Notes for Cource MLOps and production ML

Дневник прохождения курса "MLOps и production подход к ML исследованиям" и реализации финального проекта "MLOps на примере задачи матчинга геолокаций (на основе соревнования Kaggle Foursquare - Location Matching)"
Ссылки на курс:
https://ods.ai/tracks/ml-in-production-spring-22
https://yandex.ru/q/article/osnovnaia_informatsiia_dlia_uchastnikov_418642d4/

Сквозной репозиторий текущего дневника (все части дз с пометкой "Отработка ДЗ на командном репозитории" выполнялись в нем):
https://gitlab.com/mlops-23reg-team/mlops-23reg-project/

Неделя 1. Концепция воспроизводимых и масштабируемых исследований. Особенности ML разработки в production.

Видео:
https://www.youtube.com/watch?v=OAhccSmbbc0
Презентация:

P MLOps Неделя 1. Воспроизводимос... 9 MB

Неделя 2. Хранение и версионирование кода. Gitlab. Общие принципы Git-flow, Github-flow, настройка репозитория, codereview.

Видео:
https://www.youtube.com/watch?v=tRzeLx0xV9E
Презентация:

MLOps Неделя 2. Git.pdf 2 MB

ДЗ:

MLOps Task 2.docx 10 kB

- Создать локальный репозиторий со своим проектом
- Настроить gitignore в соответствии со средой и технологическим стеком проекта
- Создать удаленный репозиторий (remote)
- Синхронизировать его с локальным через SSH
- Перенести в него свой проект
- Настроить репозиторий под github-flow:
  - заблокировать main от прямых push
  - запретить approve автором кода и коммита
  - запретить merge без разрешения всех дискуссий
- Отработать github-flow:
  - Создать новую feature ветку и выполнить в нее коммит изменений
  - Сделать Pull новой ветки в remote
  - Инициировать merge/pull request
  - Эмитировать code-review добавив комментарии
  - Разрешить все комментарии

    - Сделать approve merge request
    - Удалить feature ветку
    - Повторить процесс с двумя ветками, намеренно создав конфликт и разрешить его

Опционально:

- Создать scrum/canban доску в одной из agile систем
- Отражать все работы над проектом в системе
- Под любые фичи/багфиксы создавать отдельные ветки репозитория и задачи
- Все задачи, переводимые в статус Done, сопровождать ссылками на соответствующие им PR

Работа над ДЗ (Хранение и версионирование кода. Gitlab. Общие принципы Git-flow, Github-flow, настройка репозитория, codereview).

Создание УЗ в GitLab:

1. Зарегистрировался через УЗ Google wisoffe@gmail.com
    - Full name: Ilya Klein
    - Login: @wisoffe
    - Email: wisoffe@gmail.com

Отработка ДЗ на тестовом личном репозитории (без команды):

- Установка git (for Windows):
    - https://git-scm.com/download/win
    - Пакет: 64-bit Git for Windows Setup
    - В ходе установки:
        - Default editor used by Git: Use Visual Studio Code
        - Override the default branch name ... : **main**
        - Choise: Git from the command line and from 3rd-party software
        - Choise: Use bundled OpenSSH

        - Choise: Use the OpenSSH library
        - Choise: Checkout Windows-style, commit Unix-style line endings
        - Choise: Use MinTTY
        - Choise: Default (fast-forward or merge)
        - Choise: Git Credential Manager
        - Extra options: Default (v Enable file system caching, o Enable symbolic link)
        - Experimental options: Not set (o, o)
- Первоначальная настройка git (через cmd или git bash):
    - git config --global user.name wisoffe
    - git config --global user.email wisoffe@gmail.com
- Создать локальный репозиторий со своим проектом:
    - Создал на ноутбуке директорию проекта: **MLOps-my-testrep**
    - Скачал с каггла и закинул в директорию выше свой ноутбук "Stepik-NN-and-NLP 3.3.Final (word embeddings) v6 exp with different parameters" (filename: **stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb**)
    - Через Git bash (Контекстное меню директории проекта -> "Git bash here"):
        - **git init** (этого достаточно для нашего случая, т.к. при установке выбрано global настройка имени репозитория по умолчанию "main", аналогично команде "**git config --global init.defaultBranch main**", если этого не сделано, то нужно явно указать при инициализации через "**git init --initial-branch=main**" или "**git init -b main**")
- Настроить gitignore в соответствии со средой и технологическим стеком проекта (очень хороший мануал по гитигнору с примерами https://linuxize.com/post/gitignore-ignoring-files-in-git/):
    - в корне проекта создаем файл .gitignore (кодировка UTF-8), первоначальное наполнение ниже, но нужно обязательно дополнить (venv и т.д., разобраться по ходу проекта):

```
#jupyter checkpoints
**/.ipynb_checkpoints/

# VSCODE IDE
.vscode

#Pycharm IDE
.idea
```

- Делаем initial commit (здесь и далее с все команды с Git bash):
  - **git add .**
  - **git commit -m "Initial commit"**
  - Примечание: объединить add . и commit можно через **git commit -am "Initial commit"**
- Создаем для аккунта на гитлабе SSH ключи (для дальнейшей синхронизации)
  - На гитлабе -> User settings -> SSH Keys
  - Если для аккаунта еще не созданы SSH ключи, то создаем (ключ создается под каждый ПК/устройство):
    - Go to the .ssh/ subdirectory. If the .ssh/ subdirectory doesn't exist, you are either not in the home directory, or you haven't used ssh before. In the latter case, you need to generate an SSH key pair.
    - Open a terminal (cmd for Win)
    - Type ssh-keygen -t followed by the key type and an optional comment. This comment is included in the .pub file that's created. You may want to use an email address for the comment. Для типа шифрования ED25519 (является предпочтительным):
      - **ssh-keygen -t ed25519 -C "gitlab: wisoffe@gmail.com"**
      - Enter file in which to save the key (.../.ssh/id_ed25519): Enter (т.е. оставляем предложенный вариант)
      - Enter passphrase (empty for no passphrase): вводим парольную фразу (сохранил в enote), допустимо указать пустую парольную фразу, но в этом случае, потенциальному злоумышленнику, для успешной авторизации, достаточно будет завладеть файлами сертификатов (как я понял есть способ
      - по итогу в .../.ssh/ получаем 2 файла, приватный и публичный ключи
  - Add an SSH key to your GitLab account:
    - Copy the contents of your public key file. You can do this manually or use a script. For example, to copy an ED25519 key to the clipboard:
      - Git Bash on Windows (по сути мы ниже просто копируем содержимое id_ed25519.pub):
        - **cat ~/.ssh/id_ed25519.pub | clip**
      - На гитлабе -> User settings -> SSH Keys
        - вставляем скопированный публичный ключ в поле Key
        - в поле Title указываем описание (например): lenovo nbook
        - Expiration date: например на год
        - Click Add key
  - Verify that you can connect

        - For GitLab.com, to ensure you're connecting to the correct server, confirm the SSH host keys fingerprints.
        - Open a terminal and run this command, replacing gitlab.example.com with your GitLab instance URL (ssh -T git@gitlab.example.com):
            - **ssh -T git@gitlab.com**
            - если полученный отпечаток совпадает с тем, что приведен здесь SSH host keys fingerprints, выбираем yes (это проверка на атаку MITM)
            - **ssh -T git@gitlab.com** (т.е. выполняем еще раз, после того как мы добавили отпечаток гитлаба в доверенные хосты), на запрос пассфразы вводим ее (если устанавливали)
            - По итогу должны получить сообщение вида Welcome to GitLab, @wisoffe! (если так, все отлично)
- Создать удаленный репозиторий (remote):
    - На гитлабе -> Create a project -> Create blank project:
        - Name: MLOps-my-testrep
        - Url: https://gitlab.com/wisoffe/mlops-my-testrep
        - Project deployment target (optional): None
        - Visibility Level: Public
        - Снимаем галку Initialize repository with a README
- Синхронизировать его с локальным через SSH (через Git Bash):
    - **git remote add origin git@gitlab.com:wisoffe/mlops-my-testrep.git**
    - **git push -u origin --all**
    - **git push -u origin --tags**

```
Примечание (для информации):
В целом, гитлаб после создания проекта предлагает следующие варианты.
You can also upload existing files from your computer using the instructions below.
Git global setup:
git config --global user.name "Ilya Klein"
git config --global user.email "wisoffe@gmail.com"

Create a new repository:
git clone git@gitlab.com:wisoffe/mlops-my-testrep.git
cd mlops-my-testrep
git switch -c main
```

```
touch README.md
git add README.md
git commit -m "add README"
git push -u origin main

Push an existing folder:
cd existing_folder
git init --initial-branch=main
git remote add origin git@gitlab.com:wisoffe/mlops-my-testrep.git
git add .
git commit -m "Initial commit"
git push -u origin main

Push an existing Git repository:
cd existing_repo
git remote rename origin old-origin
git remote add origin git@gitlab.com:wisoffe/mlops-my-testrep.git
git push -u origin --all
git push -u origin --tags
```

- Перенести в него свой проект:
  - Сделал это заранее, перенеся в изначальную директорию репозитория файл stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb, хотя можно было создать репозиторий только с README.md и перенести проект на этом шаге (сделать подобным образом для общего проекта)
- Настроить репозиторий под github-flow:
  - В гитлабе, переходим в репозиторий MLOps-my-testrep -> Settings ->Repository
    - В Protected branches в таблице для ветки main:
      - Allowed to push -> No one (заблокировать main от прямых push, это обязательно):
      - Allowed to merge (выставляем кому можно делать Merge request (можно выбирать как роли, так и конкретные УЗ)
      - Для реального продакшн проекта обычно выставляется Code owner approval
      - Проверяем, что не выставлена галка Allowed to force push
  - В гитлабе, переходим в репозиторий MLOps-my-testrep -> Settings ->General
    - запретить approve автором кода и коммита:

        - В Merge request approvals (данный раздел доступен в бесплатной версии gitlab только если проект является public), для реального проекта должна быть выставлена галка Prevent approval by author (не позволяет выставлять лицом принимающим решение о принятии реквеста самого автора реквеста). Исключительно на тестовом личном репозитории, если не использовать 2-ю УЗ, придется снять эту галку.
    - Настраиваем Merge (или Pull в терминологии гитхаб) request проверки, в разделе Merge requests -> Merge checks выставляем:
        - Pipelines must succeed (Merge возможен только после завершения всех пайплайнов)
        - All discussions must be resolved (запретить merge без разрешения всех дискуссий)
- (доп. задание, для себя) Клонирование созданного репозитория, для дальнейшей работы на другом ПК (авторизация через HTTPS, предполагается что гит установлен и глобальные настройки name и email выполнены):
    - Запускаем Git bash из контекстного меню директории в которой далее будет создана поддиректория проекта:
        - **git clone https://gitlab.com/wisoffe/mlops-my-testrep.git**
        - Проверяем информацию о репозитории (опционально):
            - **git status**
            - **git log**
            - **git log --pretty=oneline** (очень наглядный вывод)
            - **git branch**
- Отработать github-flow:
    - Создать новую feature ветку и выполнить в нее коммит изменений:
        - **git branch feature1** #feature1 название ветки
        - (опционально) **git branch** #проверяем, что метка создана, но мы в нее еще не перешли
        - **git checkout feature1** #переходим в ветку
        - (опционально) **git branch** или **git status** #проверяем, в какой мы ветке
        - модифицируем как-нибудь файл проекта stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb (добавил одну ячейку с print)
        - **git add .**
        - **git commit -m "Add test print"**
    - Сделать Push новой ветки в remote:
        - **git push origin feature1**
    - Инициировать merge/pull request:

        - В гитлабе, **MLOps-my-testrep -> Merge requests**
        - **New merge request**
        - Source branch: **feature1**
        - Target branch: **main**
        - **Compare branches and continue**
            - **Title:** Краткий заголовок (например Add feature ...)
            - **Description:** Хороший тон, добавлять нормальное описание для реального проекта
            - **Assignees** (ответственные за request): Указываем себя
            - **Reviewers** (кто будет осуществлять код ревью): В реальном проекте отличные от Assignees, в качестве теста, себя
            - **Milestone**: пока не трогаем
            - **Labels**: пока не трогаем
            - **Delete source branch when merge request is accepted:** По умолчанию, установлено, оставляем так
            - **Create merge request**
        - В наших уведомлениях появляются сообщения, что для меня появились запросы на Assign и Review
    - Эмитировать code-review добавив комментарии:
        - Переходим в Review созданного реквеста
            - Во вкладке Overview можно вести переписку через общие комменты
                - Добавил коммент Hello its general comment
            - Во вкладке Changes можно давать комментарии прямо к строкам кода (по факту у меня изменения внесены в юпитер файл, и гитлаб сообщает, что файл слишком большой для просмотра через данный интерфейс, его можно открыть через ... -> Web IDE, но как там давать комментарии уже не нашел).
            - Upd. в ходе выполнения последующих шагов с (мердж конфликтом), внезапно все изменения в том же файле стали отображаться, через интерфейс интуитивно понятно, как дать комментарий к конкретной строке (при наведении указателя слева от желаемой строки кода, появляется значек добавления комментария)
    - Разрешить все комментарии:
        - Upd. в ходе выполнения последующих шагов с (мердж конфликтом), получилось добавить комментарий, далее, по итогу конкретной переписки есть кнопка Resolve (т.к. в настройках выставлена галка All discussions must be resolved, все комментарии должны быть зарезолвлены мерджем)
    - Сделать approve merge request:

        - Нажимаем **Approve** на вкладке Overview (примечание: эта кнопка будет недоступна, если все проверяется под одной УЗ и не снята галка **Prevent approval by author** в Settings ->General)
    - Удалить feature ветку:
        - На вкладке Overview ставим (если не установлена) галку **Delete source branch** и жмем **Merge**
        - Важное примечание, ветка feature1 удалилась с удаленного репозитория, но осталась в моем локальном
        - В локальном Git Bash
            - **git checkout main**
            - **git pull** (или git pull origin main)
            - **git branch -d feature1**
    - Повторить процесс с двумя ветками, намеренно создав конфликт и разрешить его:
        - **git branch feature2_1**
        - **git branch feature2_2**
        - **git checkout feature2_1**
        - добавляем в файле **stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb** ячейку с кодом **print("feature2_1")**
        - **git add .**
        - **git commit -m "Add print(feature2_1)"**
        - git checkout feature2_2
        - добавляем в файле **stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb** ячейку с кодом **print("feature2_2")** (важно, что бы возник конфликт, строка обязательно должна быть добавлена в том же месте, где ранее добавляли print("feature2_1"))
        - **git add .**
        - **git commit -m "Add print(feature2_2)"**
        - **git checkout feature2_1**
        - **git push origin feature2_1**
        - После этого по аналогии с инструкцией выше, через гитлаб создаем Merge request (из feature2_1 в main), ревьювим, апрувим его, по итогу кликаем Merge (проверяем, что все прошло успешно)
        - **git checkout feature2_2**
        - **git push origin feature2_2**
        - После этого по аналогии с инструкцией выше, через гитлаб создаем Merge request (из feature2_1 в main)

- Как итог, видим неактивную кнопку **Merge** с сообщением **Merge blocked: merge conflicts must be resolved** и доп. кнопку **Resolve locally**, нажав на которую всплывает подсказка (но для наших настроек этот вариант не работает):

  Check out, review, and merge locally

  Step 1. Fetch and check out this merge request's feature branch:
  git fetch origin
  git checkout -b 'feature2_2' 'origin/feature2_2'
  Step 2. Review the changes locally.

  Step 3. Merge the feature branch into the target branch and fix any conflicts. How do I fix them?
  git fetch origin
  git checkout 'main'
  git merge --no-ff 'feature2_2'
  Step 4. Push the target branch up to GitLab.
  git push origin 'main'
  Tip: You can also check out merge requests locally. Learn more.
- Первым делом, разрешаем конфликт локально, а именно:
  - **git checkout main**
  - **git pull** (обязательно добиваемся, что бы локально ветка мейн была синхронизирована с origin/main)
  - **git checkout feature2_2**
  - **git rebase main**
  - по итогу получаем сообщение **CONFLICT (content): Merge conflict in** stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb
  - **git status**
  - видим все файлы в которых произошел конфликт, они со статусом **both modified**: (в нашем случае только файл stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb
  - после этого, нам нужно открыть все подобные файлы и разрешить конфликт (примечание: .py файлы можно открывать в IDE, а .ipynb обязательно в текстовом редакторе)
    - проблемные строки будут иметь следующий вид (где от HEAD до ======= это то, как строки представлены в main, от ======= до >>>>>>> feature2_2, как они представлены в нашей новой ветке):

<<<<<<< HEAD
#3.5
=======
#3..5
>>>>>>> feature2_2

- По итогу редактируем и сохраняем файл в том виде, котором он должен быть по итогу (обязательно убирая <<<<<<< HEAD, =======, >>>>>>> feature2_2), например:

#3.5
#3..5

- **git add .** (или конкретные файлы, например stepik-nn-and-nlp-3-3-final-word-embeddings.ipynb)
- **git commit -m "Resolve rebase conflict"**
- **git rebase --continue** (должны получить сообщение вида Successfully rebased and updated refs/heads/feature2_2)
- **git push origin feature2_2 --force** (обязательно с --force, по другому никак)

- Посте этого возвращаемся в наш вебинтерфейс гитлаба, переходим в наш Merge request (если нужно обновляем страницу)
- После этого кнопка Merge должна стать доступной, а сообщение о конфликте уйти, в окне истории мердж реквеста будет автоматически добавлена информация о наших доп комитах "Resolve rebase conflict"
- Завершаем Merge request по аналогии с инструкцией выше (ревьювим, апрувим и т.д.)

Работа по ДЗ над командным проектом:

Создание команды в GitLab:

- На гитлабе Create a Group (с УЗ wisoffe@gmail.com):
  - Group name: **MLOps-23reg-team**
  - Group URL: **mlops-23reg-team**
  - Visibility level: **Public** (изначально выставил Private, но как оказалось, в этом случае, команде доступно создание только Private проектов, а они имеют ограниченный функционал в бесплатной версии gitlab)
  - Role: Other

    - Who will be using this group?: My company or team
    - What will you use this group for?: I want to learn the basics of Git
    - Invite Members:
        - xxx@gmail.com (в последствии не смог принять участие)
        - xxx@gmail.com (в последствии не смог принять участие)
        - xxx@gmail.com (в последствии не смог принять участие)
- Конфигурация команды:
    - На гитлабе -> **Menu -> Groups -> MLOps-23reg-team -> Group information (слева) - Members**
        - Всем участникам выставляем **Max role -> Owner** (в реальном проекте выставляются действительные роли)

Настройки git/gitlab, произведенные на командном репозитории:

- На гитлабе -> Create a project -> Create blank project:
    - Name: **MLOps-23reg-project**
    - Url: **https://gitlab.com/mlops-23reg-team/mlops-23reg-project**
    - Project deployment target (optional): None
    - Visibility Level: **Public (**изначально выставил Private, но как оказалось, в этом случае, проект имеет ограниченный функционал в бесплатной версии gitlab)
    - Оставляем галку **Initialize repository with a README**
- Настраиваем минимальную видимость проекта (при этом, сохраняется весь функционал Public проекта даже в бесплатной версии gitlab, но по содержимое проекта не будет видно никому, кроме участников команды (я не уверен в этом, но похоже, что именно так)):
    - На гитлабе -> Переходим в **MLOps-23reg-project -> Settings -> General**, в подраздел **Visibility, project features, permissions:**
        - убираем галку **Users can request access**
        - Во всех полях, где выставлено **Everyone With Access** меняем на **Only Project Members**
        - Save changes
- Добавляем в корень проекта.gitignore (кодировка UTF-8, первоначальное наполнение ниже, но нужно обязательно дополнить в последствии необходимым, всякого рода venv и т.п., разобраться по ходу проекта):

```
#jupyter checkpoints
**/.ipynb_checkpoints/
```

```
# VSCODE IDE
.vscode

#Pycharm IDE
.idea
```

        - Для разнообразия делаем это через вебинтерфейс, кнопка Upload File основной страницы проекта
            - Upload new file: **.gitignore**
            - Commit message: Add elementary .gitignore
            - Target branch: **main**
- Настраиваем в гитлабе репозиторий под github-flow:
    - Переходим в репозиторий **MLOps-23reg-project -> Settings ->Repository**
        - В **Protected branches** в таблице для ветки **main**:
            - **Allowed to push -> No one** (заблокировать main от прямых push, это обязательно):
            - **Allowed to merge** (выставляем кому можно делать Merge request (можно выбирать как роли, так и конкретные УЗ): **Developers+Maintainers**
            - Проверяем, что **убрана** галка **Allowed to force push**
    - Переходим в репозиторий **MLOps-my-testrep -> Settings ->General**
        - запретить approve автором кода и коммита:
            - В **Merge request approvals** (в нашем случае, для бесплатного аккаунта и командного проекта, данный раздел недоступен) для реального проекта должна быть выставлена галка **Prevent approval by author** (не позволяет выставлять лицом принимающим решение о принятии реквеста самого автора реквеста).
        - Настраиваем Merge (или Pull в терминологии гитхаб) request проверки, в разделе **Merge requests -> Merge checks** выставляем:
            - **Pipelines must succeed** (Merge возможен только после завершения всех пайплайнов)
            - **All discussions must be resolved** (запретить merge без разрешения всех дискуссий)
        - **Save changes**
- Клонирование созданного репозитория, для дальнейшей работы на локальном ПК (авторизация через HTTPS, если нужно по SSH, пример описан в отработке ДЗ на личном репозитории, предполагается что гит установлен и глобальные настройки name и email выполнены):

- Запускаем Git bash из контекстного меню директории в которой далее будет создана поддиректория проекта:
  - **git clone https://gitlab.com/mlops-23reg-team/mlops-23reg-project.git**
  - **cd ./mlops-23reg-project**
  - Проверяем информацию о репозитории (опционально):
    - **git status**
    - **git log**
    - **git branch**

Полезное, на заметку, по итогу ДЗ:

- Провел 5 экспериментов, с целью выяснить, в каких случаях возникает merge/rebase conflict, который git не может разрешить самостоятельно (для случаев одновременного редактирования одного файла):
  - Вариант 1. Изменяем разные строки одного файла (добавляем что то в разных строках, не меняя их количество) --> Automatic merge
  - Вариант 2. Добавляем доп. строки в разных местах одного файла --> Automatic merge
  - Вариант 3. Удаляем одну или несколько строк в одной части файла, добавляем другие строки в другой части файла --> Automatic merge
  - Вариант 4. Удаляем в разных местах разные строки --> Automatic merge
  - **Вариант 5.** Добавляем в одном месте (после одной и той же строки) доп строки (разные для двух веток), либо же изменяем одни и те же строки - **Automatic merge failed**
  - **Выводы:** в большинстве реальных случаем git сможет произвести автоматическое слияние, и только, если изменения будут касаться аналогичных строк, произойдет Merge conflict, который нужно будет разрешить вручную
- Принципы работы с ветками (при их мердже в main), которые позволяют избежать излишних ветвлений и запутанной истории коммитов проекта (есть краткое полезное видео по этому поводу GIT: Merge или Rebase? В чем разница? -
  - Если работаем над веткой (например новой фичи) **в одиночку, то однозначно пользуемся git rebase**:
    - периодически, если работаем долго и кто то еще при этом может менять main
    - непосредственно, перед пушем изменений нашей feature ветки в удаленный репозиторий
    - Один из вариантов (предполагаем, что находимся в нашей feature ветке)
      - Вариант 1. С предварительной синхронизацией нашей локальной main ветки:
        - **git pull origin main:main**

            - **git rebase main**
        - Вариант 2. Без изменения состояния локальной main ветки:
            - **git fetch**
            - **git rebase origin/main**
    - Если работают над веткой 2 (максимум 3 человека), то лучше тоже делать через git rebase, но обязательно нужно договариваться и делать это синхронно (подробно в кратком видео выше)
    - В остальных случаях, скорей всего остается только варианте через merge (подробно в кратком видео выше)
- Просмотр ветвлений (графов) коммитов:
    - На гитлабе (наиболее наглядно): Project name -> (Слева) **Repository -> Graph**
    - В терминале, для всех коммитов: **git log --graph --oneline --all**
    - В терминале, для текущей ветки: **git log --graph --pretty=oneline --abbrev-commit**
- Ниже приведен процесс деплоя github-flow, для реального большого проекта (описание на 1:20:00 видеоурока #2):

    Deploy:
    • Закрываем ветку master
    • Merge master в deploy ветку
    • Build/test
    • Deploy в пре-прод
    • Deploy в прод
    • Мониторинг
    • PR в master
    • Открываем мастер

Неделя 3. Codestyle, инструменты форматирования, линтеры.

Видео:
https://youtu.be/wqfN5t2-XMk

Презентация:

MLOps Неделя 3. Codestyle.pdf 987 kB

ДЗ:

MLOps Task 3.docx 17 kB

- Провести аудит кода своего проекта на соответствие PEP 8 и хорошим практикам codestyle
- Установить локально стилистический линтер (выбрать на свое усмотрение).
- Интегрировать линтер в свою IDE (если поддерживается)
- Провести ручное форматирование части проекта
- Проверить корректность форматирования установленным линтером и внести неучтенные ранее стилистические ошибки
- Установить и настроить (так, чтобы не было конфликтов с линтером) авто-форматер (на своё усмотрение). Предпочтительно интегрировать с IDE.
- Пользуясь авто-форматером и руками провести форматирование всего проекта для его соответствия codestyle и хорошим практикам
- Создать CI пайплайн и встроить в него линтер(ы). Желательно, чтобы осуществлялся контроль:
  - стиля кода
  - семантики (best practices)
  - корректности типов

Ссылка на репозиторий, на котором Автор курса демонстрирует практические шаги: https://gitlab.com/m1f/mlops-course

Отработка ДЗ на командном репозитории (https://gitlab.com/mlops-23reg-team/mlops-23reg-project/):

- Подготовительные шаги
  - в git создал ветку ie_homework3_codestyle, перешел в нее
  - добавляем в .gitignore

```
#venv
```

```
.venv
```

- открываем директорию проекта в VSCode
- через терминал VSCode (Ctrl + `) создаем venv окружение
  - **python -m venv .venv**
  - **call .venv/scripts/activate** (cmd) или **.venv\Scripts\Activate.ps1** (PS)
  - по итогу, если мы перешли в venv, в начале командной строки должно значиться "(.venv)"
- в ходе всего урока ведем файл requirements в 2-х вариантах (нужно это для того, что бы чистом файле были видны только реально устанавливаемые пакеты, с версиями, при этом, версии зависимых от них пакетов не сохраняются, и очень редко, но может случиться ситуация, что зависимые пакеты установятся других версий, после чего воспроизводимость нарушится, для этих целей сохраняется дополнительный файл _freeze со всеми зависимостями):
  - **requirements.txt** (ведем вручную, добавляя туда информацию об устанавливаемых через pip install пакетах)
    - после установки через pip, выполняем pip freeze
    - находим строку с установленным только что пакетом и его версией, вручную добавляем эту строку в requirements.txt
  - requirements_freeze.txt, формируется по итогу всего урока, после установки всех необходимых пакетов, через:
    - **pip freeze > requirements_freeze.txt**
  - восстановить всю среду можно будет командой pip install -r .\requirements.txt
- Для запуска ноутбуков через VSCode, в виртуальное окружение нужно установить pip install ipykernel (он подтягивает очень много зависимостей, но без этого никак, в реальных проектах возможно не имеет смысла его тянуть в prod окружение, ограничиться каким-нибудь dev или отдельно средой исключительно для работы с ноутбуками)
- Провести аудит и форматирование кода своего проекта на соответствие PEP 8 и хорошим практикам codestyle
- все эксперименты и форматирование проводим на 2-х ноутбуках 0.1-foursqlm-mlops-baseline.ipynb, 0.1-ik-foursqlm-sample-submission.ipynb (в директории notebooks) и тестовом файле homework3_codestyle_sample.py (состоящем из кода 2-4 ячеек baseline ноутбука)
- начинаем с форматеров, путем предварительного сравнения, выбран Black:
  - устанавливаем (в venv) pip install black
  - выставляем в настройках VSCode как форматтер по умолчанию
    - **File -> Preferences -> Settings** (или Ctrl + ,)

        - Переключаем тип настройки с User на Workspace (опционально, можно настраивать либо для проекта, либо для текущего профиля пользователя)
        - в строке поиска вбиваем "**python formatting provider**"
        - в **Python › Formatting: Provider** выбираем **black**
        - в строке поиска вбиваем "**format on save**", ставим галку "**Format a file on save**" (форматирование при сохранении видимо не работает для ноутбуков, для них нужно форматировать вручную через через контекстное меню -> **Format Notebook** (Shift + Alt + F))
        - по итогу в файле настроек **.vscode/settings.json** будут содержаться строки:

```
{
    "python.formatting.provider": "black",
    "editor.formatOnSave": true
}
```

        - по умолчанию максимально допустимая длина строки в black = 88 символом, по мне так это мало, особенно для специфики ds, посмотрел в других проектах, например в исходниках torch, они используют длину 120 (это в том числе значение по умолчанию для pyCharm), первоначально выставил 120, но на ноутбуке в ide у меня с учетом лувого меню, строки не помещаются на экране, так же команда black настоятельно не рекомендует высталять длину более 100, в итоге решено остановиться на 100, настраиваем black (пример всех возможных настроек здесь https://github.com/psf/black/blob/main/pyproject.toml):
            - создаем корне проекта файл **pyproject.toml**, добавляем в него строки:

```
[tool.black]
line-length = 100
```

        - форматируем файлы 0.1-foursqlm-mlops-baseline.ipynb, 0.1-ik-foursqlm-sample-submission.ipynb, homework3_codestyle_sample.py через пересохранение или через контекстное меню -> Format Document/Notebook (Shift + Alt + F)
    - Переходим к настройке линтеров (в IDE)
        - В качестве основных линтеров выбраны flake8 (преимущественно стилистический), mypy (проверяет работу с типами на основе аннотаций), bandit (проверяет на предмет уязвимостей кода)

        - дополнительно в IDE можно использовать pylint (он кстати в VSCode выбран по умолчанию), для целей просмотра советов (он дает множество логических замечаний, помимо стилистических), но для реального блокировщика в ходе CI (т.е. блокировки мердж реквеста, без исправления всех замечаний) он возможно будет слишком жестким
        - все линтеры так же должны быть установлены в виртуальное окружение:
            - **pip install flake8 mypy bandit pylint**
        - создаем в корне проекта файл .flake8, следующего содержания:

```
# black-compatible flake8 configuration:
# https://black.readthedocs.io/en/stable/the_black_code_style/current_style.html
[flake8]
max-line-length = 100
select = C,E,F,W,B,B950
extend-ignore = E203, E501
```

        - активируем линтеры, через добавление в файл настроек **.vscode/settings.json** строк (альтернативный способ через Ctrl + Shift + P и вводим lint):

```
{
    ...
    "python.linting.flake8Enabled": true,
    "python.linting.enabled": true,
    "python.linting.mypyEnabled": true,
    "python.linting.banditEnabled": true,
    "python.linting.pylintEnabled": false
    ...
}
```

        - флаг "python.linting.pylintEnabled" можно менять на true в зависимости от надобности включения pylint
    - Проверяем .py файлы линтерами, устраняем замечания:
        - открываем **homework3_codestyle_sample.py** и запускаем его, в итоге во вкладке PROBLEMS видим все замечания от линтеров (альтернативный вариант проверки, без исполнения, Ctrl + Shift + P и вводим lint, выбираем "Python: run linting")
        - устраняем все замечания линтеров, в том числе доустанавливаем в виртуальное окружение все недостающие пакеты

        - что бы отключить какое то замечание **flake8** по конкретной строке, добавляем inline комментарий вида # noqa: F401 и пересохраняем файл (более подробно про исключения здесь https://flake8.pycqa.org/en/4.0.1/user/violations.html?highlight=ignoring-errors)
        - устранил замечания основных линтеров быстро, дополнительно включил **pylint**, он существенно более строгий, устранил все, что считаю нужным
            - исключения для **pylint** достаточно гибко, подробнее здесь https://pylint.pycqa.org/en/latest/user_guide/message-control.html#block-disables
            - для автоматической генерации шаблона docstring в **vscode**, устанавливаем расширение **autoDocstring**
    - Проверяем ноутбуки .ipynb линтерами (проблемный вопрос, подробности ниже):
        - Ноутбуки проверяем линтерами через
            - **nbqa (https://github.com/nbQA-dev/nbQA)**
                - **pip install nbqa**
                - **nbqa flake8 --ignore=E266, E303,E302,E305 ./notebooks**
                - проблема в том, что **nbqa flake8** не подхватывает исключения в виде комментариев типа # noqa: F401, работают только глобально переданные исключения, а это очень неудобно и не гибко
            - или через **pycodestyle_magic** (https://github.com/mattijn/pycodestyle_magic)
        - По факту, все варианты, жутко неудобные, лучше всего было бы через **pycodestyle_magic**, но попытки хоть как то передать **flake8** настройки (настройки с файла .flake8, не подхватываются, например допустимую длину строки через **%flake8_on --max_line_length 100**) приводят к переполнению буфера **pycodestyle_magic** (возможно это временный баг и его поправят)
        - Итого вопрос с адекватно и удобной проверкой ноутбуков линтерами, открыт, потратил на разбор много времени, решено сейчас более не тратить на это время, очень существенным является уже то, что ноутбук в один клик форматируется через **black**, остальное, если понадобится откладываю до реальной необходимости (возможно поправят баг выше, тогда все существенно упростится).
- Сохраняем текущее полное состояние окружения **pip freeze > requirements_freeze.txt**
- Создать CI пайплайн и встроить в него линтер(ы). Желательно, чтобы осуществлялся контроль:
    - Настраиваем локальный **CI Runner** (смысл в том, что гитлаб в бесплатном аккаунте выделяет ограниченные по ресурсам и времени мощности, для CI, но для объемов реального проекта этого недостаточно):
        - В гитлабе переходим **MLOps-23reg-project -> Settings -> CI/CD**, раскрываем раздел Runners
        - Снимаем галку **Enable shared runners for this project**

- Открываем инструкцию по ссылке Install GitLab Runner and ensure it's running и выполняем первичные мероприятия, соответствующие варианту запуска и ОС (в реальном проекте лучше разворачивать все в докере, но сейчас выберем самый простой вариант). Для Windows (https://docs.gitlab.com/runner/install/windows.html):
  - Создаем директорию для раннера (например GitLab-Runner в папке с проектами, но не в самом проекте; все виртуальное окружение в ходе CI будет создаваться в этой директории), качаем в нее x64 версию раннера, переименовываем файл в gitlab-runner.exe
  - Регистрируем раннер:
    - cmd от имени администратора, переходим в директорию с раннером, выполняем .\gitlab-runner.exe register, отвечаем на вопросы:
    - URL: https://gitlab.com/
    - Registration token: берем со страницы, откуда первоначально открывали инструкцию (MLOps-23reg-project -> Settings -> CI/CD, раздел Runners)
    - Description: что то вдумчивое, например MLOps IE-nbook windows runner
    - Tags: например, MLOps, windows, IE, runner
    - Maintenance note: можно ничего не заполнять
    - Если все норм, получаем сообщение Registering runner... succeeded
    - Enter an executor: shell
    - Должны получить сообщение Runner registered successfully.
  - Установка раннера как сервиса и запуск
    - **.\gitlab-runner.exe install** (в инструкции есть рекомендации, что сервис лучше устанавливать и запускать не от текущего пользователя, а например от системного или иного аккаунта, но т.к. в реальном проекте лучше все это делать через докер, то не заморачиваемся на это сейчас)
    - **.\gitlab-runner.exe start** (можно запустить раннер через run, тогда он будет работать пока открыто консольное окно, прервать через Ctrl + C)
    - По итогу, в разделе Runners (**MLOps-23reg-project -> Settings -> CI/CD**), слева, мы должны увидеть наш раннер с зеленым статусом (т.е. он запущен)
    - Важно, заходим в раздел Edit (значек Карандаш рядом с зеленым статусом нашего раннера), и выставить галку **Run untagged jobs**, иначе раннер будет подхватывать только задачи на основе тегов

        - Для информации, останавливается раннер через **.\gitlab-runner.exe stop**, удаляется сервис через .\gitlab-runner.exe uninstall, весь список команд можно посмотреть запустив раннер без параметров
    - Настраиваем раннер, открываем файл в директории с раннером config.toml,
        - заменяем **shell = "pwsh" на shell = "powershell"** (необходимо, добавлять команду, с помощью которой в системе из cmd можно запустить повершелл)
        - можно поднять количество одновременных задач, например до 4, т.е. concurrent = 4
        - сохраняем файл и на всякий случай перезапускаем раннер
- Настраиваем тестовый CI (только для целей урока, т.е. ограничимся проверкой линтерами)
    - На гитлабе **MLOps-23reg-project -> CI/CD -> Pipelines**, в списке предлагаемых шаблонов CI выбираем **Python -> Use template**
    - По факту CI настраивается через файл **.gitlab-ci.yml** в корне проекта, гитлаб позволяет его создать и закоммитить через веб интерфейс, но т.к. у нас есть запрет на прямые коммиты в main, просто копируем весь текст из предложенного шаблона и через IDE создаем соответствующий файл с наполнением в корне проекта (все это делаем в текущей ветке ie_homework3_codestyle урока)
    - содержание **.gitlab-ci.yml** (этот вариант подходит только для текущего раннера по windows, в реальном проекте нужно запускать раннер из под докера и изменять под него команды, либо же настраивать работу с разными вариантами раннеров через теги, в чате курса были примеры подобных настроек, но как по мне, через докер более универсальное решение):

```
# This file is a template, and might need editing before it works on your project.
# To contribute improvements to CI/CD templates, please follow the Development guide at:
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Python.gitlab-ci.yml
# Official language image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/python/tags/
image: python:3.9
# Change pip's cache directory to be inside the project directory since we can
# only cache local items.
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
# Pip's cache doesn't store the python packages
# https://pip.pypa.io/en/stable/topics/caching/
```

```
#
# If you want to also cache the installed packages, you have to install
# them in a virtualenv and cache it as well.
cache:
  paths:
    - .cache/pip
    - venv/
before_script:
  - python --version  # For debugging
  - python -m venv .venv
  - Invoke-Expression(".venv/Scripts/Activate.ps1")
  - pip install -r requirements_freeze.txt
test_lint:
  script:
    - flake8 homework3_codestyle_sample.py
    - mypy --ignore-missing-imports homework3_codestyle_sample.py
    - bandit homework3_codestyle_sample.py
    - pylint homework3_codestyle_sample.py
```

- Далее нам нужно закоммитить .gitlab-ci.yml в main ветку, через merge-request, делаем это со всеми сделанными ранее изменениями в других файлах, которые производили в рамках ветки ie_homework3_codestyle
  - **git add .**
  - **git commit -m "Add .gitlab-ci.yml and code formating"**
  - **git push origin ie_homework3_codestyle**
  - через веб-интерфейс создаем merge-request в main, аппрувим и мерджим его (все как полагается)
- Далее тестируем работу нашего пайплайна в ходе тестового merge-request:
  - В разделе **MLOps-23reg-project -> Settings -> General -> Merge requests** должны обязательно стоять галка **Pipelines must succeed** (в группе настроек Merge checks)
  - в Git bash локально, подгружаем последние изменения ветки main, удаляем ветку, которую только что смерджили, создаем еще одну ветку для тестирования CI
    - **git checkout main**
    - **git pull origin main**
    - **git branch -d ie_homework3_codestyle**
    - **git branch ie_homework3_test_ci_merge**

        - **git checkout ie_homework3_test_ci_merge**
    - Производим какое-нибудь изменение в файле homework3_codestyle_sample.py, такие, что бы какой-либо линтер выдавал замечание
    - Коммитим и пушим изменения в origin
        - **git add .**
        - **git commit -m "For test CI merge request"**
        - **git push origin ie_homework3_test_ci_merge**
    - В веб интерфейсе гитлаба, создаем мердж реквест
    - В статусе мердж реквеста должен запустить процесс работы пайплайна
        - Подробный статус выполнения можно смотреть в **MLOps-23reg-project -> CI/CD -> Jobs**
        - По итогу мердж должен быть заблокирован по причине не пройденных lint тестов в пайплайне (подробности так же видно в Jobs)
    - Исправляем все необходимое
        - Локально исправляем код в файле homework3_codestyle_sample.py, что бы линтеры не давали замечаний
        - Коммитим и пушим изменения
            - **git add .**
            - **git commit -m "Fix piplene linter errors"**
            - **git push origin ie_homework3_test_ci_merge**
        - Обновляем страницу мердж реквеста в гитлабе, только что сделанный коммит должен автоматически добавиться в историю, и пайплайн должен автоматически перезапуститься
        - По итогу удачной отработки пайпла, и после процедуры необходимых аппрувов нам станет доступна кнопка merge, после ее нажатия наши изменения попадут в main
        - На этом все, дополнительно чистим локальную ветку
            - **git checkout main**
            - **git pull origin main**
            - **git branch -d ie_homework3_test_ci_merge**

Полезное, на заметку, по итогу ДЗ:

- Дополнительно можно выставлять контроль линтерами и форматерами через git-hooks, подробности здесь https://github.com/laactech/pre-commit-config-latest , но пока мне больше нравится подход по интеграции линтеров и форматеров в ide, и отработке всех замечаний в удобном интерфейсе, и финальная проверку уже на стадии отработки пайплайна при мердж реквесте

Неделя 4. Шаблонизация. Python пакеты и CLI. Snakemake.

Видео:
https://youtu.be/VaXjCtM1XEc
Презентация:

MLOps Неделя 4. Шаблонизация.p... 935 kB

ДЗ:

MLOps Task 4.docx 18 kB

- Создать структуру каталогов для своего DS проекта (можно опираться на шаблон Cookiecutter)
- Если вы планируете использовать собственный шаблон, разворачивая его при помощи Cookiecutter, рекомендуется создать под него отдельный репозиторий.
- Произвести рефакторинг своего кода, разбив его на отдельные логические .py модули и преобразовать их в python пакет
- Реализовать для этих модулей CLI (можно использовать click)
- Создать единую точку входа для запуска вашего пайплайна. Можно использовать python скрипт, консольный файл или какой-либо workflow manager (например SnakeMake).
- С учетом масштаба изменений, под проект можно создать новый репозиторий, произведя в нем настройки необходимых линтеров

Ссылка на репозиторий, на котором Автор курса демонстрирует практические шаги: https://gitlab.com/m1f/mlops-course

Отработка ДЗ на командном репозитории (https://gitlab.com/mlops-23reg-team/mlops-23reg-project/):

- Подготовительные шаги
  - в git создал ветку ie_homework4_cookiecutter, перешел в нее (далее работаем в этой ветке)
    - **git branch ie_homework4_cookiecutter**
    - **git checkout ie_homework4_cookiecutter**
- Создать структуру каталогов для своего DS проекта (можно опираться на шаблон Cookiecutter, если вы планируете использовать собственный шаблон, разворачивая его при помощи Cookiecutter, рекомендуется создать под него отдельный репозиторий)
  - cookiecutter проект по шаблонизации, не только для проектов DS, самый распространенный шаблон именно для Data Science здесь https://github.com/drivendata/cookiecutter-data-science (мануал здесь https://drivendata.github.io/cookiecutter-data-science/)
  - основная суть cookiecutter создать однообразную структуру каталогов проекта, для этого, нужно в какой либо локальной директории (не проекта) создаем временную venv, устанавливаем туда cookiecutter, запускаем с указанием шаблона, проходим небольшой визард, после чего перемещаем полученную структуру каталогов в свой проект, а саму временную виртуальную среду с директорией удаляем:
    - переходим в какую либо директорию (не основного проекта)
    - **python -m venv ./.venvtemp**
    - **.venvtemp/Scripts/Activate.ps1**
    - **pip install cookiecutter**
    - cookiecutter https://github.com/drivendata/cookiecutter-data-science
      - project_name [project_name]: **MLOps-23reg-project**
      - repo_name [mlops-23reg-project]: **mlops-23reg-project**
      - author_name [Your name (or your organization/company/team)]: **MLOps-23reg-team**
      - description [A short description of the project.]: **MLOps cource project by 23reg team.**
      - Select open_source_license: **1 (MIT)**
      - s3_bucket [[OPTIONAL] your-bucket-for-syncing-data (do not include 's3://')]:
      - aws_profile [default]:

        - Select python_interpreter: **1 (python3)**
    - Перемещаем созданную структуру в основной проект (примечания, файлы .gitkeep находящиеся в созданных директориях, так называемые заполнители, нужны только для того, что бы "пустые" директории попали в git), с учетом следующего:
        - requirements.txt комбинируем из 2-х файлов, а именно, в текущий, из cookiecutter добавляем "-e ." и комментарии, по итогу файл будет выглядеть так:

```
# local package
-e .
# external requirements
black==22.3.0
flake8==4.0.1
...
```

        - README.md комбинируем из 2-х файлов, а именно, в текущий, добавляем все, что предлагает Cookiecutter
        - **.gitignore** комбинируем из 2-х файлов (за основу берем от Cookiecutter, добавляем туда исключение для .venv и меняем .ipynb_checkpoints/ на **/.ipynb_checkpoints/), а так же перепрописываем все исключения для директории /data/ (т.к. изначально было добавлено полное исключение /data/, вся внутренняя структура терялась в гите, а она очень полезна), по итогу получаем файл следующего содержания:

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]

# C extensions
*.so

# Distribution / packaging
.Python
env/
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
```

```
lib64/
parts/
sdist/
var/
*.egg-info/
.installed.cfg
*.egg

# Virtual environments
.venv

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt

# Unit test / coverage reports
htmlcov/
.tox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover

# Translations
*.mo
*.pot

# Django stuff:
*.log
```

```
# Sphinx documentation
docs/_build/

# PyBuilder
target/

# DotEnv configuration
.env

# Database
*.db
*.rdb

# Pycharm
.idea

# VS Code
.vscode/

# Spyder
.spyproject/

# Jupyter NB Checkpoints
**/.ipynb_checkpoints/

# exclude data from source control by default
data/*
!data/external/
!data/interim/
!data/processed/
!data/raw/
data/external/*
!data/external/.gitkeep
data/interim/*
!data/interim/.gitkeep
data/processed/*
!data/processed/.gitkeep
data/raw/*
```

```
!data/raw/.gitkeep

# Mac OS-specific storage files
.DS_Store

# vim
*.swp
*.swo

# Mypy cache
.mypy_cache/
```

- На практике столкнулся с тем, что для предложенной структуры src, мне в некоторых модулях поддиректорий, необходимо было импортировать какие-то общие для всех модулей функции, для этого я создал в корне src файл с общими функциями, при этом, я из не мог просто так импортировать функции из верхнеуровневых или соседних директорий; Решение приведено здесь https://stackoverflow.com/questions/714063/importing-modules-from-parent-folder в ответе Solution without sys.path hacks; Для этих целей в шаблоне Cookiecutter уже присутствует файл setup.py, а так же в файл requirements.txt добавлена строка "-e .". Итого, что бы все работало, делаем следующее (производим установку в текущую venv и не забываем сохранять полное состояние через freeze):
  - **pip install -e .**
  - **pip freeze > requirements_freeze.txt** (в файле freeze аналогичный импорт будет вида -e git+https://gitlab.com/mlops-23reg-team/mlops-23reg-project.git@f0ee073993a807edc4404bfb7f06f17cc8b144fb#egg=src)
- Удаляем из корня проекта временный файл homework3_codestyle_sample.py (создавался в рамках ДЗ №3)
- Раздел test_lint в файле .gitlab-ci.yml изменяем на:

```
test_lint:
  script:
    - flake8 src
    - mypy --ignore-missing-imports src
    - bandit src
    - pylint src
```

    - Удаляем следующие файлы из корня проекта (часть файлов использовать не планирую, в предназначении другой части нужно разбираться, добавлять их нужно осознанно, а не просто иметь в проекте кучу файлов непонятного предназначения; в будущем возможно имеет смысл в них разобраться, раз они попали в шаблон): **Makefile, test_environment.py, tox.ini**
    - Коммитим изменения (достаточно локально, без пуша в ремоут)
        - **git add .**
        - **git commit -m "Change structure to drivendata Cookiecutter Data Science"**
- Произвести рефакторинг своего кода, разбив его на отдельные логические .py модули и преобразовать их в python пакет
    - Произведены существенные изменения с разбивкой на отдельные модули и использование click для возможности вызова из cli (все изменения есть в репозитории)
    - В ходе рефакторинга произведены последовательные коммиты, по итогу ветка ie_homework4_cookiecutter запушена в origin, после чего смерджена в main через реквест (после чего локальная ветка ie_homework4_cookiecutter удалена)
- Создать единую точку входа для запуска вашего пайплайна. Можно использовать python скрипт, консольный файл или какой-либо workflow manager (например SnakeMake).
    - в git создал ветку ie_homework4_snakemake, перешел в нее (далее работаем в этой ветке)
        - **git branch ie_homework4_snakemake**
        - **git checkout ie_homework4_snakemake**
    - В итоге принято решение не использовать snakemake, т.к. он нормально через pip не ставится (вариант установки описанный Павлом в курсе тоже не работает, должен ставиться через конду или poetry, но не стал заморачиваться), а далее он не используется в курсе, т.к. его функционал (workflow) выполняет DVC (который более распространен и дополнительно имеет функционал версионирования файлов)
    - созданная выше ветка ie_homework4_snakemake удалена
    - Единая точка входа для запуска может быть реализована через консольный файл (например start.cmd) следующего содержания:

```
python .\src\data\raw_train_test_split.py .\data\raw\train.csv .\data\interim\split_train.csv .\data\interim\split_test.csv --n_splits=3
python .\src\data\generate_pairs.py .\data\interim\split_train.csv .\data\interim\split_train_pairs.csv .\reports\metrics_of_split_train_pairs.json --
get_metrics=true
python .\src\features\build_pair_features.py .\data\interim\split_train_pairs.csv .\data\processed\split_train_pair_feautures.csv true
python .\src\models\train_model.py .\models\model_on_split_df.pkl --pair_feautures_dataset_path .\data\processed\split_train_pair_feautures.csv

python .\src\data\generate_pairs.py .\data\interim\split_test.csv .\data\interim\split_test_pairs.csv .\reports\metrics_of_split_test_pairs.json --
get_metrics=true
```

```
python .\src\features\build_pair_features.py .\data\interim\split_test_pairs.csv .\data\processed\split_test_pair_feautures.csv true
python .\src\models\predict_model.py .\models\model_on_split_df.pkl .\data\interim\split_test.csv .\data\processed\submission_pred_split.csv --
pair_feautures_dataset_path .\data\processed\split_test_pair_feautures.csv
python .\src\models\evaluate.py .\data\interim\split_test.csv .\data\processed\submission_pred_split.csv .\reports\metrics_of_split_test_final.json
```

Неделя 5. Управление зависимостями и инструменты автоматизации на примере DVC

Видео:
https://youtu.be/tC3IdAN_hX4
Презентация:

MLOps Неделя 5. Poetry DVC.pdf 1 MB

ДЗ:

MLOps Task 5.docx 18 kB

- Выбрать менеджер зависимостей для проекта
- Создать виртуальную среду и установить все необходимые для проекта зависимости
- Сохранить конфигурацию виртуальной среды в соответствующем, выбранному менеджеру, формате и добавить в трэкинг git
- Установить и сконфигурировать DVC
- Добавить необходимые файлы данных/моделей в DVC
- Выбрать и настроить удаленное хранилище для DVC
- Настроить пайплайн запуска модулей с помощью DVC

Ссылка на репозиторий, на котором Автор курса демонстрирует практические шаги: https://gitlab.com/m1f/mlops-course

Отработка ДЗ на командном репозитории (https://gitlab.com/mlops-23reg-team/mlops-23reg-project/):

- Выбрать менеджер зависимостей для проекта
  - пробуем по рекомендации автора курса poetry
  - Подготовительные шаги, а именно в git создал ветку ie_homework5_poetry, перешел в нее (далее работаем в этой ветке)
    - **git branch ie_homework5_poetry**
    - **git checkout ie_homework5_poetry**
  - Удаляем из локальной директории проекта папку .venv
  - Устанавливаем poetry в систему (именно в текущую систему, а не виртуальное окружение) по инструкции https://python-poetry.org/docs/#installation , для win это:
    - **(Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py -UseBasicParsing).Content | python -**
    - по итогу установки проверяем версию через poetry --version (т.к. в ходе установки директория %USERPROFILE%\.poetry\bin добавляется в PATH, возможно потребуется перезапустить оболучку PS или IDE, с которой проводилась установка выше)
- Создать виртуальную среду и установить все необходимые для проекта зависимости
  - Находясь в директории проекта, с CLI:
    - **poetry init**
    - Отвечаем на вопросы визарда:
    - Package name [mlops-23reg-project]:
    - Version [0.1.0]:
    - Description []: **MLOps cource project by 23reg team.**
    - Author [wisoffe <wisoffe@gmail.com>, n to skip]: **MLOps-23reg-team**
    - License []: **MIT**
    - Compatible Python versions [^3.9]:
    - Would you like to define your main dependencies interactively? (yes/no) [yes] **no**
    - Would you like to define your development dependencies interactively? (yes/no) [yes] **no**

        - Do you confirm generation? (yes/no) [yes]
    - По итогу в файл **pyproject.toml** (где будут храниться основные настройки poetry), добавятся следующие строки:

```
...
[tool.poetry]
name = "mlops23regproject"
version = "0.1.0"
description = "MLOps cource project by 23reg team."
authors = ["MLOps-23reg-team"]
license = "MIT"

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

- Сохранить конфигурацию виртуальной среды в соответствующем, выбранному менеджеру, формате и добавить в трэкинг git
    - краткая справка по командам с виртуальной средой
        - Запуск скрипта poetry **run python < package_name>** (для случая, когда нам необходимо выполнить запуск скрипта без активации среды)
        - Активация среды (опционально) **poetry shell** (при этом среда будет создаваться в %USERPROFILE%\AppData\Local\pypoetry\Cache\virtualenvs, а не в директории проекта)
        - Деактивация среды **poetry exit** (на практике у меня не работает, говорит нет такого параметра)
        - Обновление зависимостей **poetry update**
        - Установка зависимостей **poetry install**
        - Фиксация зависимостей (в poetry.lock) **poetry lock** (применяется для полной фиксации зависимостей для прод. среды, что бы далее в прод. ставить только из этого файла, где фиксируются все версии модулей, без вариаций)
    - ставим последовательно все пакеты из requirements.txt, причем есть возможность разделять пакеты на prod и dev среды (**poetry add <package name>**, **poetry add --dev <package name>**, далее можно устанавливать только прод среду через **poetry install --no-dev**)

- **poetry shell** (активируем среду, если необходимо, возможно установка будет работать и без активации, на основании директории, с которой осуществляется запуск)
- Устанавливаем прод. модули:
  - **poetry add pandas==1.4.2**
  - **poetry add haversine==2.5.1**
  - **poetry add sklearn==0.0**
  - **poetry add fuzzywuzzy==0.18.0**
  - **poetry add xgboost==1.6.0**
- Устанавливаем dev модули:
  - **poetry add --dev black==22.3.0**
  - **poetry add --dev flake8==4.0.1**
  - **poetry add --dev mypy==0.950**
  - **poetry add --dev bandit==1.7.4**
  - **poetry add --dev pylint==2.13.8**
  - **poetry add --dev ipykernel==6.13.0**
  - **poetry add --dev nbqa==1.3.1**
- Дополнительно, т.к. у меня ранее использовалось установка локального **src** в виртуальную среду через "pip install -e .", взамен, нужно в файл **pyproject.toml** в раздел [tool.poetry] добавить следующие настройки:

```
[tool.poetry]
...
packages = [
    { include = "src", from = "." }
]
```

  - после чего выполнить **poetry install**
- Просматриваем дерево зависимостей и обновлений
  - **poetry show --no-dev**
  - **poetry show --tree**
  - **poetry show --latest** (-l)
  - **poetry show --outdated** (-o) (только имеющие апдейты)

        - по итогу, т.к. мы перешли на менеджер пакетов poetry, удаляем из проекта следующие файлы (что бы не приходилось поддерживать несколько менеджеров)
            - **requirements.txt**
            - **requirements_freeze.txt**
            - **setup.py**
        - Выполняем **poetry lock**
        - в качестве проверки, что все настроено нормально и все зависимости восстановятся в новой среде
            - закрываем IDE, удаляем из **%USERPROFILE%\AppData\Local\pypoetry\Cache\virtualenvs** директорию, соответствующую текущему проекту (mlops23regproject-...)
            - открываем IDE, из терминала выполняем
                - **poetry install**
                - **poetry shell**
            - если все отработало без ошибок, все в порядке (иногда IDE сразу не подхватывает новую виртульную среду, если так, то самое простое перезапустить IDE, либо вручную указать путь к интерпретатору в виртуальной среде)
- Корректируем файл **.gitlab-ci.yml** под poetry следующим образом:
    - Стоит обратить внимание на комментарии:
        - "Используемое кеширование на основе ключей, означает, что будет происходить проверка неизменности заданных файлов-ключей, и если они не были изменены, то директории в paths будут взяты из кэша (например если если **poetry.lock** остался прежним, то не нужно переустанавливать заново всю виртуальную среду)
        - Важно понимать, что раздел **before_script** выполняется не один раз, а перед каждым последующим стейджем (наподобии test_lint, если бы их было несколько)

```
# This file is a template, and might need editing before it works on your project.
# To contribute improvements to CI/CD templates, please follow the Development guide at:
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Python.gitlab-ci.yml
# Official language image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/python/tags/
image: python:3.9
# Change pip's cache directory to be inside the project directory since we can
# only cache local items.
```

```
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
# Pip's cache doesn't store the python packages
# https://pip.pypa.io/en/stable/topics/caching/
#
# If you want to also cache the installed packages, you have to install
# them in a virtualenv and cache it as well.
cache:
# Используемое кеширование на основе ключей, одначает, что будет происходить проверка
# неизменности заданных файлов-ключей, и если они не были изменены, то директории в paths
# будут взяты из кэша (например если если poetry.lock остался прежним, то не нужно
# переустанавливать заново всю виртуальную среду)
  key:
    files:
      - poetry.lock
      - .gitlab-ci.yml
    prefix: ${CI_JOB_NAME}
  paths:
    - .cache/pip
    - .venv
before_script:
  - python --version  # For debugging
  - python -m pip install --upgrade pip
  - pip install poetry
  - poetry config virtualenvs.in-project true
  - poetry install
test_lint:
  script:
    - poetry run flake8 src
    - poetry run mypy --ignore-missing-imports src
    - poetry run bandit src
    - poetry run pylint src
```

- Коммитим и мерджим через реквест нашу ветку ie_homework5_poetry в main:
  - **git add .**
  - **git commit -m "Complete switching from pip to poetry"**
  - **git push origin ie_homework5_poetry**

    - через веб интерфейс сливаем изменения через merge request
    - **git checkout main**
    - **git pull**
    - **git branch -d ie_homework5_poetry**
- Для дальнейшей работы с DVC создаем аккаунт в Yandex S3 (Yandex Object Storage)
    - Переходим на https://cloud.yandex.com/en-ru/services/storage
    - Get started
    - Login in using Yandex ID (авторизовался через номер телефона)
    - Cloud name: **cloud-test23team**
    - В консоли Cloud слева вверху нажимаем на точки, в выпадающем меню выбираем Object Storage
    - В открывшейся странице Create your first bucket for object storage нажимаем Create Bucket
        - name: **mlops23regs3yandex**
        - max size: **10 Gb**
        - Restrict/Restrict/Restrict/Standart
        - Create Bucket
        - По итогу предлагается завести billing account с привязкой к реальной карте (Павел упоминал, что без этого никак)
            - Account name: **test23reg**
            - Payer type: **Individual**
            - ...
            - Триал период не выбирал
            - Create
    - Далее создаем сервисный аккаунт
        - В основной консоли Cloud переходим во вкладку Service Accounts, справа вверху, на кнопку Create service account
            - Name: **dvc23reg**
            - Description: **For dvc in 23reg project**
            - Role: **storage.editor**
            - Create
        - Переходим в созданный сервисный аккаунт и кликаем (справа вверху) + Create new key -> Create static access key
            - Description: **For dvc in 23reg project**

        - Create
        - Видим наш Key ID/ Secret key, обязательно сохраняем secret key, т.е. он более показываться не будет (в случае необходимости нужно будет заново перегенерировать ключ)
- Установить и сконфигурировать DVC
    - Подготовительные шаги, а именно в git создал ветку ie_homework5_dvc, перешел в нее (далее работаем в этой ветке)
        - **git branch ie_homework5_dvc**
        - **git checkout ie_homework5_dvc**
    - Устанавливаем DVC
        - **poetry add dvc**
    - Инициализируем DVC
        - **dvc init**
    - Коммитим в гит (помимо измененных файлов poetry, добавляются следующие .dvc/.gitignore, .dvc/config, .dvcignore
        - **git add .**
        - **git commit -m "DVC init"**
- Добавить необходимые файлы данных/моделей в DVC
    - Важные примечания
        - при добавлении версионирование папки, она будет версионироваться как единая сущность, т.е. при изменении любого файла в ней, будет создаваться новая версия всей папки)
        - так же Павел упоминает, что в dvc совсем необязательно хранить все промежуточные файлы, получаемые в процессе обработки (за исключением данных получаемых долго и нужно, с внешних API, raw данных и т.п.), хотя по фокту не совсем понятно, как будут в этом случае воспроизводиться эксперименты (насколько помню из вопросов теста, в контроль dvc обязательно должны быть добавлены файлы, участвующие в stage'ах при формировании DAG), посмотрю на практике.
    - Добавляем raw файлы в отслеживание DVC
        - предварительно необходимо удалить из глобального **.gitignore** строки **data/raw/*** и **!data/raw/.gitkeep**, иначе dvc при добавлении будет выдавать ошибку вида ERROR: bad DVC file name 'data\raw\...dvc' is git-ignored.
        - **dvc add .\data\raw\pairs.csv .\data\raw\sample_submission.csv .\data\raw\test.csv .\data\raw\train.csv**
    - Коммитим изменения в гит (будут добавлены файлы 'data\raw\.gitignore' 'data\raw\test.csv.dvc' 'data\raw\pairs.csv.dvc' 'data\raw\sample_submission.csv.dvc' 'data\raw\train.csv.dvc')
        - **git add .**

        - **git commit -m "Add to dvc raw data"**
- Выбрать и настроить удаленное хранилище для DVC
    - Примечания:
        - для яндекс S3 настройки немного отличаются от классического S3 (приведена ниже)
        - dvc может хранить конфигурацию в 4-х различных уровнях (в разных конфиг файлах), подробно здесь https://dvc.org/doc/command-reference/remote/modify, самыми распространенными --project (по умолчанию, т.е. если ничего не указываем, подразумевается что работаем на этом уровне, это файл .dvc/config) и --local (файл .dvc/config.local)
            - .dvc/config синхронизируется с гит и в нем предполагается хранение общих настроек, не критичных к security (т.е. без пользовательских данных), для s3 это обычно remote по умолчанию, url, endpointurl и т.п.
            - .dvc/config.local сразу добавлен в gitignore, здесь предполагается хранение критичных данных, по типу access key, secret key и т.п.
            - по итогу dvc собирает для себя информацию из файлов всех уровней, в порядке приоритетности, так --local имеет наивысший приоритет, следующий является --project
    - Добавляем конфигурацию в файл .dvc/config через CLI:
        - **dvc remote add -d s3yandex s3://mlops23regs3yandex (где mlops23regs3yandex это имя созданного бакета)**
        - **dvc remote modify s3yandex endpointurl https://storage.yandexcloud.net**
        - при этом в файл **.dvc/config** добавится следующая конфигурация:

```
[core]
    remote = s3yandex
['remote "s3yandex"']
    url = s3://mlops23regs3yandex
    endpointurl = https://storage.yandexcloud.net
```

    - Добавляем конфигурацию в файл .dvc/config.local через CLI:
        - **dvc remote modify --local s3yandex access_key_id <key id>**
        - **dvc remote modify --local s3yandex secret_access_key <secret key>**
        - при этом в файл **.dvc/config.local** добавится следующая конфигурация:

```
['remote "s3yandex"']
    access_key_id = ...
    secret_access_key = ...
```

    - Коммитим изменения в git
        - **git add .**
        - **git commit -m "DVC config s3yandex remote"**
    - Как оказалось, для поддержки s3 необходимо доставить доп модуль dvc[s3]
        - **poetry add dvc[s3]**
        - **git add .**
        - **git commit -m "Poetry add dvc[s3]"**
    - Пушим добавленные в отслеживание файлы в remote
        - **dvc push** (в случае использования прокси, параметры прокси берутся из переменных текущей среды HTTP_PROXY, HTTPS_PROXY)
        - если все прошло успешно, проверяем через веб интерфейс яндекса, что в объектном хранилище появились файлы
- Настроить пайплайн запуска модулей с помощью DVC
    - Создаем в корне проекта файл **dvc.yaml** следующего содержания:

```
stages:
  raw_train_test_split:
    cmd: python .\src\data\raw_train_test_split.py .\data\raw\train.csv .\data\interim\split_train.csv .\data\interim\split_test.csv --
n_splits=${train_test_split.n_splits}
    deps:
      - .\data\raw\train.csv
      - .\src\data\raw_train_test_split.py
    outs:
      - .\data\interim\split_train.csv
      - .\data\interim\split_test.csv
  generate_pairs_train:
    cmd: python .\src\data\generate_pairs.py .\data\interim\split_train.csv .\data\interim\split_train_pairs.csv
.\reports\metrics_of_split_train_pairs.json --get_metrics=true
    deps:
      - .\data\interim\split_train.csv
      - .\src\data\generate_pairs.py
    outs:
      - .\data\interim\split_train_pairs.csv
    metrics:
      - .\reports\metrics_of_split_train_pairs.json:
```

```
          cache: false
  build_pair_features_train:
    cmd: python .\src\features\build_pair_features.py .\data\interim\split_train_pairs.csv .\data\processed\split_train_pair_feautures.csv true
    deps:
      - .\data\interim\split_train_pairs.csv
      - .\src\features\build_pair_features.py
    outs:
      - .\data\processed\split_train_pair_feautures.csv
  train_model:
    cmd: python .\src\models\train_model.py .\models\model_on_split_df.pkl --pair_feautures_dataset_path .\data\processed\split_train_pair_feautures.csv
    deps:
      - .\data\processed\split_train_pair_feautures.csv
      - .\src\models\train_model.py
    outs:
      - .\models\model_on_split_df.pkl
  generate_pairs_test:
    cmd: python .\src\data\generate_pairs.py .\data\interim\split_test.csv .\data\interim\split_test_pairs.csv
.\reports\metrics_of_split_test_pairs.json --get_metrics=true
    deps:
      - .\data\interim\split_test.csv
      - .\src\data\generate_pairs.py
    outs:
      - .\data\interim\split_test_pairs.csv
    metrics:
      - .\reports\metrics_of_split_test_pairs.json:
          cache: false
  build_pair_features_test:
    cmd: python .\src\features\build_pair_features.py .\data\interim\split_test_pairs.csv .\data\processed\split_test_pair_feautures.csv true
    deps:
      - .\data\interim\split_test_pairs.csv
      - .\src\features\build_pair_features.py
    outs:
      - .\data\processed\split_test_pair_feautures.csv
  predict_model:
    cmd: python .\src\models\predict_model.py .\models\model_on_split_df.pkl .\data\interim\split_test.csv .\data\processed\submission_pred_split.csv
        --pair_feautures_dataset_path .\data\processed\split_test_pair_feautures.csv
    deps:
      - .\models\model_on_split_df.pkl
```

```
      - .\data\interim\split_test.csv
      - .\data\processed\split_test_pair_feautures.csv
      - .\src\models\predict_model.py
    outs:
      - .\data\processed\submission_pred_split.csv
  evaluate:
    cmd: python .\src\models\evaluate.py .\data\interim\split_test.csv .\data\processed\submission_pred_split.csv
.\reports\metrics_of_split_test_final.json
    deps:
      - .\data\interim\split_test.csv
      - .\data\processed\submission_pred_split.csv
      - .\src\models\evaluate.py
    metrics:
      - .\reports\metrics_of_split_test_final.json:
          cache: false
```

- Создаем в корне проекта файл **params.yaml** следующего содержания:

```
train_test_split:
  n_splits: 3
```

- проверяем корректность конфигурации пайплайнов через:
  - **dvc status**
  - **dvc dag**
- выполнил dvc repro, при этом столкнулся с ошибкой "ERROR: failed to reproduce 'train_model': output 'models\model_on_split_df.pkl' is already tracked by SCM (e.g. Git). You can remove it from Git, then add to DVC." и подсказкой, что необходимо сделать для решения, делаем, To stop tracking from Git:
  - **git rm -r --cached 'models\model_on_split_df.pkl'**
  - **git commit -m "stop tracking models\model_on_split_df.pkl"**
- повторяем запуск
  - **dvc repro**
  - по итогу исполнения получаем сообщения

```
Updating lock file 'dvc.lock'

To track the changes with git, run:
```

```
    git add 'models\.gitignore' dvc.lock

To enable auto staging, run:

        dvc config core.autostage true
Use `dvc push` to send your updates to remote storage.
```

        - добавляем в гит
            - **git add .**
            - **git commit -m "Run baseline experiment with dvc"**
        - Пушим dvc в ремоут
            - dvc push
    - Примечания для понимания работы, которые удалось извлечь из нескольких экспериментов:
        - В итоге после **dvc push**:
            - в s3 запушились все промежуточные файлы (csv, model и т.п.), которые участвуют предположительно в outs/models/metrics файла dvc.yaml, при этом, что интересно, аналогичные им файлы, только с расширением .dvc созданы не были (по аналогии с тем, как они были созданы при добавлении вручную через dvc add нескольких raw датасетов); как оказалось далее, Павел упоминает про этот момент, и это является нормальным поведением, т.е. все файлы, которые генерируются стеджами в пайплайне dvc, контролируются именно через файл dvc.lock (попытка добавления их через dvc add приведет к ошибке), при этом в исключения гита они добавляются так же автоматически (если их еще нет в gitignore), а вот для версионирования остальных файлов (которые не фигурируют вообще в пайплайне, или фигурируют только в качестве deps) контролируются через dvc add и соответсвующие файлы .dvc
            - удивительно, но не запушились в s3 файлы метрик, для которых я указал доп параметр cache: false (нужно обязательно проверить на других файлах, возможно этим способом можно исключить избыточное отслеживание промежуточных файлов в s3)
        - Можем просмотреть метрики через
            - **dvc metrics show**
        - Провел небольшой эксперимент, удалив файл models\model_on_split_df.pkl
            - **dvc status** показал мне только стейджи в которых присутсвует данный файл (как в deps, так и в outs) и пометил, что файл удален

    - **dvc repro** прошелся по всем стейджам и указал для всех статус *Stage '...' didn't change, skippin*, кроме того, в котором удаленный файл находится в outs, для него указал Stage 'train_model' is cached - skipping run, checking out outputs (т.е. не стал перезапускать скрипт для его генерации, а понял, что может его восстановить из существующего кеша, что очень разумно)
- Следующий эксперимент, удалил файл metrics_of_split_test_pairs.json (для которого выставлен параметр cache: false)
    - **dvc status** показал мне только стейдж в котором фигурирует данный файл generate_pairs_test: changed outs: deleted: reports\metrics_of_split_test_pairs.json
    - **dvc repro**, по аналогии с предыдущим эксперментом, так же восстановил файл из кеша, хотя для него выставлено cache: false (подробнее поведение описано в последующих экспериментах)
    - в итоге через поиск по размеру действительно нашел этот файл в .dvc\cache\, а вот в ремоте s3 его нет, нужно понять это связано с тем, что это метрика, либо же с тем, что выставлен параметр cache: false
- Следующий эксперимент, удалил директории .dvc/cache и .dvc/tmp
    - **dvc status** подсветил все outs с пометкой changed outs: ot in cache:, причем при более детальном анализе, выяснилось, что такие статусы так же проставлены для файла data\raw\train.csv, data\raw\test.csv, но эти файлы добавлены в отслеживание через dvc add и фигурируют в пайплане только как входныйе (deps), а не выходные
    - **dvc repro** перезапустил почти весь пайплайн, кроме стейджа evaluate в котором на выходе есть только метрика (добавлена через metrics:), для которой выставлен cache: false, хотя очень странно, т.к. сами файлы были актуальны, удален был только кэш
    - по итогу исполнения, dvc предложил выполнить git add 'data\raw\train.csv.dvc' и затем dvc push
        - **git status** не видит изменений data\raw\train.csv.dvc (поменялась только его дата, но гит на это не смотрит)
        - **dvc push** сообщил, что не видит изменений и пушить нечего (это хорошо, т.к. действительно файлы все те же)
- Следующий эксперимент, удалил директории .dvc/cache и .dvc/tmp, после чего выполнил dvc pull
    - **dvc pull** загрузил 12 files fetched по факту, сами файлы он не тронул, т.к. они и так на месте, а вот кэш восстановил полностью в предыдущее состояние, за исключением файлов метрик, которые ранее были в локальном кеше, но их не было в s3, теперь их нет и в кеше
    - **dvc status** сообщает Data and pipelines are up to date. (т.е. все же параметр cache: false играет роль, хотя в некоторых моментах неоднозначно, локальный кэш все же создается)
- Следующий эксперимент, вновь удалил файл metrics_of_split_test_pairs.json (для которого выставлен параметр cache: false, и теперь его действительно нет в локальном кеше)

        - **dvc status** показал мне только стейдж в котором фигурирует данный файл generate_pairs_test: changed outs: deleted: reports\metrics_of_split_test_pairs.json (т.е. в сравнении с предыдущим случаем, поведение не изменилось)
        - **dvc repro** перезапустил только пайплайн Running stage 'generate_pairs_test': на выходе которого генерируется данная метрика, все остальные пропущены, т.к. изменений по ним не требуется.
        - в локальный кэш добавился аналогичный файл (не смотря на cache: false)
    - Следующий эксперимент, в dvc.yaml выставляем для .\data\processed\submission_pred_split.csv параметр cache: false
        - **git add .**
        - **git commit -m "DVC set cache: false for file submission_pred_split.csv"**
        - **dvc status** выдает Data and pipelines are up to date.
        - удалил файл **.\data\processed\submission_pred_split.csv**
        - **dvc status** сообщает, что файл удален и подсвечивает 2 стейджа, в которых он в качестве outs и deps
        - **dvc pull** сообщает что Everything is up to date. (но сейчас понял, что файл есть в локальном кеше)
        - **dvc repro** восстановил файл из кеша
        - повторяем удаление **.\data\processed\submission_pred_split.csv**
        - удаляем директории **.dvc/cache и .dvc/tm**p
        - **dvc pull** сообщает о 11 files fetched (в прошлый раз было 12), т.е. даже с учетом того, что файл в s3 есть, он не был подгружен в кэш, т.к. для него выставлен параметр cache: false
        - **dvc repro** перезапустил только стейдж .\src\models\predict_model.py который и генерирует удаленный файл
    - Итого, получается, что параметр cache: false как раз позволяет исключить из пересылки на внешнее хранилище файлов, для которых это делать нецелесообразно (при этом нужно иметь введу, что локально в кэш эти файлы сохраняются, но часто это даже плюс, чем минус).
- С учетом итогов экспериментов, модифицирую файл **dvc.yaml** следующим образом (что бы исключить излишние пересылки на s3, по факту файлы splits там тоже не нужны, но т.к. на данном этапе у меня нет данных с внешних апи, то оставил их в качестве искусственного примера)

```
stages:
  raw_train_test_split:
    cmd: python .\src\data\raw_train_test_split.py .\data\raw\train.csv .\data\interim\split_train.csv .\data\interim\split_test.csv --
n_splits=${train_test_split.n_splits}
    deps:
      - .\data\raw\train.csv
```

```
      - .\src\data\raw_train_test_split.py
    outs:
      - .\data\interim\split_train.csv
      - .\data\interim\split_test.csv
  generate_pairs_train:
    cmd: python .\src\data\generate_pairs.py .\data\interim\split_train.csv .\data\interim\split_train_pairs.csv
.\reports\metrics_of_split_train_pairs.json --get_metrics=true
    deps:
      - .\data\interim\split_train.csv
      - .\src\data\generate_pairs.py
    outs:
      - .\data\interim\split_train_pairs.csv:
          cache: false
    metrics:
      - .\reports\metrics_of_split_train_pairs.json:
          cache: false
  build_pair_features_train:
    cmd: python .\src\features\build_pair_features.py .\data\interim\split_train_pairs.csv .\data\processed\split_train_pair_feautures.csv true
    deps:
      - .\data\interim\split_train_pairs.csv
      - .\src\features\build_pair_features.py
    outs:
      - .\data\processed\split_train_pair_feautures.csv:
          cache: false
  train_model:
    cmd: python .\src\models\train_model.py .\models\model_on_split_df.pkl --pair_feautures_dataset_path .\data\processed\split_train_pair_feautures.csv
    deps:
      - .\data\processed\split_train_pair_feautures.csv
      - .\src\models\train_model.py
    outs:
      - .\models\model_on_split_df.pkl
  generate_pairs_test:
    cmd: python .\src\data\generate_pairs.py .\data\interim\split_test.csv .\data\interim\split_test_pairs.csv
.\reports\metrics_of_split_test_pairs.json --get_metrics=true
    deps:
      - .\data\interim\split_test.csv
      - .\src\data\generate_pairs.py
    outs:
```

```
      - .\data\interim\split_test_pairs.csv:
          cache: false
    metrics:
      - .\reports\metrics_of_split_test_pairs.json:
          cache: false
  build_pair_features_test:
    cmd: python .\src\features\build_pair_features.py .\data\interim\split_test_pairs.csv .\data\processed\split_test_pair_feautures.csv true
    deps:
      - .\data\interim\split_test_pairs.csv
      - .\src\features\build_pair_features.py
    outs:
      - .\data\processed\split_test_pair_feautures.csv:
          cache: false
  predict_model:
    cmd: python .\src\models\predict_model.py .\models\model_on_split_df.pkl .\data\interim\split_test.csv .\data\processed\submission_pred_split.csv
        --pair_feautures_dataset_path .\data\processed\split_test_pair_feautures.csv
    deps:
      - .\models\model_on_split_df.pkl
      - .\data\interim\split_test.csv
      - .\data\processed\split_test_pair_feautures.csv
      - .\src\models\predict_model.py
    outs:
      - .\data\processed\submission_pred_split.csv:
          cache: false
  evaluate:
    cmd: python .\src\models\evaluate.py .\data\interim\split_test.csv .\data\processed\submission_pred_split.csv
.\reports\metrics_of_split_test_final.json
    deps:
      - .\data\interim\split_test.csv
      - .\data\processed\submission_pred_split.csv
      - .\src\models\evaluate.py
    metrics:
      - .\reports\metrics_of_split_test_final.json:
          cache: false
```

        - **git add .**
        - **git commit -m "Final dvc configuration on this homework"**
- По итогу урока пушим нашу ветку в ориджин и проводим мердж реквест в main

- **git push origin ie_homework5_dvc**
- через веб интерфейс делаем и исполняем **мердж реквест в main**
- **git checkout main**
- **git pull**
- **git branch -d ie_homework5_dvc**

Неделя 6. Инструменты автоматизации ML исследований, DVC + MLFlow

Видео:
https://youtu.be/AHBzCQacpmQ
Презентация:

MLOps Неделя 6. MLFlow.pdf 1 MB

ДЗ:

MLOps Task 6.docx 19 kB

- Установить MLflow
- Настроить логирование:
  - параметров
  - метрик
  - моделей
- Синхронизировать удаленный репозиторий с dagshub
- Подключить проект к MLflow tracking remote (поднять собственный или использовать с DagsHub)

- Провести серию запусков эксперимента, изучить интерфейс MLflow

Ссылка на репозиторий, на котором Автор курса демонстрирует практические шаги: https://gitlab.com/m1f/mlops-course

Отработка предлагаемого подхода по версионированию на тестовом репозитории:

- Описание предлагаемого подхода по версионированию через DVC (выполнял на тестовом репозитории, что бы быстрее разобрать разные варианты и не заморачиваться сильно на документирование шагов, далее по ходу проекту обязательно будут реальные задачи по версионированию, которые будут задокументированы)
  - Предлагается решение по версионированию через комиты и дополнительные теги (теги насколько я понимаю нужны для удобства переключения между ними)
    - Краткая справка, обычно же теги используются для меток каких то важных коммитов, в частности очень часто для версионирования, причем бывает два вида тегов легковесные и аннотированные (подробнее https://git-scm.com/book/ru/v2/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%A0%D0%B0%D0%B1%D0%BE%D1%82%D0%B0-%D1%81-%D1%82%D0%B5%D0%B3%D0%B0%D0%BC%D0%B8)
      - Легковесный тег — это что-то очень похожее на ветку, которая не изменяется — просто указатель на определённый коммит.
      - Аннотированные теги (создаются с параметром -a) - хранятся в базе данных Git как полноценные объекты. Они имеют контрольную сумму, содержат имя автора, его e-mail и дату создания, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG).
      - Обычно рекомендуется создавать аннотированные теги, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные.
  - Последовательность действий для одного эксперимента:
    - создаем ветку для экспериментов
      - **git branch ie_experiments_hyperparameters**
      - **git checkout ie_experiments_hyperparameters**
    - изменяем какой-либо гиперпараметр (допустим это будет выдуманный n_splits)

        - перезапускаем пайплайн
            - **dvc repro**
        - коммитим изменения в гит, и дополнительно выставляем тег с описание (предложенная схема ниже, выдаю ее "как дана", хотя на практике я бы давал более осознанные названия тегам, по типу ExpHyp01 или ExpHyp... т.п.)
            - **git commit -m "Run experiments with tag"**
            - **git tag -a v01 -m "n_splits=2"**
        - Выполняем dvc push (если необходимо, по идее все изменения будут накапливаться в кеше и можно будет сделать финальный push, но не проверял на практике, в дальнейшем, при необходимости стоит проверить)
    - Далее по аналогии создаем следующий эксперимент
    - Для того, что бы перейти к одному из предыдущих экспериментов, выполняем
        - **git checkout v01** (где v1 это название тега)
        - **dvc checkout**
        - Важным примечанием здесь является то, то при этом мы попадаем в состояние гита HEAD detached, мы как бы уходим с ветки, и переключаемся именно на тегированный коммит (т.к. тег по сути некоторый аналог ветки, т.е. просто указатель на коммит)
            - в этом состоянии лучше не производить никаких изменений (если они необходимы, то это делается черз создание уже реальной ветки из этого состояния)
            - когда закончили переключаться по тегам, вернуться в нормальное состояние можно через git checkout на исходную или какую либо другую ветку (именно ветку, а не последний созданный тег), проверить нормальный статус можно через **git status**
    - Важное примечание, сами теги при обычном пуше ветки экспериментов в ориджин, не передаются в него, что бы их запушить выполняем
        - Пометка про то, как сразу запушить все теги в origin выполняем
            - **git push --tags** (так же пушить можно по отдельности, через указание имени тега, например git push v1)

Отработка ДЗ на командном репозитории:

- Установить MLflow
    - Подготовительные шаги, а именно в git создал ветку ie_homework6_mlflow, перешел в нее (далее работаем в этой ветке)

        - **git branch ie_homework6_mlflow**
        - **git checkout ie_homework6_mlflow**
    - Устнавливаем mlflow через poetry
        - **poetry add mlflow**
        - **poetry add matplotlib** (опционально, рекомендуется для автолога mlflow, с помощью него будут генерироваться графики)
    - Проверить работу можно запустив через одну из следующих команд
        - **mlflow server --backend-store-uri <PATH>** (где <PATH> до директории в которой будет храниться вся необходимая информация для mlflow, например можно указать ./mlflow)
        - **mlflow ai** (по сути тоже самое, только с использованием директории по умолчанию ./mlruns)
        - остановить сервер можно через Ctrl+C
        - по итогу проверки, директории можно удалить
- Настроить логирование (сценарии развертывания mlflow приведены здесь https://www.mlflow.org/docs/latest/tracking.html), провести серию запусков эксперимента, изучить интерфейс MLflow
    - Общие настройки и замечания
        - трекинг через млфлоу происходит прямо в коде модулей, соответственно нужно в необходимых модулях импортировать его
            - i**mport mlflow**
        - в текущем уроке/ДЗ рассматривается 2 варианта запуска и трекинга:
            - если мы не указываем никаких доп настроек, то предполагается трекинг через файлы и директорию по умолчанию **./mlruns**, при этом вариант сам вебсервер mlflow может быть не запущен, после его запуска он сам подтянет все прошедшие эксперименты
            - через указание в начале модуля uri сервера, например для локального сервера mlflow.set_tracking_uri("http://127.0.0.1:5000") (в этом случае сервер во время эксперимента должен работать)
        - mlflow подразумевает разбивку на серии экспериментов с каким-либо именем (если имя не указываем, используется defaults), в каждой такой серии может быть множество запусков (runs) с различными параметрами, метриками, артефактами и т.п.), для задания имени серии экспериментов, в начале модуля указываем
            - mlflow.set_experiment("exps name")

        - стоит отметить, что запуски логируются для каждого скрипта отдельными строками, что скорей всего часто будет неудобно, Павел упоминает об этом и для этих целей в видеоуроке объединяет тренировку и евалюейшн модели в один скрипт (ранее он разбивал на несколько в соответсвии с подходом кукикаттер шаблона), что бы в одной строке была информация по медели и метрикам на отложенной выборке; честно говоря походит на костыли, скорей всего должен быть какой-либо способ нормально решить эту проблему; нужно будет разобраться в этом вопросе в будущем, при реальной необходимости проведения экспериментов
    - логирование параметров
        - производится через **mlflow.log_params(<dict with params>)**
        - так же параметры логируются при использовании автолога моделей (по факту в этом случае логироется очень много различных параметров, что немного усложняет чтение, и скорей всего целесообразней использовать адресное логирование параметров, но как минимум посмотреть возможности логирования через автолог стоит, т.к. здесь же дополнительно собираются feature importance и т.п.)
            - **mlflow.xgboost.autolog()** (где xgboost это тип модели, поддерживаются все основные), указывается в начале модулей после импортов (но возможно допустимо иное место)
            - **mlflow.autolog()** (возможен такой вариант, с автоопределением основных типов моделей
            - функции имеют набор параметров, которые можно менять (трекать модель или нет, и т.п.)
    - логирование метрик
        - Примечание: некоторые названия метрик у меня содержали скобки, mlflow не принял такой формат (Names may only contain alphanumerics, underscores (_), dashes (-), periods (.), spaces ( ), and slashes (/).), переназвал метрики.
        - **mlflow.log_metrics(<dict with metrics>)**
    - логирование моделей
        - Примечание: логирование моделей ведется в отдельной вкладке mlflow, и ее работы обязательно необходима бд, допустимо использовать sqllite (это бд по факту из одного файла, доступного на чтение только из одного процесса), пример использования sqllite:
            - **mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./mlruns/artifacts --host 0.0.0.0 (при этом файл mlflow.db создается в корне проекта)**
        - пример логирования моделей

        - **mlflow.xgboost.log_model(model, artifact_path="output_model_path", registered_model_name="my_baseline_xboost")**, где model переменная с текущей моделью, artifact_path путь до ее сохранения, функция вызывается после тренировки и сохранения модели
    - логирование артефактов
        - артефактом является какой либо файл (например файл модели, если мы не хотим его сохранять через логирования модели, либо что то еще), логируется следующим образом (по факту копия файла сохраняется в run эксперимента mlflow)
            - **mlflow.log_artifact(**<path to artifact>), вызывается уже после сохранения артефакта
- По итогу тестовой настройки локального трекинга и проведения пары экспериментов, коммитим изменения в гит для истории:
    - оказалось, что в гитигноре, присутствовала строка *.db, что бы файл mlflow.db добавился в контроль гита, модифицируем следующим образом

```
*.db
!mlflow.db
```

    - **git add .**
    - **git commit -m "Complete a few test runs with: mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./mlruns/artifacts --host 0.0.0.0"**
- Синхронизировать удаленный репозиторий с dagshub (https://dagshub.com/)
    - Павел все наглядно показывает в видеоуроке (ссылка с таймметкой https://youtu.be/AHBzCQacpmQ?t=3630), в рамках текущего ДЗ не стал повторять все эти шаги, т.к. все предельно понятно по видео, а в последующем развитии проекта, будет строится отдельная локальная инфраструктура; по сути dagshub это:
        - git репозиторий (который автоматически и в рантайме может интегрироваться с основным репозиторием например на гитлабе или гитхабе),
        - с встроенными ML плюшками, в виде
            - нативной и наглядной поддержки DVC,
            - собственным хранилищем S3 (10 Гб бесплатно), либо интеграцией с существующим,
            - интегрированным в единый web интерфейс mlflow
            - уже готовым MLflow tracking remote сервером, в который напрямую можно слать эксперименты из кода (что очень удобно)
            - и многим другим

    - на самом деле действительно очень удобный ресурс для ведения проектов среднего размера (когда уже нужно все оборачивать в виде продакшена, а не ноутбуков на коленке, но проект не настолько большой, что бы разворачивать и поддерживать в рабочем состоянии свою инфраструктуру MLFlow, S3 и т.п.), в подобных проектах имеет смысл использовать.
- Подключить проект к MLflow tracking remote (поднять собственный или использовать с DagsHub)
    - решено в рамках коммита "Complete a few test runs with: mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root ./mlruns/artifacts --host 0.0.0.0", т.к. особой разницы на данном этапе нет с точки зрения кода, а более приближенную к реальности большого проекта инфраструктуру будем поднимать на докере в следующем ДЗ
- По итогу ДЗ пушим в ориджин и мерджим с main (через реквест) ветку урока
    - **git push origin ie_homework6_mlflow**
    - мерджим через реквест на гитлабе
    - **git checkout main**
    - **git pull**
    - **git branch -d ie_homework6_mlflow**

Неделя 7. Разработка сервиса на базе ML моделей. Контейнеризация с Docker.

Видео:
https://youtu.be/KZ5Cdevd-b8
Презентация:
Отсутствует

ДЗ:

MLOps Task 7.docx 19 kB

- Развернуть MLflow по 4 или 5 сценарию, используя СУБД и хранилице артифактов на свой выбор. Вместо удаленного хоста можно использовать Docker контейнеры.
- Провести эксперименты с логированием модели
- Запустить сервис с моделью стандартными средствами MLflow стандартным способом или через Docker
- Создать собственный микросервис для модели используя Docker и Flask/FastAPI

Ссылка на репозитории, на которых Автор курса демонстрирует практические шаги:
https://gitlab.com/m1f/mlops-course
https://gitlab.com/m1f/mlops_cd/

Отработка элементарных действий с докером на тестовом репозитории (на основе https://youtu.be/QF4ZF857m44):

- Предполагаем, что докер установлен с системе
- Создаем директорию с тестовым проектом
- **python -m venv .venv**
- **Создаем файл app.py**

```
print("Hello world")
```

- Пытаемся создать докер образ (выполняем находясь в нашей директории проекта)
  - **docker build -t hello-world .**
    - -t - флаг тега, по сути имени контейнера
    - . - путь
  - получаем ошибку, что необходим докерфайл, т.к. докеру необходим докерфайл, определяющий алгоритм/настройки сборки образа
- Создаем в корне проекта Dockerfile

```
# Выбираем image, причем:
# до двоеточия, мы указываем своего рода проект (в данном случае официальный проект python)
# после двоеточия указываем тег конкретного образа
FROM python:3.9
```

```
# RUN - исполнение команды
RUN mkdir -p /urs/src/app/
# Задаем рабочий каталог, все последующие команды, действия будут выпоняться находясь в ней
WORKDIR /usr/src/app/

# первый аргумент откуда (на хостовой машине)
# второй аргумент куда (в докере)
COPY . /usr/src/app/

# CMD - что нужно делать, когда мы запусти контейнер, т.е. не выполняется при создании образа
# указывается массив из последовательности параметров в данном случае выпонится "python app.py"
# Важное замечание, есть похожаня команда ENTRYPOINT, т.е. например ENTRYPOINT ["python", "app.py"]
# При этом:
# CMD запускает команды через shell (через оболочку bin.sh)
# ENTRYPOINT выполняет команды без шелла (зачем это может быть нужно, пойму в будущем)
CMD ["python", "app.py"]
```

- повторяем сборку образа
  - **docker build -t hello-world .**
  - по итогу послойно собрался необходимы образ:
    - закачиваются все слои выбранного исходного образа (в нашем случае python:3.9)
    - поверх них, создаются наши слои, причем каждая команда из файла Dokerfile (RUN, WORKDIR, COPY и т.д.) является отдельным слоем
- Выполняем docker images (примечания, в видео, в списке есть так же образ python 3.9, но у меня он не отображается, судя по мануалу, вывести все промежуточные образы можно через docker images -a):

```
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
hello-world   latest    a9937e00379b   25 minutes ago   932MB
```

- Запускаем контейнеры:
  - **docker run hello-world**
  - по итогу команда выполнилась, контейнер запустился, вывел в консоль Hello world и остановился (остановился, но не удалился, т.е. он есть в незапущенном состоянии)
  - убеждаемся, что контейнер остановился через docker ps (в выводе отсутствуют запущенные контейнеры)
  - а вот **docker ps -a** (или --all) выведет список в том числе остановленных контейнеров (с рандомно сгенерированным именем)

```
CONTAINER ID   IMAGE          COMMAND           CREATED         STATUS                     PORTS     NAMES
92abcc9b634a   hello-world    "python app.py"   5 minutes ago   Exited (0) 5 minutes ago             admiring_ride
```

    - запускаем контейнер с указанием имени (будет создан другой контейнер, но на основе того же образа)
        - **docker run --name hello hello-world**
        - **docker ps -a**

```
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS                      PORTS     NAMES
e3f7488ce82e   hello-world    "python app.py"   8 seconds ago    Exited (0) 7 seconds ago              hello
92abcc9b634a   hello-world    "python app.py"   29 minutes ago   Exited (0) 29 minutes ago             admiring_ride
```

- Удаляем контейнер
    - Один контейнер
        - **docker rm <container id or name>**
    - Несколько контейнеров
        - docker ps с флагом -q выдает нам только id контейнеров (дополнительно можно задать фильтр)
        - мы передаем результат docker ps с флагом -q в docker rm, тем самым удаляя все необходимые нам контейнеры, пример полного удаления всех контейнеров
            - **docker rm $(docker ps -qa)**
- Навыки работы с запущенными контейнерами
    - Модифицируем наш app.py для бесконечной работы

```
import time
while True:
    print("Hello world")
    time.sleep(1)
```

    - пересобираем образ с обновленным приложением
        - **docker build -t hello-world .** (обращаем внимание, что теперь скачивания базового образа python:3.6 не происходит, т.к. он уже у нас есть в локальном репозитории)

    - запускаем в фоне с параметром -d (если запустим как ранее, без -d, то наш терминал будет занят до момента остановки контейнера, причем, что интересно для кода выше, мы не будем в этом терминале видеть Hello world через каждую секунду; опытным путем выяснил, что выводы из докера в текущий терминал передаются только по результатам штатного завершения контейнера, а не интерактивно, т.е. в нашем случае с бесконечным циклом, мы штатно не завершаем наш контейнер, его приходится стопить вручную из соседнего терминала, в итоге мы не получаем выводов, а вот если сделать цикл из конечного числа итераций, то по итогу штатного завершения разом будут выведены все Hello world)
        - **docker run --name hello -d hello-world**
    - **docker ps**

```
CONTAINER ID   IMAGE         COMMAND           CREATED         STATUS         PORTS     NAMES
86fb360f1c6b   hello-world   "python app.py"   3 minutes ago   Up 3 minutes             hello
```

    - останавливаем контейнер
        - **docker stop <container id or name>**
    - **docker ps -a** (обращаем внимание что для остановленных вручную контейнеров в STATUS код возврата не 0, а Exited (137) = принудительная остановка)

```
CONTAINER ID   IMAGE         COMMAND           CREATED         STATUS                       PORTS     NAMES
86fb360f1c6b   hello-world   "python app.py"   6 minutes ago   Exited (137) 31 seconds ago            hello
```

    - удаляем контейнер через **docker rm <container id or name>**
- Запускаем контейнер так, что бы после завершения (принудительного или штатного) он автоматически удалился (без необходимости выполнения вручную docker rm), для этого в docker run есть параметр --rm
    - **docker run --name hello -d --rm hello-world**
    - **docker ps** (убеждаемся что запущен)
    - **docker stop hello**
    - **docker ps -a** (убеждаемся что отсутсвует, т.е. удален автоматически)
- Рассматриваем пример WEB приложения
    - **git clone https://github.com/amatiashov/YT-Docker-lesson.git**
    - **cd .\YT-Docker-lesson\web-hello-world**

- это приложение на Flask, когда заходим на веб страницу приложения через браузер, то веб сервер открывает файл response.json, берет оттда значение поля payload (по умолчанию там просто строка Hello, World!!!.) к нему добавляется текущее время и возвращается в качестве ответной веб страницы (из одной строки) браузеру
- **Dockerfile**

```
FROM python:3.6

RUN mkdir -p /usr/src/app/
WORKDIR /usr/src/app/

COPY . /usr/src/app/
RUN pip install --no-cache-dir -r requirements.txt

# декларируем, что этот порт внутри контейнера может быть проброшен наружу
# важно понимать, что этого не достаточно, что бы порт пробросился фактичеси
# файктический проброс происходит через параметр
# -p <порт на хостовой машине>:<порт внутри контейнера> команды docker run
EXPOSE 8080

CMD ["python", "app.py"]
```

- собираем образ (если работаем через прокси, варианты настройки здесь https://docs.docker.com/network/proxy/)
  - **docker build -t web-hello .**
- Запускаем с перенаправлением портов
  - **docker run --rm --name web -p 8088:8080 web-hello**
  - переходим в браузере http://127.0.0.1:8088/
- Решаем проблему с неверной таймзоноый, через задание переменной окружения, есть несколько вариантов задания ENV
  - на уровне образа, через ENV <env_name> < value> в докерфайле, например
    - **ENV TZ Europe/Moscow**
    - подходит для относительно статичных переменных окружения, которые не предполагают необходимости изменения для различных контейнеров
  - на уровне контейнера, через параметр -e <env_name>=< value> при выполнении docker run, например
    - **docker run --rm --name web -p 8088:8080 -e TZ=Europe/Moscow web-hello**

        - соответственно подходит для случаем, когда нам может понадобиться варьировать в зависимости от контейнеров
    - Варианты сохранения данных после установки/удаления контейнеров
        - Монтирование директории на хосте (дан комментарий, что сейчас используется довольно редко)
            - через ключ -v <абсолютный путь на хостовой машине>:<абсолютный путь в докер контейнере>, например
                - **docker run --rm --name web -p 8088:8080 -e TZ=Europe/Moscow -v d/Jupyter/mlops-docker-testrep/YT-Docker-lesson/web-hello-world/resources:/usr/src/app/resources web-hello**
        - Монтирование volume (
            - просмотр текущих volume
                - **docker volume ls**
            - создание docker volume create <name>
                - **docker volume create web**
            - монтируем по аналогии через -v <volume name>:<абсолютный путь в докер контейнере>, например
                - **docker run --rm --name web -p 8088:8080 -e TZ=Europe/Moscow -v web:/usr/src/app/resources web-hello**
- Пример развертывания временной базы данных
    - Код предложенного приложения, взаимодействующего с бд монго через локалхост и порт 27017 здесь \YT-Docker-lesson\mongodb\
    - Для того что бы развернуть временную монгу на данном порту, просто выполняем
        - **docker run --rm -d -p 27017:27017 mongo**
        - причем в локальном репозитории у нас нет образа монги, но он есть в докерхабе, и докер автоматически его выкачает и запустит
        - т.к. мы не монтировали никаких вольюмов и прочего, база данных будет затираться каждый раз при удалении контейнера, что нам и нужно
        - все супер удобно и просто, никаких хвостов в системе

Заметки по инфраструктуре Docker:

- Заметки по по докеру (основные источники - https://youtu.be/QF4ZF857m44)
    - Image and Containers

        - Image это образ (в незапущенном состоянии) упакованного "приложения" с урезанной ОС, зависимостями, настройками, самим приложением и т.п.
        - Container это работающее "приложение", созданное на базе Image, т.е. на основе одного Image, можно запустить сколько угодно контейнеров
        - Важно! для контейнера, образ, является read only системой, он никаким образом, не может его изменить (т.е. допустим, создаем на основе образа контейнер и в нем удаляем все системные файлы, по итогу данный удаляться только в сущности этого конкретного контейнера, на исходный образ это никак не повлияет)
    - Принципы Image
        - Образы представляют собой "слоеный пирог", например
            - у нас есть голый образ ubuntu, это один слой (и в том числе это полноценный образ, который мы можем запустить)
            - на этот образ мы можем поставить nginx или mongodb или python, в каждом случае это будет уже 2-й слой, по итогу это будут дополнительные 3 образа из 2-х слоев каждый
            - допустим на контейнер python мы ставим свое приложение, это будет уже 3-й слой и как следствие на выходе имеем образ из 3-х слоев.
        - Каждый слой содержит себе информацию, в объеме изменений, в сравнении с предыдущим слоем, на основе которого он основан
        - Определен единый механизм сборки приложений/имеджей (т.е. набор команд для его создания)
        - Реестры образов (Image registry)
            - локальный реестр (реестр наших локально созданных образов)
            - Docker Hub (https://hub.docker.com/), публичный реестр образов (в нем находятся как образы, созданные и поддерживаемые разработчиками ПО и докера, так и образы от любых зарегистрированных пользователей, т.е. можно создать свой образы и опубликовать его на докер хаб)
    - Принципы Containers
        - контейнер может быть запущен через
            - **docker run**
            - **через docker compose**
        - контейнер работает до тех пор, пока работает приложение внутри него (т.е. если в качестве приложения будет скрипт на питоне, который печатает Hello world, то при запуске контейнера, он запустится, выведет в консоль указанную строку и сразу остановится)

        - остановленный контейнер не удаляется, а остается в системе в незапущенном состоянии (просмотреть остановленные можно через docker ps -a)
        - если мы при запуске контейнера не указываем имя (указывается через флаг --name при docker run), докер сам выбирает имя (рандомно сгенерированные из 2-х слов через _)
        - контейнер запускается в полностью изолированной среде (это касается сетевых настроек, дисков и т.п.), т.е. если мы например явно не указываем соответствующие пробросы портов или дисков, то они не будут доступны извне
    - Итого, концепции Docker
        - единый механизм сборки образов
        - единый механизм доставки образов
            - можем создать образ и опубликовать на докер хабе
            - можем скачать в локальный реестр образов любой образ с докер хаба (чей-то, либо свой, ранее опубликованный)
        - единый механизм запуска контейнеров на основе образов (из локального реестра)
    - Основные консольные команды Docker
        - docker images - просмотр локального реестра образов (с -a показывает в том числе промежуточные)
            - теги <none> появляются при многократных билдах образов, если дилдится образ с тем же тегом, то сам тег просто переезжает на новый измененный слой, а тег прошлой версии слоя сбрасывается на none
        - удаление images
            - **docker rmi <REPOSITORY or ID>**
            - **docker rmi $(docker images -q)** - удаляет все контейнеры
        - **docker ps** - просмотр запущенных контейнеров (с -a показывает все)
- Заметки по Docker compose
    - Идет в составе Docker desktop, но если докер ставился на сервере, то возможно нужно будет дополнительно доставить docker-compose
    - Имя файла должно быть docker-compose.yaml
    - По сути файл, описывающий запуск проекта, состоящего из нескольких докер образов/контейнеров может содержать в себе
        - выбор исходного образа
            - выбор существующего образа исходника
            - билд нового образа (на основе существующих докерфайлов)
        - создание volume

        - запуск контейнеров с заданием множества параметров (по аналогии с теми, которые мы можем задавать при использовании docker run)
        - и т.д.
    - Параметры могут быть заданы как переменные, через ${ENV_NAME}, при этом значения берутся из файла .env в директории проекта
    - Важной особенностью является внутренняя изолированная инфраструктура внутри compose, например, в одной внутренней сети, контейнеры могут общаться друг с другом по именам сервиса и портам, без необходимости публикации этих портов наружу на хостовую машину (в том числе это полезно в плане безопасности)
    - Стандартная команда запуска всех контейнеров на основе docker-compose.yaml (из текущей директории)
        - **docker-compose up -d**
    - Остановка всех контейнеров
        - **docker-compose down**
    - Версия (например version: '3.8'), означает версию спецификации файла, и не имеет ничего общего с версией проекта (подробнее здесь https://docs.docker.com/compose/compose-file/compose-versioning/)
- Основные варианты запуска проекта на целевом сервере (например на выделенном сервере в хостинге)
    - через git репозиторий и сборку на стороне сервера
        - т.е. мы ведем свой проект у себя локально, все тестируем так же локально, пушим наш код проекта с Dockerfile, docker-compose.yaml (в котором для части сервисов образ строится через ключевое слово build) и т.д. в удаленный репозиторий (origin)
        - на целевом сервере клонируем репозиторий и запускаем проект через docker-compose up -d (при этом наши кастомные образы собираются на сервере и хранятся только в локальном реестре)
        - апдейт проекта происходит через изменение кода в удаленном гит репозитории, git pull на сервере и пересборке проекта
    - через docker hub
        - предварительная сборка образов, два варианта
            - Вариант 1. Мы предварительно, локально собирем все наши образы и пушим их на docker hub
                - для одного образа
                    - **docker build -t <login docker hub>/<name> .**
                    - **docker login**
                    - **docker push <login docker hub>/<name>**

        - Вариант 2. На гитхабе/гитлабе можно настроить, что бы при пуше в какую-либо ветку, на докерхабе автоматически пересобирались образы
    - на удаленном сервере нам нужен исключительно файл docker-compose.yaml, где для всех сервисов мы используем ключевое слово image(вместо build), и остальные настройки, далее поднимаем как обычно через docker-compose
    - апдейт проекта происходит, через изменение локально кода, тестирование, пуш в докерхаб новых образов (локально или через автоматический процесс гитхаб/гатлаб), далее на целевом сервере делаем docker pull

Отработка ДЗ на командном репозитории:

- Установка докера под Windows (https://docs.docker.com/desktop/windows/install/)
- Подготовительные шаги
    - git создал ветку ie_homework6_mlflow, перешел в нее (далее работаем в этой ветке)
        - **git branch ie_homework7_docker**
        - **git checkout ie_homework7_docker**
    - В корне проекта создаем:
        - (если еще не создано) файлы для переменных сред
            - **.env** (обязательно! должен быть в .gitignore)
            - **.env.example** (задается только структура оригинального .env, без самих значений, этот файл храниться контролируется гитом)
        - Директорию Docker (для хранения докерфайлов, volume и т.п.)
        - Файл docker-compose.yaml
- Развёртывание локального S3 minio в докер образе в связке с nginx (аналогично предложенному в видеолекции)
    - Предварительные пояснения
        - minio имеет 2 основных порта
            - Console, по факту порт web-интерфейса для настройки через браузер (по умолчанию 9001)
            - API, порт для взаимодействия по REST API (по умолчанию 9000)
            - порты публикуются только в рамках сети с именем "s3" внутри инфраструктуры docker-compose
        - nginx проксирует через себя аналогичные порты

        - порты доступны как внутри сети с именем "s3", так и проброшены наружу, на хостовую машину
    - Заносим в **docker-compose.yaml**
        - Примечания:
            - В видеолекции Павел задавал в docker-compose.yaml для minio переменные MINIO_ACCESS_KEY и MINIO_SECRET_KEY, при этом как оба значения использовались minioadmin/minioadmin, далее в других контейнерах для взаимодействия использовались эти же переменные. Но видимо это ошибка, по факту, переменные MINIO_ACCESS_KEY и MINIO_SECRET_KEY в контейнере minio, ничего не меняют, сам минио их абсолютно никак не использует, во всех мануалах, изначально задаются именно MINIO_ROOT_USER и MINIO_ROOT_PASSWORD (т.е. первоначальный логин/пасс для администратора). При этом, если их не задать, то эти значения по умолчанию будут как раз minioadmin/minioadmin, и как позже подтвердилось на практике, эти же учетные данные позволяют авторизоваться и писать/читать в s3, как будто через ACCESS_KEY (=логин) и SECRET_KEY (=пароль). При этом, это конечно неверный подход, УЗ администратора отдельная сущность, нежели API ключи. По итогу в настройке ниже, мы идем по пути задания первоначальных логина/пароля для администратора, далее в вебинтерфейсе генерируем API ключи и уже их значения задаем в .env для дальнейшего использования в смежных сервисах (в dvc и mlflow).

```
version: '3.8'

services:

  minio:
    container_name: minio
    hostname: minio
    image: minio/minio:RELEASE.2022-05-26T05-48-41Z.hotfix.15f13935a
    command: server --console-address ":9001" /data/
    expose:
      - "9000"
      - "9001"
    networks:
      - s3
    environment:
      MINIO_ROOT_USER: ${MINIO_ROOT_USER}
      MINIO_ROOT_PASSWORD: ${MINIO_ROOT_PASSWORD}
    healthcheck:
      test:
```

```
        [
          "CMD",
          "curl",
          "-f",
          "http://localhost:9000/minio/health/live"
        ]
      interval: 30s
      timeout: 20s
      retries: 3
    volumes:
      - ./Docker/minio/:/data

  nginx:
    image: nginx:1.19.2-alpine
    container_name: nginx
    hostname: nginx
    volumes:
      - ./Docker/nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "9000:9000"
      - "9001:9001"
    networks:
      - s3
    depends_on:
      - minio

networks:
  s3:
    driver: bridge
```

- Создаем файл **./Docker/nginx.conf**

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;
```

```
events {
    worker_connections  4096;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;
    sendfile        on;
    keepalive_timeout  65;

    # include /etc/nginx/conf.d/*.conf;

    upstream minio {
        server minio:9000;
    }

    upstream console {
        ip_hash;
        server minio:9001;
    }

    server {
        listen       9000;
        listen  [::]:9000;
        server_name  localhost;

        # To allow special characters in headers
        ignore_invalid_headers off;
        # Allow any size file to be uploaded.
        # Set to a value such as 1000m; to restrict file size to a specific value
        client_max_body_size 0;
        # To disable buffering
```

```
        proxy_buffering off;
        proxy_request_buffering off;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;

            proxy_connect_timeout 300;
            # Default is HTTP/1, keepalive is only enabled in HTTP/1.1
            proxy_http_version 1.1;
            proxy_set_header Connection "";
            chunked_transfer_encoding off;

            proxy_pass http://minio;
        }
    }

    server {
        listen       9001;
        listen  [::]:9001;
        server_name  localhost;

        # To allow special characters in headers
        ignore_invalid_headers off;
        # Allow any size file to be uploaded.
        # Set to a value such as 1000m; to restrict file size to a specific value
        client_max_body_size 0;
        # To disable buffering
        proxy_buffering off;
        proxy_request_buffering off;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
```

```
            proxy_set_header X-NginX-Proxy true;

            # This is necessary to pass the correct IP to be hashed
            real_ip_header X-Real-IP;

            proxy_connect_timeout 300;

            # To support websocket
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";

            chunked_transfer_encoding off;

            proxy_pass http://console;
        }
    }
}
```

- Добавляем в файл .env строки (аналогичные строки, но без самих значений добавляем в .env.example)

```
MINIO_ROOT_USER = admin
MINIO_ROOT_PASSWORD = miniotestpass
```

- Запускаем через docker-compose up -d --build
- Заходим через браузер http://127.0.0.1:9001 и авторизуемся
- Через веб-интерфейс создаем два бакета:
  - **Buckets -> Create bucket**
    - **mlflow**
    - **dvc**
  - Примечание: содержимое бакетов будет храниться в ./Docker/minio/<Bucket name>
- Через веб-интерфейс создаем сервисный аккаунт (по сути API Key):
  - **Identity -> Service Accounts -> Create service account**
    - Обязательно сохраняем сгенерированный secret key (т.к. его больше нельзя будет увидеть)

        - Добавляем в файл .env наши полученные Key ID/Secret Key (аналогичные строки, но без самих значений добавляем в .env.example)

```
AWS_ACCESS_KEY_ID = L6Vt9Pw72XDw26Mt
AWS_SECRET_ACCESS_KEY = IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G
```

        - По хорошему, необходимо было бы сгенерировать минимум 2 аккаунта, и настроить их права, один для бакета dvc, второй для mlflow, но в рамках данного ДЗ решено это не делать.
- Переезд dvc с yandex s3 (который использовался в прошлом ДЗ) на локальный s3 minio (для реального переезда с S3 на S3, когда уже накоплено большое количество экспериментов, версий и т.п., потребовало бы иного подхода, скорей всего переноса данных через s3 backup/restore, но для нашего случа подходит процедура ниже)
    - Синхронизируем локальный кэш dvc с s3, что бы в локальном кэше, были все необходимые файлы из s3 (без этого, даже в случае налиция оригинальных файлов, запушить их в целевой s3 стандартным способом не получится)
        - **dvc pull**
    - Изменяем файл **.dvc/config** следующим образом

```
[core]
    remote = s3minio
['remote "s3minio"']
    url = s3://dvc
    endpointurl = http://localhost:9000
```

    - Добавляем конфигурацию в файл .dvc/config.local через CLI:
        - **dvc remote modify --local s3yandex access_key_id <key id>**
        - **dvc remote modify --local s3yandex secret_access_key <secret key>**
    - Изменяем файл **.dvc/config.local** следующим образом (key id/secret key аналогичны AWS_ACCESS_KEY_ID/AWS_SECRET_ACCESS_KEY в .env):

```
['remote "s3minio"']
    access_key_id = L6Vt9Pw72XDw26Mt
    secret_access_key = IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G
```

    - Пушим все закэшированное содержимое в локальный s3 (докер контейнеры minio и nginx конечно должны быть запущены)
        - **dvc push**

        - Убеждаемся что все прошло штатно, дополнительно через веб интерфейс минио можно проверить что в бакете dvc появились файлы
- Развернуть MLflow по 4 или 5 сценарию, используя СУБД и хранилище артефактов на свой выбор. Вместо удаленного хоста можно использовать Docker контейнеры.
    - Примечания:
        - с учетом предупреждения в документации MLFlow, что сценарий 5 дает возможность любому, кто имеет доступ на сетевом уровне к mlflow так же доступ ко всем артифактам s3 с правами самого сервера mlflow, т.е. при его применении нужно оценивать риски ИБ, а так же с учетом того, что в любом случае хост проекта должен иметь доступ к S3 в рамках DVC, решено развертывать, в рамках данного ДЗ, 4-й сценарий
    - Разворачиваем контейнеры под СУБД (по аналогии с видеоуроком выбран postgres)
        - Добавляем в **docker-compose.yaml**
            - Примечания:
                - От автора курса (Павла), вольюм postgres: добавлен именно так, а не относительным путем, по аналогии с minio, из за проблем с подобным добавлением именно для постгреса на винде, если добавить относительные пути, будут проблемы с правами на запись, которые нормально пока не решаются)
                - Переменная PGADMIN_DEFAULT_PASSWORD для pgadmin, устанавливает пароль только при первом запуске контейнера, если данный в системе после первого запуска уже есть (в вольюме), то простая замена значения в этой переменной не сменит сам пароль
                - В видеоуроке для postgres вместо expose был прописан проброс портов наружу, на хостовую машину через ports: "5432:5432", но без реальной необходимости этого делать не нужно (а в нашем случае БД должна быть доступна только внутри проекта docker compose).

```
...
 db:
   container_name: postgres
   image: postgres:14.3-bullseye
   restart: always
   environment:
     POSTGRES_USER: ${POSTGRES_USER}
     POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
     POSTGRES_DB: ${POSTGRES_DB}
     PGDATA: /data/postgres
```

```
    volumes:
       - postgres:/data/postgres
    expose:
      - "5432"
    networks:
      - postgres

  pgadmin:
    container_name: pgadmin
    image: dpage/pgadmin4:2022-05-31-1
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD}
    volumes:
       - ./Docker/pgadmin/:/var/lib/pgadmin
    ports:
      - "5050:80"
    networks:
      - postgres
...
networks:
  postgres:
    driver: bridge
...
volumes:
  postgres:
```

- Добавляем в файл **.env** строки (аналогичные строки, но без самих значений добавляем в .env.example)

```
POSTGRES_USER = dbuser
POSTGRES_PASSWORD = dbtestpass
POSTGRES_DB = mlflow
PGADMIN_DEFAULT_EMAIL = "admin@admin.com"
PGADMIN_DEFAULT_PASSWORD = pgtestpass
```

- Запускаем через **docker-compose up -d --build**

        - Заходим через браузер http://127.0.0.1:5050 и авторизуемся (через PGADMIN_DEFAULT_EMAIL, PGADMIN_DEFAULT_PASSWORD)
        - Добавляем новый сервер:
            - Add New Server
                - General
                    - Name: **mlflow_backend**
                - Connection
                    - Hostname/address: **postgres** (опытным пытем проверено, что можно использовать в качестве hostname как имя сервиса, т.е. db в нашем случае, так и имя контейнера, т.е. postgres, есть ли различия или приемущества какого-либо варианта, не проверял, если будет нужно, проверю в будущем)
                    - Maintenance database: **mlflow**
                    - Username: **dbuser**
                    - Password: **dbtestpass**
                    - Save password: **Yes**
                - Save
        - По итогу мы должны подключиться к нашей БД
    - Разворачиваем docker с mlflow
        - создаем директорию **./Docker/mlflow_image**
        - в ней создаем **Dockerfile** следующего содержания

```
FROM python:3.9
# Install python packages
RUN pip install mlflow==1.26.0 boto3==1.21.21 psycopg2==2.9.3
```

        - Добавляем в **docker-compose.yaml**
            - Примечания:
                - Команда запуска mlflow взята из сценария 4 документации mlflow

- Изначально мне было непонятно, зачем нам необходимо задавать для сервера mlflow переменные AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, MLFLOW_S3_ENDPOINT_URL, если согласно рисунка из документации, описывающего сценарий 4, сам mlflow напрямую не взаимодействует с s3, а только сообщает boto3 на клиентской машине, бакет для хранения артефактов (сами credentials и endpoint url задаются так же на стороне клиента boto3); как оказалось, приведенная в документации схема описывает только процесс трекинга, а не полноценного использования mlflow web ui; Т.е. в процессе трекинга экспериментов, действительно mlflow tracking server ника не взаимодействует с s3, а вот если нам необходимо через web ui просмотреть/скачать информацию по артефактам, выполнить какие-либо действия с сохраненной моделью и т.д., то в этих случаях взаимодействие происходит именно напрямую между mlflow и s3; скорей всего для mlflow ui достаточно будет прав к s3 только на чтение (но не проверял)

```
  mlflow:
    restart: always
    build: ./Docker/mlflow_image
    container_name: mlflow_server
    ports:
      - "5000:5000"
    networks:
      - postgres
      - s3
    environment:
      - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
      - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
      - MLFLOW_S3_ENDPOINT_URL=http://nginx:9000
    command: mlflow server --backend-store-uri postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db/${POSTGRES_DB} --default-artifact-root
s3://${AWS_S3_BUCKET}/ --host 0.0.0.0
```

    - Запускаем через **docker-compose up -d --build**
    - Переходим через браузер на http://127.0.0.1:5000 и убеждаемся что ui доступен
  - Проводим изменения проекта, для логирования экспериментов по сценарию 4
    - Примечания:
      - По сценарию 4, в ходе трекинга экспериментов, непосредственно запись артефактов в s3 осуществляет локальный boto3 клиент в составе виртуальной среды python проекта

- Ниже в инструкции пара KEY ID/SECRET KEY для доступа к s3 задается для boto3 через dotenv, у меня пока этот вариант работает стабильно, но в видеолекции Павел делает оговорку, что на Windows можно столкнуться с тем, что этот способ может работать нестабильно (через раз), от чего это зависит, непонятно, и самым надежным способом является задание учетных данных через файл ~/.aws/credentials (~ это путь к домашней директории пользователя из под которого запускается python скрипт) нижеследующего содержания:

```
[default]
aws_access_key_id=...
aws_secret_access_key=...
```

- устанавливаем в виртуальное окружение python-dotenv
  - **poetry add python-dotenv**
- Добавляем в файл **.env** строки (аналогичные строки, но без самих значений добавляем в **.env.example**)

```
MLFLOW_S3_ENDPOINT_URL = "http://localhost:9000"
MLFLOW_TRACKING_URI = "http://localhost:5000"
```

- решено в в файле модуля с общими для проекта функциями (common_funcs.py) написать небольшую функцию, осуществляющую необходимые настройки, далее эту функцию вызываем после импортов в тех модулях, где необходимо выполнять трекинг экспериментов

```
import os
from dotenv import load_dotenv
import mlflow

def mlflow_set_tracking_config(experiment_name: str = "default") -> None:
    """Настраивает трекинг экспериментов через MLFlow Tracking server.
    Реализует сценарий 4. В файле .env должны быть следующие переменные:
    MLFLOW_TRACKING_URI - URL трекинг сервера MLflow (например "http://localhost:5000")
    MLFLOW_S3_ENDPOINT_URL* - URL до s3 хранилища (например "http://localhost:9000")
    AWS_ACCESS_KEY_ID* - key ID к s3 хранилищу (с правами на запись)
    AWS_SECRET_ACCESS_KEY* - secret key к s3 хранилищу
    * используется библиотекой boto3
    Args:
        experiment_name (str, optional): Название серии экспериментов в MLflow.
            Defaults to "default".
```

```
    """
    load_dotenv(override=True)
    mlflow.set_tracking_uri(os.getenv("MLFLOW_TRACKING_URI"))
    mlflow.set_experiment(experiment_name)
```

        - Примечание, опытным путем проверил, что функция load_dotenv(), даже с параметром override=True, не изменяет переменные в самой ОС, видимо они изменяются только в пределах запускаемого процесс и/или системного модуля os (например проверял следующим образом, задал через PowerShell неверное значение переменной $env:MLFLOW_S3_ENDPOINT_URL="http://", далее запустил скрипт в котором через load_dotenv(override=True) значения загружаются из файла .env, далее они используются скриптом, и по итогу завершения скрипта, $env:MLFLOW_S3_ENDPOINT_URL выдает первоначальное значение, которое и было до запуска скрипта; к слову, если не задавать override=True, то load_dotenv не будет заменять значение на указанное в .env, и при обращении к нему через os.getenv будет браться заданное изначально в ОС)
- Провести эксперименты с логированием модели
    - Тут все по аналогии с предыдущим ДЗ, за исключением следующих изменений:
        - теперь изначальные параметры mlflow задаются через собственную вспомогательную функцию mlflow_set_tracking_config (задаются только в тех модулях, где мы хотим трекать эксперименты)
        - весь код эксперимента рекомендуется выполнять внутри with mlflow.start_run() as run: (например весь код внутри основной функции модуля заносится в этот with), дополнительно в параметрах можно указывать имя запуска mlflow.start_run(run_name='...',) которое отображается в ui, и многое другое (примеры здесь https://www.mlflow.org/docs/latest/python_api/mlflow.html); На практике же, я столкнулся с тем, что, эксперимент по выходу из with не закрывается, и при следующем запске скрипта я получаю сообщение вида Exception: Run with UUID 762acca9baf44a41b7fab5323ff8ce31 is already active. To start a new run, first end the current run with mlflow.end_run(). To start a nested run, call start_run with nested=True, решение, либо принудительно закрывать эксперимент mlflow.end_run(), либо что то еще выдумывать, но т.к. такого поведения быть не должно, временно отказался от использования mlflow.start_run(), без него все работает штатно (когда возникнет реальная необходимость, и будет время, можно будет нормально разобраться, возможно это баг текущей версии)
        - производим эти изменения во всех соответствующих модулях (train_model.py, evaluate.py, generate_pairs.py)
    - По итогу проведения нескольких экспериментов, проверяем через mlflow ui (http://127.0.0.1:5000), что трекинг экспериментов и сохранение/доступность артефактов ведется штатно

- Добавляем в **.gitignore** следующие строки:

```
# Docker
Docker/minio/
Docker/pgadmin/
```

- Делаем промежуточный коммит
  - **git add .**
  - **git commit -m "Configuration docker complete"**
- Запустить сервис с моделью стандартными средствами MLflow стандартным способом или через Docker
  - Примечания и предварительные действия:
    - т.к. для публикации модели, при ее трекинге обязательно нужно собирать сигнатуры, пришлось произвести существенный рефакторинг модулей train_model.py, predict_model.py, evaluate.py, по итогу при исполнении train_model дополнительно производится предсказание и оценка качества (сбор метрик) на отложенной выборке, а так же подгружаются метрики сохраняемые на этапе формирования пар кандидатов; в текущем варианте логи экспериментов выглядят более полными, без разбивки на множество модулей, как это было ранее (возможно этого можно было добиться иными способами, например через объединение каким-либо способом в один runs mlflow из разных скриптов, но т.к. мы дополнительно используем dvc repro и запускаются не все, а только измененные стейджи, то реализовал по самому быстрому на текущий момент варианту)
    - трекинг сигнатур выглядит примерно следующим образом

```
signature = infer_signature(X, y_pred) #param1 = inputs, param2 = outputs
mlflow.<model_type>.log_model(..., signature=signature)
```

      - в итоге, в сигнатурах хранятся данные, описывающие:
        - структуру того, что модель принимает на вход (в нашем случае это датафрейм с набор колонок и их типами)
        - структуру того, что получаем на выходе из модели
    - по итогу рефакторинга, делаем соответствующие коммит, более подробно все изменения можно посмотреть через git по указанному ниже коммиту
      - **git add .**
      - **git commit -m "Refactoring train_model.py, predict_model.py, evaluate.py, dvc.yaml, for improve track experiments"**
  - Запуск сервиса MLflow стандартным способом:

- Заходим через mlflow ui (http://127.0.0.1:5000) в какой-либо runs, где есть сохраненная модель
- Слева в списке актефактов выбираем "директорию" с моделью, и копируем строку из Full Path: вида s3://mlflow/4/1b1d9d8ce5514f758e8aa0a19e936ca5/artifacts/output_model_path
- Обязательно экспортируем переменные среды AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, MLFLOW_S3_ENDPOINT_URL (проще всего прямо из терминала IDE через повершелл $env:name="value")
- Выполняем команду вида (порт указываем любой не занятый)
  - **mlflow models serve --env-manager=local -m s3://mlflow/4/1b1d9d8ce5514f758e8aa0a19e936ca5/artifacts/output_model_path -h 0.0.0.0 -p 8001**
- Проверяем через браузер доступность порта http://127.0.0.1:8001 (ответ должен быть Not Found, значит порт запущен)
- далее через приложение postman (https://www.postman.com/downloads/) проверяем работу API (подробное описание mlflow model API здесь (https://mlflow.org/docs/latest/models.html#built-in-deployment-tools)
  - **Collections -> New collections -> New Request**
  - Выбираем медот POST и указываем в адресной строке http://127.0.0.1:8001/invocations
  - Переходим во вкладку **Body -> Raw**, выбираем тип контента JSON
  - В текст запроса вводим 3 строки из нашего тестового датасета, в следующем формате

```
{
    "columns": ["ftr_geo_distance", "ftr_name_levenshtein"],
    "data": [[0.011162535561185155,100], [0.05291212507643972,26],[0.04997659641777344,11]]
}
```

  - Нажимаем Send, по итогу должны получить ответ вида:

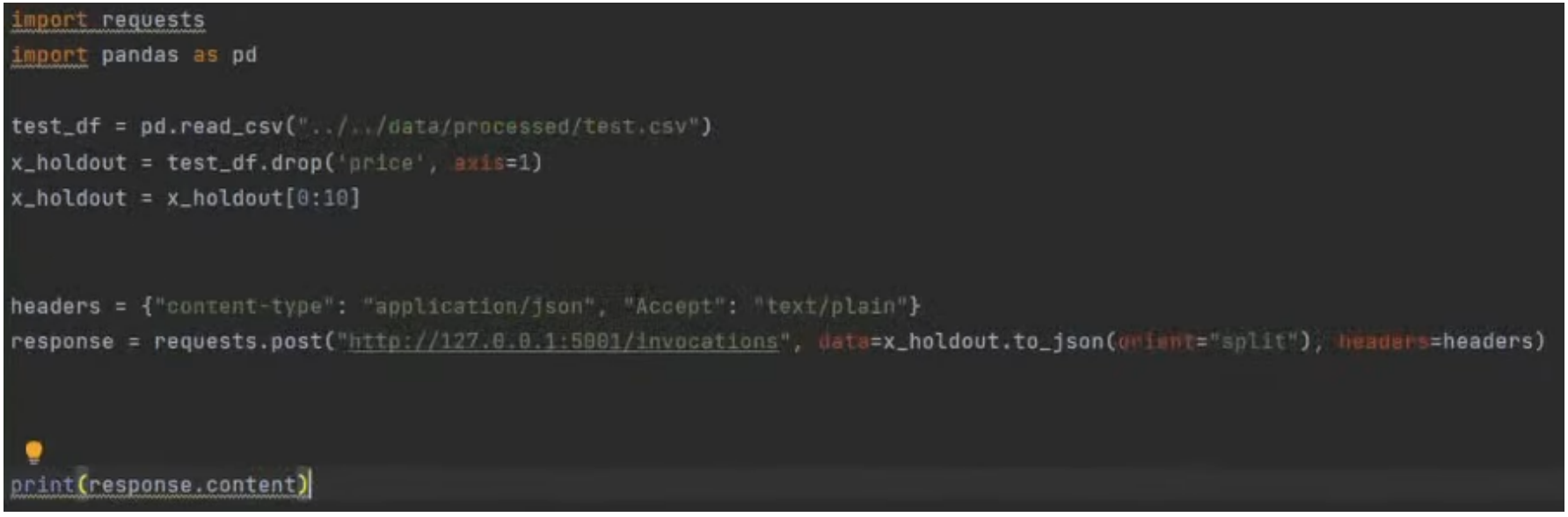
```
[
    1,
    0,
    0
]
```

  - Примечание, формат принимаемых данных видев в сигнатуре сохраненной в mlflow ui модели, если выбрать сам файл модели, запрос выше актуален для следующей сигнатуры

```
signature:
  inputs: '[{"name": "ftr_geo_distance", "type": "double"}, {"name": "ftr_name_levenshtein", "type": "long"}]'
```

```
outputs: '[{"type": "tensor", "tensor-spec": {"dtype": "int32", "shape": [-1]}}]'
```

- Ниже пример кода из видеолекции для взаимодействия с api через requests (из оговорок, могут быть ограниечение на размер ответа, нужно это учитывать)



- Стопим сервис через Ctrl+C

- Запуск сервиса MLflow через Docker
  - **mlflow models build-docker -m s3://mlflow/4/1b1d9d8ce5514f758e8aa0a19e936ca5/artifacts/output_model_path -n "xgboost_model"**
  - После того, как контейнер сбилдится запускаем его наиболее подходящим образом (через docker run, compose и т.п.), не забываем сделать проброс порта наружу
  - https://youtu.be/KZ5Cdevd-b8?t=4063

- Создать собственный микросервис для модели используя Docker и **Flask/FastAPI**
  - Примечания:
    - В рамках текущего ДЗ решено ограничиться только сервисом голой модели, т.е. без препроцессинга фиче генерации и т.д., но в итоговом проекте планирую его переписать, что бы на вход подавались сырые данные, на выходе данные в формате требуемом в задаче
    - Несколько вариантов выбора и подгрузки актуальной модели:

        - в mlflow для каждой версии моедли можно назначать Stage = Staging, Production, Archived (в единицу времени, каждая метка может ссылаться только на одну модель в пределах имени модели)
        - подходы по актуализации модели в докере микросервиса (на Fast API):
            - модель инициализируется в начале модуля сервиса (через mlflow.pyfunc.load_model(f"models:/{model_name}/{model_stage}")), при необходимости деплоя новой версии модели, это версии выставляется необходимый стедж в mlflow, далее запускается команда пересборки контейнера (в данном ДЗ используем этот подход)
            - модель инициализируется внутри функции обработчика POST запроса, при необходимости обновления, достаточно изменить стейдж в mlflow, и при следующем моделе уже подтянеться новая модель; данный вариант может подходить только для легковесных моделей и малонагруженных сервисов
            - написание собственного скрипта, который будет следить за обновлениями стейджей в mlflow и при его изменении, переинициализировать модель
    - В mlflow для одной из моделей выбираем stage = Staging
    - Доставляем необходимые пакеты
        - **poetry add uvicorn**
        - **poetry add fastapi**
        - **python-multipart**
    - Создаем директорию, ./src/app и в ней
        - пустой файл **__init__.py**
        - **inference.py** следующего содержания

```
"""Module for ML service on FastAPI"""
import os
import sys
import pandas as pd
import mlflow
from dotenv import load_dotenv
from fastapi import FastAPI, File, UploadFile, HTTPException
# Load the environment variables from the .env file into the application
load_dotenv(override=True)
# os.environ["MLFLOW_S3_ENDPOINT_URL"] = os.getenv("MLFLOW_S3_ENDPOINT_URL")
# Initialize the FastAPI application
```

```
app = FastAPI()
# Create a class to store the deployed model & use it for prediction
class Model:  # pylint: disable=too-few-public-methods
    """General model class for inference"""
    def __init__(self, model_name, model_stage):
        """
        To initialize the model
        model_name: Name of the model in registry
        model_stage: Stage of the model
        """
        # Load the model from Registry
        self.model = mlflow.pyfunc.load_model(f"models:/{model_name}/{model_stage}")
    def predict(self, data):
        """
        To use the loaded model to make predictions on the data
        data: Pandas DataFrame to perform predictions
        """
        predictions = self.model.predict(data)
        return predictions
# Create model
model = Model("general_model_xboost", "Staging")
# Create the POST endpoint with path '/invocations'
@app.post("/invocations")
async def create_upload_file(file: UploadFile = File(...)):
    """Generate prediction by file .csv
    Args:
        file (UploadFile, optional): file .csv with columns of model signatures.
            Defaults to File(...).
    Raises:
        HTTPException: If file not .csv
    Returns:
        _type_: Json of model prediction
    """
    # Handle the file only if it is a CSV
    if file.filename.endswith(".csv"):  # pylint: disable=no-else-return
        # Create a temporary file with the same name as the uploaded
        # CSV file to load the data into a pandas Dataframe
        with open(file.filename, "wb") as input_file:
```

```
            input_file.write(file.file.read())
        data = pd.read_csv(file.filename)
        os.remove(file.filename)  # Return a JSON object containing the model predictions
        # без преобразования каждого элемента в int, получал ошибку
        # TypeError: 'numpy.int32' object is not iterable
        return list(map(int, model.predict(data)))
    else:
        # Raise a HTTP 400 Exception, indicating Bad Request
        # (you can learn more about HTTP response status codes here)
        raise HTTPException(status_code=400, detail="Invalid file format. Only CSV Files accepted.")
# Check if the environment variables for AWS access are available.
# If not, exit the program
if os.getenv("AWS_ACCESS_KEY_ID") is None or os.getenv("AWS_SECRET_ACCESS_KEY") is None:
    sys.exit(1)
```

- Примечание, оттестировать работу сервиса локально, можно через вызов **uvicorn src.app.inference:app --host 0.0.0.0 --port 80**
- Создаем директорию **./Docker/ml_service**, в ней создаем:
  - **Dockerfile** следующего содержания

```
FROM python:3.9

WORKDIR /code

RUN pip install --upgrade pip
RUN pip install poetry

COPY ./pyproject.toml /code/
COPY ./poetry.lock /code/
COPY ./src /code/src
COPY ./Docker/ml_service/.env /code/.env

RUN poetry config virtualenvs.create false \
    && poetry install --no-dev --no-interaction --no-ansi

CMD ["uvicorn", "src.app.inference:app", "--host", "0.0.0.0", "--port", "80"]
```

  - Файл **.env**

```
AWS_ACCESS_KEY_ID = L6Vt9Pw72XDw26Mt
AWS_SECRET_ACCESS_KEY = IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G
```

- Файл **.env.example**

```
AWS_ACCESS_KEY_ID =
AWS_SECRET_ACCESS_KEY =
```

- Файл **.gitignore**

```
# DotEnv configuration
.env
```

- Собираем образ через
  - **docker build -f Docker/ml_service/Dockerfile -t ml_service .**
- Добавляем в файл **docker-compose.yaml**

```
...
app:
    image: ml_service
    container_name: ml_service
    ports:
      - "8003:80"
    networks:
      - s3
    environment:
      - MLFLOW_TRACKING_URI=http://mlflow:5000
      - MLFLOW_S3_ENDPOINT_URL=http://nginx:9000
    depends_on:
      - mlflow
      - nginx
...
```

- Запускаем образ через
  - **docker-compose up -d**
- Проверяем через postman
  - **Collections -> New collections -> New Request**

    - Выбираем медот POST и указываем в адресной строке http://127.0.0.1:8003/invocations
    - Переходим во вкладку Body -> from-data
        - в первой строке, в поле key вводим "file" (это имя параметра функции async def create_upload_file(file: UploadFile = File(...)), в этой же ячейке сперва в выпадающем списке выбираем File
        - указываем предварительно подготовленный файл .csv с несколькими строками, формата датафрейма, который принимает на вход модель (т.е. только с колонками "ftr_geo_distance", "ftr_name_levenshtein")
        - нажимаем send, по итогу должны получить ответ

```
[
    1,
    0,
    0
]
```

- По итогу урока делаем merge request нашей ветки урока
    - **git add .**
    - **git commit -m "Complete homework7"**
    - **git push origin ie_homework7_docker**
    - **мерджим через реквест на гитлабе**
    - **git checkout main**
    - **git pull**
    - **git branch -d ie_homework7_docker**

Неделя 8. Методы и инструменты тестирования кода и данных.

Видео:
https://youtu.be/tC3IdAN_hX4
Презентация:
-

ДЗ:
-

Ссылка на репозиторий, на котором Автор курса демонстрирует практические шаги: https://gitlab.com/m1f/mlops_cd/

Отработка ДЗ на командном репозитории (https://gitlab.com/mlops-23reg-team/mlops-23reg-project/):

- Подготовительные шаги
  - git создал ветку ie_homework8_ci_tests, перешел в нее (далее работаем в этой ветке)
    - **git branch ie_homework8_ci_tests**
    - **git checkout ie_homework8_ci_tests**
- Возможен вариант тестов прямо в коде через assert, самими разработчиками (см. замечание ниже), например, код проверки типов может быть реализован следующим образом
  - **assert isinstance(<variable>, <type>)**, "Сообщение, которое будет выведено, если тип другой" (пример assert isinstance(train_df, pd.DataFrame), "input[0] must be a valid dataframe")
  - Замечание: данный подход в самых простых случаях может работать, но основная цель тестов - защитить репозиторий от плохого/некачественного кода (особенно критичные ветки прода), и когда мы отдаем тестирование самим разработчикам, то мы можем столкнуться множеством проблем:
    - разработчики в команде могут быть разного уровня, необходимые проверки могут быть не добавлены в код по разным причинам, может быть произведен рефректоринг, который потребует дополнительных проверок в других модулях, сторонних сервисах/микросервисах и т.п.)
    - наилучшей практикой является разделение разработчиков и тестеров, и тесты должны быть написаны таким образом, что бы они фейлились при изменениях в коде, которые предполагают каких-либо дополнительных действий (возможно доработок соседних сервисов), которые сделаны не были; т.е. код меняется, тесты остаются теми же, и мы получаем фейл, если измененный код им не соответсвует
- Общие заметки по юнит-тестам (тесты, которые тестируют элементарную функциональность), через отдельные скрипты в директории tests
  - Общие заметки:

- Для того, что бы pytest автоматически запускал все необходимые тесты (т.е. когда мы просто выполняем с консоли pytest), тесты должны быть реализованы в виде .py файлов, они должны находится в директории tests и имена файлов должны либо начинаться с "test_", либо заканчиваться на "_test"
  - Примечание, для больших проектов можно создать отдельный конфиг для pytest (через pyproject.toml), в котором задать настройки, где и какие файлы искать, но для простых проектов имеет смысл придерживаться стандартного подхода
- Кроме того, принятой практикой, является указание в имени файла теста, имени файла модуля, который он тестирует (например для тестирования модуля model_train.py, создаем файл теста test_model_train.py)
- Самыми общими тестами могут являться
  - проверка штатной работы модуля (т.е. по итогу отработки он возвращает код возврата 0), который вызывается через cli (например с тестовыми данными)
    - для этого используется функция .invoke(...) экземпляра класса CLIRunner из модуля click.testing
    - Пример из репозитория Павла
      - Важно: clean_data это именно функция, которая обернута в декоратор click, а не имя модуля, так же в пути к файлам, слеши используем в правую сторону, вне зависимости от ОС

```
def test_cli_command():
    result = runner.invoke(clean_data, 'data/interim/data_regional.csv data/interim/data_cleaned.csv')
    assert result.exit_code == 0
```

  - проверка данных (например требуемых форматов output файлов, создаваемых по итогу работы модулей)
    - самым простым вариантом является написание функции загрузки файлов, получаемых после отработки скрипта и проверки их структуры (например, если это таблицы, то проверка колонок, типов данных, отсутствие пропусков np.NaN и т.п.)

```
def test_output():
    df = pd.read_csv('data/interim/data_cleaned.csv')
    df_ge = ge.from_pandas(df)

    expected_columns = ["price", "date", "geo_lat", "geo_lon", "building_type", "level", "levels", "rooms", "area",
                        "kitchen_area", "object_type"]
    assert df_ge.expect_table_columns_to_match_ordered_list(column_list=expected_columns).success is True
    assert df_ge.expect_column_values_to_be_unique(column="id").success is True is True
```

```
    assert df_ge.expect_column_values_to_not_be_null(column="geo_lat").success is True is True
    assert df_ge.expect_column_values_to_be_of_type(column="kitchen_area", type_="float64").success is True
    assert df_ge.expect_column_values_to_match_strftime_format(
        column="date", strftime_format="%Y-%m-%d %H:%M:%S").success is True
```

        - Кроме того, есть большая оговорка, по небольшим проектам, для таких проектов, в большинстве случаев (за исключением сервиса инференса), будет достаточно запуска исполнения DAG (например через dvc repro), и прохождения теста в случае его корректного завершения
    - В идеале и реальном проекте, должно быть 100% покрытие всего кода тестами, но так же нужно понимать, что это дополнительная трудоемкость и если нет отдельной команды QA (тестирования), то нужно взвесить все за/против (в небольших исследованиях, особенно, когда разработчик один, возможно стоит тестирование пропустить, либо выполнять только самое основное, например только код сервиса модели, который отвечает для инференс)
- Кроме юнит тестов, есть еще интеграционные теста, когда проверяется взаимодействие между собой разных модулей, обращения к БД, сторонним сервисам и т.п., в самом простом случае, корректное исполнение нашего DAG, в какой то мере может считаться подобным тестом
- Реализация тестов в текущем проекте
    - Важное замечание, т.к. мы всю инфраструктуру поднимаем локально, и нормальное исполнение кода зависит от ее доступности, то в наших условиях, все наши вспомогательные докер контейнеры должны быть запущены на той же машине, на которой у нас крутится раннер (в реальном проекте, инфраструктура будет поднята отдельно, но раннер так же должен будет обязательно иметь к ней доступ)
    - Устанавливаем самый распространенный пакет pytest
        - **poetry add --dev pytest**
        - **poetry add --dev great_expectations**
    - Создаем в корне проекта директорию tests, внутри
        - пустой файл **__init__.py**
        - **test_evaluate.py**

```
"""Unit test for ./src/model/evaluate.py"""
from click.testing import CliRunner
from src.models.evaluate import evaluate_cli
INPUT_ORIGINAL_DATASET_PATH = "./data/interim/split_test.csv"
INPUT_SUBMISSION_PRED_PATH = "./data/processed/submission_pred_split.csv"
```

```
OUTPUT_METRICS_PATH = "./reports/metrics_of_split_test_final.json"
# Initialize runner
runner = CliRunner()
def test_cli_command():
    """Test execute module from CLI"""
    result = runner.invoke(
        evaluate_cli,
        "{} {} {}".format(  # pylint: disable=consider-using-f-string
            INPUT_ORIGINAL_DATASET_PATH,
            INPUT_SUBMISSION_PRED_PATH,
            OUTPUT_METRICS_PATH,
        ),
    )
    assert result.exit_code == 0
```

- **test_predict_model.py**

```
"""Unit test for ./src/model/predict_model.py"""
from click.testing import CliRunner
import pandas as pd
import great_expectations as ge
from src.models.predict_model import predict_model_cli
INPUT_MODEL_PATH = "./models/model_on_split_df.pkl"
INPUT_ORIGINAL_DATASET_PATH = "./data/interim/split_test.csv"
INPUT_PAIR_FEATURES_DATASET_PATH = "./data/processed/split_test_pair_feautures.csv"
OUTPUT_SUBMISSION_PATH = "./data/processed/submission_pred_split.csv"
# Initialize runner
runner = CliRunner()
def test_cli_command():
    """Test execute module from CLI"""
    result = runner.invoke(
        predict_model_cli,
        "{} {} {} {}".format(  # pylint: disable=consider-using-f-string
            INPUT_MODEL_PATH,
            INPUT_ORIGINAL_DATASET_PATH,
            INPUT_PAIR_FEATURES_DATASET_PATH,
            OUTPUT_SUBMISSION_PATH,
        ),
    )
```

```
    assert result.exit_code == 0
def test_output():
    """Test format output files"""
    submission_df = pd.read_csv(OUTPUT_SUBMISSION_PATH)
    df_ge = ge.from_pandas(submission_df)
    expected_columns = ["id", "matches"]
    assert (
        df_ge.expect_table_columns_to_match_ordered_list(column_list=expected_columns).success
        is True
    )
    assert df_ge.expect_column_values_to_be_unique(column="id").success is True is True
    assert df_ge.expect_column_values_to_not_be_null(column="matches").success is True is True
```

- Проверяем выполнимость тестов локально через выполнение из командной строки
  - **pytest** (или pytest --disable-warnings)
- Добавляем в gitlab CI/CD variables переменные среды (значения дерем из файла .env), которые нам понядобятся для выполнение CI пайплайна
  - на гитлабе **Settings -> CI/CD -> Variables**, через Add variable
    - Замечание, в настоящий момент паплайн запускается в момент пуша любой ветки, не дожидаясь мердж реквеста, если не выставить Protected в False (Export variable to pipelines running on protected branches and tags only), то они не передаются в паплайн, т.к. текущая ветка не защищенная. Нужно понимать, что при Protected:False, любой, кто имеет права на создание и пуш любой ветки в репозиторий, в теории может написать скрипт (и в случае необходимости модифицируя gitlab-ci), который может запустясь через раннер сграбить значения всех текущих переменных и передать их например на какой-либо сторонний ресурс. Возможно все не так просто, и подобный вектор атаки невозможен, но в любом случае в реальном проекте стоит подробнее разобраться в данном вопросе, и в целом, по хорошему, нужно добиться работы пайплайна с Protected переменными.
    - **AWS_ACCESS_KEY_ID** (Protected: False, Masked: True)
    - **AWS_SECRET_ACCESS_KEY** (Protected: False, Masked: True)
    - **MLFLOW_S3_ENDPOINT_URL** (Protected: False, Masked: False)
    - **MLFLOW_TRACKING_URI** (Protected: False, Masked: False)
- В файл **.gitlab-ci.yml** добавляем юнит тесты

        - Примечание: если данных очень много и выполнение всего DAG (через dvc repro) на полных данных может занимать значительное время, а этого не нужно, то допустимо создать отдельный dvc-ci.yaml и отдельный набор урезанных входных данных

```
pytest:
 script:
   - poetry run dvc remote modify --local s3minio access_key_id ${AWS_ACCESS_KEY_ID}
   - poetry run dvc remote modify --local s3minio secret_access_key ${AWS_SECRET_ACCESS_KEY}
   - poetry run dvc pull
   - poetry run dvc repro
   - poetry run pytest --disable-warnings
```

    - Делаем коммит, пуш в ориджин и мердж ветки в main (как напоминание, что бы пайплан отработал корректно, с раннера должны быть доступны все смежные ресурсы в виде поднятых докер контейнеров, в нашем случае, они должны быть запущены на одной машине)
        - **git add .**
        - **git commit -m "Complete homework8"**
        - **git push origin ie_homework8_ci_tests**
        - мерджим через реквест на гитлабе (только после того как убедимся что пайплайн отработал успешно, на самом деле на этом моменте должна стоять галка, что мердж возможен только при успешном пайплайне)
        - **git checkout main**
        - **git pull**
        - **git branch -d ie_homework8_ci_tests**

P.S. Что интересно, после того как merge реквест уже произошел (при этом перед тем, как нажать на кнопку merge я дождался успешного исполнения предваряющего его пайплайна), запустился еще один пайплайн (я заметил это в последствии по истории), который зафейлился (по странной причине, возможно я просто закрыл ноутбук с раннером и задача прервалась), но не суть, а суть в том, что первый пайплан, перед мердж реквестом имеет название Complete homework8 (это мой последний коммит) и относится к ветке ie_homework8_ci_tests, а запущенный позже, имеет название Merge branch 'ie_homework8_ci_tests' into 'main' и относится к main. По хорошему, нужно будет разобраться с этим в финальном уроке (сообщение из будущего: разобрался, подробности в следующем уроке), и скорей всего настроить так, что бы пайплан отрабатывал только при мердже в мейн, а не каждом пуше любой ветки (+ сделать переменные защищенными, об этом упоминал выше).

Чуть позже перезапустил зафейленый пайплайн, в итоге с первого раза выполнился job test_lint, со второго раза pylint (возможно не хватает ресурсов для одновременного исполнения, возможно есть какие-либо конфликты при одновременной установке/обновлении сред), в ходе последнего ДЗ или проекта, постараться определить причину подобной нестабильности (возможно причиной является работа раннера на Win).

Неделя 9. CI/CD (GitLab, nexus)

Видео:
https://youtu.be/M1P7j0qsedM
Презентация:
-
ДЗ:
-
Ссылка на репозиторий, на котором Автор курса демонстрирует практические шаги: https://gitlab.com/m1f/mlops_cd/

Отработка ДЗ на командном репозитории (https://gitlab.com/mlops-23reg-team/mlops-23reg-project/):

- Подготовительные шаги
  - git создал ветку ie_homework9_ci_cd, перешел в нее (далее работаем в этой ветке)
    - **git branch ie_homework9_ci_cd**
    - **git checkout ie_homework9_ci_cd**
- Общие теоретические заметки по теме
  - Заметки по файлу пайплайна **gitlab-ci.yml** (по сути это основной компонент CI/CD от гитлаба)
    - Павел упоминает, что по умолчанию пайплайн запускается при изменении файла gitlab-ci.yml, но провел эксперимент изменим в текущем ветке файл readme.md, провел коммит с именем "Experiment - run pipeline on any push?" и запушил ветку в гитлаб, в итоге автоматически запустился пайплайн, получается по умолчанию пайплайн запускается при каждом комите в любую ветку. Мне такое поведение не нравится, скорей всего имеет смысл запускать пайплайн только для определенных "защищаемых" веток, в нашем случае main; необходимо разобраться с этим по ходу урока;

- Один из ключевых компонентов это стейджи/стадии (позволяют настраивать как параллельное, так и последовательное исполнение стейджей)
  - задаются через ключевой тег stages: в котором перечислены стадии, которые должны проходить, например:

```
stages:
 - test
 - build
```

  - при таком синтаксисте, стейджы будут выполняться последовательно, в указанном порядке
  - далее эти стеджи указываются уже в именованных задачах (job) по типу:

```
docker_build:
  stage: build
...

docker_deploy:
  stage: deploy
...
```

    - в большинстве случаев развертывание микросервиса в виде докер готового докер образа (размещенного в реестре докер образов) в какой либо инфраструктуре докер контейнеров (например в кластере кубернетс)

```
docker_build:
  stage: build
  tags:
    - mlops
...

docker_deploy:
  stage: deploy
  tags:
    - prod
...
```

  - Так же теги назначаются раннерам, и раннер будет брать в работу только задачи с делегированными ему тегами

        - Таги назначаются раннеру через **Settings -> CI/CD -> Runners**, далее на значек редактирования раннера (карандаш), и указываем их через запятую в поле tags
        - Так же есть отдельная настройка **Run untagged jobs**, флажек снят по умолчанию, отвечает за то, будет ли раннер подбирать нетегированные задачи или нет (в реальном проекте обычно галка снята, т.к. при нормальной настройке, все наши задачи должны быть протегированы)
    - Хорошей идеей будет указание названий веток к которым применима задача, в этом случае, пайплайн будет запускаться только при мердже в эти ветки, а не при любом запушенном комите (упоминал об этом поведении по умолчанию ранее), кроме того, в случае наличия более сложной структуры проекта (например с препрод средой), мы можем более гибко варьировать и настраивать задачи, которые требуется выполнять для разных стадий/мерджей
        - Возможны следующие варианты
            - Указание веток к которым будет применяться задача (через only:)
            - Указание веток исключений, т.е. задача будет применятся ко всем, кроме них (через except:)

```
lint_test:
...
except:
    - main
...

docker_build:
...
  only:
    - main
...
```

    - Гитлаб позволяет в ходе CI/CD работать с переменными среды, находящимися в самом гитлабе, из документации (https://docs.gitlab.com/ee/ci/variables/index.html) они могут быть
        - You can use predefined CI/CD variables or define custom (список встроенных переменных здесь https://docs.gitlab.com/ee/ci/variables/predefined_variables.html):
            - Variables in the .gitlab-ci.yml file.
            - Project CI/CD variables.
            - Group CI/CD variables.

        - Instance CI/CD variables.
    - Так же есть примечание, про которое лучше не забывать, а именно что переменные из GitLab UI автоматически не прокидываются в сервисные контейнеры (имеется ввиду или контейнер в котором запущен сам раннер, если он запущен через докер или любые другие контейнеры тоже, тут проверить нужно на практике, но держать в голове, эту оговорку нужно), переинициализировать нужно в файле .gitlab-ci.yml следующим образом:

```
variables:
  SA_PASSWORD: $SA_PASSWORD
```

    - Напоминание, раздел **before_script** выполняется не один раз, а перед каждой задачей (job), соответственно заносим в этот раздел только то, что действительно нужно
        - Провел небольшой эксперимент с настройками ниже, выводы представлены ниже

```
before_script:
  - echo "Before script run!"

stages:
  - stage1
  - stage2

job1:
  stage: stage2
  script:
  - echo "job1 of stage2 run!"
  - mkdir .cache/pip/
  - ls .cache/pip/
  - echo "job1 of stage2 run!" >> .cache/pip/job1.txt

job2:
  stage: stage2
  script:
  - echo "job2 of stage2 run!"
  - mkdir .cache/pip/
  - ls .cache/pip/
  - echo "job2 of stage2 run!" >> .cache/pip/job2.txt

job4:
```

```
  stage: stage2
  script:
  - echo "job4 of stage2 run!"
  - mkdir .cache/pip/
  - ls .cache/pip/
  - echo "job4 of stage2 run!" >> .cache/pip/job4.txt

job3:
  stage: stage1
  script:
  - echo "job3 of stage1 run!"
  - mkdir .cache/pip/
  - ls .cache/pip/
  - echo "job3 of stage1 run!" >> .cache/pip/job3.txt
```

- **before_script** запускается перед каждой задачей (job)
- **stages** (стейджи) выполняются последовательно, в порядки их указания (по факту выполняются не стейджи а задания отнесенные к стейджам, но пока полностью не запершатся все задачи предыдущего стейджа, задачи из следующего не запустятся)
- задачи в рамках одного стейджа запускаются параллельно, максимальное количество параллельно запускаемых раннером задач задается в конфигурационном файле раннера ключевым словом concurrent), к слову, если есть проблемы с ресурсами на тестовом раннере, можно выставить в 1 для снижения нагрузки
- каждая задача запускается в отдельной временной среде/директории раннера (для Win раннера, в его директории builds\...\0, builds\...\1, builds\...\2 и т.д. по количеству одновременно запускаемых задач
  - перед запуском задачи, директория зачищается, после чего, в нее может быть скопирован только кэш с прошлого запуска этой же задачи
    - директории и файлы, которые необходимо кэшировать указываются в разделе cache: файла gitlab-ci.yml
    - кэши прошлых запусков хранятся в директории cache раннера, в виде архивов
  - полезно то, что директория зачищается именно перед запуском задачи, а не после ее исполнения (т.е. по итогу задачи, в том числе зафейленной, можно найти ее директорию и дополнительно проанализировать что в ней не так)
  - после зачистки директории в нее производится git clone состояния репозитория инициирующей паплайн ветки/коммита

- Развертывание репозитория nexus в качестве docker registry
  - **Sonatype Nexus** - платформа для создания самых различных репозиториев, реестров и т.п. (docker, pypi, linux repos и многое многое другое, поддерживаемых репозиториев под сотню), но стоит отметить, что т.к. с определенного момента в гитлабе появился свой репозиторий под докер образы, то сейчас все чаще используют именно его, но зачастую в уже имеющихся инфраструктурах уже развернут локальный nexus, в связи с чем ниже кратко приводится пример работы с ним
  - Развернуть можно через добавление в docker-compose.yaml (2 порта, 8081 для веб менеджмент консоли, 8123 для взаимодействия в репозиторием через API)

```
nexus:
    image: sonatype/nexus3:latest
    container_name: nexus
    ports:
      - "8081:8081"
      - "8123:8123"
    volumes:
      - nexus:/nexus-data

...
volumes:
  ...
  nexus:
```

  - Заходим через браузер на http://127.0.0.1:8081
  - Для первичной авторизации логин admin, а пароль будет сгенерирован и лежать в самом докер контейнере в файле **/nexus-data/admin.password**
    - проваливаемся в контейнер через **docker exec -it nexus bash**
    - **cat /nexus-data/admin.password**
  - В веб интерфейсе заходим в Repositories -> Create repository: docker (hosted)
    - Name: **mlops_nexus**
    - HTTP: **8123** (т.е. указываем здесь порт, который ранее прокидывали в докер композе)
    - Можно активировать **Enable Docker V1 AVI**
    - **Create**

    - Далее, для работы с любым сторнним репозиторием, нужно предварительно залогинится в нем через docker login -u <login> -p <password> <host>:<port>, для нашего случая:
        - **docker login -u admin -p <пароль, установленный после первого входа в nexus> 127.0.0.1:8123**
    - Для примера, пушим наш контейнер mlflow_server в nexus
        - **docker tag mlflow_server:latest 127.0.0.1:8123/mlflow_server:latest**
        - **docker push 127.0.0.1:8123/mlflow_server:latest**
    - В вебинтерфейсе nexus, разделе Browse репозитория, можно смотреть текущие образы (если провели действия выше, должен появиться mlflow_server)
    - Загрузить образ из репозитория можно через:
        - из командной строки
            - **docker pull 127.0.0.1:8123/mlflow_server:latest**
        - из докер композ файла
            - **image: 127.0.0.1:8123/mlflow_server:latest**
- Для ведения репозитория докер образов в репозиториях, нужно обязательно проводить версионирование, основные варианты
    - хэш коммита
    - версии в соответствии с каким-либо подходом по версионированию (SemVer или PEP 440)
- Использование репозитория образов встроенного в gitlab (наиболее популярный сейчас способ для внутренних проектов, для публичных обычно используется docker hub)
    - Реестр образов в гитлабе доступен через раздел **Packages & Registries -> Container Registry**
        - В пустом разделе сразу приведены примеры логина, билда и пуша, приводим их для информации
            - **docker login registry.gitlab.com**
            - **docker build -t registry.gitlab.com/mlops-23reg-team/mlops-23reg-project .**
            - **docker push registry.gitlab.com/mlops-23reg-team/mlops-23reg-project**
    - Сборку образов и запуск контейнеров будем проводить через gitlab pipeline
    - Ниже рабочий **gitlab-ci.yml** (содержащий в себе стейджи, тестирования, доставки и деплоя)
        - Примечания пайплайну:
            - важно понимать, что при мердж реквесте, имеет место 2 исполнения 2-х разных пайплайнов
                - первый пайплайн запускается при пуше любой ветки в удаленный репозиторий (т.е. он запускается сразу при пуше, и даже тога, когда мы не делаем мердж реквест)

        - при этом, для удачного мердж реквесте, этот предварительно запученный пайплайн должен удачно завершиться
        - после всех аппрувов и нажатии по итогу кнопки merge, запустится следующий пайплайн, уже для ветки main
    - применяем 3 стеджа (напоминаю, что задачи из разных стейджей выполняются последовательно)
        - **tests** - выполняется при пуше в любую ветку, в рамках него запускаются (параллельно) следующие задачи
            - **test_lint** - аудит кода линтерами
            - **pytest** - юнит-тесты
        - **build** (с одной задачей **docker_build**) - сборка и доставка в gitlab registry докер образа нашего сервиса dev_ml_service
            - выполняется только для коммитов в ветку main, в нашем случае, т.к. прямые коммины запрещены, будет запускаться после мерджа в main (именно после, т.е. предвариательно должен пройти весь процесс мердж реквеста)
        - **deploy** (с одной задачей **docker_deploy**) - выполняется загрузка готового докер образа нашего сервиса с gitlab registry и его деплой (запуск соответсвующего контейнера)
            - выполняется только для коммитов в ветку main и только после успешного завершения стеджа build
            - предварительно тушаться и удаляются предыдущие версии контейнеров с заданным префиксом, так же удаляются их volume (в части реальных проектов, возможно они должны оставаться)
    - очень часто к контейнерам применяют префикс (в нашем случае "dev"), это полезно для более удобного управления группой контейнерами (для массвовой остановки/перезапуска и т.п.)
    - необходимый для контейнера файл .env генерируем налету используя занесенные в хранилище gitlab переменные (столкнулся с тем, что dotenv требует, что бы файл .env выл в кодировке UTF-8, а PowerShell создает файл в другой кодировке, в итоге добавил команду конвертирования кодировки файла)

```
# This file is a template, and might need editing before it works on your project.
# To contribute improvements to CI/CD templates, please follow the Development guide at:
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Python.gitlab-ci.yml
# Official language image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/python/tags/
image: python:3.9
# Change pip's cache directory to be inside the project directory since we can
# only cache local items.
```

```yaml
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
  DOCKER_REGISTRY: "registry.gitlab.com/mlops-23reg-team/mlops-23reg-project"
  CONTAINER_PREFIX: "dev"
# Pip's cache doesn't store the python packages
# https://pip.pypa.io/en/stable/topics/caching/
#
# If you want to also cache the installed packages, you have to install
# them in a virtualenv and cache it as well.

cache:
# Используемое кеширование на основе ключей, одначает, что будет происходить проверка
# неизменности заданных файлов-ключей, и если они не были изменены, то директории в paths
# будут взяты из кэша (например если если poetry.lock остался прежним, то не нужно
# переустанавливать заново всю виртуальную среду)
  key:
    files:
      - poetry.lock
      - .gitlab-ci.yml
    prefix: ${CI_JOB_NAME}
  paths:
    - .cache/pip
    - .venv

before_script:
# Важно понимать, что этот раздел выполняется не один раз, а перед каждым последующим стейджем (наподобии test_lint, если бы их было несколько)
  - python --version  # For debugging

stages:
  - tests
  - build
  - deploy

test_lint:
  stage: tests
  tags:
    - mlops
  except:
```

```
    - main
  script:
    # preparing python virtual environment
    - python -m pip install --upgrade pip
    - pip install poetry
    - poetry config virtualenvs.in-project true
    - poetry install
    # job scripts
    - poetry run flake8 src
    - poetry run mypy --ignore-missing-imports src
    - poetry run bandit src
    - poetry run pylint src

pytest:
  stage: tests
  tags:
    - mlops
  except:
    - main
  script:
   # preparing python virtual environment
   - python -m pip install --upgrade pip
   - pip install poetry
   - poetry config virtualenvs.in-project true
   - poetry install
   # job scripts
   - poetry run dvc remote modify --local s3minio access_key_id $AWS_ACCESS_KEY_ID
   - poetry run dvc remote modify --local s3minio secret_access_key $AWS_SECRET_ACCESS_KEY
   - poetry run dvc pull
   - poetry run dvc repro
   - poetry run pytest --disable-warnings

docker_build:
  stage: build
  tags:
    - mlops
  only:
    - main
```

```
  script:
    - echo $CI_COMMIT_SHA
    - echo "# For UTF8 with BOM read" >> Docker/ml_service/.env
    - echo "AWS_ACCESS_KEY_ID = $AWS_ACCESS_KEY_ID" >> Docker/ml_service/.env
    - echo "AWS_SECRET_ACCESS_KEY = $AWS_SECRET_ACCESS_KEY" >> Docker/ml_service/.env
    - (Get-Content -path Docker/ml_service/.env) | Set-Content -Encoding UTF8 -Path Docker/ml_service/.env
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -f Docker/ml_service/Dockerfile -t $DOCKER_REGISTRY/dev/app:$CI_COMMIT_SHA .
    - docker push $DOCKER_REGISTRY/dev/app:$CI_COMMIT_SHA

docker_deploy:
  stage: deploy
  tags:
    - prod
  only:
    - main
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    # Scripts for Windows runner:
    - docker rm $(docker stop $(docker ps -a -q --filter name=${CONTAINER_PREFIX}_*)); if (-not $?) {cd .}
    - docker volume rm $(docker volume ls -q --filter name=${CONTAINER_PREFIX}_*); if (-not $?) {cd .}
    # Scripts for linux runner:
    # - docker stop $(docker ps -a | grep ${CONTAINER_PREFIX}_ | awk '{print $1}') || true
    # - docker rm $(docker ps -a | grep ${CONTAINER_PREFIX}_ | awk '{print $1}') || true
    # - docker volume rm $(docker volume ls | grep ${CONTAINER_PREFIX}_ | awk '{print $2}') || true
    # Scripts for any runner:
    - docker-compose -p $CONTAINER_PREFIX -f docker-compose.dev.yaml up -d
```

- Также необходимо создать в корне проекта отдельный **docker-compose.dev.yaml**
  - Примечания:

- есть проблема с нашей развернутой тестовой инфраструктурой, что ниже сервис должен обращаться к mlflow и s3 на хостовой машине (127.0.0.1 не подойдет, т.к. докер контейнер в этом случае будет обращаться к самому себе), в реальной инфраструктуре здесь должны быть указаны hostname/порты реальных серверов, на которых развернуты соответсвующие сервисы, но у нас иной случай; если наш тестовый раннер работал на linux машине и соответсвенно докер крутился бы тоже на ней, в качестве hostname можно было бы указать host.docker.internal, но на Windows это нормально не работает, в целом нормального и автоматического решения под Win я не нашел, в итоге остается только подставлять wsl2 ip адрес, для однообразия решил делать это сразу в переменных гитлаб MLFLOW_TRACKING_URI/MLFLOW_S3_ENDPOINT_URL, т.е. заменил их значения (ранее в них был указан localhost), подробности в слудующем пункте

```
version: '3.8'
services:
  app:
    restart: always
    image: $DOCKER_REGISTRY/dev/app:$CI_COMMIT_SHA
    container_name: dev_ml_service
    ports:
      - "8004:80"
    environment:
      - MLFLOW_TRACKING_URI=$MLFLOW_TRACKING_URI
      - MLFLOW_S3_ENDPOINT_URL=$MLFLOW_S3_ENDPOINT_URL
```

  - из за причин, описанных выше, решил использовать одни и те же переменные гитлаб MLFLOW_TRACKING_URI/MLFLOW_S3_ENDPOINT_URL как для юнит тестов, так и для нашего итогового сервиса, в связи с этим изменяем значения наших переменных в гитлабе (только в гитлабе Settings -> CI/CD -> Variables, нигде более менять их не нужно)
    - Примечание для Windows и Docker Desktop на основе WSL2:
      - в качестве <ip_docker_server_host> должен быть указан ip адрес wsl2 (обычно начинается с октета 172.)
        - ip адрес wsl2 можно узнать через PowerShell выполнив wsl hostname -I (должна быть именно заглавная английская "и")
      - стоит учитывать, что при каждой перезагрузке хостовой машины, этот адрес будет меняться, но если необходимо, его можно выставить статическим (инструкцию приведить не буду, многое зависит от версий ОС/WSL, но все гуглится)

        - **MLFLOW_TRACKING_URI**, значение: http://<ip_docker_server_host>:5000
        - **MLFLOW_S3_ENDPOINT_URL** , значение: http://<ip_docker_server_host>:9000
    - Дополнительно, косметически (логика не изменилась) скорректировал файл Docker/ml_service

```
FROM python:3.9
WORKDIR /code
COPY ./pyproject.toml /code/
COPY ./poetry.lock /code/
COPY ./src /code/src
COPY ./Docker/ml_service/.env /code/.env
RUN pip install --upgrade pip
RUN pip install poetry

# Variant 1. Installing packages via poetry:
RUN poetry config virtualenvs.create false \
    && poetry install --no-dev --no-interaction --no-ansi

# Variant 2. Installing packages via pip (get deps from poetry):
# RUN poetry export --without-hashes -f requirements.txt --output requirements.txt
# RUN pip install -r requirements.txt

CMD ["uvicorn", "src.app.inference:app", "--host", "0.0.0.0", "--port", "80"]
```

    - в настройках раннера выставляем ему теги mlops, prod, а так же убираем галку**Run untagged jobs**
    - так же в конфигурационном файле раннера выставил параметр concurrent = 1, что бы не было проблем с производительностью (если проблем нет, то выставлять не нужно)
    - По итогу всех настроек, убеждаемся что на машине с раннером подняты все необходимые контейнеры смежных сервисов, и производим следующие действия:
        - **git add .**
        - **git commit -m "Complete homework9 CI/CD"**
        - **git push origin ie_homework9_ci_cd**
        - при этом в через вебинтерфейс гитлаба, можно убедится, запустился пайплайн для ветки ie_homework9_ci_cd (в нем параллельно запустились 2 задания test_lint и pytest и стеджа tests)
        - через вебинтерфейс гитлаба делаем мердж реквест ветки в main, при этом

        - если к этому моменту запущенные задания уже успешно выполнились, то нам станет доступна кнопка merge (при условии получения необходимого количества аппрувов, в случае командной работы)
        - если задания еще не завершились (а аппрувы уже есть), то нам будет доступна кнопка merge after pipeline succeeded
    - после проведения мерджа, запустится следующий пайплайн, уже для ветки main, в котором уже последовательно выполнится задание docker_build стейджа build, затем задание docker_deploy стеджа deploy (причем задание стейджа deploy запустится только в случае успешного завершения задания стейджа build, иначе оно будет пропущено и для него выставится статус skipped)
    - убеждаемся что пайплайн для main полностью отработал
    - по итогу, проверяем, что у нас поднялся контейнер из нашего dev docker-compose, и сервис доступен через браузер или postman по адресу http://127.0.0.1:8004/invocations
    - **git checkout main**
    - **git pull**
    - **git dranch -d ie_homework9_ci_cd**

Финальный проект. Завершение финального проекта "MLOps на примере задачи матчинга геолокаций (на основе соревнования Kaggle Foursquare - Location Matching)"

Репозиторий: https://gitlab.com/mlops-23reg-team/mlops-23reg-project/

Доработки финального проекта на командном репозитории:

- Подготовительные шаги
    - git создал ветку ie_complete_project, перешел в нее (далее работаем в этой ветке)
        - **git branch ie_complete_project**
        - **git checkout ie_complete_project**
- Переписал код проекта в части изменения функционала основного app сервиса ml_service:

    - ранее сервис представлял из себя голую опубликованную модель, принимал на вход уже предобработанный .csv файл с набором готовых фичей, на выходе возвращал json response, в виде массива бинарной классификации на основе попарного сравнения выбранных POI (0 - объекты различны, 1 - объекты являются дублями)
    - после изменений, сервис осуществляет end-to-end обработку, т.е. на вход получает .csv файл оригинального формата (с колонками id,name,latitude,longitude,address,city,state,zip,country,url,phone,categories), по итогу возвращает .csv файл, требуемого по условиям задачи (соревнования на kaggle) формата (с колонками id, matches)
- В связи с тем, что сервис теперь имеет end-to-end функционал, и модель в его составе должна обрабатывать датасет с тем набором фичей, на котором она обучалась, при этом сам проект должен подразумевать возможность удобного проведения экспериментов и итерационное развитие модели, включая изменение пайплайна (в том числе набора фичей), проект должен позволять разграничивать версии пайплайна (кода) с которым работает текущая модель и с которым могут проводиться текущие эксперименты. Как следствие добавлены следующие походы по версионированию:
    - При изменении пайплайна (набора фичей, способа их расчета и т.п.), **необходимо изменять версию проекта, делается это путем:**
        - предварительного создания ветки (это стандартная практика)
        - изменения [tool.poetry].version в файле pyproject.toml (формат SemVer, т.е. например 0.1.0)
        - пересборка текущего проекта через poetry (например через poetry add mlops23regproject@latest), что бы в виртуальной среде так же обновилась версия проекта
        - внесения необходимых изменений в код пайплайна
        - тестирование нормально работы кода (возможно проведение экспериментов)
        - коммита в гит, с сообщением, что меняем версию пайплайна
        - пуш ветки в удаленный репозиторий и проведение процедуры merge request
        - после удачного проведения merge request добавляем к мердж коммиту соответствующий tag в формате v0.1.0, например локально это можно сделать через
            - находясь в **main**
            - **git pull**
            - **git tag -af v0.1.0 -m "Project version 0.1.0"**
            - **git push v0.1.0** (или **git push --tags**, в этом случае запушатся все локальные теги)

- Кроме того, код проекта изменен таким образом, что при трекинге экспериментов и моделей в mlflow так же учитывается текущая версия проекта (как следствие пайплана), это выражается в автоматическом добавлении к имени эксперимента и имени модели постфикса версии (в формате _v0.1.0), информация о версии добавляется автоматически путем получения текущей версии пакета через pkg_resources.get_distribution("mlops23regproject").version
- Важные замечания по изменению процесса CI/CD
  - в переменные гитлаб (и только в них, т.е. в .env добавлять не нужно) обязательно добавляется переменная **APP_PROJECT_VERSION**, в которую, в формате 0.1.0 заносится версия проекта (пайплайна) с на основе которого будет поднят нам сервис **dev_ml_service**
  - в ходе выполнения задания (job) docker_build нашего ci/cd пайплайна, версия кода будет "откачена" на соответствующую значению переменной **APP_PROJECT_VERSION** версию (производится это через **git checkout v$APP_PROJECT_VERSION**, как следствие для успешного исполнения должен сучествовать соответсвующий тег вида v0.1.0)
  - по итогу будет поднят соответствующий сервис на порту 8004, проверить работу можно через postman
    - POST
    - http://127.0.0.1:8004/invocations
    - BODY -> from-data
    - в первой строке, в поле key вводим "file" (это имя параметра функции async def create_upload_file(file: UploadFile = File(...)), в этой же ячейке справа в выпадающем списке выбираем File
    - указываем предварительно подготовленный файл .csv с несколькими строками, исходного формата датафрейма, например такого содержания (важно, что для текущего варианта бейзлайна, как минимум 2 POI должны находиться географически рядом, ближе чем на 100м друг к другу, иначе алгоритм не сможет сгенерировать парный файл, это легко дорабатывается в коде, но в текущем проекте не было задачи править подобное поведение):

```
id,name,latitude,longitude,address,city,state,zip,country,url,phone,categories
E_00001118ad0191,Jamu Petani Bagan Serai,5.012169,100.535805,,,,,MY,,,Cafés
E_000020eb6fed40,Johnny's Bar,40.43420889065938,-80.56416034698486,497 N 12th St,Weirton,WV,26062,US,,,Bars
E_00002f98667edf,QIWI,47.215134,39.686088,"Межевая улица, 60",Ростов-на-Дону,,,RU,https://qiwi.com,+78003011131,ATMs
E_001b6bad66eb98,"Gelora Sriwijaya, Jaka Baring Sport City",-3.01467472168758,104.79437444575598,,,,,ID,,,Stadiums
E_0283d9f61e569d,Stadion Gelora Sriwijaya,-3.021726757527373,104.78862762451172,Jalan Gubernur Hasan Bastari,Palembang,South
Sumatra,11480.0,ID,,,Soccer Stadiums
E_00002f98667edf_copy,QIWI,47.215134,39.686088,"Межевая улица, 60",Ростов-на-Дону,,,RU,https://qiwi.com,+78003011131,ATMs
E_001b6bad66eb98_copy,"Gelora Sriwijaya, Jaka Baring Sport City",-3.01467472168758,104.79437444575598,,,,,ID,,,Stadiums
```

- нажимаем send, по итогу должны получить ответ в виде отданного .csv файла, для примера выше, ответ будет следующим:

```
id,matches
E_00001118ad0191,E_00001118ad0191
E_000020eb6fed40,E_000020eb6fed40
E_00002f98667edf,E_00002f98667edf E_00002f98667edf_copy
E_001b6bad66eb98,E_001b6bad66eb98 E_001b6bad66eb98_copy
E_0283d9f61e569d,E_0283d9f61e569d
E_00002f98667edf_copy,E_00002f98667edf_copy E_00002f98667edf
E_001b6bad66eb98_copy,E_001b6bad66eb98_copy E_001b6bad66eb98
```

Итоговый технологический стек проекта:

- Язык разработки: Python 3.9.
- Менеджмент зависимостей: poetry.
- Шаблон проекта: cookiecutter data science.
- Система контроля версий кода – git/gitlab.
- Система версионирования данных – dvc + minio.
- workflow менеджер – dvc.
- Инструмент контроля codestyle.
  - Линтеры – pylint, flake8, mypy, bandit;
  - Форматтер - black.
- Трекинг экспериментов – mlflow по сценарию 4 (СУБД PostgreSQL, s3 minio).
- CLI: click.
- Модель – XGBClassifier.
- Тестирование: pytest, great expectations, dvc repro.
- Api – FastAPI+Uvicorn.
- Runtime – docker.

Инструкция по запуску всей инфраструктуры финального проекта на новой машине (инструкция под Windows):

- Примечания: ниже описаны шаги от самого начала и до запуска целевого микросервиса, при этом, для запуска целевого микросервиса, необходимо будет связать локальный репозиторий со своим удаленным, т.к. доставка и деплой сервиса осуществляется через гитлаб пайплайн, и какой-либо комит в main ветку, для нижеуказанного репозитория права есть только у участников команды mlops-23reg-team
- Устанавливаем предварительно в систему:
  - **python 3.9**
  - **poetry**
  - **Docker Desktop (в виде бэкенда WSL2)**
- **git clone https://gitlab.com/mlops-23reg-team/mlops-23reg-project.git**
- все нижеуказанные команды выполняем находясь в корневой директории проекта
- создаем в корне проекта файл mlops-23reg-team следующего содержания:

```
MINIO_ROOT_USER = admin
MINIO_ROOT_PASSWORD = miniotestpass
AWS_ACCESS_KEY_ID = L6Vt9Pw72XDw26Mt
AWS_SECRET_ACCESS_KEY = IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G
AWS_S3_MLFLOW_BUCKET = mlflow
AWS_S3_DVC_BUCKET = dvc
MLFLOW_S3_ENDPOINT_URL = "http://localhost:9000"
MLFLOW_TRACKING_URI = "http://localhost:5000"
POSTGRES_USER = dbuser
POSTGRES_PASSWORD = dbtestpass
POSTGRES_DB = mlflow
PGADMIN_DEFAULT_EMAIL = "admin@admin.com"
PGADMIN_DEFAULT_PASSWORD = pgtestpass
```

- создаем файл **.dvc/config.local** следующего содержания

```
['remote "s3minio"']
    access_key_id = L6Vt9Pw72XDw26Mt
    secret_access_key = IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G
```

- **docker-compose up -d --build** (дожидаемся старта контейнеров)

- производим предварительную настройку minio s3
  - Заходим через браузер http://127.0.0.1:9001 и авторизуемся admin/miniotestpass
  - Через веб-интерфейс создаем два бакета:
    - **Buckets -> Create bucket**
      - **mlflow**
      - **dvc**
  - Через веб-интерфейс создаем сервисный аккаунт (по сути API Key):
    - **Identity -> Service Accounts -> Create service account**
    - Name: **L6Vt9Pw72XDw26Mt**
    - Pass: **IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G**
- Добавляем в проект исходные данные (датасеты)
  - Заходим в соревнование https://www.kaggle.com/competitions/foursquare-location-matching/data
  - копируем из этого соревнования файлы train.csv, test.csv, sample_submission.csv, pairs.csv в директорию проекта ./data/raw/
- Создаем виртуальную среду
  - **poetry install**
  - **poetry shell**
- Запускаем эксперимент
  - **dvc repro**
- Заходим в mlflow через http://127.0.0.1:5000
  - проверяем что трекинг эксперимента успешно добавлен
  - заходим в модели и выставляем для текущей модели **stage = staging**
- Все последующие шаги возможны, только после привязки локального репозитория к удаленному в гитлабе, в который есть полный доступ (перепривязываем в какой-либо свой)
- Для запуска целевого сервиса, на текущей машине необходимо поднять гитлаб раннер (подробная инструкция есть в подразделе "НастраиваемлокальныйCI Runner" дневника курса)
- В удаленном репозитории gitlab в Settings -> CI/CD -> Variables создаем следующие переменные:
  - **AWS_ACCESS_KEY_ID** (Protected: False, Masked: True): **L6Vt9Pw72XDw26Mt**
  - **AWS_SECRET_ACCESS_KEY** (Protected: False, Masked: True): **IEnnVlaJqfKgNWKTFu1RE9Uk8jSfd52G**

    - **MLFLOW_S3_ENDPOINT_URL** (Protected: False, Masked: False): http://<ip_docker_server_host>:9000 (<ip_docker_server_host> узнаем через PowerShell выполнив wsl hostname -I (должна быть именно заглавная английская "и")
    - **MLFLOW_TRACKING_URI** (Protected: False, Masked: False): http://<ip_docker_server_host>:5000
    - **APP_PROJECT_VERSION**(Protected: False, Masked: False): 0.1.0
- **git tag -af v0.1.0 -m "Project version 0.1.0"**
- **git push v0.1.0** (или git push --tags, в этом случае запушатся все локальные теги)
- **git push origin main**
- По итогу отработки всех gitlab пайплайнов должны получить работающий контейнер dev_ml_service и API по адресу **http://127.0.0.1:8004/invocations**:
- Проверить работу API можно через postman
    - POST
    - http://127.0.0.1:8004/invocations
    - BODY -> from-data
    - в первой строке, в поле key вводим "file" (это имя параметра функции async def create_upload_file(file: UploadFile = File(...)), в этой же ячейке справа в выпадающем списке выбираем File
    - указываем предварительно подготовленный файл .csv с несколькими строками, исходного формата датафрейма, например такого содержания (важно, что для текущего варианта бейзлайна, как минимум 2 POI должны находиться географически рядом, ближе чем на 100м друг к другу, иначе алгоритм не сможет сгенерировать парный файл, это легко дорабатывается в коде, но в текущем проекте не было задачи править подобное поведение):

```
id,name,latitude,longitude,address,city,state,zip,country,url,phone,categories
E_00001118ad0191,Jamu Petani Bagan Serai,5.012169,100.535805,,,,,MY,,,Cafés
E_000020eb6fed40,Johnny's Bar,40.43420889065938,-80.56416034698486,497 N 12th St,Weirton,WV,26062,US,,,Bars
E_00002f98667edf,QIWI,47.215134,39.686088,"Межевая улица, 60",Ростов-на-Дону,,,RU,https://qiwi.com,+78003011131,ATMs
E_001b6bad66eb98,"Gelora Sriwijaya, Jaka Baring Sport City",-3.01467472168758,104.79437444575598,,,,,ID,,,Stadiums
E_0283d9f61e569d,Stadion Gelora Sriwijaya,-3.021726757527373,104.78862762451172,Jalan Gubernur Hasan Bastari,Palembang,South
Sumatra,11480.0,ID,,,Soccer Stadiums
E_00002f98667edf_copy,QIWI,47.215134,39.686088,"Межевая улица, 60",Ростов-на-Дону,,,RU,https://qiwi.com,+78003011131,ATMs
E_001b6bad66eb98_copy,"Gelora Sriwijaya, Jaka Baring Sport City",-3.01467472168758,104.79437444575598,,,,,ID,,,Stadiums
```

    - нажимаем send, по итогу должны получить ответ в виде отданного .csv файла, для примера выше, ответ будет следующим:

```
id,matches
```

```
E_00001118ad0191,E_00001118ad0191
E_000020eb6fed40,E_000020eb6fed40
E_00002f98667edf,E_00002f98667edf E_00002f98667edf_copy
E_001b6bad66eb98,E_001b6bad66eb98 E_001b6bad66eb98_copy
E_0283d9f61e569d,E_0283d9f61e569d
E_00002f98667edf_copy,E_00002f98667edf_copy E_00002f98667edf
E_001b6bad66eb98_copy,E_001b6bad66eb98_copy E_001b6bad66eb98
```

На этом все, удачи!