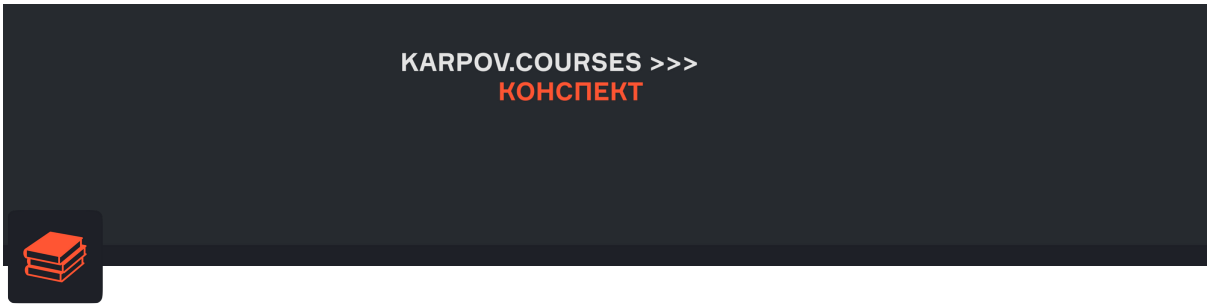


> Конспект > 11 урок > Airflow: Обзор платформы

> Оглавление

> Airflow.Зачем?

Проблема batch-запуска

Представим, что вы с командой довели ML-модель до рабочего состояния, увидели первые результаты в юпитер-ноутбуке (или на простых скриптах) и теперь хотите поставить на поток расчеты.

Мы уже умеет выставить API для доступа к модели через веб. Тем не менее, использование модели через API означает, что модель будет считаться по запросу, а не регулярно. Как же быть, если мы захотим считать предсказания каждый день?

Такой режим запуска называется batch-режимом.

Простое решение

Кажется, что можно решить вопрос в два этапа:

1. Сделать скрипт/обертку над ноутбуком, который можно запустить простой командой и результатом будут все предсказания.
2. Руками запускать скрипт каждый день.

Есть как минимум три проблемы. Во-первых, модель вы будете запускать, скорее всего, не на рабочем компьютере, а на сервере. Значит, вам придется каждый день подключаться к серверу, запускать скрипт и затем каким-то образом получать результаты с сервера. Во-вторых, моделей может стать много, и запускать каждую из них может быть утомительно (поверьте, через неделю вам это надоест). В-третьих, вы не имеете наблюдаемости - скрипт модели может упасть через час после того, как вы вручную его запустите, и вы узнаете об этом только на следующий день. Если, скажем, захочется иметь автоматический перезапуск - придется его реализовывать самостоятельно.

Готовые решения

Для решения проблемы с регулярного запуском существует несколько готовых инструментов. Вот два самых популярных:

- Luigi
- Airflow

В этом курсе мы остановимся на Airflow.

> Основные концепции

Airflow базируется на следующих принципах:

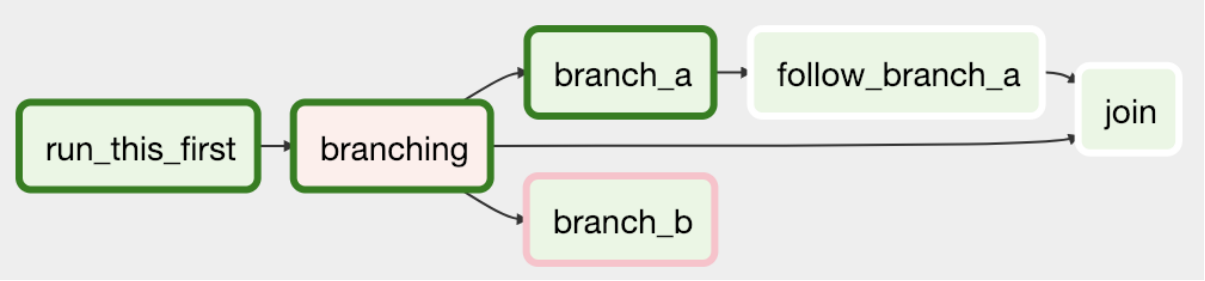
- Сложный процесс обработки данных разбивается на шаги.
- Из шагов выстраивается граф зависимостей - какие шаги выполнять первыми и кому чьи результаты нужны.
- Этот граф зависимостей должен быть без циклов. В терминологии Airflow он называется DAG (directed acyclic graph).
- DAG описывается программно из Python. То есть, весь граф можно динамически построить в цикле, а можно заранее прописать кодом.
- DAG имеет информацию по своем запуску:
- когда начать запускать
- с какой периодичностью
- сколько раз пытаться
- кому писать email в случае провала
- сколько раз пытаться перезапустить в случае провала
- когда начать запускать

- с какой периодичностью
- сколько раз пытаться
- кому писать email в случае провала
- сколько раз пытаться перезапустить в случае провала
- Система самостоятельно запускает DAG в соответствии с информацией по запуску.

Airflow поддерживает запуск не только Python-кода, но и shell скриптов, docker контейнеров - словом, может запускать много чего.

DAG

DAG хранит описание процесса обработки данных. Пример DAG ниже:



DAG может ветвиться, задачи в нем могут иметь условия (например, если сегодня выходной, то пропустить стадию из DAG).

Посмотрим на пример DAG из документации (не запустится в Jupyter):

```
"""
Test documentation
"""
from datetime import datetime, timedelta
from textwrap import dedent

# Для объявления DAG нужно импортировать класс из airflow
from airflow import DAG

# Операторы - это кирпичики DAG, они являются звеньями в графе
# Будем иногда называть операторы тасками (tasks)
from airflow.operators.bash import BashOperator
with DAG(
    'tutorial',
    # Параметры по умолчанию для тасок
    default_args={
        # Если прошлые запуски упали, надо ли ждать их успеха
        'depends_on_past': False,
        # Кому писать при провале
        'email': ['airflow@example.com'],
        # А писать ли вообще при провале?
        'email_on_failure': False,
        # Писать ли при автоматическом перезапуске по провалу
        'email_on_retry': False,
        # Сколько раз пытаться запустить, далее помечать как failed
        'retries': 1,
        # Сколько ждать между перезапусками
        'retry_delay': timedelta(minutes=5),  # timedelta из пакета datetime
    },
    # Описание DAG (не тасок, а самого DAG)
    description='A simple tutorial DAG',
```

```
    # Как часто запускать DAG
    schedule_interval=timedelta(days=1),
    # С какой даты начать запускать DAG
    # Каждый DAG "видит" свою "дату запуска"
    # это когда он предположительно должен был
    # запуститься. Не всегда совпадает с датой на вашем компьютере
    start_date=datetime(2022, 1, 1),
    # Запустить за старые даты относительно сегодня
    # https://airflow.apache.org/docs/apache-airflow/stable/dag-run.html
    catchup=False,
    # теги, способ помечать даги
    tags=['example'],
) as dag:

    # t1, t2, t3 - это операторы (они формируют таски, а таски формируют даг)
    t1 = BashOperator(
        task_id='print_date',  # id, будет отображаться в интерфейсе
        bash_command='date',  # какую bash команду выполнить в этом таске
    )

    t2 = BashOperator(
        task_id='sleep',
        depends_on_past=False,  # переопределили настройку из DAG
        bash_command='sleep 5',
        retries=3,  # тоже переопределили retries (было 1)
    )
    t1.doc_md = dedent(
        """\
    #### Task Documentation
    You can document your task using the attributes `doc_md` (markdown),
    `doc` (plain text), `doc_rst`, `doc_json`, `doc_yaml` which gets
    rendered in the UI's Task Instance Details page.
    ![img](http://montcs.bloomu.edu/~bobmon/Semesters/2012-01/491/import%20soul.png)

    """
    )  # dedent - это особенность Airflow, в него нужно оборачивать всю доку

    dag.doc_md = __doc__  # Можно забрать докстрингу из начала файла вот так
    dag.doc_md = """
    This is a documentation placed anywhere
    """  # а можно явно написать
    # формат ds: 2021-12-25
    templated_command = dedent(
        """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
    {% endfor %}
    """
    )  # поддерживается шаблонизация через Jinja
    # https://airflow.apache.org/docs/apache-airflow/stable/concepts/operators.html#concepts-jinja-templating

    t3 = BashOperator(
        task_id='templated',
        depends_on_past=False,
        bash_command=templated_command,
    )

    # А вот так в Airflow указывается последовательность задач
    t1 >> [t2, t3]
    # будет выглядеть вот так
    #      -> t2
    #  t1 |
    #      -> t3
```

> Оператор

Оператор - это описание действия, которые нужно выполнить. Операторы образуют таски (tasks, задачи), они могут зависеть друг от друга. Мы уже видели `BashOperator`.

Каждый оператор должен иметь **уникальный ID**. Помимо этого оператор может иметь много информации о себе: число retry, документацию и т.д.

Познакомимся с основными операторами.

BashOperator

Служил для выполнения bash-команд (команд консоли Linux). Может исполнять одну команду, а может целый скрипт.

Документация

```
# вот так можно попросить Airflow подставить логическую дату
# в формате YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id="test_env",
    bash_command="/tmp/test.sh ",  # обратите внимание на пробел в конце!
    # пробел в конце нужен в случае BashOperator из-за проблем с шаблонизацией
    # вики на проблему https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=62694614
    # и обсуждение https://github.com/apache/airflow/issues/1017
    dag=dag,  # говорим, что таска принадлежит дагу из переменной dag
    env={"DATA_INTERVAL_START": date},  # задает переменные окружения
)
```

`BashOperator` может понадобиться, когда у вас есть готовые shell-скрипты или shell команды, которые нужно регулярно выполнять. Это может быть выгрузка данных (если она идет не через Python) или очищение временных папок.

PythonOperator

Запускает python-функцию с аргументами.

Документация

```
def print_context(ds, **kwargs):
    """Пример PythonOperator"""
    # Через синтаксис **kwargs можно получить словарь
    # с настройками Airflow. Значения оттуда могут пригодиться.
    # Пока нам не нужно
    print(kwargs)
    # В ds Airflow за нас подставит текущую логическую дату - строку в формате YYYY-MM-DD
    print(ds)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='print_the_context',  # нужен task_id, как и всем операторам
    python_callable=print_context,  # свойственен только для PythonOperator - передаем саму функцию
)
```

Пример взят из официальной документации.

В `PythonOperator` можно попросить передавать аргументы функции при вызове:

```
def my_sleeping_function(random_base):
    """Заснуть на random_base секунд"""
    time.sleep(random_base)
```

```
# Генерируем таски в цикле - так тоже можно
for i in range(5):
    # Каждый таск будет спать некое количество секунд
    task = PythonOperator(
        task_id='sleep_for_' + str(i),  # в id можно делать все, что разрешают строки в python
        python_callable=my_sleeping_function,
        # передаем в аргумент с названием random_base значение float(i) / 10
        op_kwargs={'random_base': float(i) / 10},
    )
    # настраиваем зависимости между задачами
    # run_this - это некий таск, объявленный ранее (в этом примере не объявлен)
    run_this >> task
```

Пример взят из официальной документации.

Красота состоит в том, что в таски можно передавать шаблоны (например, строку `"{{ ds }}"`, куда Airflow подставит дату), можно использовать f-strings, вычисления из Python, использовать kwargs для приема настроек Airflow и контекста (набора параметров сервера "здесь и сейчас") - и с использованием всего этого писать функции с очень динамичным поведением.

Справочник по шаблонам Airflow.

> Про даты

В Airflow имеется логическая дата (передается как `ds`). Она говорит, какую дату обрабатывает конкретный запуск.

Логическая дата не всегда совпадает с датой запуска! В нормальном функционировании Airflow запускает даг после окончания логической даты, чтобы убедиться в доступности всех данных.

Например, если даг настроен на ежедневный запуск, то в день 2022-02-12 даг запустится с логической датой 2022-02-11 (вчера), потому что за 12-ое число данных еще может не быть.

> Передача информации

Бывает необходимым передать информацию от одной задачи к другой. В целом, Airflow задуман больше как оркестратор, а не полная платформа для контроля исполнения задач. Поэтому Airflow имеет только простой интерфейс передачи данных - XCom (сокращенно от Cross Communication). XCom передает данные в формате ключ-значение и предназначен для хранения небольших данных (примерно до 1 Гб).

Если вам нужно передавать большие данные между задачами, стоит сохранять их в отдельную от Airflow базу данных, либо в некое отдельное хранилище (S3, HDFS и т.п.).

XCom

Построен очень просто: данные можно push (положить) и pull (получать).

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

import requests
import json

url = 'https://covidtracking.com/api/v1/states/'
```

```
state = 'wa'

def get_testing_increase(state, ti):
    """
    Gets totalTestResultsIncrease field from Covid API for given state and returns value
    """
    res = requests.get(url + '{0}/current.json'.format(state))
    testing_increase = json.loads(res.text)['totalTestResultsIncrease']
    # в ti уходит task_instance, его передает Airflow под таким названием
    # когда вызывает функцию в ходе PythonOperator
    ti.xcom_push(
        key='testing_increase',
        value=testing_increase
    )

def analyze_testing_increases(state, ti):
    """
    Evaluates testing increase results
    """
    testing_increases = ti.xcom_pull(
        key='testing_increase',
        task_ids='get_testing_increase_data_{0}'.format(state)
    )
    print('Testing increases for {0}:'.format(state), testing_increases)

# Default settings applied to all tasks
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'xcom_dag',
    start_date=datetime(2021, 1, 1),
    max_active_runs=2,
    schedule_interval=timedelta(minutes=30),
    default_args=default_args,
    catchup=False
) as dag:
    opr_get_covid_data = PythonOperator(
        task_id = 'get_testing_increase_data_{0}'.format(state),
        python_callable=get_testing_increase,
        op_kwargs={'state':state}
    )
    opr_analyze_testing_data = PythonOperator(
        task_id = 'analyze_data',
        python_callable=analyze_testing_increases,
        op_kwargs={'state':state}
    )

    opr_get_covid_data >> opr_analyze_testing_data
```

Пример взят с astronomer.

Кстати, можно не класть явно результаты в XCom. Все, что возвращает функция, будет автоматически положено в XCom, соответствующий этой стадии (такое поведение можно убрать, если выставить в настройках `{'do_xcom_push': False}`).

> Connections, Variables

В Airflow есть несколько полезных вещей, которые помогают разворачивать код

Connections

Система хранения подключений. Стоит использовать, чтобы безопасно хранить логины/пароли.

```
from airflow.hooks.base_hook import BaseHook

connection = BaseHook.get_connection("conn_name")
conn_password = connection.password
conn_login = connection.login
```

Variables

Динамические переменные, которые задаются в Airflow и которые можно доставать из кода:

```
from airflow.models import Variable

is_prod = Variable.get("is_prod")  # необходимо передать имя, заданное при создании Variable
# теперь в is_prod лежит значение Variable
```

> Best practices

Есть несколько правил "хорошего" DAG.

Документация по хорошим практикам.

Идемпотентность

Повторные перезапуски задачи в DAG за тот же день должны возвращать одинаковые результаты. Это не всегда просто, но выстраивать свои процессы в Airflow стоит именно так, чтобы они обладали идемпотентностью.

Мотивация: задачи могут перезапускаться в следующие дни, могут случайно перезапуститься - в обоих случаях хочется, чтобы запуски выдавали одинаковые значения.

Один таск = одна законченная операция

Стоит смотреть на задачи как на транзакцию: проведение некой атомарной операции от начала до конца. Таск не должен оставлять систему в промежуточном состоянии.

Осторожнее с `datetime.now()`

Питоновская `datetime.now()` возвращает текущее время. Оно может отличаться от "логического" времени на таск и на DAG, и это неприемлимо для идемпотентности. Можно использовать, например, шаблонизированный `{{ ds }}`, в который Airflow автоматически подставит логическую дату.

Не хранить много в XCom

XCom для хранения данных использует СУБД (ее поднимают отдельно), и эти СУБД могут быть не спроектированы для хранения больших результатов. К тому же, большие данные в XCom могут привести к замедлению работы Airflow. Официальная документация не рекомендует хранить большие данные в XCom.

Использовать Airflow как оркестратор

Иногда есть соблазн обернуть все коды внутри DAG и Task в Airflow. Это кажется быстрым решением, но не стоит забывать, что Airflow все-таки ***оркестратор***, а не система "все-в-одном".

Поэтому стоит хранить код в отдельном месте, затем доставлять его на Airflow (через импорты библиотек или докер-образы) и в Airflow хранить только коды с логикой Airflow (а не вашего приложения).

Тестировать код Airflow

Стоит относиться к коду DAG с той же серьезностью, что и к коду всего проекта. Его точно так же стоит покрывать тестами, проверять DAG на валидность, делать тестовые прогоны.

Подробнее