Проект "MLOps на примере задачи матчинга геолокаций (на основе соревнования Kaggle Foursquare - Location Matching)"

Ссылка на курс: https://ods.ai/tracks/ml-in-production-spring-22
Репозиторий проекта: https://gitlab.com/mlops-23reg-team/mlops-23reg-project

1. Краткое описание ML проекта:

Реальная цель проекта - освоить подходы MLOps на основе практической ML-задачи. Важно понимать, что целью проекта не является реализация эффективной ML части, а только поэтапное построение ее обертки в виде MLOps, в свою очередь, данная обертка должна позволять, в дальнейшем, эффективно проводить исследования и итеративно улучшать работу целевого сервиса. Как следствие, ML часть сводится, к написанию наиболее простого, но работающего бейзлайна.
В качестве ML задачи, выбрана задача матчинга геолокации из активного соревнования на Kaggle Foursquare - Location Matching (https://www.kaggle.com/competitions/foursquare-location-matching). На входе имеем датасет, состоящий из точек интереса (POI) и информации о них. Датасет сформирован автоматизированными методами, путем получения информации из различных открытых и не только источников. Информация о POI (из разных источников) часто имеет несоответствия, избыточность, конфликты, двусмысленность, позиционную неточность и т.п. Необходимо решить задачу определения дубликатов POI (для дальнейшей их обработки, получения максимально полной и достоверной информации о каждой POI).

Постановка задачи, в рамках текущего проекта (легенда)

Представим себе, что разработчики проекта являются сотрудниками компании Foursquare (один из лидеров в части поставок информации по глобальным POI). Необходимо создать внутренний сервис сопоставления POI, доступный внутри организации посредством API, а так же дизайн проекта, позволяющий итеративно улучшать работу сервиса, в том числе, эффективно проводить исследования (не затрагивая при этом работу основного сервиса).
Рамки проекта и задач:

- сервис должен функционировать в виде api, пользовательский интерфейс не требуется;
- сервис должен принимать на вход .csv файл, заданного формата (формат описан ниже, в соответствующем разделе), содержащий в себе информацию о собранных POI;
- сервис должен возвращать в качестве ответа .csv файл, заданного формата, с данными о найденных дубликатах POtest_lintl;

- сервис должен функционировать исключительно внутри организации и доверенной инфраструктуры, предпринимать меры безопасности, требуемые для возможности публикации сервиса в Интернет, не требуется;
- проект должен предоставлять возможность проведения исследований и экспериментов (включая всевозможные изменения ML пайплайна, предобработки, фиче-инжиниринга и т.п.), не затрагивая при этом штатную работу текущей версии сервиса;
- проект должен иметь возможность итерационно улучшать работу сервиса, путем выкатки новых версий пайплайна и моделей, показавших лучшие результаты в ходе исследований;
- допустимая численность команды, отвечающей за данный сервис 1-5 чел.

Описание исходных данных

Имеется набор данных из более чем полутора миллионов записей мест (POI), в датасете присутствует шум, дублирование, посторонняя и иногда неверная информация.

train.csv - представляет собой обучающий набор, состоящий из одиннадцати атрибутивных полей для более чем миллиона записей о POI, в том числе:

- *id* - уникальный идентификатор для каждой записи;
- *point_of_interest* - уникальные идентификаторы POI, сформированные предварительной разметкой (преимущественно экспертной); т.е. все строки, с одним и тем же значением *point_of_interest*, являются дублями; по сути, колонка является источником нашей целевой переменной, на основе которой мы сможем обучить модель; в реальных данных, с которыми в дальнейшем будет работать сервис, эта колонка отсутствует;
- *latitude, longitude* - географические координаты;
- name, address, city, state, zip, country, url, phone, categories- данные о названии, адресе, городе, регионе, индексе, стране, url-адресе, телефонном номере и категории соответственно.

pairs.csv - файл сгенерирован только на основе информации из train.csv и не несет в себе никаких доп. данных. Цель этого файла, показать возможное решение задачи, путем ее сведения к задаче бинарной классификации. Принцип формирования файла, рандомное сопоставление 2-х разных строк из train.csv (с доп. постфиксами _1, _2), за исключением point_of_interest, дополнительно в колонка match проставляется True, если объекты имели одинаковый point_of_interest, иначе False. Файл содержит 578 907 строк и не используется в рамках текущего проекта.

sample_submission.csv - файл-пример требуемого выходного формата, количество строк, должно соответствовать количеству строк поданного на вход датасета, содержит в себе следующие колонки

- *id* - исходные id
- matches - список id найденных дублей, через пробел (список в том числе должен включать в себя сам исходный id, если дублей не найдено, то в ячейке будет значиться только исходный id)

Результаты анализа и обработки данных

Был проведен краткий EDA-анализ.
Структура данных:

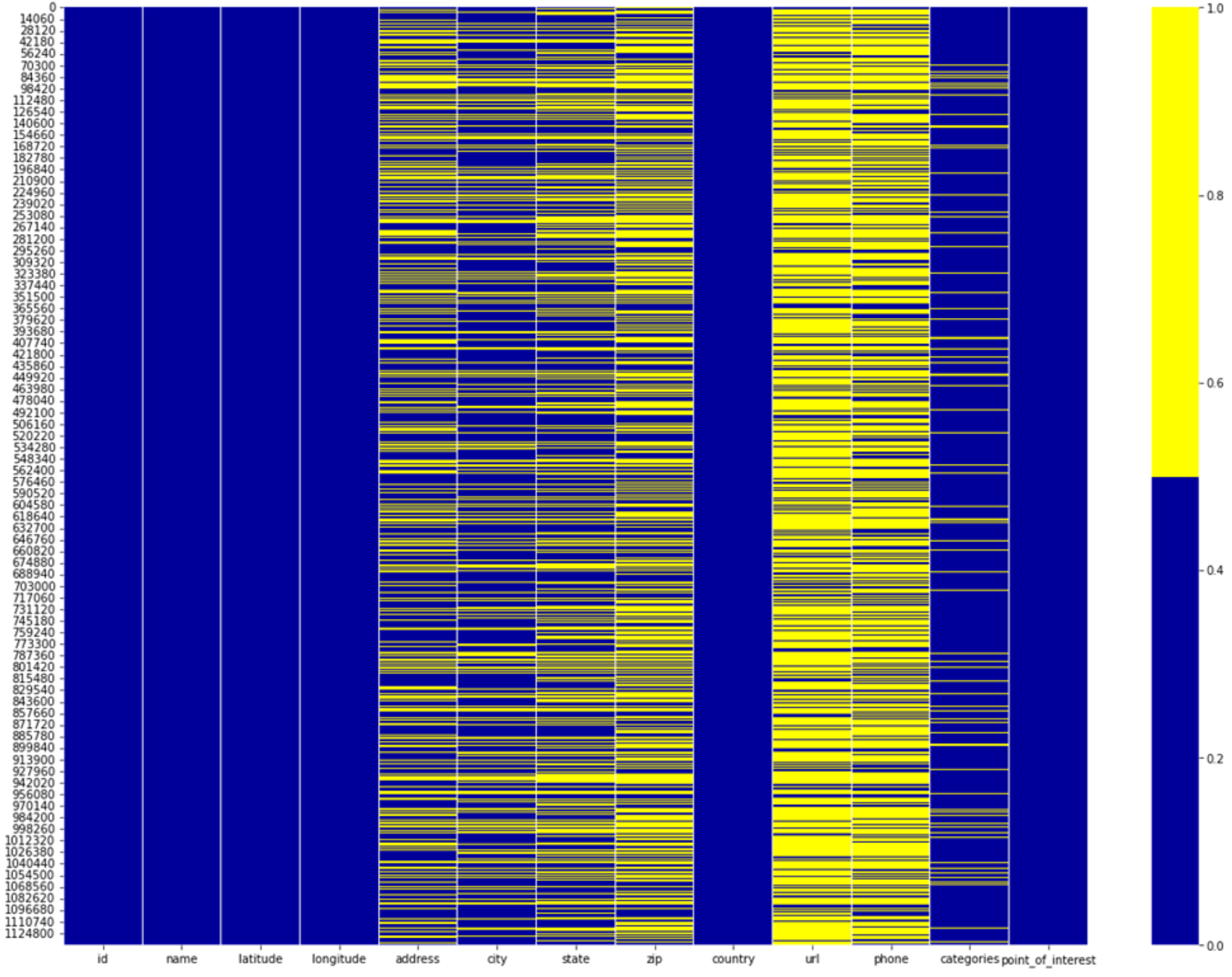
id	name	latitude	longitude	address	city	state	zip	country	url	phone	categories	p
E_ed82ad65edea24	coral coffee	36.308522	126.515174	NaN	NaN	NaN	NaN	KR	NaN	NaN	Cafés	
E_108ad4a9c1a3fa	สำนักงานสัสดีจังหวัดยะลา	6.539897	101.281741	ศาลากลางจังหวัดยะลา	NaN	ยะลา	NaN	TH	NaN	NaN	Offices	F
E_3bd862c1fe5deb	Halte Geluwe Leiestraat	50.808224	3.084615	NaN	NaN	West-Vlaanderen	NaN	BE	https://www.delijn.be	+3270220200	Bus Stops	F
E_340df1dc263d33	Radio Zammú	37.506310	15.078397	Ex Monastero dei Benedettini	Catania	Sicilia	95124	IT	NaN	NaN	Student Centers	
E_041bc4707bc81c	Quem Disse Berenice?	-22.731375	-43.454578	Av. Governador Roberto	Nova Iguaçu	Rio de Janeiro	NaN	BR	NaN	NaN	Cosmetics Shops	

Словесные описания заполнены на разных языках.

И общая информация о датасете:

```
RangeIndex: 1138812 entries, 0 to 1138811
Data columns (total 13 columns):
 #   Column             Non-Null Count    Dtype  
---  ------             --------------    -----  
 0   id                 1138812 non-null  object 
 1   name               1138811 non-null  object 
 2   latitude           1138812 non-null  float64
 3   longitude          1138812 non-null  float64
 4   address            742191 non-null   object 
 5   city               839623 non-null   object 
 6   state              718226 non-null   object 
 7   zip                543386 non-null   object 
 8   country            1138801 non-null  object 
 9   url                267724 non-null   object 
 10  phone              342855 non-null   object 
 11  categories         1040505 non-null  object 
 12  point_of_interest  1138812 non-null  object 
dtypes: float64(2), object(11)
memory usage: 112.9+ MB
```

Видно, что есть много пропущенных значений в разных колонках. Тепловая карта пропущенных значений.


0
14060
28120
42180
56240
70300
84360
98420
112480
126540
140600
154660
168720
182780
196840
210900
224960
239020
253080
267140
281200
295260
309320
323380
337440
351500
365560
379620
393680
407740
421800
435860
449920
463980
478040
492100
506160
520220
534280
548340
562400
576460
590520
604580
618640
632700
646760
660820
674880
688940
703000
717060
731120
745180
759240
773300
787360
801420
815480
829540
843600
857660
871720
885780
899840
913900
927960
942020
956080
970140
984200
998260
1012320
1026380
1040440
1054500
1068560
1082620
1096680
1110740
1124800
id
name
latitude
longitude
address
city
state
zip
country
url
phone
categories
point_of_interest
1.0
0.8
0.6
0.4
0.2
0.0

Первичный EDA-анализ показал, что в исходном датасете существуют колонки с множеством пропущенных значений, все составляющие ключа POI - категориальные, кроме географических координат долготы и широты. Больше всего пропущенных значений в колонках url, phone, и zip. В колонках имени, долготы и широты нет пропущенных значений, но могут быть неверные данные, кроме того, для одних и тех же POI, но полученных из разных источников мы можем наблюдать существенные различия по всем колонкам.

Описание бейзлайна

Сводим задачу к бинарной классификации, путем отбора кандидатов (строк) для последующего их сравнения на предмет идентичности. Сравнить все пары строк между собой не представляется возможным, т.к. мы для нашего случая получили бы парный датасет из более чем 10^12 строк.
В бейзлайне, отбор кандидатов производим путем округления координат *latitude, longitude* до заданного знака после запятой (соответствующего например расстоянию 100м, 1 км, и т.п.), с последующим объединением строк в пары, через merge на основе одинаковых значений колонки = конкатенации округленных latitude, longitude.
Далее, для каждой строки полученного парного датасета, формируем следующие фичи:

- географическое расстояние в км. между исходными точками;
- расстояние Левенштейна между строками-названиями исходных точек.

Колонку целевой переменной формируем путем сравнения point_of_interest_1, point_of_interest_2, если они идентичны, целевая переменная выставляется в 1, иначе в 0.
Обучаем XGBClassifier на полученных данных.
На основе полученных предсказаний, формируется выходной .csv файл формата sample_submission.csv.

Метрики качества

В качестве основной метрики используется Jaccard score, который равен среднему значению из Jaccard index всех строк. Формула расчета Jaccard index приведена с статье на википедии https://en.wikipedia.org/wiki/Jaccard_index

В процессе обучения, Jaccard score расчитывается на двух стадиях:

- на стадии формирования датасета пар кандидатов, это своего рода максимальный Jaccard score, который мы сможем получить, решая задачу бинарной классификации на текущем парном датасете;
- на финальной стадии, после получения предсказаний модели, это реальный, итоговый Jaccard score

2. Описание MLOPS подходов

Система контроля версий
В качестве системы контроля версий, выбран git. Причина выбора - наиболее распространенная система, отвечающий всем заявленным к проекту требованиям.
В качестве хранилища удаленного репозитория, выбран GitLab. Причины выбора - необходимость получения практического опыта работы с GitLab (до курса, имел только незначительный опыт работы с GitHab).

Инструменты контроля codestyle
В качестве инструментов codestyle выбраны следующие:

- форматеры - Black (максимальная длина строки 100 символов); кроме black были опробованы форматеры autopep8 и yapf (от Google), в итоге выбран black, как наиболее безкомпромисный, и действительно смособный привести код разных разработчиков в единообразный формат;
- линтеры:
  - pylint - наиболее строгий и требовательный линтер (осуществляет как логические, так и стилистические проверки);
  - flake8 - очень распространенный и гибко настраиваемый линтер;
  - mypy - осуществляет логические проверки, касающиеся корректности работы с типами данных (основной источник информации это аннотации типов);
  - bandit - выполняет анализ кода на предмет выявления небезопасного кода

В качестве точек контроля выбраны:

- IDE (в моем случае VSCode), конфигурация выглядит следующим образом:

```
{
    "python.formatting.provider": "black",
    "editor.formatOnSave": true,
    "python.linting.flake8Enabled": true,
    "python.linting.enabled": true,
    "python.linting.mypyEnabled": true,
```

```
    "python.linting.banditEnabled": true,
    "python.linting.pylintEnabled": true
}
```

- в пайплайне gitlab-ci.yml в задаче (job) test_lint стейджа tests, следующим образом

```
    - poetry run flake8 src
    - poetry run mypy --ignore-missing-imports src
    - poetry run bandit src
    - poetry run pylint src
```

Кроме того, был рассмотрен вариант добавления дополнительно точки проверки линтерами через git pre commit hooks, но посчитал этот вариант избыточным, т.к. у нас основная цель защитить основные ветки удаленного репозитория (в нашем случае main) от "плохого" кода, в свою очередь, иногда требуется сделать быстрый локальный коммит, без отработки всех замечаний линтеров, а в случае pre commit hooks этого сделать не удастся.

Шаблон проекта / шаблонизаторы / структура проекта

В качестве шаблонизатора и шаблона был выбран Cookiecutter с шаблоном для Data Science от DrivenData (https://github.com/drivendata/cookiecutter-data-science)
В качестве менеджера зависимостей выбран poetry.
Исходный код проекта в соответствии с выбранным шаблоном располагается в подкаталогах директории src и оформлен в виде модуля.
Для сборки модуля в пакет в фале pyproject.toml для poetry добавлены следующие настройки:

```
[tool.poetry]
name = "mlops23regproject"
version = "0.1.0"
description = "MLOps cource project by 23reg team."
authors = ["MLOps-23reg-team"]
license = "MIT"
packages = [
    { include = "src", from = "." }
]
```

Данные, обрабатываемые в рамках проекта располагаются в подкаталогах директории data и так же соответствуют концепциям Cookiecutter DS шаблона.

```
├── LICENSE
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number (for ordering),
│                         the creator's initials, and a short `-` delimited description, e.g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in reporting
│
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
```

```
|   └── visualization  <- Scripts to create exploratory and results oriented visualizations
|       └── visualize.py
```

Workflow менеджеры

В качестве workflow менеджера и инструмента версионированиий данных выбран DVC.

Заметки к версионированию данных:

- бэкэндом для версионирования данный является локально развернутый s3 сервис на базе minio (сервис состоит из 2-х докер контейнеров, непосредственно minio, и nginx в качестве кэширующего прокси);
- для DVC в сервисе s3 выделен бакет dvc;
- учетные данные для доступа к s3 хранятся не в конфигурационном файле уровня проекта .dvc/config, а в локальном типе конфигурационного файла .dvc/config.local, который исключен из контроля git; для целей unit тестирования в ходе исполнения gitlab ci пайплайна, соответствующий файл генерируется "налету" в ходе исполнения самой задачи раннером, путем исполнения следующих команд:
  - 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
  - переменные $AWS_ACCESS_KEY_ID/$AWS_SECRET_ACCESS_KEY хранятся в разделе Variables гитлаба
- пайплайн DVC настроен таким образом, что бы в s3 версионировались не все промежуточные файлы, а только те, для которых это действительно целесообразно
  - исходные файлы датасетов, а так же файлы после разбиения на train/test;
  - файлы метрик (исключительно в образовательных целях);
  - файлы моделей (так же больше в образовательных целях, т.к. итоговое версионирование моделей осуществляется через MLFlow);
  - для всех остальных файлов, фигурирующих в outs пайплайна dvc.yaml выставлен параметр cache: false (в этом случае dvc продолжает хранить копии файлов в локальном кэше, но не отправляет их на хранение в s3; при запуске dvc repro и отсутствии подобных файлов, стейджи пайплайна, которые генерируют необходимые файлы, будут перезапущены).

Заметки к workflow пайплайну:

- пайплайн реализован через файл dvc.yaml и дополнительный файл params.yaml;
- код проекта разбит на логические модули, предполагающие возможность их исполнения из CLI (реализовано путем декарирования соответствующих функций с помощью click);

- dvc.yaml содержит описание стейджей (стадий), в виде запуска соответствующих им python модулей с заданными параметрами, а так же описанием файлов от которых зависит каждый стейдж и выходных файлов, которы должны быть сгенерированы по итогу исполнения стеджа;
- исполнение пайплайна осуществляется через вызов команды из CLI: dvc repro;
- после получения соответствующие команды, DVC строит DAG, и запускает стейджи в соответствующем порядке (в том числе параллельно, если это допустимо).

Инструменты трекинга экспериментов

Трекинг экспериментов и моделей осуществляется с помощью MLFlow. MLFlow развернут локально, в докер контейнере, по сценарию 4:

- в качестве СУБД (хранилище трекинга экспериментов) выступает локально развернутый контейнер с PostgreSQL;
- в качестве s3 хранилища сертификатов выступает локально развернутый minio (используется бакет mlflow).

Методы и инструменты тестирования

Unit-тестирование осуществляется слудующими инструментами:

- pytest, с двумя скриптами в директории ./tests
  - test_evaluate.py - осуществляет имитацию вызова функции обернутой в декораторы click из CLI;
  - test_predict_model.py - осуществляет имитацию вызова функции обернутой в декораторы click из CLI, дополнительно, через библиотеку great_expectations, проверяет формат выходного файла, генерируемого по итогу запуска модуля (конкретно добавлены проверки на соответсвие списку колонок, уникальность значений, отсутствие пропусков NaN);
- через запуск всего пайплайна командой dvc repro (тест будет пройден при отсутствии ошибок).

Указанные тесты выполняются в ходе исполнения задачи (job) pytest, стеджа tests пайплайна gitlab-ci.yml

Описание CI пайплайна

Важно помнить, по умолчанию, что в ходе merge-request, имеет место 2 запуска пайплайна gitlab-ci.yml:

- первый раз пайплайн запускается при любом коммите/пуше ветки в удаленный репозиторий (т.е. он запускается сразу после коммита/пуши, даже тога, когда мы не собиремся делать merge-request в main);
- если мы решим сделать merge-request, то этот предварительно запущенный пайплайн, должен обязательно удачно завершиться (при соответствующих рекомендованных настройках защиты ветки main, когда выставляется параметр, что merge требует удачного завершения пайплайна);

- по итогу отработки merge-request пайплайн будет вновь замущен на исполнение, но уже для ветки main.

Пайплайна gitlab-ci.yml состоит из 3-х стейджей, 2 из которых относятся к CI (будут описаны в этом разделе), и 1 относится к CD (будет описан в последующем разделе).
CI стейджи:

- tests - выполняется при коммите/пуше в любую ветку (кроме main), в рамках него параллельно запускаются следующие задачи:
  - test_lint - аудит кода линтерами;
  - pytest - юнит-тесты;
- build - выполняется только для коммитов в ветку main, в нашем случае, т.к. прямые коммиты запрещены, будет запускаться сразу после успешного отработанного merge-request в main; в рамках него запускается единственная задача:
  - docker_build - сборка и доставка в gitlab registry докер образа нашего сервиса dev_ml_service.

3. Описание получившегося сервиса/продукта

Сервис сервис реализован в виде докер контейнера dev_ml_service, предоставляющий API, функционал которого описан ниже.

API

API доступен по адресу http://<ip or hostname>:8004/invocations
Принимает на вход POST запрос входными данными в виде одного параметра file представляющего из себя .csv файл исходного формата, пример входного файла:

```
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
```

По итогу работы, возвращает файл требуемого по условию задачи формата, пример выходного файла:

```
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
```

Описание CD пайплайна

CD часть пайплайна реализована в виде следующего стейджа:

- deploy - выполняется загрузка готового докер образа нашего сервиса с gitlab registry и его деплой (запуск соответствующего контейнера); в рамках стейджа запускается единственная задача docker_deploy:
  - выполняется только удачного merge request в ветку main и только после успешного завершения CI стейджа build;
  - предварительно останавливаются и удаляются предыдущие версии контейнеров с заданным префиксом (_dev).

Итоговый технологический стек

Язык разработки: Python 3.9.
Менеджмент зависимостей: poetry.
Шаблон проекта: cookiecutter data science.
Система контроля версий кода – git/gitlab.
Система контроля версий 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.

4. Проблемы и недостатки текущего workflow и получившихся результатов. Возможные улучшения.

- Для стабильного функционирования API сервиса необходимо дополнительно использовать какой-либо менеджер очередей, возможно в связке nginx;
- Необходимо добавить end-to-end тестирование работы сервиса, в том числе напрашивается dev и prod среда;
- Работу сервиса необходимо добавить в какую-либо систему мониторинга (которая по легенде уже обязательно должны быть в организации);
- При каждом удачно разрешенном merge-request в ветку main, докер контейнер нашего сервиса будет остановлен и перезапущен, даже если внесенные изменения не затрагивают функционал самого сервиса, необходимо проводить перезапуск только в случае необходимости.