Лабораторна робота №5. Оркестрація ML-пайплайнів: Від CI/CD до Continuous Training

0. Передумови та правила академічної доброчесності

Передумови: лабораторна робота є інтеграційною та базується на результатах виконання ЛР1–ЛР4: налаштовано MLflow Tracking для логування експериментів, організовано DVC-конвеєри для версіонування даних, реалізовано гіперпараметричну оптимізацію (Optuna/Hydra), налаштовано CI/CD (GitHub Actions) та автоматичне тестування.

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.
- Зміни в API мають бути задокументовані у коміт-повідомленнях та CHANGELOG.
- У звіті необхідно пояснювати архітектурні рішення (вибір операторів Airflow, стратегія тестування DAG тощо).

1. Мета роботи

1. Зрозуміти концепцію MLOps Maturity Model та важливість переходу до повністю автоматизованих систем.
2. Опанувати принципи Workflow Orchestration та побудови спрямованих ациклічних графів (DAG) за допомогою Apache Airflow.
3. Навчитися створювати оптимізовані Docker-контейнери для ML-задач (multi-stage builds).
4. Реалізувати Continuous Integration для перевірки якості коду та коректності DAG-ів.
5. Забезпечити автоматичне виконання ML-пайплайну (Training → Evaluation → Registration) під управлінням оркестратора.

Важливо: ви може обрати довільний фреймворк або бібліотеку на власний розсуд. Основна мета - дотримання принципів та етапів MLOps, а не використання конкретного інструменту.

2. Теоретичні відомості

2.1. Рівні зрілості MLOps (Google MLOps Maturity Model)

Автоматизація машинного навчання (ML) у виробничих системах зазвичай відбувається як поступова, еволюційна трансформація практик розроблення та експлуатації - від індивідуальної дослідницької роботи до стандартизованих, керованих і відтворюваних інженерних процесів. Така еволюція зумовлена потребою підвищувати надійність, масштабованість і контроль якості моделей, а також зменшувати ризики, пов’язані з людським фактором, фрагментарністю артефактів і несистемністю розгортання. У методології MLOps часто виокремлюють кілька рівнів зрілості автоматизації, які відрізняються ступенем формалізації життєвого циклу моделі та глибиною інтеграції з інженерними практиками.

Рівень 0: Повністю ручний процес

На початковому рівні ML-розроблення має переважно експериментальний характер і зосереджується навколо індивідуальної роботи Data Scientist. Навчання моделі здійснюється локально, найчастіше в середовищі Notebook, де код, дані та результати експериментів співіснують у слабко структурованому форматі. Відтворюваність експериментів ускладнюється через відсутність стандартизованого управління залежностями, версіонування набору даних і формалізованого журналу метрик.

Перенесення результатів у виробниче середовище (production) зазвичай відбувається шляхом ручної передачі артефактів: файлів моделі, скриптів передобробки даних або конфігурацій. Цей підхід характеризується високими операційними ризиками: можливі розбіжності між середовищами розроблення та випуском, неконсистентність версій бібліотек, складність аудиту змін, а також значна залежність від конкретного виконавця. У підсумку, час виходу змін зростає, а контроль якості та стабільність розгортання залишаються обмеженими.

Рівень 1: Автоматизація ML-пайплайна та фокус на Continuous Training (CT)

Наступним етапом є перехід до автоматизації ML-конвеєра (ML Pipeline Automation), коли процес підготовки даних, навчання, валідації та реєстрації

моделі оформлюється як послідовність формально визначених кроків. Центральною ідеєю цього рівня стає Continuous Training (CT) - безперервне (або регулярне) перенавчання моделей у відповідь на появу нових даних, зміну розподілів або деградацію якості.

На цьому рівні експерименти виконуються не вручну, а через автоматизовані запуски - за розкладом або подіями (наприклад, надходженням нового набору даних). З'являється поняття оркестрації (Orchestration), тобто керування виконанням конвеєрів як керованого графа завдань: визначення залежностей між етапами, контролю їхньої послідовності, повторів у разі збоїв та моніторингу статусів. Оркестрація сприяє стандартизації робочого процесу, підвищує відтворюваність результатів і зменшує кількість ручних операцій.

Водночас повна інтеграція з інженерними практиками ще не є завершеною: зміни в коді конвеєрів можуть вноситися без належного циклу тестування та збірки, а розгортання компонентів інфраструктури може залишатися частково ручним. Однак навіть за цих умов організація отримує істотні переваги: регулярне оновлення моделей, систематичний контроль метрик навчання та більш прозорий життєвий цикл експериментів.

Рівень 2: Повна автоматизація через CI/CD та інтеграцію з CT

Найвищий із наведених рівнів передбачає повноцінну автоматизацію, де ML-процеси узгоджуються з усталеними практиками розроблення програмного забезпечення. На цьому етапі зміни в коді конвеєра, компонентів передобробки, сервісів інференсу та конфігурацій інфраструктури проходять через CI/CD-процеси: автоматизовані тести (unit, integration, e2e), перевірки якості коду, збірку артефактів та контрольоване розгортання. Таким чином, оновлення «системи» (pipeline code) відбувається за правилами безперервної інтеграції та доставки.

Паралельно оновлення «моделей» здійснюється через Continuous Training (CT): нові версії моделей створюються у стандартизованому конвеєрі, проходять формалізовані етапи оцінювання (валідація, порівняння з базовою моделлю, перевірки на регресію якості), після чого можуть бути автоматично зареєстровані та, за визначених політик, розгорнуті. У межах цього підходу розмежовуються дві площини керування: (1) інженерна - керування змінами коду та інфраструктури через CI/CD; (2) модельна - керування життєвим циклом моделей через CT, включно з версіонуванням, трасуванням походження (provenance) та аудитом.

Зрілість рівня 2 проявляється також у підсиленні спостережуваності (observability): системи моніторингу якості інференсу, контролю data drift / concept drift, відстеження SLA/latency, а також процедур реагування (наприклад, автоматизоване переведення на попередню стабільну версію). У результаті організація отримує керований, масштабований і відтворюваний процес доставки ML-рішень, де мінімізується ручне втручання, а якість забезпечується формалізованими контролями на всіх етапах.

2.2. Контейнеризація у ML

Docker є інструментом контейнеризації, який забезпечує ізоляцію програмного середовища та підвищує відтворюваність розгортання застосунків. Завдяки контейнерам залежності (бібліотеки, системні пакети, конфігурації) фіксуються на рівні образу, що мінімізує розбіжності між середовищами розроблення, тестування та продакшну. Така ізоляція зменшує ризики конфліктів версій і спрощує перенесення застосунку між різними інфраструктурами (локальні машини, сервери, хмарні платформи).

Важливим механізмом оптимізації контейнерних образів є багатоступеневі збірки (multi-stage builds). Їхня ключова мета полягає в тому, щоб розділити процес компіляції/збірки та середовище виконання, уникаючи перенесення в кінцевий образ зайвих компонентів. Це дозволяє отримувати значно легші, безпечніші та швидше завантажувані образи, що є критичним для CI/CD-процесів і масштабованого розгортання.

2.3. Оркестрація: Apache Airflow

Оркестратори робочих процесів призначені для керування складними залежностями між задачами, забезпечуючи контроль послідовності виконання, узгодження етапів і повторюваність запусків. Одним із поширених інструментів цього класу є Apache Airflow, який реалізує підхід *pipeline-as-code*: конвеєри описуються мовою Python і формалізуються у вигляді DAG (*Directed Acyclic Graph*), тобто спрямованого ациклічного графа. У такій моделі вузли відповідають задачам, а ребра - їхнім залежностям, що дозволяє явно визначати порядок виконання та уникати циклічних сценаріїв.

Архітектура Apache Airflow включає кілька ключових компонентів, які разом забезпечують планування, виконання та спостережуваність конвеєрів:

- ***Scheduler*** - компонент планування, який інтерпретує DAG, відстежує стан задач і ініціює їхній запуск відповідно до розкладу або тригерів.

- ***Executor*** - механізм виконання, що визначає середовище, у якому запускаються задачі, зокрема локально (Local Executor), у розподіленому режимі через Celery або у контейнеризованій інфраструктурі Kubernetes.
- ***Webserver*** - вебінтерфейс для керування конвеєрами, перегляду журналів, статусів виконання та діагностики помилок.
- ***Operators*** - параметризовані шаблони задач, які інкапсулюють типові сценарії виконання (наприклад, запуск Bash-команд, Python-функцій або контейнерів), що спрощує побудову конвеєрів і підвищує повторне використання компонентів.

Таким чином, Apache Airflow надає структурований і формалізований спосіб опису та виконання конвеєрів із чітко визначеними залежностями, що є критично важливим для надійної автоматизації процесів обробки даних і ML-життєвого циклу.

3. Завдання для виконання

В рамках роботи необхідно побудувати комплексну MLOps-систему, яка поєднує CI/CD для коду та Оркестрацію для даних/моделей.

Крок 1: Контейнеризація середовища (Docker)

1. Створити Dockerfile для вашого ML-проєкту.
2. Застосувати паттерн multi-stage build:
   - Використати проміжний етап для встановлення важких залежностей.
   - Фінальний образ має бути максимально легким (наприклад, на базі python:slim) і містити лише необхідне для запуску скриптів (dvc, mlflow, requirements).

Крок 2: Розгортання Оркестратора (Apache Airflow)

1. Налаштувати локальне середовище Apache Airflow за допомогою Docker Compose.
2. Забезпечити зв’язок між Airflow та вашим ML-кодом (через монтування томів або DockerOperator).

Крок 3: Проєктування та розробка DAG

Створити Python-файл з описом DAG (наприклад, ml_training_pipeline.py), який реалізує наступний процес:

1. Sensor / Check: перевірка доступності даних або оновлень у DVC.

2. Data Preparation: запуск стадії підготовки даних (dvc repro stage: prepare).

3. Model Training: запуск тренування моделі.

4. Evaluation & Branching: використання BranchPythonOperator для прийняття рішення: Якщо модель краща за поріг - перехід до реєстрації. Якщо гірша - завершення або сповіщення.

5. Model Registration: реєстрація успішної моделі у MLflow Model Registry зі стадією Staging.

Крок 4: Автоматизація CI (GitHub Actions)

Створити конфігурацію CI/CD (.github/workflows/main.yaml), яка виконується при Push у репозиторій:

1. Linting: перевірка якості Python-коду (flake8/pylint).

2. DAG Integrity Test: перевірка, що файли DAG не містять синтаксичних помилок і коректно завантажуються Airflow (наприклад, тест на import_errors у об'єкті DagBag).

3. Docker Build: перевірка, що Docker-образ успішно збирається.

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)
```

4. Методичні вказівки до виконання

4.1. Інтеграція Airflow та ML-проєкту

Інтеграція Apache Airflow з ML-проєктом зазвичай зводиться до вибору способу виконання навчання, підготовки даних і відтворення експериментів із урахуванням керування залежностями та відтворюваності середовища. На практиці можна виокремити два базові патерни запуску ML-задач із Apache Airflow, які відрізняються рівнем ізоляції та вимогами до інфраструктури.

1. BashOperator у поєднанні з VirtualEnv

У цьому сценарії Apache Airflow встановлюється в те саме середовище, що й ML-залежності, або принаймні має доступ до відповідного віртуального оточення (venv). Виконання задач реалізується як виклик командного рядка через BashOperator, наприклад запуск відтворення кроків пайплайна командою dvc repro train. Перевагою підходу є простота реалізації та зручність для локального запуску, оскільки він мінімізує кількість додаткових шарів абстракції. Водночас такий варіант вимагає дисципліни у керуванні залежностями й коректного налаштування шляхів, щоб Airflow гарантовано використовував потрібне середовище виконання.

2. DockerOperator

Альтернативний підхід полягає в запуску кожної задачі як окремого Docker-контейнера. Це забезпечує більш «чисту» інтеграцію завдяки ізоляції залежностей і зменшенню впливу системних відмінностей між середовищами. Однак контейнерний запуск потребує додаткової конфігурації: зокрема, доступу до Docker daemon (через docker socket) або використання Docker-in-Docker у залежності від обраної інфраструктури. Тому, попри інженерні переваги, цей підхід може бути складнішим для навчальних або обмежених середовищ.

Для роботи доцільно обрати більш простий варіант із BashOperator, за умови належного налаштування шляхів і гарантії, що Apache Airflow запускає команди саме в тому оточенні, де встановлено необхідні ML-бібліотеки та інструменти.

4.2. Тестування DAG-ів у CI

Щоб перевірити, чи немає в DAG помилок імпорту, створіть простий pytest тест:

```
from airflow.models import DagBag

def test_dag_import():
    dag_bag = DagBag(dag_folder='dags/', include_examples=False)
```

```
    assert len(dag_bag.import_errors) == 0, f"DAG import errors:
{dag_bag.import_errors}"
```

Цей тест слід запускати в GitHub Actions, попередньо встановивши apache-airflow у середовищі CI.

4.3. Реєстрація моделі з умовами

Для реалізації логіки розгалуження (Branching) створіть Python-функцію:

```
def check_accuracy(**kwargs):
    ti = kwargs['ti']
    metrics = ti.xcom_pull(task_ids='evaluate_model')
    if metrics['accuracy'] > 0.85:
        return 'register_model'
    return 'stop_pipeline'
```

Використовуйте BranchPythonOperator та передайте цю функцію як python_callable.

5. Контрольні запитання

1. Поясніть, як взаємодіють компоненти CI/CD (GitHub Actions) та Оркестратор (Airflow) у побудованій системі? Що запускає що?
2. Які переваги дає використання DAG порівняно з лінійним скриптом (`run_all.sh`)?
3. Навіщо потрібен `BranchPythonOperator` і в яких сценаріях MLOps він використовується?
4. Як Multi-stage build впливає на безпеку та швидкість розгортання контейнерів?
5. Чому важливо тестувати DAG-файли на етапі CI/CD? Які типи помилок це дозволяє виявити?
6. Опишіть архітектуру Apache Airflow: за що відповідають Scheduler, Webserver та Executor?
7. Що таке XComs в Airflow? Які обмеження існують на передачу даних через XComs і чому не варто передавати великі датасети таким чином?
8. Яка різниця між операторами `Sensor` та звичайними операторами (Task)? Наведіть приклад використання сенсора в MLOps.
9. Поясніть поняття «ідемпотентність» (idempotency) стосовно завдань в DAG. Чому це критично для автоматизованих пайплайнів?

10. Як реалізувати версіонування не тільки коду, а й інфраструктури (Infrastructure as Code) у контексті MLOps?
11. У чому різниця між поняттями `Build Artifact` та `Model Artifact`?
12. Які стратегії обробки помилок (Failure Handling) надає Airflow (retries, alerts, SLAs)?
13. Що таке `Backfill` в Airflow і коли виникає потреба у його застосуванні?
14. Як забезпечити безпеку конфіденційних даних (API keys, passwords) при використанні CI/CD та оркестраторів?

6. Рекомендовані матеріали

- Google Cloud Architecture Center (MLOps: Continuous delivery and automation pipelines in machine learning)
- Apache Airflow Documentation (Core Concepts & DAGs)
- Docker Documentation (Best practices for writing Dockerfiles - Multi-stage builds)
- MLflow Documentation (Model Registry Concepts)
- GitHub Actions Documentation (Building and testing Python)