Лабораторна робота №2. Версіонування даних та побудова пайплайнів (DVC)

1. Мета роботи

1. Опанувати інструмент DVC (Data Version Control) для управління версіями великих файлів даних.
2. Навчитися розділяти версіонування коду (Git) та даних (DVC).
3. Спроектувати та реалізувати відтворюваний ML-пайплайн (Pipeline), що складається з етапів підготовки даних та навчання моделі.
4. Засвоїти принципи роботи з DAG (Directed Acyclic Graph) у контексті виконання ML-задач.

2. Завдання для виконання

1. Встановити та ініціалізувати DVC у проєкті, створеному в Лабораторній роботі №1.
2. Налаштувати локальне віддалене сховище (Local Remote) для імітації хмарного сховища (S3/Google Drive).
3. Додати файл сирих даних під контроль DVC.
4. Виконати рефакторинг коду: розділити монолітні скрипти, для прикладу train.py на два окремі етапи: prepare.py (підготовка даних) та train.py (навчання).
5. Описати пайплайн у файлі dvc.yaml, визначивши входи (dependencies) та виходи (outputs) для кожного етапу.
6. Запустити виконання пайплайну командою dvc repro.
7. Продемонструвати роботу механізму кешування DVC: змінити параметри або дані та перевірити, що виконуються лише необхідні кроки.

3. Теоретичні відомості

3.1. Проблема версіонування даних

Git ідеально підходить для текстових файлів (код), але погано працює з великими бінарними файлами (датасети, моделі). Зберігання гігабайтів даних у Git призводить до розростання репозиторію (.git folder) та сповільнення роботи. DVC вирішує цю проблему, зберігаючи у Git лише легкі метафайли (.dvc), які містять хеш-суми реальних даних, а самі дані зберігаються у зовнішньому сховищі.

3.2. ML-пайплайни та DAG

ML-експеримент рідко складається з одного скрипта. Зазвичай це послідовність кроків: Data Ingestion -> Cleaning -> Feature Engineering -> Training -> Evaluation. Цю послідовність можна представити як DAG (Directed Acyclic Graph) - орієнтований ациклічний граф. DVC дозволяє описати цей граф, де кожен вузол - це скрипт, а ребра - це файли даних, що передаються між скриптами.

3.3. Відтворюваність (Reproducibility)

Головна перевага використання DVC-пайплайнів - гарантія того, що результат (модель) отримано саме з цієї версії даних і саме цим кодом. Команда dvc repro автоматично визначає, які кроки потрібно перезапустити, якщо змінилися вхідні дані або код, а які можна взяти з кешу.

4. Покрокова інструкція

4.1. Встановлення та ініціалізація DVC

1. Активуйте віртуальне середовище (якщо не активоване): Для Windows:

   ```
   .\venv\Scripts\activate
   ```

   Для Linux/MacOS:

   ```
   source venv/bin/activate
   ```

2. Встановіть DVC:

   ```
   pip install dvc
   ```

3. Ініціалізуйте DVC у корені вашого проєкту (там, де папка .git):

   ```
   dvc init
   ```

4. Зафіксуйте зміни в Git:

   ```
   git commit -m "Initialize DVC"
   ```

4.2. Налаштування сховища (Remote Storage)

Для лабораторної роботи ми будемо використовувати локальну папку як імітацію хмарного сховища (S3/Azure Blob).

1. Створіть папку для сховища (бажано за межами папки проєкту, наприклад, на рівень вище):

   ```
   mkdir ../dvc_storage
   ```

2. Додайте цю папку як "remote" сховище:

   ```
   dvc remote add -d mylocal ../dvc_storage
   ```

3. Зафіксуйте зміни конфігурації в Git:

   ```
   git add .dvc/config
   git commit -m "Setup local remote storage"
   ```

4.3. Версіонування сирих даних

1. Переконайтеся, що ваш датасет знаходиться у data/raw/dataset.csv (або інша назва).

2. Додайте файл під контроль DVC:

   ```
   dvc add data/raw/dataset.csv
   ```

3. Додайте файл .dvc у Git, а оригінальний файл даних — у .gitignore (DVC зазвичай робить це автоматично):

   ```
   git add data/raw/dataset.csv.dvc data/raw/.gitignore
   git commit -m "Track raw dataset with DVC"
   ```

4. Надішліть дані у сховище:

   ```
   dvc push
   ```

4.4. Рефакторинг коду (Створення стадій)

Вам необхідно розділити логіку вашого src/train.py на два файли.

1. Скрипт підготовки (src/prepare.py):

- **Вхід:** Сирі дані (data/raw/dataset.csv).
- **Дії:** Очистка, обробка пропусків, генерація ознак (Feature Engineering), розділення на train/test (опціонально, або це можна робити в train).
- **Вихід:** Підготовлені дані, збережені, наприклад, у data/prepared/train.csv та data/prepared/test.csv.

Приклад структури src/prepare.py:

```
import pandas as pd
import sys
import os

input_file = sys.argv[1]  # data/raw/dataset.csv
output_dir = sys.argv[2]  # data/prepared

os.makedirs(output_dir, exist_ok=True)
df = pd.read_csv(input_file)

# ... Ваша логіка обробки ...

df.to_csv(os.path.join(output_dir, "train.csv"), index=False)
```

2. Скрипт навчання (src/train.py):

- **Вхід:** Підготовлені дані (data/prepared/train.csv).
- **Дії:** Навчання моделі, логування в MLflow.
- **Вихід:** Модель (артефакт) та метрики.

4.5. Створення пайплайну (dvc.yaml)

Створіть у корені проєкту файл dvc.yaml під вашу задачу на основі вибраного датасету та моделі (необхідно визначити кількість кроків та обробки та імплементувати). Для прикладу цей файл описує стадії виконання.

```
stages:
  prepare:
    cmd: python src/prepare.py data/raw/dataset.csv data/prepared
    deps:
      - data/raw/dataset.csv
      - src/prepare.py
    outs:
      - data/prepared

  train:
    cmd: python src/train.py data/prepared data/models
    deps:
      - data/prepared
      - src/train.py
    outs:
      - data/models
```

4.6. Запуск пайплайну

1. Виконайте команду:

```
dvc repro
```

DVC проаналізує граф, побачить, що стадії ще не виконувалися, і запустить їх послідовно.

2. Перевірте створення файлу dvc.lock. Цей файл фіксує точні версії (хеші) всіх залежностей та результатів на момент успішного виконання.

```
git add dvc.yaml dvc.lock src/prepare.py src/train.py
git commit -m "Create DVC pipeline"
```

4.7. Перевірка відтворюваності

1. Спробуйте запустити dvc repro ще раз. *Очікуваний результат:* DVC скаже Stage '...' didn't change, skipping, оскільки нічого не змінилося.
2. Внесіть дрібну зміну в src/train.py (наприклад, змініть гіперпараметр або додайте print).
3. Запустіть dvc repro. *Очікуваний результат:* DVC пропустить стадію prepare (бо дані та код підготовки не змінилися) і запустить тільки train. Це економить час!

5. Контрольні запитання

1. Чому не рекомендується робити git add для файлів розміром 1GB+?
2. Яку роль виконує файл .dvc (наприклад, dataset.csv.dvc)?
3. Що таке dvc.lock і чому його потрібно комітити в Git?

4. Як DVC розуміє, що стадію prepare не потрібно перезапускати при повторному виклику dvc repro?
5. Ви змінили сирі дані (data/raw/dataset.csv). Які кроки потрібно виконати, щоб оновити пайплайн і зафіксувати нову версію даних?
6. Чим відрізняється dvc push від git push?
7. Сценарій: Ви працюєте над експериментом, змінили параметри препроцесингу, запустили dvc repro. Результат погіршився. Як за допомогою DVC та Git повернутися до попереднього стану (і коду, і даних, і моделі)?
8. Інтеграція: Як можна використати файл dvc.lock у CI/CD пайплайні (наприклад, GitHub Actions) для перевірки того, чи модель була дійсно навчена на зафіксованих даних?

6. Рекомендовані матеріали

1. DVC Get Started: Official Tutorial — Офіційний курс.
2. YouTube: DVC Course by Iterative.