-

Przetwarzanie danych i odkrywanie wiedzy

Na licencji Pixab

L3: Narzędzia zarządzania eksperymentami

W ramach materiałów zajmiemy się zagadnieniem analizy nacechowania wypowiedzi, dla którego zbudujemy potok przetwarzania dla tego zadania oraz wykorzystamy narzędzia do zarządzania eksperymentami: DVC oraz MLFlow. Na potrzeby laboratorium tym razem wykorzystamy zbiór `clarin-pl/polemo2`, który jest również udostępniony na repozytorium `huggingface`. Zbiór `polemo2` jest przeznaczony do analizy nacechowania wypowiedzi zawiera 8216 recenzji napisanych w języku polskim pochodzących z różnych domen: `hotels` (tripadvisor.com), `medicine` (znanylekarz.com), `school` (polwro.pl), `products` (ceneo.pl). Do rencenzji przypisano cztery klasy nacechowania wypowiedzi:

- zero - neutralna
- minus - negatywna
- plus - pozytywna
- amb - nieokreślona

DVC

DVC można najkrócej opisać jako system kontroli wersji dla danych lub jako `git` dla danych i modeli. Jest to narzędzie o otwartych źródłach, które pozwala nam m.in.:

- wersjonować dane i modele (do śledzenia wersji plików lub folderów dvc wykorzystuje sumy kontrolne `md5`)
- zarządzać potokami przetwarzania i zapewnia interfejs do śledzenia metryk
- przesyłać dane na zewnetrzne serwery oraz zarządzać nimi lokalnie

> **Important**
>
> DVC jest dalej w trakcie rozwoju, dlatego poniższa instrukcja ogranicza się do zastosowania DVC w wersji 2.x. Nowsze wersje biblioteki np. 3.x, mogą wprowadzić duże zmiany do poszczególnych funkcjonalności. Najnowsza wersja dokumentacji jest dostępna pod adresem https://dvc.org/doc

Instalacja

DVC możemy najprościej zainstalować poprzez menadżera paczek pip

```
pip install dvc
```

Dodatkowo do obsługi zewnętrzych serwerów wymaga zainstalowania dodatkowych zależności. Wspierane konfiguracje to `s3, gs, azure, oss, ssh` lub `all` - instaluje wszystkie opcjonalne zależności. Jeżeli chcemy zainstalować dodatkowe zależności np. dla s3, polecenie przyjmie następującą postać:

```
pip install dvc[s3]
```

W ramach materiałów nie będziemy wykorzystywać zewnętrznego serwera, dlatego wystarczy nam podstawowa wersja.

> **Important**
>
> Poniższa instrukcja dotyczy wykorzystania DVC w wersji 2.0 i została przetestowana z wykorzystaniem wersji 2.1.0

Inicjalizacja repozytorium dvc

Inicjalizacja repozytorium dvc występuję z wykorzystaniem komendy `dvc init`, przechodząc do głównego katalogu projektu.

```
$ dvc init

Initialized DVC repository.

You can now commit the changes to git.

+---------------------------------------------------------------------+
|                                                                     |
|        DVC has enabled anonymous aggregate usage analytics.         |
|     Read the analytics documentation (and how to opt-out) here:     |
|             <https://dvc.org/doc/user-guide/analytics>              |
|                                                                     |
+---------------------------------------------------------------------+

What's next?
------------
- Check out the documentation: <https://dvc.org/doc>
- Get help and share ideas: <https://dvc.org/chat>
- Star us on GitHub: <https://github.com/iterative/dvc>
```

Contents

Print to PDF

> **Important**
>
> `dvc init` domyślnie wykonuje inicjalizację w głównym folderze, jeżeli chcemy umieścić go poza głównym folderem musimy skorzystać z flagi `--subdir`, jednak zazwyczaj nie jest to potrzebne. Pełną listę konfiguracji komendy `dvc init` znajdziemy w dokumentacji. LINK

Sprawdźmy, jakie pliki są śledzone przez `git`.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .dvc/.gitignore
        new file:   .dvc/config
        new file:   .dvc/plots/confusion.json
        new file:   .dvc/plots/confusion_normalized.json
        new file:   .dvc/plots/default.json
        new file:   .dvc/plots/linear.json
        new file:   .dvc/plots/scatter.json
        new file:   .dvc/plots/smooth.json
        new file:   .dvcignore
```

Pliki trackowane przez repozytorium git

- `.dvc/.gitignore` - Deklaracja plików/folderów, które mają nie być trackowane przez gita
- `.dvc/config` - Konfiguracja dvc, tutaj umieszczana jest np. konfiguracja zewnętrznego serwera
- `.dvc/plots/*.json` - Szablony dla wykresów udostępnianych przez dvc. Więcej informacji: [LINK]
- `.dvcignore` - Deklaracja plików/folderów, która mają nie być trackowane przez dvc

Pliki nietrackowane przez git:

- `.dvc/cache/` - Folder cache, tutaj zawierają się lokalne pliki repozytorium. Więcej informacji: [LINK]
- `.dvc/tmp/` - Pliki tymczasowe

Szczegółowy opis plików znajduje się pod adresem: [LINK]

Dodanie pierwszego etapu potoku przetwarzania do DVC

Przygotowanie skryptu

Najpierw zaczniemy od przygotowania skryptu do pobrania zbioru danych. Konfigurację dla tego skryptu umieścimy w zewnętrzntym pliku `params.yaml` w głównym katalogu. Parametrami skryptu będą domeny dla danego podziału danych (train, dev i test) oraz typ tekstów (zdania lub pełne recenzje). Działanie tego skryptu będzie polegać na pobraniu odpowiedniej konfiguracji zbioru danych z repozytorium i jego transformacja go do prostszego formatu. Przetworzony plik umieśćmy w folderze `data/`

`scripts/download_data.py`

```
import pickle

import datasets
import yaml


def main():
    with open("params.yaml", "r") as f:
        cfg = yaml.safe_load(f)

    dataset = datasets.load_dataset(
        "clarin-pl/polemo2-official", **cfg["download_data"]
    )

    dataset = {
        "train": (dataset["train"]["text"], dataset["train"]["target"]),
        "validation": (dataset["validation"]["text"], dataset["validation"]["target"]),
        "test": (dataset["test"]["text"], dataset["test"]["target"]),
    }

    with open("data/dataset.pkl", "wb") as f:
        pickle.dump(obj=dataset, file=f)


main()
```

Jako domyślną konfigurację wykorzystajmy recenzje pochodzące z hoteli i zdania.

`params.yaml`

```
download_data:
  train_domains:
    - hotels
  dev_domains:
    - hotels
  test_domains:
    - hotels
  text_cfg: sentence
```

Dodanie skryptu do DVC

Etapy przetwarzania dodajemy do potoku za pomocą komendy `dvc run` [LINK] lub `dvc stage add` [LINK].

Głównymi elementami danego etapu definiowanego w dvc są:

- zależności etapu (dane wejściowe, skrypty pythonowe etc.), definiowane za pomocą parametru `-d`
- parametry (w celu śledzenia zmian), definiowane za pomocą parametru `-p`
- pliki wyjściowe, definiowane za pomocą parametru `-o`
- skrypt który będziemy wykonywać podawany na końcu polecania w naszym przypadku będzie to `python3 scripts/download_data.py`

Wróćmy do naszego skryptu pobierającego zbiór danych. Nazwę etapu przypisujemy za pomocą parametru `-n`.

> **Important**
>
> Etapy w dvc podczas dodawania za pomocą `dvc run` są automatycznie wykonywane, aby zmienić to zachowanie musimy dodać flagę `--no-exec` lub możemy wykorzystać polecenie `dvc stage add`

Zdefiniujmy teraz elementy etapu:

- Zależności:
  - Skrypt `scripts/download_data.py`
- Parametry:
  - train_domains
  - test_domains
  - dev_domains
  - text_cfg
- Wyjście:
  - Plik `data/dataset.pkl`
- Skrypt do wykonania: `PYTHONPATH=. python3 scripts/download_data.py`

Zbierzmy teraz wszystko w ramach polecenia i wykonajmy je.

```
$ dvc run  \
    -n download_data \
    -d scripts/download_data.py \
    -o data/dataset.pkl \
    -p download_data.train_domains \
    -p download_data.dev_domains \
    -p download_data.test_domains \
    -p download_data.text_cfg \
    --no-exec \
    PYTHONPATH=. python3 scripts/download_data.py

Creating 'dvc.yaml'
Adding stage 'download_data' in 'dvc.yaml'

To track the changes with git, run:

        git add data/.gitignore dvc.yaml
```

> **Important**
>
> Przy pracy z repozytorium DVC szczególną uwagę należy zwrócić na pliki `.gitignore` generowane podczas dodawania następnych etapów. Pozwala to uniknąć dodania nieporządanych plików do repozytorium git.

> **Important**
>
> Plik `params.yaml` umieszczony w głównym katalogu repozytorium lub w odpowiednim podfolderze, jeżeli dvc było inicjalizowane z flagą –subdir jest domyślną ścieżką do definicji parametrów w dvc. Pozwala to na odwoływanie się do parametrów umieszczonych w tym pliku bezpośrednio. Jeżeli nasza konfiguracja byłaby w innej lokalizacji np. w `configs/download_data.yaml` musielibyśmy ją podać. Przykładowa dekleracja parametru miałaby wtedy postać `-p configs/download_data.yaml:train_domains`

Wszystkie elementy potoku przetwarzania są definiowane w pliku `dvc.yaml`. Spójrzmy na jego sygnaturę w tym momencie.

`dvc.yaml`

```yaml
stages:
  download_data:
    cmd: PYTHONPATH=. python3 scripts/download_data.py
    deps:
    - scripts/download_data.py
    params:
    - download_data.dev_domains
    - download_data.test_domains
    - download_data.text_cfg
    - download_data.train_domains
    outs:
    - data/dataset.pkl
```

> **Hint**
>
> Kolejne etapy możemy również dodawać lub edytować bezpośrednio modyfikąc plik `dvc.yaml`. Jednak wtedy musimy sami zadbać o poprawność definicji oraz o zarządzanie plikami `.gitignore

Sprawdzenie statusu potoku

Stan potoku możemy sprawdzić za pomocą polecenia `dvc status`. Daje nam to pogląd na zmiany, które występują w naszym potoku.

```
$ dvc status

download_data:
        changed deps:
                modified:           scripts/download_data.py
                params.yaml:
                        new:                download_data.dev_domains
                        new:                download_data.test_domains
                        new:                download_data.text_cfg
                        new:                download_data.train_domains
        changed outs:
                deleted:            data/dataset.pkl
```

W tym przypadku `dvc status` pokazuje nam w naszym etapie `download_data` tutaj zmiany zarówno w zależnościach, parametrach jak i w pliku wyjściowym.

Wykonywanie etapów

Do wykonania etapu lub etapów służy polecenie `dvc repro`. Domyślna konfiguracja polecenia `dvc repro` oznacza przejrzenie całego potoku i wykonanie tylko tych etapów w których wykryto zmiany. Możemy również wykonać jeden etap lub wybraną część potoku przekazując nazwy etapów. Pełna dokumentacja: [LINK]

Przydatne parametry:

- `--dry` - pozwala podejrzeć które etapy mają byc wykonane
- `-f` - wymuszenie wykonania wszystkich lub wybranych potoków, nawet jeżeli `dvc` nie reportuje zmian
- `-s` - wykonanie pojedynczego etapu z pominięciem poprzednich, nawet jeżeli `dvc` reportuje zmiany

Wykonujmy teraz reprodukcję potoku

```
$ dvc repro

Running stage 'download_data':
> PYTHONPATH=. python3 scripts/download_data.py
Using custom data configuration default-d3f574c876938804
Reusing dataset pol_emo2
Generating lock file 'dvc.lock'
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.lock
Use `dvc push` to send your updates to remote storage.
```

Stan potoku jest zapisywany w pliku dvc.lock

```
schema: '2.0'
stages:
  download_data:
    cmd: PYTHONPATH=. python3 scripts/download_data.py
    deps:
    - path: scripts/download_data.py
      md5: 79bddadefd2d6422e47f5b9190b1e26e
      size: 585
    params:
      params.yaml:
        download_data.dev_domains:
        - hotels
        download_data.test_domains:
        - hotels
        download_data.text_cfg: sentence
        download_data.train_domains:
        - hotels
    outs:
    - path: data/dataset.pkl
      md5: 5bd49b39290147a1cfd193b4356890d0
      size: 2524424
```

Wysyłanie i pobieranie danych z dvc

> **Important**
>
> Wymaga konfiguracji serwera zewnętrznego. Więcej szczegółów na temat konfiguracji serwera: [LINK]

Pliki śledzone przez dvc takie jak np. pliki wyjściowe możemy przesyłać na zewnętrzny serwer za pomocą polecenia `dvc push`, a pobierać lokalnie za pomocą komendy `dvc pull`. Podobnie jak w przypadku `dvc repro` zakres operacji `push` i `pull` obejmuje domyślnie cały potok lub jego wybraną część poprzez przekazanie nazw(y) etapów jako argumentu polecenia.

Kolejne etapy przetwarzania

Dodajmy teraz kolejne etapy przetwarzania extract_features i evaluate_model. Odpowiednie parametry zostały dodane do pliku `params.yaml`.

`extract_features` dodamy podobnie jak wcześniej `download_data`

```
$ dvc run  \
    -n extract_features \
    -d scripts/extract_features.py \
    -d data/dataset.pkl \
    -o data/featurized.pkl \
    --no-exec \
    PYTHONPATH=. python3 scripts/extract_features.py

Adding stage 'extract_features' in 'dvc.yaml'

To track the changes with git, run:

        git add dvc.yaml data/.gitignore
```

W przypadku `evaluate_model` dodamy mechanizm śledzenie metryk. Pełny opis jest dostępny pod adresem [LINK]. Metryki specyfikuje się za pomocą flagi `-M`. Obecnie wspierany jest format `json`, `toml` lub `yaml`.

```
$ dvc run  \
    -n evaluate_model \
    -d scripts/evaluate_model.py \
    -d data/featurized.pkl \
    -M data/results.json \
    -p evaluate_model.random_state \
    -p evaluate_model.max_iter \
    --no-exec \
    PYTHONPATH=. python3 scripts/evaluate_model.py

Adding stage 'evaluate_model' in 'dvc.yaml'

To track the changes with git, run:

        git add dvc.yaml
```

Zreprodukujmy teraz cały potok

```
$ dvc repro

Stage 'download_data' didn't change, skipping
Running stage 'extract_features':
> PYTHONPATH=. python3 scripts/extract_features.py
Updating lock file 'dvc.lock'

Running stage 'evaluate_model':
> PYTHONPATH=. python3 scripts/evaluate_model.py
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.lock
Use `dvc push` to send your updates to remote storage.
```

Przeglądanie potoku przetwarzania, parametrów i metryk

Mając już zdefiniowany potok przetwarzania możemy go zwizualizować w konsoli za pomocą komendy `dvc dag` [LINK]

```
$ dvc dag

  +---------------+
  | download_data |
  +---------------+
          *
          *
          *
+------------------+
| extract_features |
+------------------+
          *
          *
          *
 +----------------+
 | evaluate_model |
 +----------------+
```

Możemy również śledzić graph zależności pomiędzy wyjściami za pomocą flagi `--outs`

```
$ dvc dag --outs

  +------------------+
  | data/dataset.pkl |
  +------------------+
            *
            *
            *
+---------------------+
| data/featurized.pkl |
+---------------------+
            *
            *
            *
 +-------------------+
 | data/results.json |
 +-------------------+
```

Listę etapów możemy podejrzeć za pomocą polecenia `dvc stage list` [LINK]

```
$ dvc stage list
download_data     Outputs data/dataset.pkl
extract_features  Outputs data/featurized.pkl
evaluate_model    Reports data/results.json
```

Metryki możemy teraz wyświetlić bezpośrednio z konsoli za pomocą komendy `dvc metrics show` [LINK]

```
$ dvc metrics show
Path               accuracy    f1-score
data/results.json  0.74165     0.67661
```

Parametry możemy wyświetlić za pomocą `dvc params diff --all` [LINK]

```
$ dvc params diff --all

Path         Param                        Old         New
params.yaml  download_data.dev_domains    ['hotels']  ['hotels']
params.yaml  download_data.test_domains   ['hotels']  ['hotels']
params.yaml  download_data.text_cfg       sentence    sentence
params.yaml  download_data.train_domains  ['hotels']  ['hotels']
params.yaml  evaluate_model.max_iter      200         200
params.yaml  evaluate_model.random_state  441         441
```

Wprowadzanie tymczasowych zmian

Czasami potrzebujemy wprowadzić szybkie zmiany, ale niekoniecznie chcemy je przesłać później na repozytorium. W celu ćwiczenia zmieńmy parametry `download_data.text_cfg` na `text` oraz `evaluate_model.max_iter` na `50`.

Zmiany możemy sprawdzić za również za pomocą `dvc params diff`.

```
$ dvc params diff

Path         Param                    Old       New
params.yaml  download_data.text_cfg   sentence  text
params.yaml  evaluate_model.max_iter  200       50
```

Zamiast wszystkich elementów, polecenie teraz wyświetla nam tylko te w których dokonana była zmiana.

Parametry się zgadzają, dokonajmy reprodukcji całego potoku za pomocą `dvc repro`. Zmiany wymagają wykonania całego potoku od nowa.

Po wykonaniu się wszystkich elementu potoku, możemy sprawdzić zmiany.ożemy sprawdzić jak zmieniły się metryki do tego wykorzystamy polecenia `dvc metrics diff`

```
$ dvc metrics diff

Path               Metric    Old      New      Change
data/results.json  accuracy  0.74165  0.81013  0.06848
data/results.json  f1-score  0.67661  0.80187  0.12526
```

Czas na cofnięcie zmian, zacznijmy od odrzucenia zmian w repozytorium git, zmienione pliki to `results.json`, `params.yaml` i `dvc.lock`.

```
$ git restore .
```

Teraz zostaje nam cofnięcie zmian w plikach `dvc` do tego wykorzystamy komendę `checkout`, która działa w podobny sposób jak `git checkout`. Przydatna jest również w sytuacji, kiedy przełączamy się pomiędzy różnymi branchami.

```
$ dvc checkout
```

Inne przydatne komendy DVC

- dvc add - Pozwala na śledzenie pliku / folderu poprzez utworzenie pliku `{nazwa_pliku}.dvc`.
  Przykład

```
$ dvc add data/
```

`data.dvc`

```
outs:
- md5: 48c6715169a5ed3b78e278f5f2c6055e.dir
  size: 57675468
  nfiles: 9
  path: data
```

- dvc commit - Pozwala uaktalnić stan danego etapu. `dvc commit` jest szczególnie przydatny w sytuacjach:
  - Kiedy wprowadziliśmy zmiany w zależnościach etapu, które nie mają wpływu na pliki wyjściowe etapu
  - Kiedy wykonaliśmy skrypt danego etapu z pominięciem polecenia `dvc repro`
- dvc remove - Usuwa dany etap

Kod do części I

Rozwiązanie części I znajduje się pod adresem: [LINK]

MLFlow

MLFlow jest również dedykowanym narzędziem do zarządzania eksperymentami. Jego funkcjonalnosc obejmuje następujące moduły:

- `MLFlow Tracking` - zarządzanie eksperymentami, logowanie parametrych, metryk, wykresów i innych artefaktów

- `MLFlow Projects` - przygotowanie środowiska eksperymentalnego dla projektu, wykonywanie eksperymentów
- `MLFlow Deployment` - Wdrażanie modeli
- `MLFlow Registry` - Rejestr modeli

Dokumentacja: [LINK]

W ramach zajęć skupimy się na module `MLFlow Tracking`

Instalacja

Biblioteka mlflow również możemy zainstalować za pomocą menadżera paczek

```
pip install mlflow
```

MLFlow Tracking

Moduł jest oparty o monitorowanie przebiegów uczenia (ang. runs), które opcjonalnie możemy grupować w eksperymenty.

W ramach pojedynczego przebiegu zbierane są takie komponenty jak:

- Wersja kodu (aktualny hash commitu z gita)
- Czas rozpoczęcia i zakończenia przebiegu
- Źródło uruchomienia np. nazwa skryptu do uczenia
- Parametry
- Metryki
- Artefakty - pliki wyjściowe w dowolnym formacie, można logować obrazy, wykresy, modele itp.

Do logowania eksperymenty potrzebujemy zdefiniowania dwóch magazynów danych:

- `backend store` - przechowuje dane dotyczące ekspetymentów (przebiegi, parametry, metryki, tagi, metadane itp.)
  Wspierane typy magazynów danych:
  - lokalna ścieżka
  - baza danych: mysql, mssql, sqlite lub postgresql
- `artifact store` - przechowuje artefakty (pliki, modele, wykresy, obrazy itp.) Wspierane typy magazynów danych: lokalna ścieżka, Amazon S3, Azure Blob Storage, Google Cloud Storage, serwer FTP i SFTP, NFS i HDFS

Więcej informacji na temat magazynów danych mlflow: [LINK]

Ustawienia potrzebne do relizacji laboratorium

W ramach laboratorium wykorzystamy baze danych sqllite, a artefakty będziemy logować do folderu `artifacts`

Uruchomienie serwera do śledzenia przebiegów uczenia

Pierwszym krokiem, który musimy zdefiniować jest uruchomienie serwera, który będzie nam obsługiwał proces śledzenia przebiegów uczenia. Do tego wykorzystamy komendę `mlflow server`. Aplikacja domyślnie uruchamia się na porcie 5000. Szczegółowa dokumentacja: [LINK]

```
$ mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./artifacts --host 0.0.0.0
```

Następnym krokiem jest przekazanie adresu serwisu do śledzenia do biblioteki mlflow. Możemy umieścić bezpośrednie odwołanie w kodzie:

```
mlflow.set_tracking_uri("http://localhost:5000")
```

lub za pomocą zmiennej środowiskowej

```
MLFLOW_TRACKING_URI=http://localhost:5000
```

Monitorowanie przebiegów uczenia

Nowy przebieg tworzymy za pomocą metody `mlflow.start_run()`. Najprościej jest umieścić go w ramach bloku kodu `with`.

```
with mlflow.start_run():
    ...
```

Automatyczne logowanie

MLflow dostarcza moduły, które pozwolają automatycznie logować parametry, sam model oraz metryki, odbywa się to za pomocą metody `autolog()`. Metoda `autolog()` musi być wywołana przed rozpoczęciem procesu uczenia.

> **Important**
>
> Funkcjanolność związana z autologowaniem za pomocą metody autolog() jest w fazie eksperymentalnej i zależy od wykorzystywanej biblioteki.

Autolog obecnie nie wspiera automatycznego wykrywania modelu, musimy zaimportować odpowiedni moduł przeznaczony dla naszego modelu. W naszym przypadku będzie to sklearn. `mlflow.sklearn.autolog()`.

Szczegóły dotyczące autologowania i lista obecnie wspieranych bibliotek znajduję się pod linkiem: [LINK]

Autolog w przypadku `sklearn` loguje metryki wyłącznie dla danych uczących dla danych testowych lub walidacyjnych również to wykonać. Do tego wykorzystamy funkcję `mlflow.sklearn.eval_and_log_metrics`

Ręczne logowanie

- Logowanie parametrów: `mlflow.log_param` lub `mlflow.log_params`
  Przykłady:
  - `mlflow.log_param("train_domains", cfg["download_data"]["train_domains"])`
  - `mlflow.log_params(cfg["download_data"])`
- Logowanie metryk `mlflow.log_metric` lub `mlflow.log_metrics`
  Przykłady:
  - `mlflow.log_metric("accuracy", metrics["accuracy"])`
  - `mlflow.log_metrics(metrics=metrics)`
- Logowanie wykresów `mlflow.log_figure` Przykłady:
  - `mlflow.log_figure(fig, artifact_file="metrics.png")`
- Logowanie modeli `mlflow.{moduł_wykorzystywanej_biblioteki}.log_model` Przykłady:
  - `mlflow.sklearn.log_model(clf, "model", registered_model_name="LogisticRegression")`

Dodanie MLFlow do skryptu

Dodajmy teraz obsługę mlflow do wcześniej przygotowane skryptu `evaluate_model`.

Elementy, które będziemy logować:

- autologowanie
- logowanie parametru `train_domains` i `test_domains`
- logowanie metryk `accuracy` i macro `f1-score`
- logowanie wykresu przedstawiającego metryki `precision`, `recall`, `f1-score` per klasa

Bazowa wersja skryptu `evaluate_model`

```
import json
import pickle

import yaml
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report


def main():
    with open("data/featurized.pkl", "rb") as f:
        dataset = pickle.load(f)

    with open("params.yaml", "r") as f:
        cfg = yaml.safe_load(f)

    clf = LogisticRegression(**cfg["evaluate_model"])
    clf.fit(dataset["train"]["X"], dataset["train"]["y"])

    y_pred = clf.predict(dataset["test"]["X"])

    report = classification_report(
        y_pred=y_pred, y_true=dataset["test"]["y"], output_dict=True
    )

    metrics = {
        "accuracy": report["accuracy"],
        "f1-score": report["macro avg"]["f1-score"],
    }
    with open("data/results.json", "w") as f:
        json.dump(obj=metrics, fp=f)


main()
```

Kod po dodaniu obsługi mlflow

```
import json
import pickle
+ from typing import Dict, Any, List

+ import matplotlib.pyplot as plt
+ import mlflow
+ import pandas as pd
+ import seaborn as sns
import yaml
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report


+ def plot_metrics_per_class(report: Dict[str, Any], labels: List[str]):
+     report = pd.DataFrame(
+         [
+             {
+                 "class": labels[class_id],
+                 "metric": metric,
+                 "value": report[str(class_id)][metric],
+             }
+             for class_id in range(len(labels))
+             for metric in ["precision", "recall", "f1-score"]
+         ]
+     )
+
+     sns.barplot(data=report, x="class", y="value", hue="metric")
+
+     plt.title("Metrics per class")
+     fig = plt.gcf()
+     return fig


def main():
    with open("data/featurized.pkl", "rb") as f:
        dataset = pickle.load(f)

    with open("params.yaml", "r") as f:
        cfg = yaml.safe_load(f)

+    mlflow.set_tracking_uri("http://localhost:5100")

+    with mlflow.start_run():
+        mlflow.sklearn.autolog()

        clf = LogisticRegression(**cfg["evaluate_model"])
        clf.fit(dataset["train"]["X"], dataset["train"]["y"])

        y_pred = clf.predict(dataset["test"]["X"])

        report = classification_report(
            y_pred=y_pred,
            y_true=dataset["test"]["y"],
            output_dict=True,
        )
        metrics = {
            "accuracy": report["accuracy"],
            "f1-score": report["macro avg"]["f1-score"],
        }

+        fig = plot_metrics_per_class(report, labels=dataset["labels"])
+        mlflow.log_params(cfg["download_data"])
+        mlflow.log_metrics(metrics=metrics)
+        mlflow.sklearn.eval_and_log_metrics(
+            clf, X=dataset["test"]["X"], y_true=dataset["test"]["y"], prefix="test_"
+        )
+        mlflow.log_figure(fig, artifact_file="metrics.png")

    with open("data/results.json", "w") as f:
        json.dump(obj=metrics, fp=f)


main()
```

Kod do części II

Pełny kod pokrywający część II znajduje się pod adresem: [LINK]

Eksperymenty w DVC 2.0

Ostatnim omawianym elementem jest definiowanie eksperymentów w ramach DVC w wersji 2.0 Eksperymenty oparte są o moduł `dvc exp` pozwalają na łatwą zmianę konfiguracji potoku przetwarzania oraz umożliwiają definicję kilku konfiguracji. Aktualny opis interfejsu `experiments` znajduje się pod adresem [LINK]

Eksperymenty są uruchamiane za pomocą komendy `dvc exp run` [LINK] zmienione parametry podajemy za pomocą flagi `-S`.

Zmieńmy teraz liczbę iteracji na 50.

```
$ dvc exp run -S evaluate_model.max_iter=50

...

Reproduced experiment(s): exp-b605e
Experiment results have been applied to your workspace.

To promote an experiment to a Git branch run:

        dvc exp branch <exp>
```

Teraz możemy podejrzeć zmainę metryk za pomocą `dvc exp diff`

```
$ dvc exp diff

Path               Metric    Value    Change
data/results.json  accuracy  0.73964  -0.0020121
data/results.json  f1-score  0.66816  -0.0084543

Path         Param                    Value    Change
params.yaml  evaluate_model.max_iter  50       -150
```

Alternatywnie możemy zdefiniować kolejkę eksperymentów i dopiero później je wykonać. Dodajmy teraz inne konfigurację. W celu zakolejkowania eksperymentu wykorzystujemy flagę `--queue`.

```
$ dvc exp run --queue -S evaluate_model.max_iter=300 -S download_data.train_domains="[hotels,medicine]"
Queued experiment 'dd51be5' for future execution.
$ dvc exp run --queue -S evaluate_model.max_iter=300 -S download_data.train_domains="[medicine,products]"
Queued experiment '9887d0f' for future execution.
$ dvc exp run --queue -S evaluate_model.max_iter=300 -S download_data.train_domains="[all]"
Queued experiment 'db4350f' for future execution.
```

Uruchumienie eksperymentów

```
$ dvc exp run --run-all
```

Podsumowanie wyników eksperymentów możemy uzyskać za pomocą polecenia `dvc exp show` [LINK]

```
$ dvc exp show
```

Możemy ograniczyć listę wyświetlanych kolumn przez polecenie `dvc exp show` poprzez zignorowanie czasu wykonania eksperymentu (flaga `--no-timestamp`), wyspecifikowanie parametrów na których nam zależy (flaga `--include-params`)

```
$ dvc exp show --no-timestamp --include-params download_data.train_domains,evaluate_model.max_iter

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Experiment                ┃ accuracy ┃ f1-score ┃ download_data.train_domains ┃ evaluate_model.max_iter ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━┩
│ workspace                 │ 0.74165  │ 0.67661  │ ['hotels']                  │ 200                     │
│ main                      │ 0.74165  │ 0.67661  │ ['hotels']                  │ 200                     │
│ ├── 3080e90 [exp-58a4b]   │ 0.72716  │ 0.66199  │ ['hotels', 'medicine']      │ 300                     │
│ ├── 6846c4f [exp-410e9]   │ 0.57223  │ 0.51545  │ ['medicine', 'products']    │ 300                     │
│ └── 72921d1 [exp-dd6cb]   │ 0.73119  │ 0.6675   │ ['all']                     │ 300                     │
└───────────────────────────┴──────────┴──────────┴─────────────────────────────┴─────────────────────────┘
```

Wyświetlanie listy eksperymentów

```
$ dvc exp list
main:
        exp-58a4b
        exp-410e9
        exp-dd6cb
```

Możemy teraz wybrać najlepszy eksperyment, akurat w tym przypadku żaden z przeprowadzonych eksperymentów nie okazał się lepszy od konfiguracji zastosowanej na głównym branchu. Więc pozostawimy bez zmian. Ale jeżeli chcielibyśmy zaaktualizować potok na bazie któregoś z przeprowadzonych elementów możemy wykorzystać komendę `dvc exp apply` podając identyfikator eksperymentu

Do usunięcia eksperymentu służy komenda `dvc remove`, możemy również usunąć stare eksperymenty za pomocą `dvc exp gc --workspace`, dodatkowo aby wyczyścić pamięc podręczną musimy również wykonać polecenie `dvc gc`

By Tomasz Kajdanowicz, Kamil Tagowski, Krzysztof Rajda, Albert Sawczyn, Piotr Bielak