Zarządzenie projektami

Cel: krótki opis typowych praktyk pomagających w zespołowej pracy nad (większymi) projektami, a następnie utrzymywaniu projektów.

Typowy cykl życia projektu:

$$\text{Eksperymenty} \rightarrow \text{praca nad projektem} \rightarrow \text{utrzymywanie projektu}$$

Typowe problemy:

- Brakujące zależności.
- Zmieniające się wersje używanych bibliotek.
- Śledzenie (i commitowanie) niechcianych plików.
- „U mnie działa”.

Zarządzenie projektami

Typowa struktura katalogów dla małego/średniego projektu z ML w Pythonie.
Założenie: używamy gita do kontroli wersji.

```
src/ # źródła
    main.py
    utils.py
    ...
data/ # dane
    samples.csv
    metadata.txt
    ...
tests/ # testy sprawdzające poprawność
    test.py
    ...
README.md
.gitignore
requirements.txt
```

Zarządzenie projektami

Przypomnienie: systemy kontroli wersji (konkretnie git):

Katalog roboczy z projektem (lokalny)

$\downarrow$ add/rm

Poczekalnia (lokalna)

$\downarrow$ commit

Repozytorium lokalne

$\downarrow$ push

Repozytorium zdalne

Komunikacja w drugą stronę: checkout, pull.

Zarządzenie projektami

Dobra praktyka – każdy commit jako snapshot (migawka) *działającego* projektu (a nie po prostu robocza kopia zapasowa).

- Wymóg można osłabić dla gałęzi, które nie są główne.
- Można zignorować dla commitów w repozytorium lokalnym, pod warunkiem że zostaną ściśnięte (squash) przed pushem (wysłaniem commitów do zdalnego repozytorium).
- GitHub pozwala na ściśnięcie commitów z gałęzi przed scaleniem tej gałęzi – squash and merge (tutaj live demo).
- Ściśnięcie commitów lokalnych (przed pushem) wymaga linii poleceń (np. `git reset --soft ...`).

Niechciane pliki

Praca nad projektem zazwyczaj prowadzi do istnienia plików i katalogów, które nie powinny stawać się częścią repozytorium, np.

- Skompilowany kod, linkowane obiekty etc. (Python: katalog __pycache__, pliki .pyc, .pyd, .pyo; C++: .a, .o, .obj, .exe, .dll).
- Pliki edytora (PyCharm: katalog .idea, VS Code: katalog .vscode, Jupyter: katalog .ipynb_checkpoints).
- Różne pliki generowane lokalnie przez uruchomiony projekt (np. pliki z konfiguracją, pliki robocze).
- Wirtualne środowisko (o tym później): katalogi typowo nazwane venv lub .venv
- Inne specyficzne pliki (MacOS/iOS: .DS_Store, Windows: Thumbs.db)
- itp.

git pozwala na utworzenie list wykluczeń dla całego projektu (.gitignore – tu live demo) lub dla pojedynczego użytkownika.

Wirtualne środowisko

Klasyczny problem: projekty mają zależności (np. biblioteki typu numpy). Biblioteki (jak każdy inny projekt) są rozwijane: w starszych wersjach będzie brakować pewnych funkcjonalności. Niektóre elementy funkcjonalności mogą zostać wycofywne (brak wstecznej kompatybilności).

W konsekwencji projekt może wymagać konkretnych wersji (lub konkretnego układu wersji) danych bibliotek.

Rozwiązanie: *wirtualne środowisko* – minimalistyczna kopia interpretera wraz z niezależnym układem bibliotek, po jednej kopii na projekt (krótkie live demo).

Wirtualne środowisko

Repozytorium przechowuje informację o bibliotekach używanych przez moduł, ale nie przechowuje samych bibliotek. Typowy schemat pracy (live demo):

1. Klonowanie repozytorium.
2. Utworzenie środowiska standardowymi narzędziami (venv, conda, etc.).
3. Aktywacja środowiska.
4. Zautomatyzowana instalacja bibliotek.
5. (...)
6. Dezaktywacja środowiska.