Лабораторна робота №4. CI/CD для ML-проєктів: автоматизація тестування та звітності з GitHub Actions та CML

0. Передумови та правила академічної доброчесності

Передумови: лабораторна робота охоплює результати ЛР1–ЛР3: базове тренування/оцінювання моделі та логування експериментів (MLflow), структуру репозиторію, версіонування даних і відтворюваність через DVC pipeline (dvc.yaml + dvc.lock), а також керування конфігураціями (Hydra) і HPO (Optuna).

Академічна доброчесність:

- Результати (метрики/графіки) мають бути відтворюваними: фіксуйте seed, версію даних (DVC hash), версію коду (git commit hash) і конфігурацію запуску.
- Заборонено підмінювати або «покращувати» метрики вручну (редагування metrics.* після тренування). У висновках пояснюйте отримані значення та фактори, що на них вплинули.
- Усі зміни мають проходити через Git (commit + PR) так, щоб історія експериментів була прозорою.

1. Мета роботи

1. Зрозуміти принципи Continuous Integration (CI) та Continuous Delivery (CD) в контексті Machine Learning (з урахуванням коду, даних, конфігурацій і артефактів моделі).
2. Навчитися створювати автоматизовані workflows за допомогою GitHub Actions.
3. Опанувати інструмент CML (Continuous Machine Learning) для автоматичної генерації звітів по експериментах у Pull Request.
4. Впровадити автоматичне тестування коду, даних та артефактів (pytest) у процес розробки.
5. Реалізувати концепцію Quality Gate: формалізоване правило (поріг/критерій), яке автоматично визначає, чи приймається модель/зміна.

2. Завдання для виконання

1. Підготувати репозиторій на GitHub з базовим ML-проєктом (продовжити проєкт з ЛР2-ЛР3).
2. Додати набір тестів (Unit/Smoke Tests) для перевірки:
   - якості/структури вхідних даних (data validation),
   - коректності створення артефактів (наприклад, model.pkl, metrics.json, confusion_matrix.png),
   - проходження Quality Gate за метриками (наприклад, f1 >= threshold).
3. Створити конфігураційний файл GitHub Actions: .github/workflows/cml.yaml.
4. Налаштувати workflow для виконання кроків при pull_request та/або push:
   - встановлення залежностей,
   - лінтинг (flake8/black),
   - запуск швидких тестів,
   - тренування (через dvc repro або python train.py),
   - запуск пост-тренувальних тестів (Quality Gate),
   - формування та публікація звіту (CML) у PR.
5. (Опційно, елемент CD) Для гілки main/master опублікувати артефакти моделі як workflow-artifact або виконати реєстрацію моделі у вашому інструменті життєвого циклу (наприклад, MLflow Registry).

3. Теоретичні відомості

3.1. CI/CD у Machine Learning

CI - автоматична перевірка якості змін (код + конфігурації + тести), яка запускається при кожному коміті або Pull Request. Для ML-проєктів CI часто включає:

- перевірки стилю коду (lint/format),
- data validation (схема, пропуски, діапазони),
- відтворюваність пайплайна (DVC),

- перевірку метрик (Quality Gate).

CD - автоматизація етапів після успішного CI: підготовка/публікація артефактів, реєстрація моделі, розгортання у staging тощо. У цьому курсі «CD» розглядається як публікація/промоція артефактів (мінімальний практичний рівень) або інтеграція з інструментами на кшталт MLflow Registry (розширення).

3.2. GitHub Actions

- Workflow - YAML файл у .github/workflows/.
- Event - подія запуску (наприклад, push, pull_request, workflow_dispatch).
- Job - набір кроків (steps), що виконуються на одному runner.
- Runner - VM/контейнер, що виконує job.

3.3. CML (Continuous Machine Learning)

CML - open-source CLI-інструмент від Iterative.ai, який дозволяє інтегрувати ML-звіти у Git-платформи (GitHub/GitLab/Bitbucket). Типовий сценарій: workflow тренує модель, генерує метрики/візуалізації та залишає коментар у Pull Request, що спрощує Code Review для Data Science задач.

Важливо: щоб зображення коректно відображались у PR-коментарі, їх потрібно публікувати у звіт як URL (наприклад, командою cml publish ... --md).

4. Покрокова інструкція

4.1. Підготовка проєкту

1. Переконайтеся, що ваш проєкт має файл requirements.txt. Додайте необхідні бібліотеки.

2. У вашому проєкті має бути скрипт, який:

- зберігає артефакт моделі у model.pkl,
- зберігає метрики у metrics.json (або metrics.txt, для Quality Gate зручніше JSON),
- зберігає візуалізацію у confusion_matrix.png.

Приклад збереження метрик у JSON (рекомендовано для автоматичної перевірки):

```
import json

metrics = {"accuracy": float(accuracy), "f1": float(f1)}
with open("metrics.json", "w", encoding="utf-8") as f:
    json.dump(metrics, f, ensure_ascii=False, indent=2)
```

3. Якщо ви використовуєте DVC pipeline (ЛР2), переконайтеся, що:

- файл dvc.yaml описує стадії підготовки/тренування,
- файл dvc.lock у віддаленому репозиторію,
- дані доступні для CI-runner:
- або датасет малий і зберігається в Git,
- або налаштовано DVC remote (S3/Azure Blob/Google Drive тощо) і в CI виконується dvc pull.

4.2. Написання тестів (pytest)

У CI важливо мати швидкі перевірки, які не потребують тренування (pre-train), і перевірки результатів (post-train). Це робить пайплайн стабільнішим і дає швидший зворотний зв'язок у Pull Request.

- Pre-train (до тренування): структура/якість даних, наявність колонок, діапазони значень, відсутність критичних пропусків, валідність конфігурації (Hydra) тощо.
- Post-train (після тренування): артефакти (model.pkl, metrics.json, *.png) і Quality Gate.

1. Створіть папку tests/ та файл tests/test_model.py.

2. Реалізуйте дві групи перевірок:

- *Швидкі перевірки (pre-train)*: наприклад, схема даних або наявність потрібних колонок.
- *Пост-тренувальні перевірки (post-train)*: наявність артефактів та Quality Gate по метриках.

Приклад pre-train тесту для базової перевірки даних (без зовнішніх бібліотек). Адаптуйте під ваш датасет:

```
import os
import pandas as pd

def test_data_schema_basic():
    data_path = os.getenv("DATA_PATH", "data/train.csv")
    assert os.path.exists(data_path), f"Data not found: {data_path}"

    df = pd.read_csv(data_path)

    required_cols = {"text", "label"}
    missing = required_cols - set(df.columns)
    assert not missing, f"Missing columns: {sorted(missing)}"

    assert df["label"].notna().all(), "label contains spaces"
    assert df.shape[0] >= 50, "Too few lines for a learning
experiment"
```

Приклад (спрощений, але робочий) Quality Gate для `metrics.json`:

```
import json
import os
import os.path

def test_artifacts_exist():
    assert os.path.exists("model.pkl"), "model.pkl not found"
    assert os.path.exists("metrics.json"), "metrics.json not found"
    assert os.path.exists("confusion_matrix.png"),
"confusion_matrix.png not found"

def test_quality_gate_f1():
    # Мінімально допустимий рівень якості задаємо як частину політики.
    # Для навчальних прикладів можна почати з 0.70 і підбирати поріг
під конкретний датасет.
    threshold = float(os.getenv("F1_THRESHOLD", "0.70"))

    with open("metrics.json", "r", encoding="utf-8") as f:
        metrics = json.load(f)

    f1 = float(metrics["f1"])
    assert f1 >= threshold, f"Quality Gate not passed: f1={f1:.4f} <
{threshold:.2f}"
```

4.3. Налаштування GitHub Actions workflow

На практиці повне тренування може бути занадто повільним для CI. Для лабораторної роботи:

- використовувати малу підвибірку в CI (наприклад, --max-rows 5000),
- додати режим CI=true, який скорочує кількість епох/ітерацій,
- запускати повний retraining за розкладом (schedule) або на merge у main.

Це дозволяє зберегти повний цикл і не перевантажувати runner.

1. У корені репозиторію створіть директорію .github/workflows.

2. Створіть файл .github/workflows/cml.yaml.

Нижче наведено рекомендований приклад, який:

- запускається на pull_request (коментар у PR) та push (перевірки),
- встановлює Python,
- завантажє дані через DVC,
- тренує модель,
- запускає тести,
- формує CML-звіт і публікує зображення через cml publish.

```yaml
name: Model CI (Train, Test, Report)

on:
  push:
  pull_request:

permissions:
  contents: read
  pull-requests: write

jobs:
  ci:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
```

```
        with:
          python-version: "3.11"
          cache: "pip"

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Lint (flake8)
        run: |
          flake8 . --count --select=E9,F63,F7,F82 --show-source --
statistics

      - name: Format check (black)
        run: |
          black . --check

      # DVC remote:
      # - name: DVC pull data
      #   run: |
      #     dvc pull
      # env:
      #   AZURE_STORAGE_CONNECTION_STRING: ${{
secrets.AZURE_STORAGE_CONNECTION_STRING }}

      - name: Train model
        run: |
         python train.py

      - name: Run tests (incl. Quality Gate)
        env:
          F1_THRESHOLD: "0.70"
        run: |
          pytest -q

      - name: Create CML report (PR only)
        if: github.event_name == 'pull_request'
        env:
          REPO_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: |
          echo "## Model Metrics" > report.md
          cat metrics.json >> report.md
          echo "" >> report.md
          echo "## Visualization" >> report.md
          cml publish confusion_matrix.png --md >> report.md
          echo "" >> report.md
          echo "### Quality Gate" >> report.md
          echo "F1 threshold: $F1_THRESHOLD" >> report.md

          cml comment create report.md
```

Примітка. Для PR з forks або обмежених налаштувань репозиторію `GITHUB_TOKEN` може не мати прав на створення коментарів. У такому випадку використайте PAT (Personal Access Token) як секрет (наприклад, `CML_TOKEN`) і передайте його у `REPO_TOKEN`.

4.4. (Опційно) CD: публікація артефактів після merge

Додайте крок, який запускається лише на push у main/master і публікує model.pkl як артефакт:

```
      - name: Upload model artifact (main only)
        if: github.ref == 'refs/heads/main' && github.event_name ==
'push'
        uses: actions/upload-artifact@v4
        with:
          name: model
          path: model.pkl
```

4.5. (Опційно) Baseline comparison у Pull Request

Щоб зробити Code Review змістовнішим, додайте порівняння метрик PR-гілки з baseline з main/master.

1. У гілці main зберігайте файл baseline/metrics.json (метрики «еталонного» запуску).
2. У PR workflow після тренування згенеруйте metrics.json і порівняйте його з baseline/metrics.json (наприклад, коротким Python-скриптом).
3. Додайте у CML-звіт таблицю «було/стало» та Δ.

Це демонструє принцип regression testing для ML-метрик і формалізує обговорення змін.

5. Контрольні запитання

1. Що таке CI і яку проблему воно вирішує у ML-проєктах?
2. У чому основна різниця між CI та CD (у контексті цієї лабораторної)?
3. Які основні складові GitHub Actions: workflow, job, step, runner?
4. Чому для ML-проєктів важливо зберігати метрики у машинно-читаному форматі (наприклад, metrics.json), а не лише в консолі?

5. Що таке Quality Gate і навіщо він потрібен в процесі перевірки змін у Pull Request?
6. Поясніть різницю між pre-train та post-train тестами. Наведіть по 2 приклади для кожного типу.
7. Які типові перевірки data validation ви б додали у pre-train тести (схема, пропуски, діапазони тощо) і чому саме вони?
8. Які permissions потрібні workflow, щоб залишати коментарі у Pull Request, і що станеться, якщо їх не вистачає?
9. Чому в CML-звіті зображення краще додавати через cml publish ... --md, а не як локальне посилання на файл у репозиторії?
10. Як ви забезпечите доступність даних у CI-раннері, якщо дані не зберігаються в Git (DVC remote)? Які категорії секретів можуть знадобитися?
11. Які ризики "хибно зеленого" Quality Gate (false positive) і "хибно червоного" (false negative) у ML-CI? Як їх мінімізувати?
12. Запропонуйте підхід до обґрунтування порогу (threshold) для Quality Gate: на які дані/експерименти ви спиратиметесь і чому?
13. Опишіть дизайн baseline comparison у PR: як зберігати baseline, як порівнювати метрики, і як показати зміни (Δ) у CML-звіті так, щоб це було корисно для review.
14. Як зробити CI швидким і стабільним, якщо повне тренування займає довго (підвибірка, CI-режим, кеші, schedule, розділення job'ів)? Які компроміси виникають?
15. Як би ви реалізували policy-as-code для ML-якості: щоб правила (пороги, мінімальні вимоги до даних, вимоги до артефактів) були версіоновані та виконувались автоматично? Які файли/механізми ви б використали?

6. Рекомендовані матеріали

- GitHub Docs - Workflow syntax for GitHub Actions (події on, jobs, steps, if, env, permissions) (GitHub Docs)
- GitHub Docs - Managing GitHub Actions settings for a repository (налаштування Actions, "Workflow permissions", важливо для GITHUB_TOKEN) (GitHub Docs)

- CML Docs - Command reference: cml publish (публікація зображень для коректного відображення у PR-коментарі) (CML · Continuous Machine Learning)
- GitHub Marketplace - Setup CML (Continuous Machine Learning) Action (практичне використання cml comment create у GitHub Actions) (GitHub)
- DVC Docs - dvc pull command reference (завантаження даних/артефактів з remote у CI середовищі) (Data Version Control · DVC)
- pytest Docs - How to use fixtures (організація тестів для pre-train/post-train перевірок) (docs.pytest.org)