. Мовні моделі генерують текст із характерними статистичними патернами, що відрізняються від природної варіативності людського мовлення.

2. Аналіз ймовірностей генерації, який використовує послідовність ймовірностей токенів, обчислених моделлю під час генерації. Алгоритми типу DNA-GPT агрегують ці ймовірності в дискримінативні ознаки для класифікації походження тексту.

3. Криптографічні водяні знаки. Спеціалізовані методи вбудовування прихованих маркерів безпосередньо в процес генерації. Потребують модифікації моделі та кооперації з її розробниками.

У даній роботі фокус зроблено на перших двох підходах через їх універсальність та практичну застосовність. Ці методи не вимагають доступу до внутрішньої архітектури моделі чи співпраці з провайдерами API, що робить їх придатними для реальних сценаріїв використання.

1.9 WhiteBox vs BlackBox

Як зазначалося в розділі 2, існує фундаментальне розділення методів детекції за рівнем доступу до моделі-генератора.

BlackBox (чорний ящик): детектор працює лише з готовим текстом, без доступу до внутрішньої структури моделі. Єдине що ми маємо – це текст який згенерувала модель. Перевагами рішень запроваджених за цим принципом є швидка інтеграція, відсутність потреби знання архітектури моделі.

WhiteBox (білий ящик): алгоритм має доступ до ваг, активацій , внутрішніх проміжних шарів моделі, розподілу імовірностей вихідних токенів моделі. Перевагами рішень запроваджених за цим принципом є можливість виявити глибинні патерни генерації, вища точність. Але такі підходи мають більшу залежність від конкретної архітектури, потребу у значних обчислювальних ресурсах і праві на ваги моделі.

Залежно від категорії задачі, доведеться використати різні підходи. В цій роботі розглянуті і адаптовані методи для вирішення задач обох категорій.

1.10 DNA-GPT

Першим з алгоритмів, що був розглянутий і в подальшому адаптований під українську мову став DNA-GPT. Цей алгоритм не потребує додаткового навчання на даних і може бути імпелментований для вирішення як Black-box так і White-box задач.

Суть алгоритму полягає в тому, що мовні моделі, при генерації текста роблять це токен за токеном, при цьому кожний наступний токен вибирається серед найбільш імовірних після попереднього. Завдяки цьому, текст написаний мовною моделлю стає набагато більш передбачуваним, ніж текст написаний людиною. Цю передбачуваність можна виміряти декількома способами, залежно від задачі, що буде розглянуто пізніше. Також важливо, що хоч алгоритм не потребує додаткового навчання, для роботи йому потрібна мовна модель, вибір якої прямо впливає на результат. Якщо використана мовна модель, і мовна модель яку ми підозрюємо, що це вона генерувала текст – однакові, то ймовірність правильного виявлення значно зростає. Але загалом алгоритм ділиться на наступні етапи:

1) Сегментація тексту. Вихідний текст ділиться на дві частини: префікс X — усічена початкова послідовність і оригінальне продовження Y_0 — фрагмент тексту, який насправді йшов далі. Ілюстрація цього розділення зображена на Рисунку 3.1

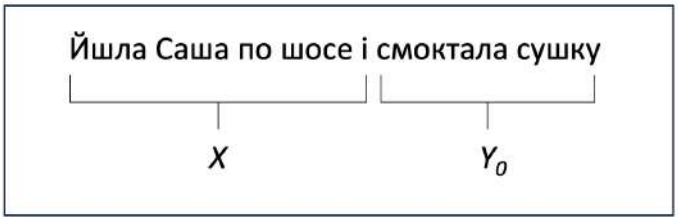

Рисунок 3.1 – Розділення початкового речення на 2 частини

2) Генерація альтернатив. На основі префікса X модель генерує K варіантів продовження $\{Y_1, Y_2, \ldots, Y_K\}$. Це проілюстровано на Рисунку 3.2

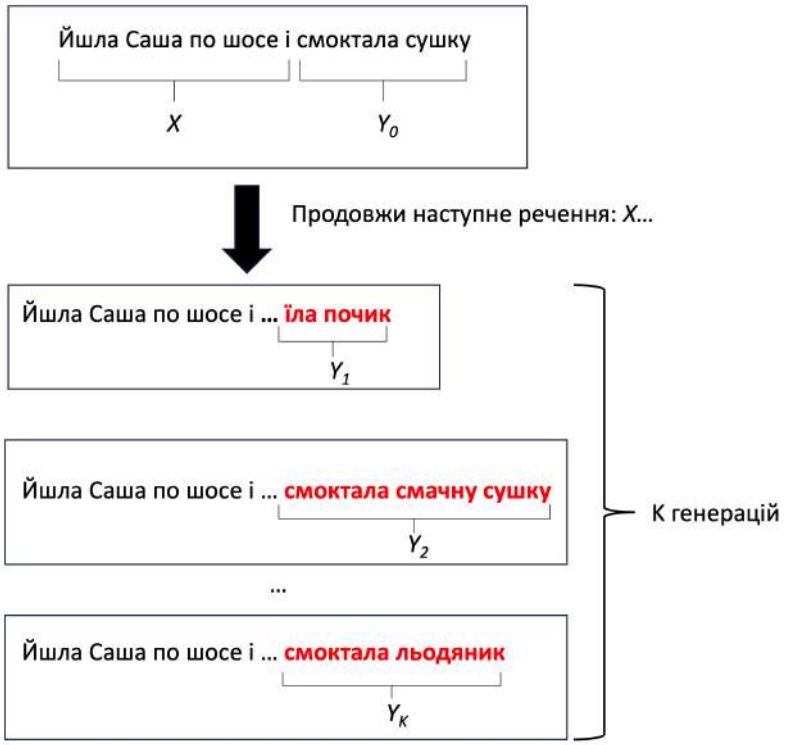

Рисунок 3.2 – Генерація K нових закінчень речення за допомогою мовної моделі

3) Наступний крок залежить від задачі яка перед нами стоїть.

3.1) Якщо перед нами стоїть задача white-box детекції, тобто ми маємо доступ до ваг моделей і ймовірності генерації токенів, ми можемо порахувати умовний WScore за наступною формулою:

$$WScore(S, \Omega) = \frac{1}{K} \sum_{k=1}^{K} \quad log \ \frac{p(X)}{p(X)} \qquad (1.4)$$

де S – повний тестовий уривок, який ми перевіряємо (префікс + оригінальне продовження);

X – префікс (усічена початкова частина тексту), який подається на вхід моделі;

Y0 – справжнє (оригінальне) продовження тексту після X;

Ω={Y1,Y2,…,YK} – множина з K нових продовжень, згенерованих моделлю на основі того ж префікса X;

p(Y|X) – умовна ймовірність послідовності Y за даного префікса X, яку видає модель (product of token probabilities);

K – число повторних генерацій (розмір множини Ω).

Після підрахунку WSCore, по заданій межі можна буде розділити на текст написаний людиною і мовною моделлю. Людський текст набагато більш випадковий, тому у нього цей WScore буде близький до нуля чи навіть негативний, на відміну від мовної моделі, у якої в більшості випадків цей WScore буде вищим.

3.2) Якщо ж перед нами стоїть Black-box задача, то метрика по якій ми будемо визначати чи є текст згенерований мовною моделлю буде інша. Оскільки тепер у нас нема доступів до імовірностей генерації токенів, будемо дивитися на перетин n-грам. Логічно, що якщо текст згенерований моделлю, то при схожому запиті вона буде видавати відповідь зі схожими формулюваннями. Чим більше спільних і довших n-грам у згенерованих закінченнях тексту та в оригінальному, тим іморвірніше цей текст був згенерований ШІ. Формула 1.5 наведена нижче.

$$BScore(S,\Omega) = \frac{1}{K}\sum_{k=1}^{K} \quad \sum_{n=n_0}^{N} \quad f(n)\frac{|grams(Y_k,n)\cap grams(Y_0,n)|}{|Y_k||grams(Y_0,n)|} \quad (1.5)$$

де S – тестовий уривок (префікс X + оригінальне продовження Y0);

$\Omega=\{Y_1,\ldots,Y_K\}$ – множина з K варіантів продовження, згенерованих моделлю;

grams(T,n) – мн|ожина усіх n-грам у послідовності T;

$|Y_k|$ – довжина (кількість токенів) у послідовності Yk;

n0 і N – мінімальна та максимальна довжина n-грам (наприклад, n0=4, N=25);

f(n) – вагова функція для n-грам різної довжини (в цій роботі, f(n)=nlog(n)).

Для оцінки ефективності алгоритмів детекції в роботі використовуються стандартні метрики бінарної класифікації, такі як TPR, FPR, AUROC.

TPR (True Positive Rate) – частка правильно ідентифікованих текстів серед усіх згенерованих мовними моделями текстів. Розраховується як TPR = TP/(TP+FN), де TP – кількість текстів, правильно класифікованих як згенеровані мовними моделями, FN – тексти згенеровані мовними моделями, помилково класифіковані як людські. Висока TPR свідчить про здатність алгоритму надійно виявляти машинну генерацію. Одиниця виміру – %.

FPR (False Positive Rate) – частка людських текстів, помилково класифікованих як згенеровані мовними моделями. Обчислюється як FPR = FP/(FP+TN), де FP – людські тексти, визначені як машинні, TN – правильно ідентифіковані людські тексти. Низька FPR критично важлива для уникнення хибних звинувачень у використанні мовних моделей. Одиниця виміру – %.

AUROC (Area Under the Receiver Operating Characteristic curve) – інтегральна метрика, що оцінює загальну якість класифікатора незалежно від порогу прийняття рішення. ROC-крива будується як залежність TPR від FPR при різних порогах класифікації. AUROC = 0.5 відповідає випадковому вгадуванню, AUROC = 1.0 – ідеальній класифікації. Ця метрика дозволяє порівнювати різні алгоритми незалежно від обраного порога класифікації. Одиниця виміру – площа під кривою.

В оригінальній статті цей алогоритм застосувався до англомовного набору даних **Xsum** і показав наступні результати по підходам, при пошуку тексту згенерованого різними моделями, для генерації суфіксів була використана та сама модель:

Алгоритм	text-davinci-003(White-box)			text-davinci-003(Black-box)			GPT-4-0314(Black-box)		
Метрика	AUROC	TPR	FPR	AUROC	TPR	FPR	AUROC	TPR	FPR
Результат	98.64%	90.0%	<1%	86.6%	26.00%	<1%	91.72%	32.67%	<1%

Таблиця 3.1 – результати роботи алгоритму DNA-GPT в оригінальній статті

Ми бачимо, що модель показує непогані результати і фокусується на найменшій кількості неправдиво позитивних результатів, навіть якщо це іде за рахунок низького відсотка правдиво позитивних результатів. При адаптації алгоритму на українську мову, будемо триматися такого самого підходу – якомога менше неправдиво позитивних результатів, адже в умовах академічної доброчесності та медійної відповідальності хибні звинувачення у використанні мовних моделей можуть завдати значно більше шкоди, ніж

пропущені випадки автоматичної генерації. Помилково позначений як згенерований моделлю студентський твір чи журналістська стаття підриває довіру до нього та створює серйозні юридичні й етичні ризики.

1.11 Класифікатор на базі BERT

Наступним досліджуваним алгоритмом став класифікатор на основі архітектури BERT, який на відміну від розглянутих раніше підходів, належить до методів з навчанням і потребує попереднього тренування на розмічених даних.

BERT (Bidirectional Encoder Representations from Transformers) використовує двонаправлений механізм аналізу контексту, одночасно обробляючи інформацію як з лівого, так і з правого боку від цільового токена. Це дозволяє формувати глибокі контекстуалізовані векторні представлення, що враховують повний семантичний контекст.

Для адаптації BERT до задачі детекції, над попередньо навченою моделлю додається повнозв'язний нейронний шар та функції активації softmax. Вся система проходить етап донавчання на збалансованому наборі даних, який містить приклади як людської, так і машинної генерації текстів. Оптимізація параметрів відбувається через мінімізацію крос-ентропійної функції втрат для бінарної класифікації.

Під час безпосереднього застосування навчена модель обчислює ймовірнісні оцінки приналежності вхідного тексту до кожного класу. Фінальне рішення приймається на основі порівняння цих ймовірностей – текст відноситься до категорії з вищою оцінкою.

Ключовою перевагою BERT класифікатора є здатність виявляти складні лінгвістичні патерни та синтаксичні особливості, характерні для тексту згенерованого мовними моделями.

Розділ 2: Збір даних

2.1 Критерії вибору набору даних

Для даного експерименту критично важливо підібрати правильний набір даних. Щоби коректно навчити модель розрізняти текст зненерований мовною моделю від написаного людиною чи просто провалідувати результати, треба бути впевненим, що текст, який ми вважаємо написаний людиною точно є таким. Саме тому перший критерій, це дата створення текста. Текст, промаркований як людський, повинен бути опублікований до 2018 року, щоб зменшити вплив створення і виходу в маси моделі ChatGPT-3, що була поштовхом до “буму” мовних моделей. Таким чином ми мінімізуємо кількість прикладів в наборі даних, що являють штучно згенеровані, але марковані як людський текст.

Дослідження зосереджено на літературній українській мові з акцентом на формальних жанрах: публіцистичних статтях, книжкових текстах, журнальних матеріалах та новинах. Розширення тематичного спектру набору даних потребувало б значних часових та фінансових ресурсів, включаючи доступ до комерційних високоякісних наборів даних. З огляду на обмежені можливості дослідження, було прийнято рішення про фокусування на вищезазначених категоріях як репрезентативній вибірці для поставлених завдань.

2.2 Збір даних

За основу взяти набір даних brown-uk/corpus[33], що є збіркою текстів з різних журналів і статей, написаних до 2018 року, до виходу ChatGPT-3, отже до масового використання мовних моделей, що значно зменшує шанс того, що якісь елементи цього набору даних згенеровані мовною моделлю. Також цей датасет додатково в ручну провалідований людьми, що збільшує загальну якість даних.

Дані були організовані в набір даних за наступним форматом. Кожен елемент бази складався з кількох стовпців:

1. Безпосередній текст
2. Тема тексту. Вона була згенерована ШІ з тексту статті за запитом “Згенеруй короткий але змістовний опис того, про що ідеться в цій статті”.
3. Клас тексту. Людський чи згенерований мовною моделлю.

2.3 Генерація штучних даних

Для формування збалансованого датасету, що містить приклади тексту, згенерованого мовними моделями, було створено шість додаткових наборів даних з використанням двох мовних моделей: ChatGPT-4o-mini та Llama3:70b.

Кожна модель використовувалася для створення трьох окремих наборів даних з різними значеннями параметра температури, а саме 0.1, 0.5 і 0.8, що дозволило дослідити вплив випадковості генерації на можливості виявлення тексту згенерованого мовними моделями.

Генерація здійснювалася за стандартизованим промптом:

“Напиши новину з {length} слів для наступного заголовку: {zagolovok}”, де {length} – довжина відповідного оригінального людського тексту; {zagolovok} – тематичний дескриптор, згенерований для оригінальної статті.

Такий підхід гарантує збереження ключових структурних характеристик, як довжина та тематика, між парами людських та згенерованих мовними моделями текстами.

Розділ 3: Імплементація алгоритмів. Проведення експериментів. Аналіз результатів

3.1 Імплементація DNA-GPT

Для імплементації цього алгоритму було створено наступні класи:

1. Gpt4oMiniCompletion, CompletionModelLLama, DavinchiCompletion – ці класи відповідали за генерацію продовження закінчення оригінального текста.
2. DavinchiEstimation – клас для отримання лог проб генерації даного тексту. Реалізований лише для моделі Davinchi-002, оскільки лише вона має інструменти для отримання ймовірностей генерації токенів не згенерованого тексту, а вже написаного. З мінусів цієї моделі - вона дуже дорога, тому вона не була використана для створення датасету, а лише для експерименту.

Повний код вищезазначених класів можна знайти в Додатку А. Експеримент був розділений на 2 частини – Black-box та White-box експерименти.

Порядок проведення Black-box експерименту:

1. Ітеруємося по набору навчальних даних. Беремо опис і текст. Текст розділяємо на 2 частини. Використовуємо один з Completion класів, залежно від експерименту, і генеруємо по N нових закінчень тексту;
2. Після цього, по Формулі 1.5 рахуємо BScore і зберігаємо результат;
3. Підбираємо такий поріг BScore, щоби FPR був <1%;
4. Валідація результатів на тестових даних.

На виході отримуємо поріг BScore для даної конфігурації алгоритму та результати тестування. Детальніше про результати в наступних главах.

Порядок проведення White-box експерименту:

1. Ітеруємося по набору навчальних даних. Беремо опис і текст. Текст розділяємо на 2 частини;

2. Використовуємо один з Completion класів, залежно від експерименту, і генеруємо по N нових закінчень тексту;
3. Використовуємо DavinchiEstimation клас для визначення імовірностей кожного з токенів початкового закінчення тексту;
4. Рахуємо WScore за Формулою 1.4;
5. Підбираємо такий поріг WScore, щоби FPR був <1%;
6. Валідація результатів на тестових даних.

На виході отримуємо поріг WScore для даної конфігурації алгоритму та результати тестування. Детальніше про результати в наступних главах.

3.2 Вибір гіперпараметрів. Проведення експерименту на DNA-GPT

Вибір гіперпармаетрів алгоримтму, таких як кількість генерацій закінчень N, використана мовна модель, пропорції розділення оригінального тексту, мінімальна і максимальна довжини n-грам та інші є критично важливими для роботи алгоритму. Саме через це було проведено ряд експериментів з різними комбінаціями цих гіпер параметрів, для виявлення оптимальних значень. Основний фокус був на температуру тексту зненерованого мовною моделлю і на те, як алгоритми, з різними мовними моделямі в основі, працюють з різними даними. Параметри, такі як кількість генерацій, довжина n-грам та пропорції розділення тексту залишені як в оригінальній статті, адже їх вплив на результат далеко не такий сильний. Гіперпараметри для кожного з експериментів вказані в таблицях нижче.

N експерименту	1	2	3	4	5	6	7	8	9	10	11	12	13
Модель для геенарції датасету	gpt-4o-mini	gpt-4o-mini	gpt-4o-mini	gpt-4o-mini	gpt-4o-mini	gpt-4o-mini	llama3:70b	llama3:70b	llama3:70b	llama3:70b	llama3:70b	llama3:70b	gpt-4o-mini + llama3:70b
Температура моделі генерації	0.1	0.5	0.8	0.1	0.5	0.8	0.1	0.5	0.8	0.1	0.5	0.8	0.1 + 0.1
Completion Model	GPTCompl	GPTCompl	GPTCompl	LlamaCo	LlamaCo	LlamaCo	LlamaCo	LlamaCo	LlamaCo	GPTCompl	GPTCompleti	GPTCompletion	GPTCompletion

	etion	etion	etion	mpletion	mpletion	mpletion	mpletion	mpletion	mpletion	etion	on		
Температура Completion model	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Пропорція розділення оригінального тексту	70:30	70:30	70:30	70:30	70:30	70:30	70:30	70:30	70:30	70:30	70:30	70:30	70:30
Кількість генерацій нового закінчення	10	10	10	10	10	10	10	10	10	10	10	10	10
n0 - мінімальна довжина n-грам	4	4	4	4	4	4	4	4	4	4	4	4	4
N - максимальна довдина n-грам	25	25	25	25	25	25	25	25	25	25	25	25	

Таблиця 5.1 – Гіперпараметри Black-box експериментів

N експерименту	1	2	3	4	5	6
Модель для геенарції датасету	gpt-4o-mini	gpt-4o-mini	gpt-4o-mini	llama3:70b	llama3:70b	llama3:70b
Температура моделі генерації	0.1	0.5	0.8	0.1	0.5	0.8
CompletionModel	Davinchi Completion	Davinchi Completion	Davinchi Completion	Davinchi Completion	Davinchi Completion	Davinchi Completion
Температура Completion model	0.1	0.1	0.1	0.1	0.1	0.1
EstimationModel	Davicnhi Estimation	Davicnhi Estimation	Davicnhi Estimation	Davicnhi Estimation	Davicnhi Estimation	Davicnhi Estimation
Температура EstimationModel	0.1	0.1	0.1	0.1	0.1	0.1
Пропорція	70:30	70:30	70:30	70:30	70:30	70:30

розділення оригінального тексту						
Кількість генерацій нового закінчення	10	10	10	10	10	10

Таблиця 5.2 – Гіперпараметри White-box експериментів

WhiteBox можливо проводити лише з моделями класу Davinchi, оскільки лише вони мають доступ до розрахунку імовірностей генерації токенів уже згенерованого тексту (що необхідно для оцінки імовірностей генерації ориганільного закінчення тексту). Згенерувати великий датасет саме для цієї моделі було б занадто дорого, тому довелося тестувати на наборах даних згенерованих іншими моделями.

3.3 Валідація результатів експерименту

Провівши ряд експериментів описаних в попередній главі, було отримано ряд певних граничних значень для класифікації. Після чого, експеримент з даними граничними значеннями було проведено на окремому тестовому наборі даних. Результати наведені нижче.

Номер експерименту	AUROC	FPR	TPR
1	0.74	<1%	48.6%
2	0.70	<1%	41.1%
3	0.65	<1%	31.5%
4	0.53	<1%	6.6%
5	0.53	<1%	6.3%
6	0.53	<1%	6.5%
7	0.63	<1%	27%
8	0.63	<1%	25.1%
9	0.60	<1%	20.8%
10	0.50	<1%	<1%
11	0.50	<1%	<1%
12	0.50	<1%	<1%
13	0.50	<1%	<1%

Таблиця 5.3 – Результати Black-box експериментів

Номер експерименту	AUROC	FPR	TPR
1	0.55	<1%	10.4%

2	0.54	<1%	10.2%
3	0.54	<1%	9.4%
4	0.5	<1%	<1%
5	0.5	<1%	<1%
6	0.5	<1%	<1%
7	0.5	<1%	<1%

Таблиця 5.4 – Результати White-box експериментів

3.4 Імплементація BERT класифікатора

У межах роботи було розроблено клас BERTDetector, що інкапсулює всю логіку побудови та інференсу моделі.

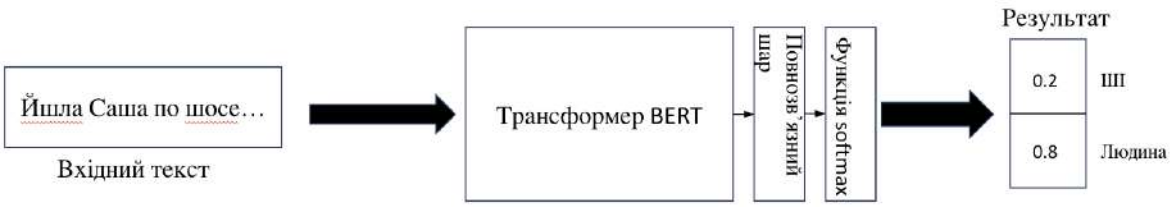

Рисунок 5.1 – Схема роботи алгоритму на базі моделі BERT

Як базову архітектуру було використано попередньо-натреновану трансформерну модель BERT. Вона була натренована на величезному багатомовному наборі даних, в тому числі й українською. До моделі був доданий класифікаційний шар, що являє собою однорівневий feed-forward[34] шар та активацією Softmax для бінарного виходу. Клас надає методи fit, predict_proba, predict, що спрощують інтеграцію в пайплайн експериментів.

Повний код класу BERTDetector наведено в Додатку Б.

3.5 Вибір гіперпараметрів та процес тренування

В ході цього експерименту було перевірено декілька конфігурацій наборів даних. Також були протестовані різні конфігурації навчання моделі, де перевірявся вплив вибора оптимізатора, коефіцієнта навчання, планувальника коефіцієнту навчання, але вони практично не впливали на результат, тому їх не буде вказано в роботі. Набагато сильніший вплив мала комбінація навчального і тестового наборів даних. Ці комбінації і стали головними гіперпараметрами експериментів. Вони наведені в таблиці нижче.

N експерименту	1	2	3	4	5	6	7	8	9	10	11
Тренувальний набір даних	Human + gpt-4o-mini температури 0.1, 0.5, 0.8	Human + gpt-4o-mini температури 0.1	Human + gpt-4o-mini температури 0.5	Human + gpt-4o-mini температури 0.8	Human + gpt-4o-mini температури 0.1, 0.5, 0.8	Human + llama3:70b температури 0.1, 0.5, 0.8	Human + llama3:70b температури 0.1	Human + llama3:70b температури 0.5	Human + llama3:70b температури 0.8	Human + llama3:70b температури 0.1, 0.5, 0.8	Human + gpt-4o-mini + llama3:70b, температури 0.1,

											0.5, 0.8
Тестовий набір даних	Human + gpt-4o-mini температури 0.1, 0.5, 0.8	Human + gpt-4o-mini температури 0.1	Human + gpt-4o-mini температури 0.5	Human + gpt-4o-mini температури 0.8	Human + gpt-4o-mini + llama3:70b, температури 0.1, 0.5, 0.8	Human + llama3:70b температури 0.1, 0.5, 0.8	Human + llama3:70b температури 0.1	Human + llama3:70b температури 0.5	Human + llama3:70b температури 0.8	Human + gpt-4o-mini + llama3:70b, температури 0.1, 0.5, 0.8	Human + gpt-4o-mini + llama3:70b, температури 0.1, 0.5, 0.8

Таблиця 5.5 – конфігурації експериментів для BERT-моделей

Щодо конфігурації тренування – результати тренувань, що будуть зазначені нижче, були отримані за використання оптимізатора AdamW[36], з коефіцієнтом навчання 2e-5, кількістю епох 10 і розміром батчінгу в 32. Результати інших параметрів навчання практично не відрізняються, тому не будуть вказані.

3.6 Проведення і результати експерименту

У Таблиці 5.6 представлені результати експериментальних досліджень продуктивності моделі, що базується на архітектурі BERT. Аналізуючи наведені дані, можна чітко побачити, що сценарії, в яких для навчання та тестування моделі використовувалися виключно дані, згенеровані людьми, а також дані, отримані від однієї конкретної моделі при фіксованому значенні температури, демонструють майже бездоганні результати для gpt-4o-mini та досить високі показники для llama3. Така тенденція спостерігається у випадку використання однорідних наборів даних як для тренування, так і для валідації, що свідчить про високу здатність цих моделей до узагальнення в межах вузькоспеціалізованих розподілів даних. Причини такої високої ефективності, а також детальний аналіз отриманих результатів будуть розглянуті в наступній главі, де буде проведено глибше дослідження впливу однорідності даних на продуктивність мовних моделей.

Номер експерименту	TPR	FPR	AUROC
1	90.4%	<1%	94.7
2	99.3%	<1%	99.2
3	98.4%	<1%	98.7
4	98.6%	<1%	98.9
5	31.5%	<1%	65.3
6	52.1%	<1%	75.6
7	78.2%	<1%	88.6
8	80.1%	<1%	89.6
9	77.9%	<1%	88.5

10	22.1%	<1%	60.6
11	77.5%	<1%	88.3

Таблиця 5.6 – результати експериментів з моделью на основі BERT

3.7 Аналіз результатів експериментів

3.7.1 DNA-GPT BlackBox підхід

Оптимальні результати детекції спостерігалися при використанні ідентичної моделі для генерації продовжень (CompletionModel) та створення тестових даних. У експерименті з GPT-4o-mini в обох ролях досягнуто AUROC 0.74 та TPR 48.6% при збереженні FPR нижче 1%.

Параметр температури продемонстрував значний вплив на ефективність детекції. Підвищення температури, яке призводить до більш випадкової генерації, супроводжувалося помітним зниженням ефективності алгоритму. Так, для GPT-4o-mini спостерігалося падіння TPR з 48.6% при τ=0.1 до 31.5% при τ=0.8.

Розбіжність між моделлю для продовжень та моделлю-генератором тестових даних призводила до критичного погіршення результатів. В таких конфігураціях фіксувалися низькі значення як AUROC, так і TPR, що свідчить про фундаментальну залежність BlackBox DNA-GPT від відповідності між детектором та цільовою моделлю.

Важливо відзначити, що в усіх проведених експериментах алгоритм стабільно утримував FPR на рівні менше 1%, демонструючи високу специфічність та мінімальну кількість помилкових тривог при ідентифікації людських текстів як машинних.

3.7.2 DNA-GPT WhiteBox підхід:

Експерименти з WhiteBox варіантом DNA-GPT продемонстрували суттєво нижчу ефективність порівняно з BlackBox підходом. Максимальний досягнутий AUROC становив лише 0.55 з відповідним TPR 10.4% для датасету gpt-4o-mini.

Головною причиною низької продуктивності є архітектурна невідповідність між моделлю оцінки ймовірностей (EstimationModel) та моделями-генераторами тестових текстів. У проведених експериментах для оцінки ймовірностей використовувалася модель Davinci-002, тоді як тестові набори даних створювалися моделями GPT-4o-mini та Llama3:70b. Оптимальна конфігурація WhiteBox підходу передбачає використання ідентичної моделі для обох завдань – генерації тексту та обчислення ймовірностей токенів, проте, як уже було зазначено, через обмеження в ресурсах і фінансах, таку конфігурацію не було створено.

Як і в BlackBox, вдалося утримати FPR <1%.

3.7.3 Аналіз результатів BERT класифікатора:

Навчання класифікатора на базі BERT продемонструвало найвищу ефективність серед усіх підходів, особливо коли тренувальний та тестовий набори даних були однорідними.

В експерименті 1, де модель тренувалася та тестувалася на даних, згенерованих gpt-4o-mini (з різними температурами), було досягнуто AUROC 94.7% та TPR 90.4% при FPR <1%. Це свідчить про високу здатність BERT

виявляти специфічні патерни українського тексту, згенерованого конкретною моделлю, після відповідного донавчання.

Експерименти з навчанням на змішаних наборах даних виявили зниження ефективності BERT-класифікатора. При тренуванні на комбінованих даних від GPT-4o-mini та Llama3:70b (експеримент 2) або виключно на текстах Llama3:70b (експеримент 3), спостерігалося падіння основних метрик: AUROC знизився до 75.2% та 75.6%, а TPR – до 51.5% та 52.1% відповідно. Хоча ці показники залишаються все ще значущими при збереженні FPR менше 1%, вони помітно поступаються результатам спеціалізованого навчання. Ця тенденція свідчить про обмежену здатність BERT-класифікатора до узагальнення між різними мовними моделями. Висока ефективність на текстах конкретної моделі при конкретній температурі може вказувати на перенавчання – модель занадто точно адаптується до специфічних патернів окремої LLM замість виявлення універсальних ознак машинної генерації.

Практичний висновок полягає в тому, що для ефективного застосування BERT-детектора необхідна попередня інформація про цільову модель-генератор. Спроби виявити тексти від невідомих або не представлених у тренувальній вибірці моделей можуть призвести до непередбачуваного зниження точності, що обмежує універсальність підходу в реальних сценаріях використання.

3.7.4 Загальні висновки з результатів

Навчання моделі BERT на специфічному для завдання наборі українських текстів (людських та згенерованих мовною моделлю) показало найкращі результати, особливо коли модель навчалася і тестувалася на

текстах, згенерованих однією і тією ж мовною моделлю без варіацій температури. До того ж, результати детекції тексту згенерованого моделью llama стабільно нижчі, ніж gpt-4o-mini. З цього можна зробити висновок, що моделі з роду gpt мають менші показники перплексії для українського корпусу і є більш стабільними та передбачуваними. Цей висновок є цілком логічним, оскільки gpt моделі є більш потужними прямо з коробки, а моделі на відкритому коді як llama більше призначені для подальшої доробки і оптимізації під конкретну задачу.

Ефективність DNA-GPT, особливо в BlackBox варіанті, сильно залежить від того, наскільки CompletionModel (та EstimationModel для WhiteBox) відповідає моделі, що згенерувала аналізований текст. При їх розбіжності точність значно знижується, що є очікуваним результатом. Цікаво, що в оригінальній імплементації цього алгоритму, різниця в моделях генерації і оцінки не така висока. У всіх експериментах вдалося досягти поставленої мети – False Positive Rate менше 1%. Це критично важливо для уникнення хибних звинувачень, особливо в контексті академічної доброчесності.

Тексти, згенеровані з нижчою температурою (більш детерміновані), легше детектуються як DNA-GPT, так і детектором на основі BERT, що логічно, оскільки алгоритм базується на передбачуваності генерації.

Практичне застосування WhiteBox DNA-GPT для української мови ускладнене через обмежений доступ до моделей, що надають імовірності для довільного тексту, та їхню вартість. Потенційно, за наявності відповідних інструментів, цей метод міг би показати кращі результати при збігу моделей.

Висновки

Підбиваючи підсумки, можна сказати, що задача визначення, чи є текст згенерованим мовною моделлю, для української мови є абсолютно вирішуємою і існує багато підходів, як це зробити. В цій роботі були досліджені деякі з таких методів, такі як алгоритм DNA-GPT та власний класифікатор на основі мовної моделі BERT. Вони показали доволі непогані результати і змогли бути адаптовані для застосуванні на текстах написаних українською мовою. В ході роботи були знайдені приклади даних і згенеровані синтетичні додаткові тексти, що дозволило провести понад 30 різних експериментів, і окрім отримання готових алгоритмів, що показували стабільний хороший результат, емпірично довели гіпотезу сильну недетермінованість мовних моделей при обробці і генерації українського тексту.

Основні наукові та практичні результати:

1) Розроблено реалізації DNA-GPT з гнучкими гіперпараметрами та адаптовано їх під українську мову.
2) Створено клас BERTDetector з підтримкою донавчання і оцінки, який демонструє високу точність для конкретних LLM-сценаріїв.
3) Визначено сильні та слабкі сторони кожного підходу: BERT – для контрольованих умов, DNA-GPT – для швидкої інтеграції без доступу до ваг моделі.
4) Досліджено підходи виявлення тексту згенерованого мовними моделями.

Література

1. https://platform.openai.com/docs/models/
2. https://chatgpt.com/
3. https://www.llama.com/
4. https://claude.ai/
5. https://gemini.google.com/
6. https://www.zerogpt.com/
7. https://copyleaks.com/
8. https://originality.ai/
9. Sherstinsky, Alex. "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network." Physica D: Nonlinear Phenomena 404 (2020): 132306.
10. Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.
11. Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014).
12. Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).
13. Provilkov, Ivan, Dmitrii Emelianenko, and Elena Voita. "BPE-dropout: Simple and effective subword regularization." arXiv preprint arXiv:1910.13267 (2019).
14. Song, Xinying, et al. "Fast wordpiece tokenization." arXiv preprint arXiv:2012.15524 (2020).
15. Kudo, Taku, and John Richardson. "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing." arXiv preprint arXiv:1808.06226 (2018).
16. Rong, Xin. "word2vec parameter learning explained." arXiv preprint arXiv:1411.2738 (2014).

17. Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.
18. Ethayarajh, Kawin. "How contextual are contextualized word representations? Comparing the geometry of BERT, ELMo, and GPT-2 embeddings." arXiv preprint arXiv:1909.00512 (2019).
19. https://openai.com/api/
20. https://cloud.llamaindex.ai/
21. https://www.techtarget.com/whatis/feature/GPT-4o-explained-Everything-you-need-to-know
22. https://www.gpu-mart.com/blog/top-open-source-llms-for-2024
23. Yang, Xianjun, et al. "Dna-gpt: Divergent n-gram analysis for training-free detection of gpt-generated text." arXiv preprint arXiv:2305.17359 (2023).
24. https://www.python.org/downloads/release/python-3110/
25. https://pandas.pydata.org/
26. https://www.nltk.org/
27. https://matplotlib.org/
28. https://seaborn.pydata.org/
29. https://huggingface.co/docs/tokenizers/en/index
30. https://huggingface.co/docs/hub/en/index
31. https://pytorch.org/
32. https://huggingface.co/docs/transformers/index
33. https://github.com/brown-uk/corpus
34. Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." Neural networks 61 (2015): 85-117.
35. Loshchilov, Ilya, and Frank Hutter. "Decoupled weight decay regularization." arXiv preprint arXiv:1711.05101 (2017).

Додаток А

```
class DavinchiCompletion:
    def __init__(self, openai_key: str, model: str):
        self.openai_key = openai_key
        self.model = model
        self.headers = {
            "Authorization": f"Bearer {self.openai_key}",
            "Content-Type": "application/json"
        }

    def complete_text(self, prefix: str, predict_log_proba: bool, max_tokens:
int, num_generations: int = 1) -> Union[
        Tuple[List[str]], Tuple[List[str], np.array]]:
        """
        Generates the remaining suffix for the given beginning of the text.
        :param prefix: The beginning of some text.
        :param predict_log_proba: Whether the model should return log
probabilities of underlying text.
        :param max_tokens: Maximum tokens to generate.
        :param num_generations: Number of completions to generate.
        :return: Tuple of tokens (list of strings) and optionally an array of log
probabilities of generated text.
        """
        payload = {
            "model": self.model,
            "prompt": prefix,
            "max_tokens": max_tokens,
            "echo": False,
```

```
            "logprobs": 1,
            "n": num_generations,
            "temperature": 0.3
        }
        response = requests.post(OPENAI_COMPLETION_ENDPOINT,
headers=self.headers, json=payload)
        response.raise_for_status()
        data = response.json()

        tokens_list = []
        logprobs_list = []

        for choice in data['choices']:
            tokens = choice['logprobs']['tokens']
            tokens_list.append(tokens)

            if predict_log_proba:
                logprobs = choice['logprobs']['token_logprobs']
                logprobs_list.append(np.array(logprobs))

        if predict_log_proba:
            return tokens_list, logprobs_list  # Return tokens and log
probabilities in arrays
        return (tokens_list,)



class DavinchiEstimation:
    def __init__(self, openai_key: str, model: str):
        self.openai_key = openai_key
```

```
        self.model = model
        self.headers = {
            "Authorization": f"Bearer {self.openai_key}",
            "Content-Type": "application/json"
        }

    def get_text_log_proba(self, text: str) -> Tuple[List[str], np.array]:
        """
        Returns log token probabilities for the given text.
        :param text: Input text.
        :return: Tuple of text split into model's tokens and np.array representing probabilities of corresponding tokens.
        """
        # Prepend the special token to the prompt as in the original code.
        payload = {
            "model": self.model,
            "prompt": "<|endoftext|>" + text,
            "max_tokens": 0,
            "echo": True,
            "logprobs": 1
        }
        response = requests.post(OPENAI_COMPLETION_ENDPOINT, headers=self.headers, json=payload)
        response.raise_for_status()
        data = response.json()

        tokens = data['choices'][0]['logprobs']['tokens'][1:]  # Remove special token "<|endoftext|>"
```

```
        logprobs = data['choices'][0]['logprobs']['token_logprobs'][1:]  # Remove special token "<|endoftext|>"

        return tokens, np.array(logprobs)

import json
from pprint import pprint

import numpy as np
import requests
from typing import List, Tuple, Union

class Gpt4oMiniCompletion:
    def __init__(self, openai_key: str, model: str):
        """
        :param openai_key: Your OpenAI API key.
        :param model: The model name (e.g. "gpt-4o-mini").
        """
        self.api_key = openai_key
        self.model = model
        self.endpoint = "https://api.openai.com/v1/chat/completions"
        self.headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {self.api_key}"
        }

    def complete_text(
            self,
            prefix: str,
```

```
        predict_log_proba: bool,
        max_tokens: int,
        num_generations: int = 1
    ) -> Union[
        Tuple[List[List[str]]],
        Tuple[List[List[str]], List[np.ndarray]]
    ]:
        """
        Sends a prompt to the ChatCompletion API and parses the token-level log probabilities
        from the "logprobs.content" section of each choice (if available).
        - If `predict_log_proba=False`, returns only the tokens.
        - If `predict_log_proba=True`, returns tokens and logprobs in parallel lists.

        :param prefix: The text prompt to send to the model.
        :param predict_log_proba: Whether to parse and return token-level log probabilities.
        :param max_tokens: Maximum number of tokens to generate.
        :param num_generations: Number of completions (choices) to generate.
        :return:
          - If predict_log_proba is False:
            A tuple containing a list of lists of tokens (one per choice).
              e.g. ( [ ["Hello", "!", " How", " can", ...], ... ], )
          - If predict_log_proba is True:
            A tuple of two lists, (tokens_list, logprobs_list),
            where tokens_list is a list of lists of tokens,
            and logprobs_list is a list of NumPy arrays of logprobs.
        """
```

```
        payload = {
            "model": self.model,
            "messages": [
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user", "content": prefix}
            ],
            "max_tokens": max_tokens,
            "n": num_generations,
            "temperature": 0.3,
            "logprobs": True,
            # The user-provided example JSON had "logprobs" data
            # in the output, so your model/endpoint must support that.
            # If your model does not produce it by default, you may need
            # to add a custom parameter or tweak the server config.
        }

        response = requests.post(self.endpoint, headers=self.headers,
data=json.dumps(payload))
        if response.status_code != 200:
            raise Exception(f"Request failed with status
{response.status_code}:\n{response.text}")

        data = response.json()
        # We'll store all completions in parallel lists:
        #   tokens_list[i] will correspond to logprobs_list[i]
        tokens_list: List[List[str]] = []
        logprobs_list: List[np.ndarray] = []

        # "choices" is a list of completions
```

```
for choice in data["choices"]:
    # Expecting choice["logprobs"]["content"] to be a list of token dicts
    if "logprobs" not in choice:
        # Possibly fallback or raise an error if 'logprobs' is missing
        tokens_list.append([choice["message"]["content"]])
        if predict_log_proba:
            logprobs_list.append(np.array([]))
        continue

    logprobs_obj = choice["logprobs"]
    content_array = logprobs_obj.get("content", [])
    # If there's nothing in "content", fallback similarly
    if not content_array:
        tokens_list.append([choice["message"]["content"]])
        if predict_log_proba:
            logprobs_list.append(np.array([]))
        continue

    # Parse out tokens, and optionally their log probabilities
    choice_tokens = []
    choice_logprobs = []
    for item in content_array:
        # Each item has fields: "token", "logprob", "bytes", etc.
        choice_tokens.append(item["token"])
        if predict_log_proba:
            choice_logprobs.append(item["logprob"])

    tokens_list.append(choice_tokens)
    if predict_log_proba:
```

```
                # Convert to numpy array, just like your old snippet
                logprobs_list.append(np.array(choice_logprobs))

        # Return format depends on predict_log_proba
        if predict_log_proba:
            return tokens_list, logprobs_list
        else:
            return (tokens_list,)

class LLama3Completion(CompletionLanguageModel):
    def __init__(
        self,
        api_key: str,
        model: str = "llama3-70b",
        base_url: str = "https://api.llmapi.com/"
    ):
        """
        :param api_key: Your LLMAPI / OpenAI-compatible API key.
        :param model: The model name, e.g. "llama3-70b" or "llama3.1-70b".
        :param base_url: The base URL for your LLMAPI endpoint.
        """
        self.client = OpenAI(api_key=api_key, base_url=base_url)
        self.model = model

    def complete_text(
        self,
        prefix: str,
        predict_log_proba: bool,
        max_tokens: int,
```

```
    num_generations: int = 1
) -> Union[
    Tuple[List[List[str]]],
    Tuple[List[List[str]], List[np.ndarray]]
]:
    if predict_log_proba:
        # Use the completions endpoint to get token-level logprobs
        resp = self.client.completions.create(
            model=self.model,
            prompt=prefix,
            max_tokens=max_tokens,
            n=num_generations,
            temperature=0.3,
            logprobs=0  # return token logprobs
        )
        tokens_list: List[List[str]]   = []
        logprobs_list: List[np.ndarray] = []

        for choice in resp.choices:
            lp = choice.logprobs
            # lp.tokens is the list of generated tokens
            # lp.token_logprobs is the list of their log-probabilities
            tokens_list.append(lp.tokens)
            logprobs_list.append(np.array(lp.token_logprobs))

        return tokens_list, logprobs_list

    else:
```

```
        # Use chat endpoint for plain text (and support multiple completions
via n)
        resp = self.client.chat.completions.create(
            model=self.model,
            messages=[
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user",   "content": prefix}
            ],
            max_tokens=max_tokens,
            n=num_generations,
            temperature=0.3,
            stream=False
        )
        tokens_list: List[List[str]] = []
        for choice in resp.choices:
            # treat the entire reply as one “token”
            tokens_list.append([choice.message.content])
        return (tokens_list,)
```

Додаток Б

```
import torch
import numpy as np
from typing import List
from torch.optim import AdamW
from transformers import BertTokenizer, BertForSequenceClassification
from tqdm.auto import tqdm
from sklearn.metrics import (
    roc_curve, f1_score, precision_score, recall_score,
    accuracy_score
)
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.utils.data import DataLoader

class BERTDetector:
    def __init__(self, model_name: str = "bert-base-uncased", num_labels: int
= 2):
        # Use GPU if available
        self.device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        print(f"Device used: {self.device}")

        # Load tokenizer and model
        self.tokenizer = BertTokenizer.from_pretrained(model_name)
        self.model = BertForSequenceClassification.from_pretrained(
            model_name, num_labels=num_labels
        )
        self.model.to(self.device)
```

```
    # Label names
    self.labels = [f"Class {i}" for i in range(num_labels)]

def predict_proba(self, text: str) -> np.ndarray:
    """Returns the softmax probability distribution over all classes."""
    self.model.eval()
    inputs = self.tokenizer(
        text,
        return_tensors="pt",
        truncation=True,
        padding=True,
        max_length=512
    )
    inputs = {k: v.to(self.device) for k, v in inputs.items()}
    with torch.no_grad():
        outputs = self.model(**inputs)
        logits = outputs.logits
    probs = torch.softmax(logits, dim=1)
    return probs.cpu().numpy().flatten()

def get_labels(self) -> List[str]:
    """Returns the list of label names."""
    return self.labels

def save_weights(self, path: str = "bert_detector_state.pt") -> None:
    """Saves model state_dict."""
    torch.save(self.model.state_dict(), path)
    print(f"Model weights saved to {path}")
```

```
    def load_weights(self, path: str = "bert_detector_state.pt") -> None:
        """Loads model state_dict."""
        try:
            state_dict = torch.load(path, map_location=self.device)
            self.model.load_state_dict(state_dict)
            print(f"Model weights loaded from {path}")
        except Exception as e:
            raise ValueError(f"Could not load weights from {path}") from e

    def train(
        self,
        train_dataset,
        val_dataset=None,
        epochs: int = 3,
        batch_size: int = 8,
        lr: float = 2e-5,
    ):
        self.model.train()
        train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
        optimizer = AdamW(self.model.parameters(), lr=lr)
        scheduler = (
            ReduceLROnPlateau(optimizer, mode="min", factor=0.5,
patience=1)
            if val_dataset
            else None
        )
```

```
        for epoch in range(1, epochs + 1):
            total_loss = 0.0
            pbar = tqdm(train_loader, desc=f"Epoch {epoch}/{epochs}",
leave=False)
            for batch in pbar:
                optimizer.zero_grad()
                inputs = {
                    "input_ids": batch["input_ids"].to(self.device),
                    "attention_mask": batch["attention_mask"].to(self.device),
                    "labels": batch["labels"].to(self.device),
                }
                outputs = self.model(**inputs)
                loss = outputs.loss
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
                pbar.set_postfix(loss=loss.item())

            avg_train_loss = total_loss / len(train_loader)
            print(f"Epoch {epoch}/{epochs} — Train Loss:
{avg_train_loss:.4f}")

            if val_dataset:
                val_loss, stats = self.evaluate(val_dataset, batch_size,
return_stats=True)
                print(
                    f"Epoch {epoch}/{epochs} — Val Loss: {val_loss:.4f} — "
                    f"Overall Acc: {stats['overall_accuracy']:.4f}"
                )
```

```
            scheduler.step(val_loss)

    def evaluate_per_class(
        self,
        y_true: np.ndarray,
        y_probs: np.ndarray,
        target_classes: List[int]
    ) -> dict:
        """
        One-vs-rest metrics for each class in `target_classes`.
        For cls>0: finds the threshold with FPR<=1% maximizing F1.
        For cls==0: uses default threshold=0.5.
        Returns a dict mapping cls -> stats dict.
        """
        per_class_stats = {}
        for cls in target_classes:
            y_true_bin = (y_true == cls).astype(int)
            scores = y_probs[:, cls]

            if cls == 0:
                thr = 0.5
                y_pred = (scores >= thr).astype(int)
                fp = ((y_pred == 1) & (y_true_bin == 0)).sum()
                tn_fp = (y_true_bin == 0).sum() or 1
                tp = ((y_pred == 1) & (y_true_bin == 1)).sum()
                fn_tp = (y_true_bin == 1).sum() or 1

                per_class_stats[cls] = {
                    "threshold":   thr,
```

```
                "FPR":        fp / tn_fp,
                "TPR":        tp / fn_tp,
                "Precision":   precision_score(y_true_bin, y_pred,
zero_division=0),
                "Recall":     recall_score(y_true_bin, y_pred, zero_division=0),
                "F1":         f1_score(y_true_bin, y_pred, zero_division=0),
            }
        else:
            fpr_arr, tpr_arr, thr_arr = roc_curve(y_true_bin, scores)
            best = {"F1": 0.0}
            for fp, tp, thr in zip(fpr_arr, tpr_arr, thr_arr):
                if fp <= 0.01:
                    y_pred = (scores >= thr).astype(int)
                    f1 = f1_score(y_true_bin, y_pred, zero_division=0)
                    if f1 > best["F1"]:
                        best = {
                            "threshold":  thr,
                            "FPR":        fp,
                            "TPR":        tp,
                            "Precision":  precision_score(y_true_bin, y_pred,
zero_division=0),
                            "Recall":     recall_score(y_true_bin, y_pred,
zero_division=0),
                            "F1":         f1,
                        }
            per_class_stats[cls] = best

    return per_class_stats
```

```
def evaluate(self, dataset, batch_size: int = 8, return_stats=False):
    """
    Evaluates the model on `dataset`:
      - Computes overall multiclass accuracy via argmax.
      - Computes per-class one-vs-rest metrics with FPR<=1% constraints.
    """
    self.model.eval()
    loader = DataLoader(dataset, batch_size=batch_size)
    all_probs, all_labels = [], []

    with torch.no_grad():
        for batch in tqdm(loader, desc="Evaluating", leave=False):
            inputs = {
                "input_ids": batch["input_ids"].to(self.device),
                "attention_mask": batch["attention_mask"].to(self.device),
            }
            labels = batch["labels"].to(self.device)
            outputs = self.model(**inputs)
            logits = outputs.logits

            probs = torch.softmax(logits, dim=1).cpu().numpy()
            all_probs.append(probs)
            all_labels.append(labels.cpu().numpy())

    y_probs = np.concatenate(all_probs, axis=0)
    y_true  = np.concatenate(all_labels, axis=0).astype(int)

    # Overall multiclass accuracy
    y_pred_argmax = np.argmax(y_probs, axis=1)
```

```
        overall_acc = accuracy_score(y_true, y_pred_argmax)
        print(f"\nOverall Accuracy (argmax): {overall_acc:.4f}")

        # Per-class metrics
        target_classes = list(range(self.model.config.num_labels))
        per_class = self.evaluate_per_class(y_true, y_probs, target_classes)

        print("\nPer-Class One-vs-Rest Metrics (FPR≤1% for AI classes):")
        for cls, stats in per_class.items():
            label = self.labels[cls] if cls < len(self.labels) else f"Class {cls}"
            print(
                f" • {label} (id={cls}): "
                f"Thr={stats['threshold']:.4f}, FPR={stats['FPR']:.4f}, "
                f"TPR={stats['TPR']:.4f}, Prec={stats['Precision']:.4f}, "
                f"Rec={stats['Recall']:.4f}, F1={stats['F1']:.4f}"
            )

        self.model.train()

        if return_stats:
            return overall_acc, {"overall_accuracy": overall_acc, "per_class":
per_class}

    def find_best_threshold(self, dataset, batch_size: int = 8):
        """
        Standalone per-class threshold finder via evaluate().
        """
        _, stats = self.evaluate(dataset, batch_size, return_stats=True)
        return stats["per_class"]
```