


ML Pipeline

Гущин Александр
DMIA Production ML  весна 2021

Как вывести модель в продакшн?

Обычно, DS обучают модель у себя на ноутбуке, затем сами или с помощью разработчиков встраивают её в текущие системы.

Проблема: обученная локально модель может оказаться невоспроизводимой

Решение: давайте завернем скрипты по подготовке данных и обучению модели в докер и зафиксируем окружение. Это обычно и называют **ML Pipeline**.

Другие пайплайны, с которыми мы имеем дело:

- Расчет прогнозов в оффлайн, batch inference (тоже ML pipeline)
- Шаги в CI/CD (запуск ML pipeline - часть CI/CD)
- Расчет сложных фичей для ускорения online inference

Будем думать о пайплайне как о наборе скриптов и зафиксированном окружении - для простоты и по причине распространенности такого подхода. Например, пайплайн в нашем курсе может выглядеть так

```
.
├── Dockerfile
├── Pipfile
├── Pipfile.lock
├── config.yaml
├── scripts
│   ├── get_data.py
│   ├── generate_features.py
│   ├── split_train_val.py
│   ├── train.py
│   ├── validate.py
│   └── save_results.py
├── run_all.sh
└── README.md
```

А его запуск выполняться так

```
1 | ~ bash ./run_all.sh
```

Наша задача - заставить условный `run_all.sh` работать в проде на регулярной основе. Это достигается в два этапа:

1. Переезжаем из ноутбуков в скрипты (из Jupyter в Pycharm/VScode)
2. Запускаем скрипты на удаленной машине с помощью некоторой системы оркестрации (Circle CI, Airflow, Kubeflow, или даже Cron)

Чтобы быть воспроизводимым и стабильно работающим, такой пайплайн должен

1. не содержать рандома (заданные random seed, детерминированные вычисления)
2. иметь зафиксированную среду исполнения (используем контейнеры) и окружение (фиксируем зависимости и версии библиотек).

В нашем курсе мы будем выполнять этот пайплайн в рамках шага CI/CD на Gitlab.

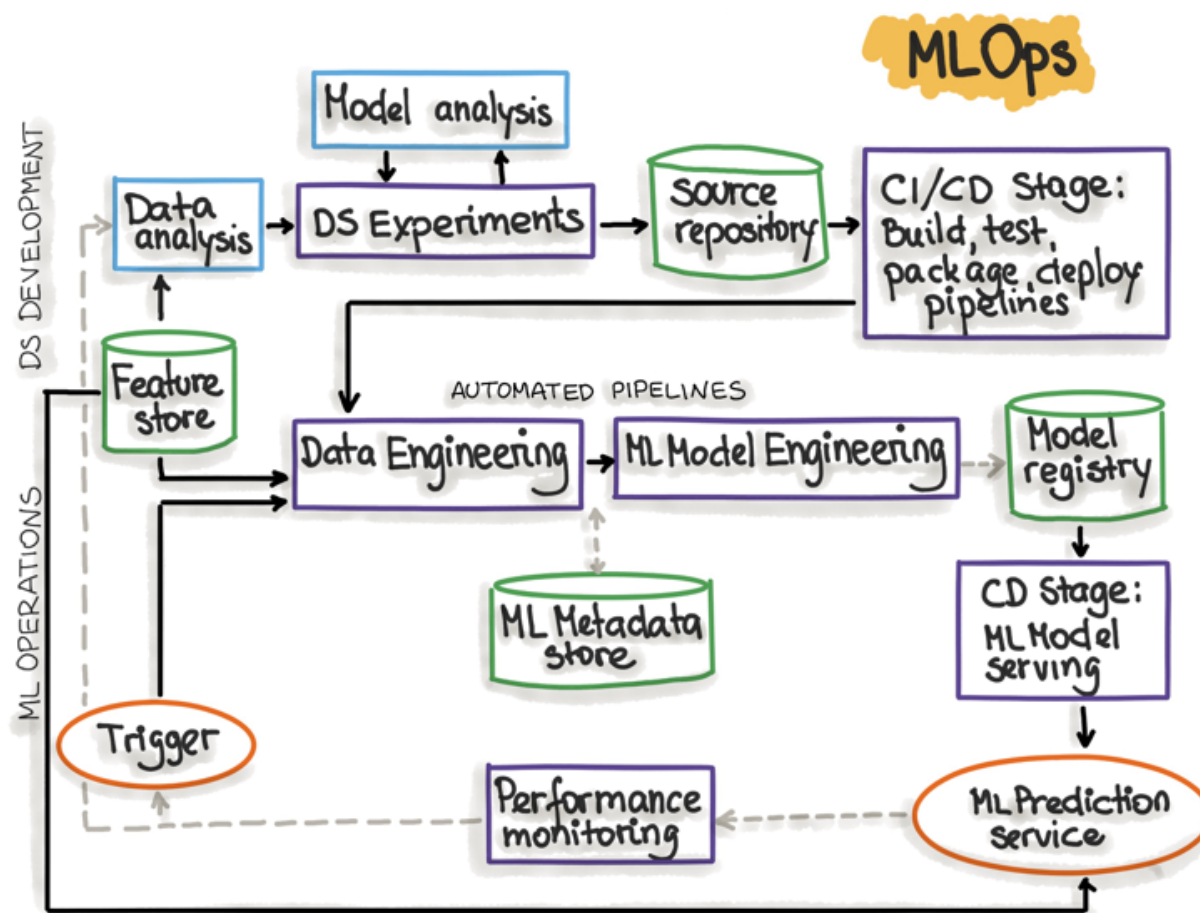
Зачем?

Зачем создавать ML Pipeline и настраивать его регулярный запуск?

Почему не обойтись без этого и не запускать всё вручную?

1. Непроизвольные ошибки и невоспроизводимость
2. Overhead на регулярные переобучения - модель будет устаревать
3. Зависимость от конкретных людей (bus factor)

Из каких этапов должен состоять пайплайн?



ml-ops.org

*Continuous Training

Step	ML Pipeline	ML API	Batch inference
Data extraction & validation & splitting	✓		✓
Feature generation	✓	✓	✓
Model training	✓		
Model inference	✓	✓	✓
Model evaluation & validation	✓		

Отличие кода при обучении и применении - источник технических ошибок. Чтобы избежать этого этапы генерации признаков и инференса модели можно выделять в отдельную переиспользуемую компоненту[ы].

Написание ML Pipeline

Для написания нашего пайплайна мы можем воспользоваться одним из нескольких подходов:

- Процедурное программирование
- OOP

Каждый подход имеет свои сильные и слабые стороны.

Процедурное программирование

```
1 PARAMETERS = {...}
2
3 def count_vectorize_data(raw_data, params):
4     ...
5     return vectorized_data
6
7 def tfidf_transform_data(vectorized_data, params):
8     ...
9     return transformed_data
10
11 def train_sgd_classifier(train_data, params):
12     ...
13     return model
14
15 vectorized_data = count_vectorize_data(raw_data, PARAMETERS['vect'])
16 transformed_data = tfidf_transform_data(vectorized_data, PARAMETERS['tfidf'])
17 model = train_sgd_classifier(transformed_data, PARAMETERS['model'])
```

Связывать эти кусочки необязательно на Python. Часто это делают на bash (как в нашем примере выше).

OOP

```
1 from sklearn.feature_extraction.text import CountVectorizer
2 from sklearn.feature_extraction.text import TfidfTransformer
3 from sklearn.linear_model import SGDClassifier
4 from sklearn.pipeline import Pipeline
5
6 pipeline = Pipeline([
7     ('vect', CountVectorizer()),
8     ('tfidf', TfidfTransformer()),
9     ('clf', SGDClassifier()),
10 ])
11 pipeline.fit(X_train, y_train)
```

Здесь мы используем `sklearn.Pipeline`, но можем написать и свой класс с методами `fit`, `predict`, `process_data`, `serialize_model`, etc, etc.

Как выбрать подход

- Процедурное программирование
 - Разделение на этапы более наглядно
 - Проще добавлять новую логику
 - Если логика сложная, проще ошибиться
 - В некоторых ситуациях подход необходим - например, если пайплайн сложный и неоднородный и все собирается вместе скриптом на bash
- ООР
 - Легче осуществлять подбор оптимальных параметров и сериализовывать их
 - Реализовывается в рамках одного языка, сложно "подружить" разнородные компоненты
 - Легче писать тесты, более надежный подход к разработке, особенно в сложных пайплайнах

Системы оркестрации

- они могут требовать, чтобы мы оформили наш код как-то особенно, например
 - написали Python операторы, как в Airflow,
 - собрали разные этапы в разные контейнеры, как в Kubeflow,
 - явно указали создаваемые в процессе артефакты
- в разных компаниях и командах могут использоваться разные инструменты, что обычно продиктовано особенностями инфраструктуры и задач, например
 - когда Airflow уже широко применяют в компании для других задач (ETL), используют его
 - вместе с Kubernetes (особенно на GCP) используется Kubeflow
 - Где-то есть собственные инструменты, вроде Nirvana в Яндексе

Это значит, что на работе вы столкнетесь с одним из этих инструментов. Но основы для выстраивания пайплайнов одинаковы, и для их изучения нам не потребуется ни один из них. Наш ML Pipeline будет представлять из себя набор скриптов.

Этот пакет будет выполняться локально командой вроде `pipenv run run_pipeline.py`. Позже мы запакуем его в Docker контейнер и будем запускать как часть CI/CD на Gitlab.

Pipeline, API и пакеты

- Зачем вообще использовать здесь пакеты?
- Должен ли ML pipeline быть отделен от ML api? Почему бы не использовать **один** питоновский пакет?

Примеры:

- Один пакет (lstm_model) - наш репозиторий
- Несколько пакетов (пакеты ml_api, regression_model, neural_network_model) - <https://github.com/trainindata/deploying-machine-learning-models>

- Плюсы подхода “совсем без пакетов”:
 1. Удобен на начальном этапе - так быстрее
 2. Так проще, если у вас всего один проект
- Что при этом нужно иметь в виду:
 1. Для pipeline могут быть нужны зависимости, которые не нужны для api
 2. В разделенные pipeline и api более проще и безопаснее вносить изменения
 3. Если ваша команда поддерживает несколько API для разных моделей, их может быть проще объединить в один сервис
 4. Разделенные pipeline и api позволят сделать более гибкую конфигурацию - например, более “прицельный” запуск CI/CD

Саммари

1. Процесс, который обеспечивает переобучение модели, называют ML пайплайном. Тем же термином иногда называют процесс регулярного расчета прогнозов.
2. ML pipeline создают, чтобы получить стабильный во времени и как можно более независимый от человеческого вмешательства процесс. На практике это значит, что ML pipeline запускается на удаленном сервере под управлением системы оркестрации.
3. Существует два подхода к созданию пайплайнов - пайплайны в стиле OOP и в стиле процедурного программирования. Эти подходы имеют свои плюсы и минусы, а также часто используются вместе.

Семинар

1. Смотрим наш процедурный пайплайн, показываю, как это работает, запускаю его, показываю сохраненные артефакты. Смотрим без dvc и с dvc. Смотрим studio.
 2. Наш первый пайплайн по паролям - OOP
 3. Ещё один OOP https://github.com/trainindata/deploying-machine-learning-models/tree/master/packages/regression_model/regression_model
 4. Скажу, почему выбрали функциональный (пройду по \pm которые мы написали выше и объясню, как они проявляются для этого случая)
 5. Наконец, покажу, как это запускается в докере - <https://gitlab.com/production-ml/password-complexity/-/blob/master/Dockerfile>
 6. Напомню, что в конце концов мы запустим этот пайплайн в CI/CD, а в шляпе данные будут обновляться каждый день, так что слушателям нужно будет его закодировать.
- Выдам ДЗ