Сколковский институт науки и технологий

Руководство специалиста по анализу данных

Платформа DATASKAI

версия: 0.0.1

Москва
2020

Оглавление

Глава 1

Введение

1.1 Описание роли пользователя

Мы придерживаемся разделения дата-сайентистов (здесь и далее в этом документе - DS), работающих в вашем проекте, на три группы:

1. Джуниор дата-сайентист

   Типичные задачи: проведение разведочного анализа данных (EDA), создание и отправка моделей по задачам анализа данных

2. Мидл дата-сайентист

   Типичные задачи: валидация моделей, создание и проверка построителей признаков, запуск сервисов DATASKAI.

3. Сеньор дата-сайентист

   Типичные задачи: управление задачами по анализу данных, создание и проверка моделей предметной области, отслеживание результатов решения задач

Глава 2

Определения

Глава 3

Типовые задачи джуниор дата-сайентиста

Для сохранения соревновательного духа проектов по анализу данных, большинство задач джуниор дата-сайентиста ставятся в состязательной манере. Такой подход позаимствован у платформ по проведению соревнований по машинному обучению (Kaggle и другие). Упрощенно, подход состоит из цикла вида:

3.1 Получение Docker-контейнера с ML задачей

Вся работа дата-сайентиста должна проходить внутри Docker-контейнера. Чтобы получить ссылку на такой контейнер, обратитесь к мидл/сеньор дата-сайентисту вашей команды.

Как только вы получите ссылку, пройдите по ней в своём браузере. Откроется стандартный интерфейс Jupyter-ноутбука.

Рекомендация

Для подключения к Docker-контейнеру со своего локального компьютера рекомендуется использовать переадресацию портов с помощью команды „autossh“. Преимущества такого подключения:

1. Автоматическое восстановление ssh-соединения с Jupyter-ноутбуком в случае кратковременного разрыва соединения
2. Нет необходимости запоминать IP-адрес вашего внутреннего сервера

При выполнении команды „autossh“ используйте флаги -L и -N:

```
autossh -N -L 8888:some_serv_ip:8989 some_serv_ip
```

3.2 Файловая структура проекта

Поскольку работа над проектом по анализу данных является долгосрочной, важно поддерживать единообразную файловую структуру проекта внутри команды. Предлагаемая по умолчанию структура проекта задаёт расположение всех важных частей:

```
project_name                - name of your project
├── .git                    - git repository standard directory
├── configs                 - configs that would be used as starting point for DATASKAI tools␣
↪initialization
```

(continues on next page)

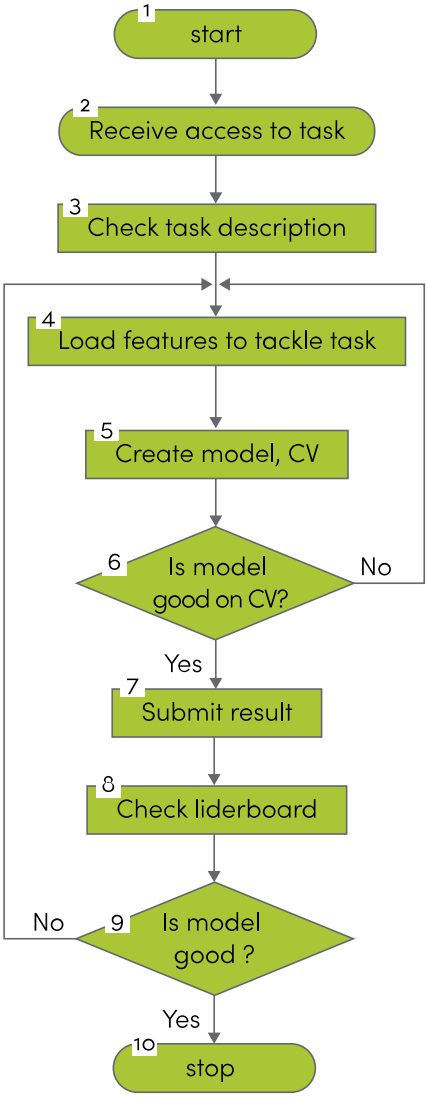


Рис.3.1: Схема решения ML задач джуниором дата-сайентистом

(продолжение с предыдущей страницы)

```
├── dockerfiles            - dockerfile for DS project container
├── data                   - folder for data
│   ├── external           - folder for small external data (small catalogs, references,␣
↪dicts) data, attached to git
│   ├── processed          - folder for processed data, derived from raw and loaded with raw_
↪data_loader (should
│   │                        be used as tmp folder for processed data, don't store actual␣
↪data here)
│   └── raw                - folder for raw data downloaded from somewhere and used in project␣
↪(should be used as tmp
│                            folder for data, don't store actual data here)
├── documents              - valuable pdf|other formats documents for current project
├── modules                - python modules, used in project
│   ├── subject_domain     - subject domain models for project {{cookiecutter.project_name}}␣
↪written in python
│   ├── evaluation_tools   - DATASKAI tools
│   ├── metrics            - metrics python code used by metrics_service to calculate metrics␣
↪for project
│   └── feature_extractors - feature_extractor modules of this project
├── notebooks              - notebooks with some research|exploration|model creation performed␣
↪by data scientists
├── scripts                - misc helping scripts
├── services               - DATASKAI services docker and compose files
├── tools                  - binary tools (if they're small, and couldn't be obtained in␣
↪simple ways) required for data
│                            processing
├── .env                   - DATASKAI environment file with variables for client and server␣
↪side
└── README.md              - README file for project
```

Внутри Docker-контейнера директория проекта project_name также содержит стандартную .git поддиректорию, так что сам проект является Git репозиторием.

3.3 Директория с Jupyter-ноутбуками

Внутри директории с Jupyter-ноутбуками в проекте рекомендуется использовать поддиректорию с вашим именем:

```
project_name               - name of your project
├── notebooks                   - folder for data
│   ├── elon.musk
│   ├── andrew.ng
│   └── stanislav.semenov
...
```

Также полезно нумеровать ваши Jupyter-ноутбуки, с помощью как минимум двух цифр, например:

```
01__Simple_EDA.ipynb
02__First_model.ipynb
03__First_CV_model.ipynb
...
```

Такая система именований облегчит вашим коллегам навигацию внутри вашей поддиректории.

3.4 Загрузка ML задачи в Jupyter-ноутбук

Создайте внутри поддиректории с вашим именем первый Jupyter-ноутбук и импортируйте в него класс TaskLoader:

```
import sys
sys.path.append('../../modules/evaluation_tools/')
from task_loader import TaskLoader
```

После этого вы можете инициализировать TaskLoader для конкретной ML задачи с помощью имени задачи и конфигурации по умолчанию:

```
TASK_CONFIG = json.load(open('../../configs/tasks_default_config.json'))
TASK_NAME = 'raw_data__aero__fw_classification_v1'

task_loader = TaskLoader(TASK_NAME, TASK_CONFIG['mongo_config'])
```

Для работы этих команд нужно, чтобы проект содержал файл с конфигурацией по умолчанию задачи и чтобы уже была поставлена задача с указанным именем.

3.5 Получение постановки задачи

После инициализации объекта task_loader, вы можете получить описание постановки задачи с помощью метода display_description_in_jupyter:

```
task_loader.display_description_in_jupyter()
```

Выполнение этой команды отобразит описание задачи в формате Markdown:

3.6 Загрузка исходных данных

Для загрузки исходных данных (например для проведения разведочного анализа данных), используйте компонент raw_data_loader из task_loader:

```
raw_data_loader = task_loader.raw_data_loader
```

Чтобы получить список всех доступных источников данных, выполните:

```
feature_records = [x['feature_manager_config']['feature_records'] for x in raw_data_loader.
↪list_configs()]
record_names = [x['name'] for x in feature_records[0]]
record_names
```

Теперь переменная record_names содержит имена всех доступных источников исходных данных для данной ML задачи.

Для получения pandas-датафрейма с исходными данными из конкретного источника, используйте:

```
some_raw_data = raw_data_loader.load_data_one_record('some_available_source')
```

Где „some_available_source“ — имя из списка record_names, полученного выше.

Предупреждение

Исходные данные обычно загружаются со строковым типом данных (в Pandas-датафрейме тип objects). Это

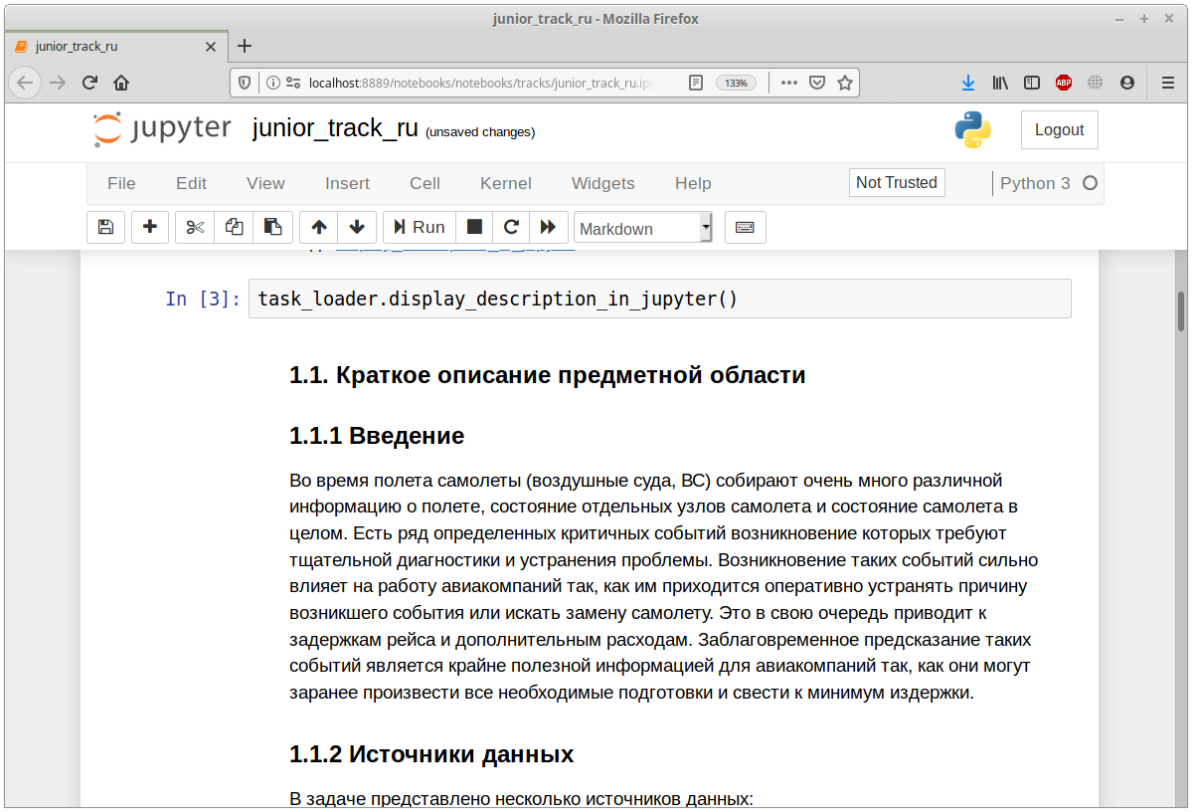


Рис.3.2: Описание постановки задачи в Jupyter-ноутбуке

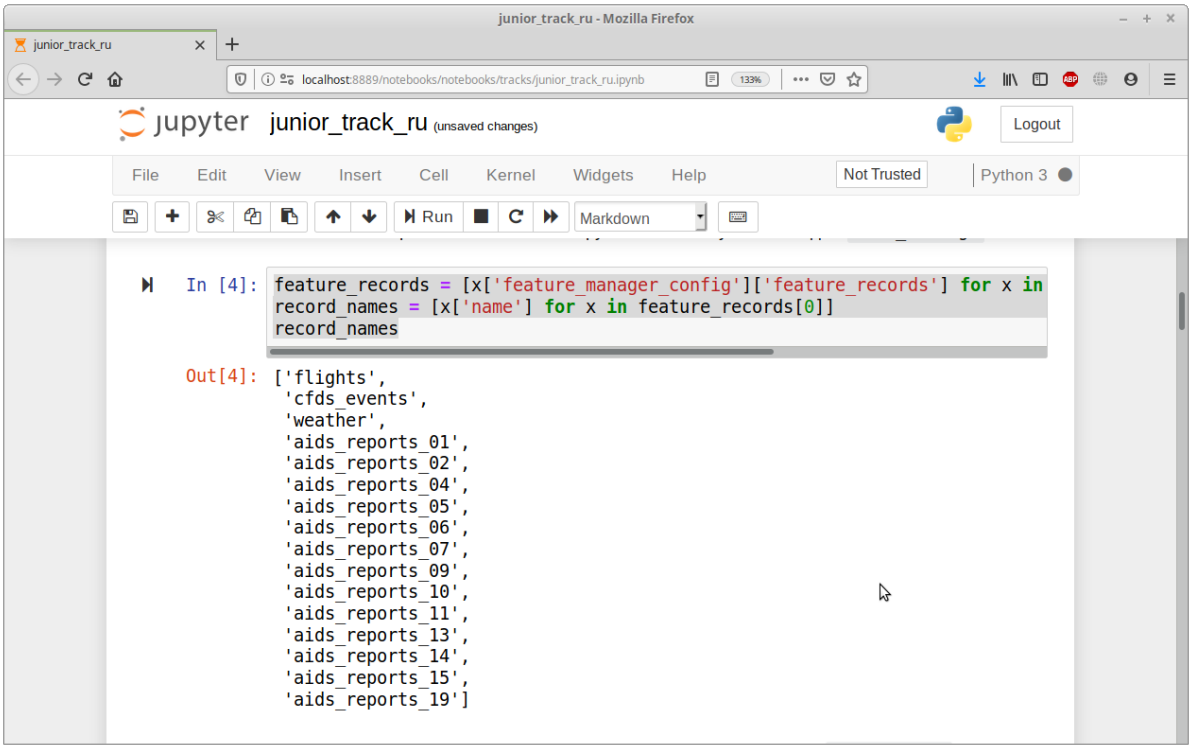
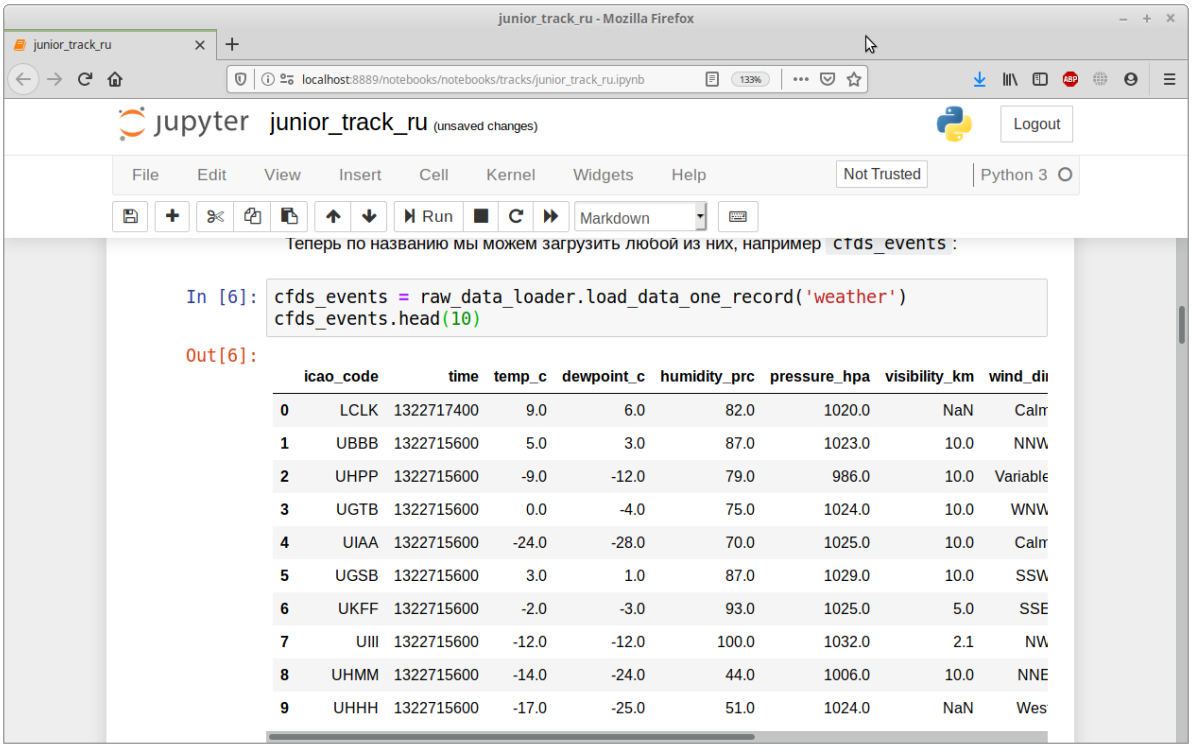


Рис.3.3: Список источников для исходных данных



	icao_code	time	temp_c	dewpoint_c	humidity_prc	pressure_hpa	visibility_km	wind_di
0	LCLK	1322717400	9.0	6.0	82.0	1020.0	NaN	Calr
1	UBBB	1322715600	5.0	3.0	87.0	1023.0	10.0	NNW
2	UHPP	1322715600	-9.0	-12.0	79.0	986.0	10.0	Variable
3	UGTB	1322715600	0.0	-4.0	75.0	1024.0	10.0	WNW
4	UIAA	1322715600	-24.0	-28.0	70.0	1025.0	10.0	Calr
5	UGSB	1322715600	3.0	1.0	87.0	1029.0	10.0	SSW
6	UKFF	1322715600	-2.0	-3.0	93.0	1025.0	5.0	SSE
7	UIII	1322715600	-12.0	-12.0	100.0	1032.0	2.1	NW
8	UHMM	1322715600	-14.0	-24.0	44.0	1006.0	10.0	NNE
9	UHHH	1322715600	-17.0	-25.0	51.0	1024.0	NaN	Wes

Рис.3.4: Пример загрузки исходных данных в Jupyter-ноутбуке

может занять много оперативной памяти, поэтому полезно удалять такие датафреймы из памяти после использования.

3.7 Загрузка признаков

Для загрузки Pandas-датафрейма с признаками, используйте атрибут feature_loader:

```
feature_loader = task_loader.feature_loader
```

Это клиент к удалённому или локальному хранилищу признаков. Он позволяет получить нужное подмножество признаков с помощью механизма фильтрации. Для этого нужно создать функцию, которая отфильтровывает нужные признаки по их именам, и передать эту функцию как аргумент feature_selector в метод load_features:

```
def feature_filtering_func(feature_name):
    return feature_name.startswith('cur_val__aids_rep_01') or feature_name.startswith('event__
↪cfds')

features = feature_loader.load_features(feature_selector=feature_filtering_func)
```

Полученные признаки представляют собой единый pandas.DataFrame, без разделения на обучающую/тестовую выборки.

Получение единого датафрейма, без разделения на обучающую/тестовую выборки, сделано специально на уровне дизайна системы, для того чтобы избежать следующей частой ошибки: неопытные дата-сайентисты применяют преобразования к обучающей выборке, и забывают преобразовать таким же образом тестовую выборку, что приводит к проблемам отладки кода.

Примечание

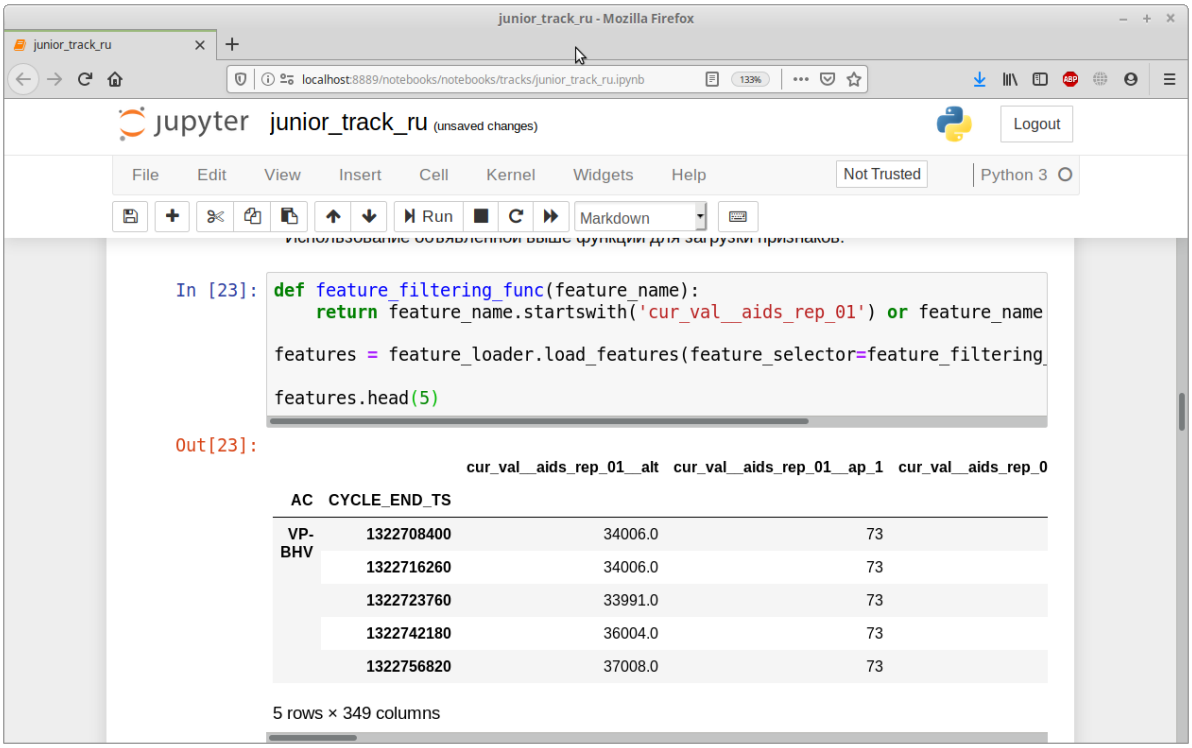


Рис.3.5: Пример загрузки признаков с помощью feature_loader

1. Feature loader сохраняет индексы строк датафрейма и их порядок.
2. Feature Loader автоматически применяет преобразование типов согласно конфигурации ML задачи, так что вам можно не беспокоиться о типах возвращаемых данных и о снижении потребления памяти (уже выбраны оптимальные типы данных).
3. Feature Loader отслеживает Ваши вызовы метода load_features и кэширует на диск загруженные Pandas-датафреймы.

3.8 Соглашение об именовании признаков

Мы рекомендуем использовать следующее соглашение об именовании признаков:

```
[oper_N]__ ... __[oper_2]__[oper_1]__[src]__[parameter]
```

Такое имя следует читать справа налево, а двойные нижние подчеркивания в нем используются в качестве разделителей. Имя признака кодирует порядок преобразований, в результате которых он был получен: к параметру [parameter] из источника [src] применили преобразование [oper_1], затем к результату применили преобразование [oper_2], и так далее до преобразования [oper_N].

Например:

```
cur_val__aids_rep_01__alt
```

Следует читать как «взять параметр alt из источника aids_rep_01 и применить к нему преобразование cur_val».

Примечание

Обратите внимание, что имя признака получается уникальным, независимо от того из какой конфигура-

ционной записи был взят исходный признак. Если бы это было не так, было бы невозможно определить из какой конфигурационной записи следует загружать признак.

*Внимание**

Всегда придерживайтесь единых принципов именования признаков, это позволит вам и вашей команде:

1. Эффективно находить и использовать нужные признаки для решения задач по анализу данных
2. Выстроить понятную систему именования признаков для удобства интерпретации и обсуждений с экспертами.
3. Использовать систему именования признаков при построении автоматических сервисов по интерпретации и объяснении признаков для экспертов и менеджеров.

3.9 Загрузка целевого признака

Для получения pandas-датафрейма с целевым признаком для обучения с учителем и для получения датафрейма, который нужно заполнить предсказаниями (ответами) на тестовой выборке, используйте объект target_loader:

```
target_loader = task_loader.target_loader
```

И вызовите метод get_train_test:

```
y_train, y_test = target_loader.get_train_test()
```

Этот метод вернет два pandas.DataFrame. Первый датафрейм содержит целевую переменную, необходимую для алгоритмов машинного обучения с учителем, и используемую при создании схем валидаций и обучения моделей:

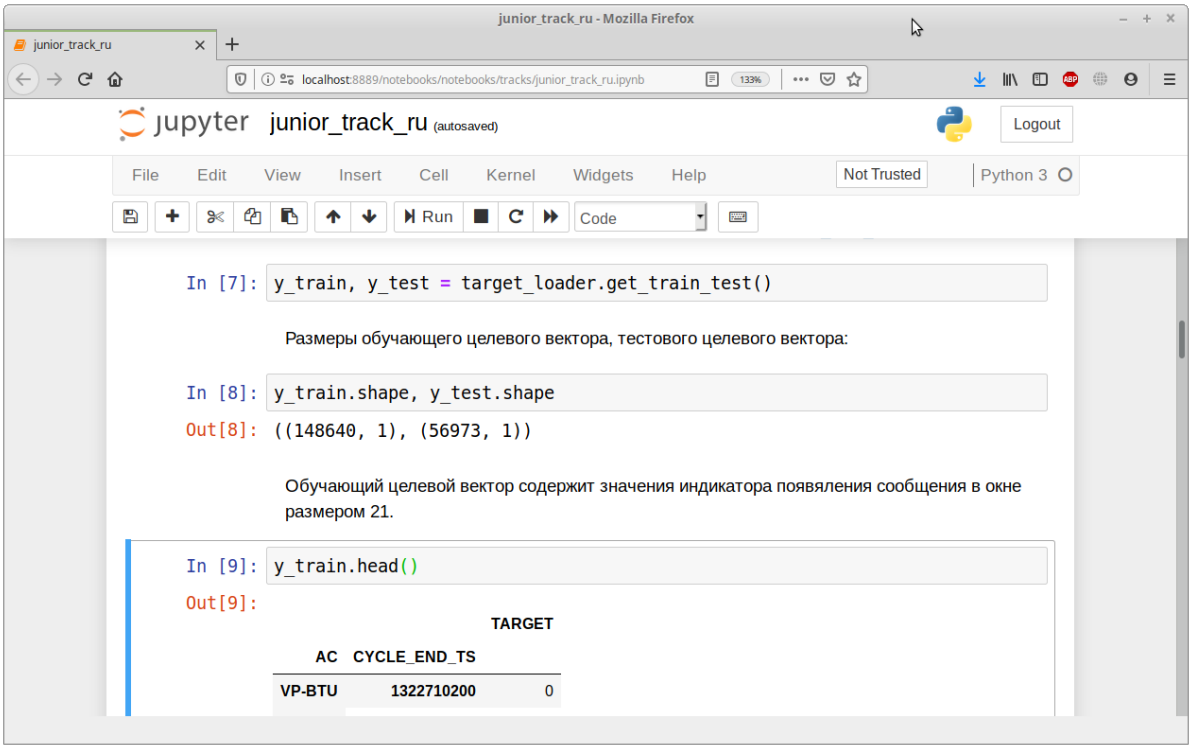
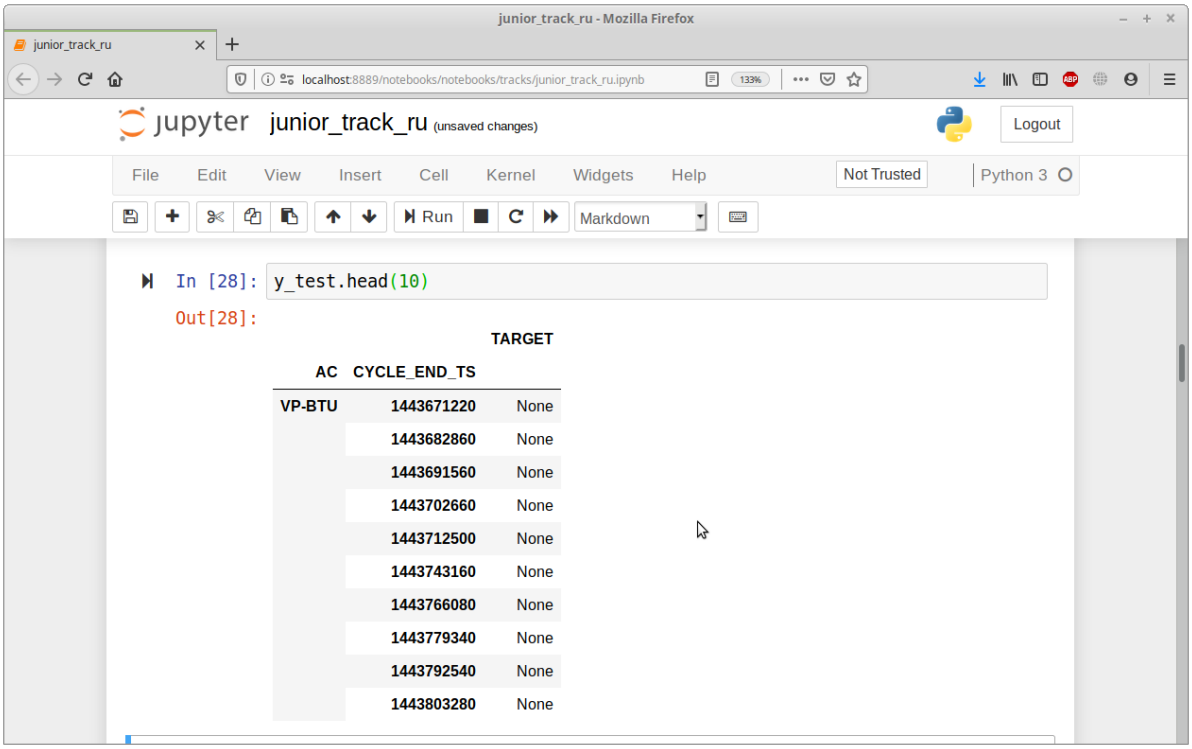


Рис.3.6: Использование target_loader для получения y_train и y_test

Второй датафрейм предназначен для заполнения предсказаниями (ответами) на тестовой выборке, поэтому он не будет содержать значений целевой переменной:



		TARGET
AC	CYCLE_END_TS	
VP-BTU	1443671220	None
	1443682860	None
	1443691560	None
	1443702660	None
	1443712500	None
	1443743160	None
	1443766080	None
	1443779340	None
	1443792540	None
	1443803280	None

Рис.3.7: Целевая переменная для тестовой выборки удалена системой

Примечание

1. Target Loader сохраняет индексы строк датафрейма и их порядок.
2. Target Loader кэширует датафреймы с целевой переменной для обучающей/тестовой выборок в базе данных согласно конфигурационному файлу ML задачи.
3. Если целевая переменная для обучающей/тестовой выборок не найдена в кэше, target_loader построит ее и положит в кэш заново, используя спецификацию предметной области проекта.

3.10 Использование схем валидаций из модуля model_selection

The DATASKAI contains its own tools to support cross-validation. Big part of them is not presented in scikit-learn library, but might be useful in different researches.

This tools are presented in the form of Python classes and assembled into a separate module named model_selection. Classes split input data into folds according to a certain rule and return these folds to the user. At the same time, the input data is sorted by time (or must already be sorted by user, but we will fix it in the nearest future) so the result folds contain data close to each other in time.

Also in some classes we use grouping data by entity (object class, category or something similar) and then build the folds so that the data of each entity are evenly distributed between different folds.

There are five base classes, TimeRangeBase, TimeRangeBaseExtraFold, TimeRangeByEntityBase, TimeRangeByEntityBaseExtraFold and TimeRangeConsecutiveOnesBase. Each of them is designed to pre-split the input data to folds, which are then combined into the train and test using the method *split* of the children classes. The diagram also shows the SignalAggregationKFold class, the logic of which is similar to the TimeRangeConsecutiveOnesBase, but covers additional functions.

The work of all classes will be discussed in more detail below.

A class diagram be represented as:

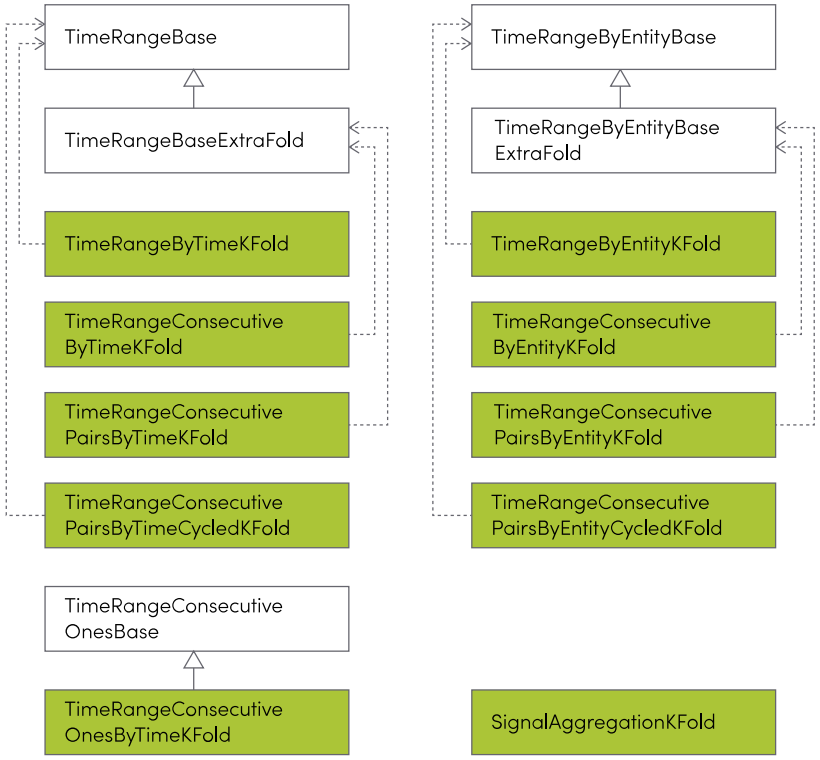


Рис.3.8: Class diagram for cross-validation classes.

3.10.1 Base classes

They are used to construct data folds, from which then the splits will be crested using various algorithms. The primary folds contain approximately equal amount of data.

TimeRangeBase

TimeRangeBase class works as shown below. It takes input data, sort it and makes split for **n** folds (user can custom **n** parameter when creating a class instance).

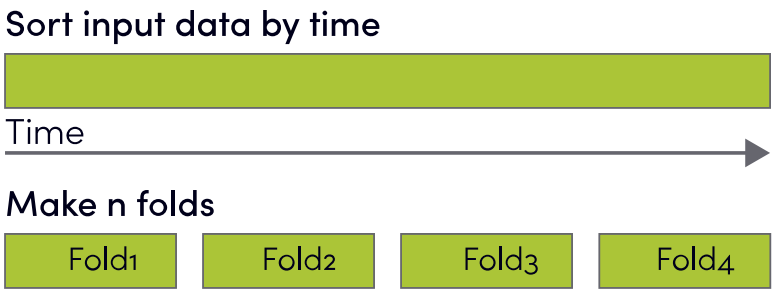


Рис.3.9: The operating principle of the class TimeRangeBase.

TimeRangeBaseExtraFold

TimeRangeBaseExtraFold works like previous class, but creates +1 extra fold. This happens because classes TimeRangeConsecutiveByTimeKFold and TimeRangeConsecutivePairsByTimeKFold, which are inheritors of class TimeRangeBaseExtraFold, require +1 initial fold to create the expected number of splits.

TimeRangeByEntityBase

Another base class is TimeRangeByEntityBase. Besides the sorting by time, it also performs grouping of data by entity name. Thus, we guarantee that the each entity data will be distributed across all the folds as evenly as possible.

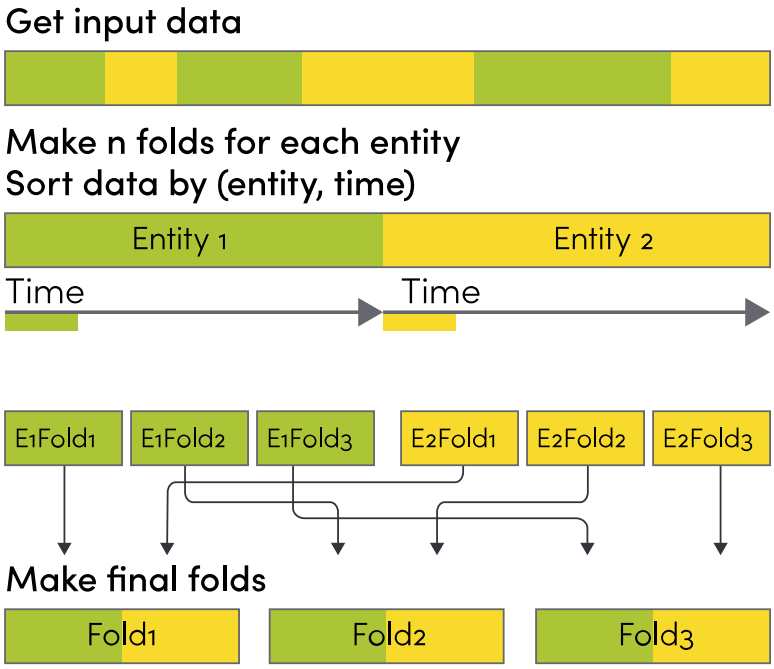


Рис.3.10: The operating principle of the class TimeRangeByEntityBase.

TimeRangeByEntityBaseExtraFold

TimeRangeByEntityBaseExtraFold works like previous class, but creates +1 extra fold. This happens because classes TimeRangeConsecutiveByEntityKFold and TimeRangeConsecutivePairsByEntityKFold, which are inheritors of class TimeRangeByEntityBaseExtraFold, require +1 initial fold to create the expected number of splits.

TimeRangeConsecutiveOnesBase

The third base class is the TimeRangeConsecutiveOnesBase. It is expected that this class will be used in the case when the input data contains a feature in the form of a vector of zeros and ones, where *one* means the occurrence of any event, and zero means its absence. After splitting each data fold contains a sequence of ones and preceding in time zeros. For example, the sequence [1, 1, 0, 0, 1, 1, 0, 1, 1, 0] will be divided into 4 folds with the following indices: [0 1] [2 3 4 5] [6 7 8] [9]

Важно: In current realization input data should be already sorted in (entity, time) manner.

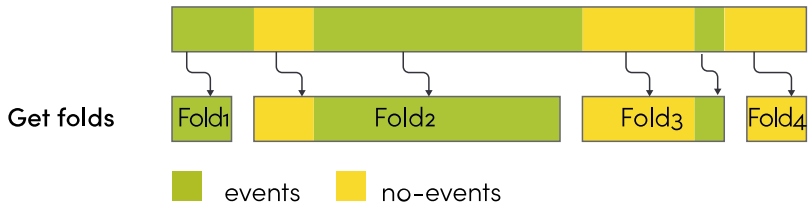


Рис.3.11: The operating principle of the class TimeRangeConsecutiveOnesBase.

The following is a description of four pairs of folds generators classes. The classes of one pair work on similar algorithms, but one of the classes groups data by the name of the entity, and the other does not.

Work principles of **TimeRangeConsecutiveOnesBase** and **SignalAggregationKFold** classes will be given in the end of this section.

3.10.2 KFold generation

TimeRangeByTimeKFold and **TimeRangeByEntityKFold** classes make approximately equal folds from data and create splits for validation by sequentially selecting each of the parts to the test sample. For example, if you use n_splits=4 in base class and have 4 data folds (1, 2, 3, 4), after generating you will receive the following splits: (2 + 3 + 4, 1), (1 + 3 + 4, 2), (1 + 2 + 4, 3), (1 + 2 + 3, 4).

TimeRangeByTimeKFold

In the case of using the TimeRangeByTimeKFold (inherited from TimeRangeBase class) splitting is performed with end-to-end sorting of data by time.

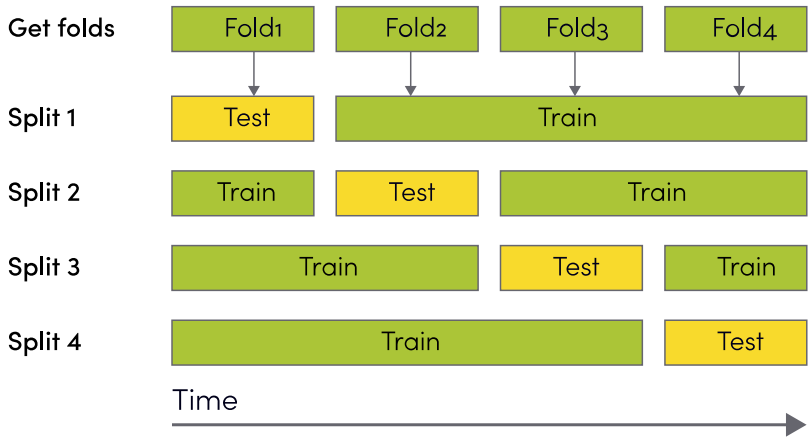


Рис.3.12: The operating principle of the class TimeRangeByTimeKFold.

TimeRangeByEntityKFold

If you use the TimeRangeByEntityKFold, an additional division is happened: the data for each entity is evenly distributed over the initial folds (see base class TimeRangeByEntityBase). Then primary folds are aggregated to

test / train folds. All data is sorted by time within the subsets of each entity.

Важно: In current realization input data should be already sorted in (entity, time) manner.

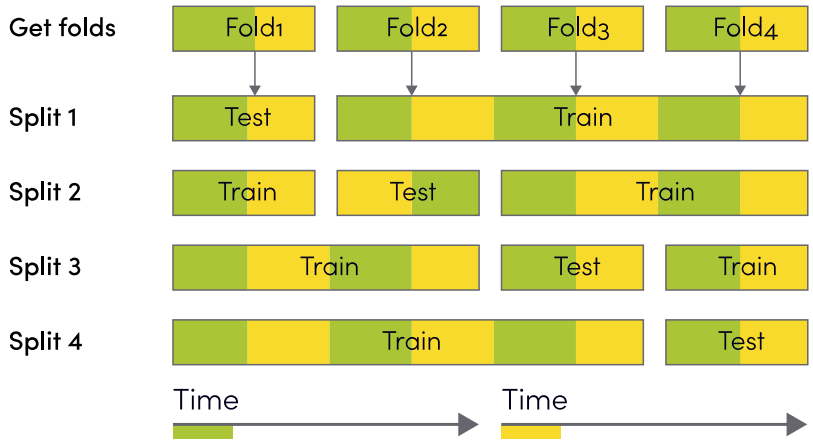


Рис.3.13: The operating principle of the class TimeRangeByEntityKFold.

3.10.3 Time series split

Classes of this type make splits in which the test fold is always beyond the train fold on time. In the first iteration the test fold is the second input data part, then the third, fourth, and so on. For example, if you split data to 4 folds (1, 2, 3, 4), at the output you will get the following splits: (1, 2), (1 + 2, 3), (1 + 2 + 3, 4)

TimeRangeConsecutiveByTimeKFold

When using the TimeRangeConsecutiveByTimeKFold (inherited from TimeRangeBase class) you will get splitting with end-to-end sorting of data by time as presented below:

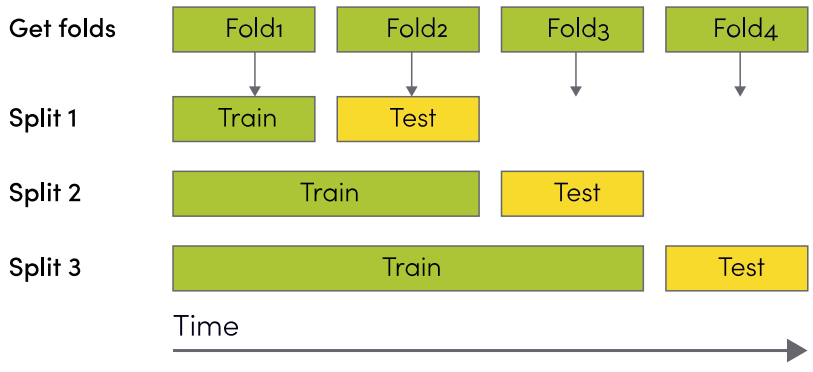


Рис.3.14: The operating principle of the class TimeRangeConsecutiveByTimeKFold.

TimeRangeConsecutiveByEntityKFold

Also you can evenly distribute each entity data over the different folds using TimeRangeConsecutiveByEntityKFold (inherited from TimeRangeByEntityBase class). All data is sorted by time separately for each entity.

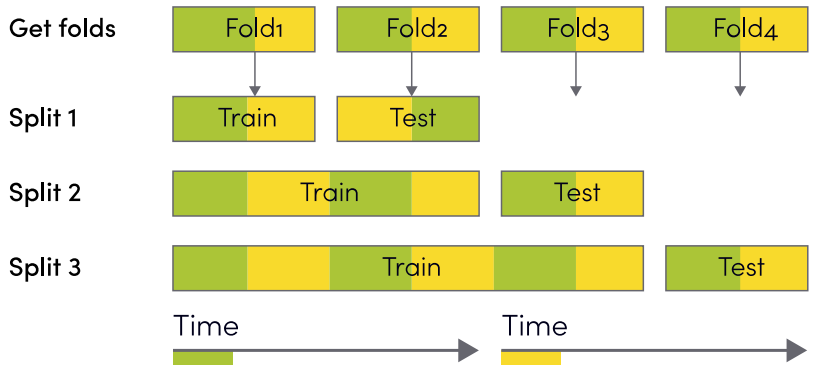


Рис.3.15: The operating principle of the class TimeRangeConsecutiveByEntityKFold.

3.10.4 Consecutive pairs

TimeRangeConsecutivePairsByTimeKFold and **TimeRangeConsecutivePairsByEntityKFold** classes create sets, that contain of neighbor data parts pairs - one as a training, other - a test fold. For example, if you split data to 4 folds (1, 2, 3, 4), at the output you will get the following splits: (1, 2), (2, 3), (3, 4)

TimeRangeConsecutivePairsByTimeKFold

Class TimeRangeConsecutivePairsByTimeKFold (inherited from TimeRangeBase class) returns splits with common sorting by time.

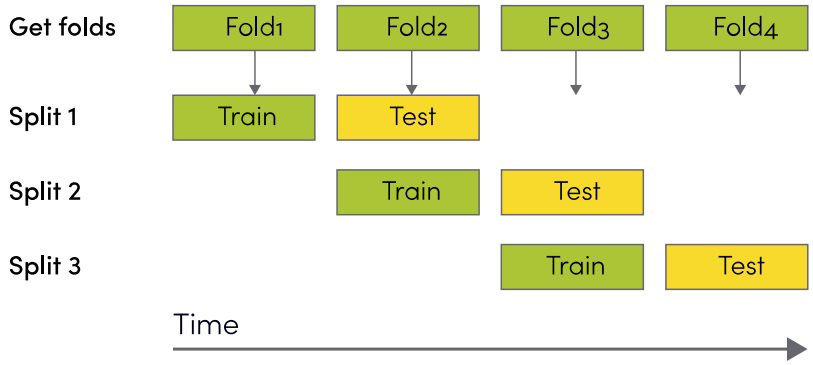


Рис.3.16: The operating principle of the class TimeRangeConsecutivePairsByTimeKFold.

TimeRangeConsecutivePairsByEntityKFold

Class TimeRangeConsecutivePairsByEntityKFold (inherited from TimeRangeByEntityBase class) additionally distributes the data of each entity over different folds, data is sorted by time separately for each entity.

3.10.5 Cycled consecutive pairs

Classes **TimeRangeConsecutivePairsByTimeCycledKFold** and **TimeRangeConsecutivePairsByEntityCycledKFold** return pairs consisting of two neighbor data folds — one as a training, and the other — a test fold, and when the last data fold is reached, the sequence loops, creating a pair from the last and first in sequence folds. For example, if you split data to 4 folds (1, 2, 3, 4), at the output you will get the following splits: (4, 1), (1, 2), (2, 3), (3, 4).

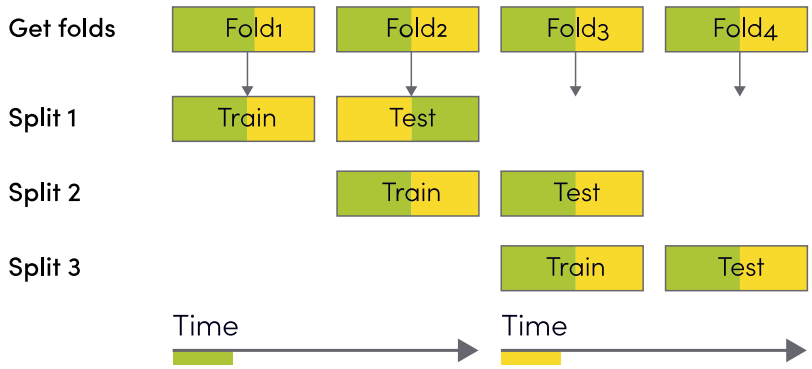


Рис.3.17: The operating principle of the class TimeRangeConsecutivePairsByEntityKFold.

TimeRangeConsecutivePairsByTimeCycledKFold

TimeRangeConsecutivePairsByTimeCycledKFold (inherited from TimeRangeBase class) returns cycled pairs with common sorting data by time.

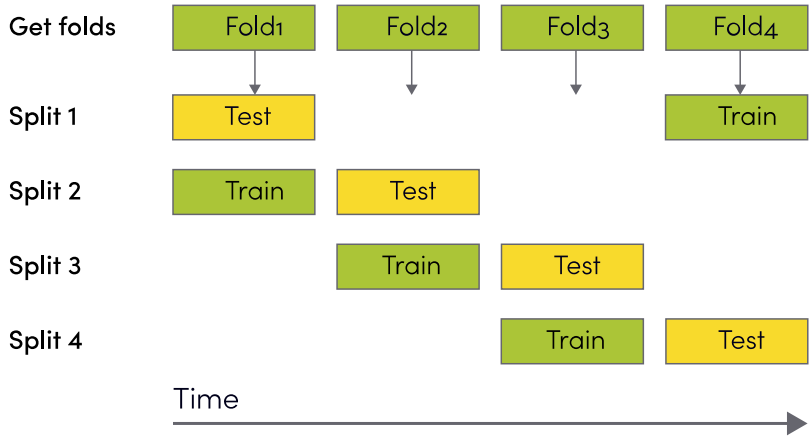


Рис.3.18: The operating principle of the class TimeRangeConsecutivePairsByTimeCycledKFold.

Also you can custom the number of data folds that will be aggregated in training fold. For example, if this parameter equals 2, then on 4 data folds (1, 2, 3, 4) you will have (3+4, 1), (4+1, 2), (1+2, 3), (2+3, 4) result splits as shown below.

TimeRangeConsecutivePairsByEntityCycledKFold

TimeRangeConsecutivePairsByEntityCycledKFold (inherited from TimeRangeByEntityBase class) additionally distributes the data of each entity over different folds, data is sorted by time separately for each entity.

Work scheme for TimeRangeConsecutivePairsByEntityCycledKFold with aggregation of two neighbor data folds:

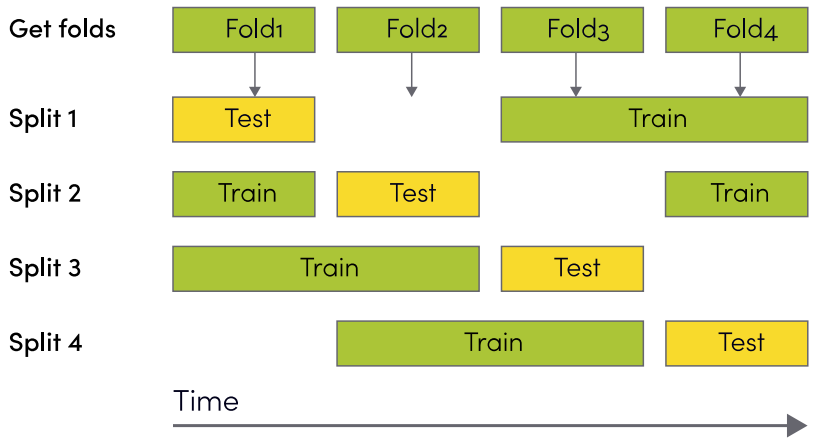


Рис.3.19: The operating principle of the class TimeRangeConsecutivePairsByTimeCycledKFold with aggregation of two neighbor data folds.

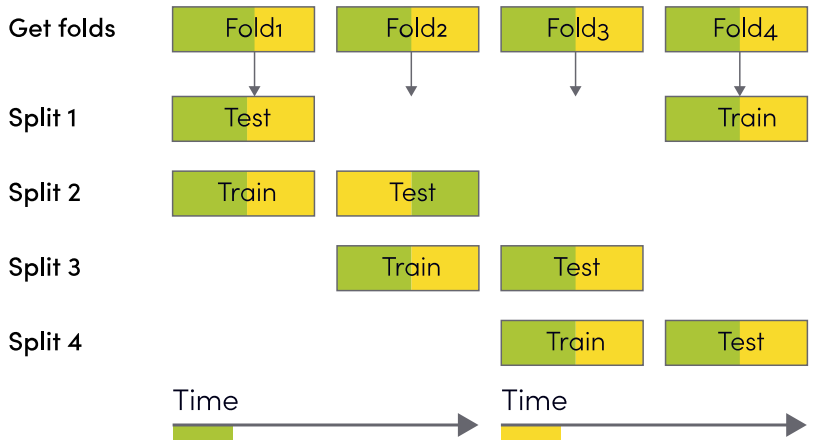


Рис.3.20: The operating principle of the class TimeRangeBase.

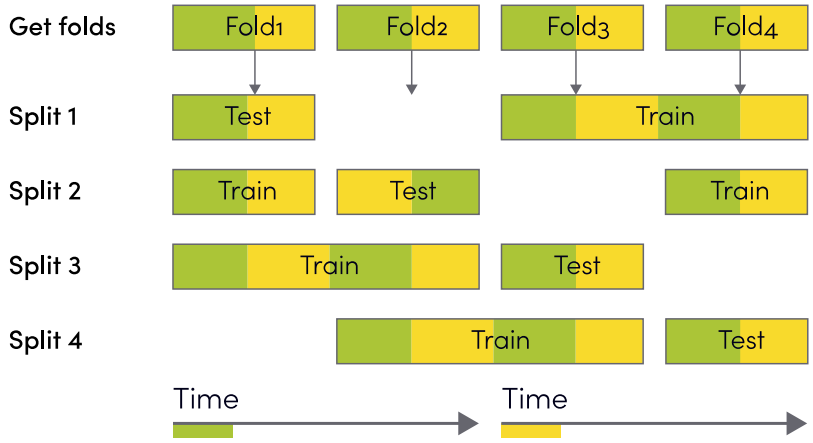


Рис.3.21: The operating principle of the class TimeRangeConsecutivePairsByEntityCycledKFold with aggregation of two neighbor data folds.

3.10.6 Consecutive events

TimeRangeConsecutiveOnesByTimeKFold

Class TimeRangeConsecutiveOnesByTimeKFold uses an algorithm to allocate groups of sequential events and is based on the **TimeRangeConsecutiveOnesBase** class. When receiving the final folds, the following logic is used: the test fold is always ahead on time. For example, if you split data to 4 folds (1, 2, 3, 4), at the output you will get the following splits: (1, 2), (1 + 2, 3), (1 + 2 + 3, 4)

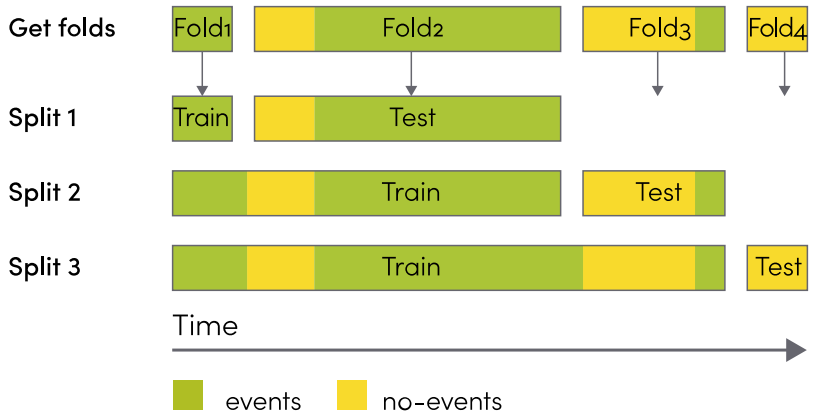


Рис.3.22: The operating principle of the class TimeRangeConsecutiveOnesByTimeKFold.

SignalAggregationKFold

Class SignalAggregationKFold also works based on allocation of groups of events (ones), but you can custom a number of splits and the *window width*, which is a number of previous and following values for each group of events.

SignalAggregationKFold allows to evenly distribute data across the splits. If you have a bid gap between groups of events (long zero sequence), then you guaranteed will not have a fold without events. h parameter, the *window width*, acts as a context, which shows how the process develops before and after the group of events.

So, this class detects the groups of ones, adds h previous and h following (for each group) values to themselves, distributes groups over n_splits folds (n_splits and h parameters are custom by user) and deletes repeated values. Then it extends data folds with the still not involved zero (no-event) values of the input data. Data in each separate fold is sorted by time.

If the number of required splits is equal to number of separate groups of events it works as presented on the diagram. Note that the training fold is, in essence, an array of input data from which the test fold was excluded, with the initial sort preserved.

In the case if the number of splits (n_splits parameter) is less then number of groups of events, groups are aggregated by number_of_groups % number_of_splits rule.

3.11 Отправка Jupyter-ноутбука с результатами

Когда модель готова и проверена с помощью процедуры кросс-валидации, наступает этап отправки в таблицу результатов. DATASKAI предоставляет компонент Submitter для отсылки предсказаний на тестовой выборке и дополнительной информации в подсистему хранения сабмитов:

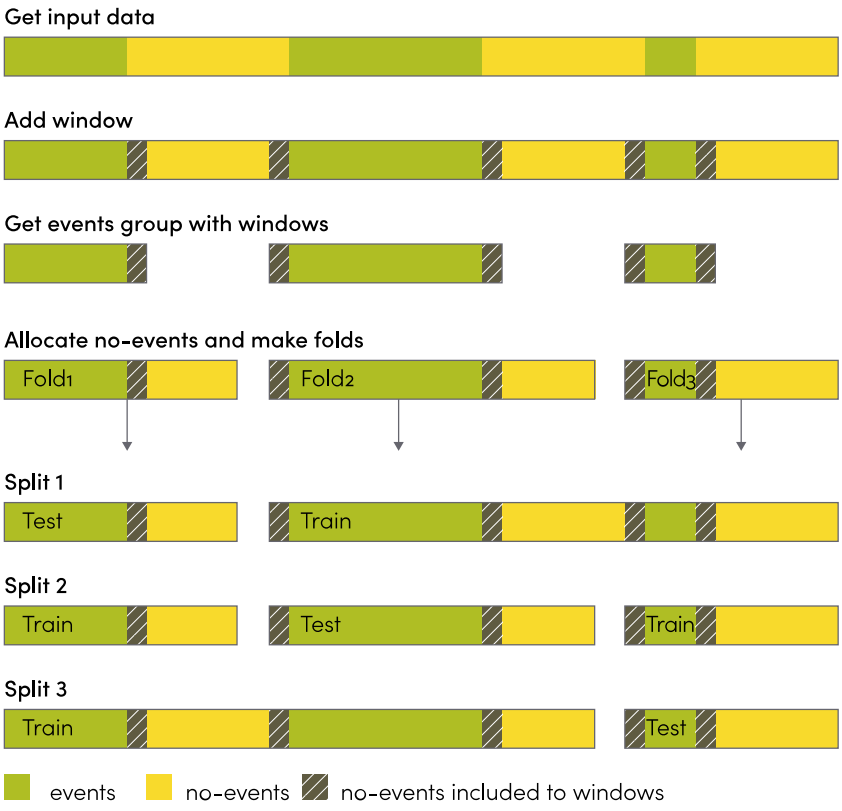


Рис.3.23: The operating principle of the class SignalAggregationKFold.

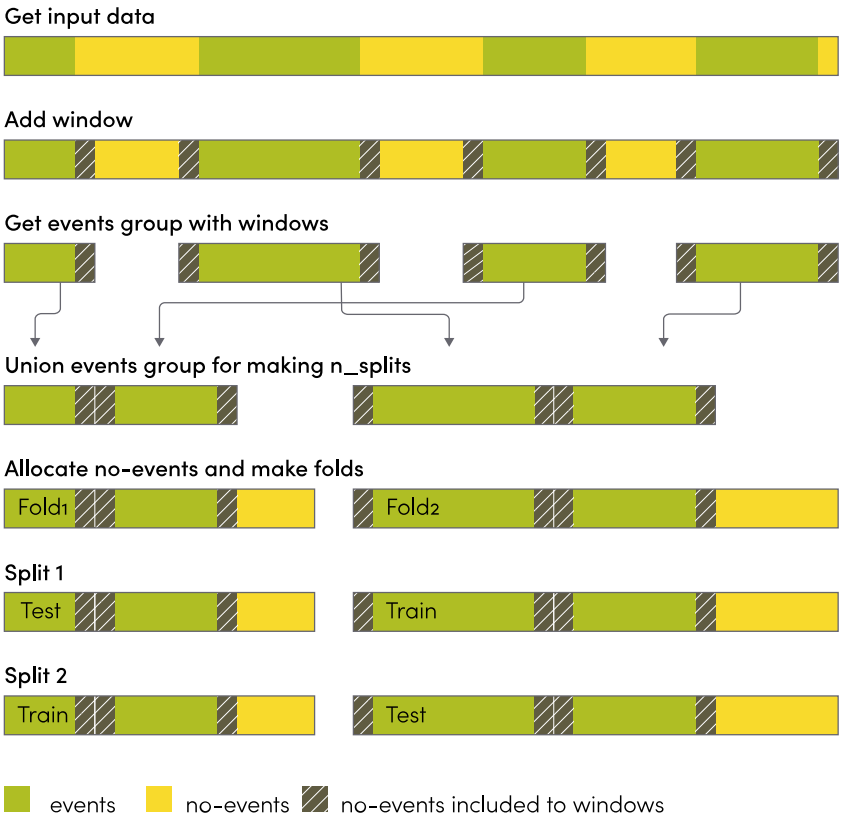


Рис.3.24: The operating principle of the class SignalAggregationKFold with creating a specific number of splits.

```
submitter = task_loader.submitter
```

После того как вы заполнили датафрейм y_test предсказаниями (ответами) на тестовой выборке, вызовите метод submit_results:

```
submitter.submit_results(
    results_df=y_test,
    author_name=AUTHOR_NAME,
    model=logistic_regression,
    model_name='Logistic Regression',
    tags=['baseline'],
    model_feature_columns=feature_columns.tolist(),
    model_fill_na_values=[0] * len(feature_columns),
    model_version='1.0'
)
```

Этот метод содержит набор именованных аргументов, каждый из которых имеет определенную цель:

1. results_df - (обязательный, pandas-датафрейм) - заполненный y_test с предсказаниями (ответами) на тестовой выборке
2. author_name - (обязательный, str) - имя автора модели
3. model - (обязательный, object) - экземпляр объекта обученной модели
4. model_name - (обязательный, str) - название модели для отображения в таблице результатов
5. tags - (обязательный, list[str]) - ключевые слова для быстрого поиска в подсистеме результатов
6. model_feature_columns - (обязательный, list[str]) - упорядоченный список имен используемых признаков
7. model_fill_na_values - (обязательный, list[float]) - числа, используемые при необходимости для заполнения пропущенных значений признаков, этот список упорядочен согласно соответствующим именам используемых признаков
8. model_version - (обязательный, str) - версия модели, для быстрого поиска и сортировки на лидерборде

Примечание

Submitter автоматически сохраняет текущее состояние Jupyter-ноутбука при вызове метода submit_results. Также Submitter отправляет сам файл Jupyter-ноутбука в подсистему хранения сабмитов, вместе с перечисленными выше именованными аргументами. Таким образом, нет необходимости после отсылки результатов версионировать Jupyter-ноутбук с помощью Git или системы именования, а можно прямо из него продолжать отправлять новые результаты.

В случае успешного выполнения, метод submit_results возвращает уникальный идентификатор сабмита. Этот идентификатор удобно использовать в качестве ключевого слова при поиске в графическом интерфейсе таблицы результатов.

Метод submit_results также поддерживает следующие необязательные параметры:

1. notebook_name - (необязательный, str) - имя файла Jupyter-ноутбука, из которого вызывается метод submit_results. В большинстве случаев нет необходимости задавать этот параметр, так как Submitter пытается определить его автоматически.
2. data_scaler - (необязательный, объект) - объект класса по масштабированию данных из пакета sklearn.preprocessing, если используется.
3. model_feature_importances - (необязательный, список вещественных чисел) - веса признаков (в зависимости от используемой модели), некоторые модели поддерживаются автоматически (такие как XGBoost, sklearn RandomForest и некоторые другие), для них не нужно передавать этот параметр.

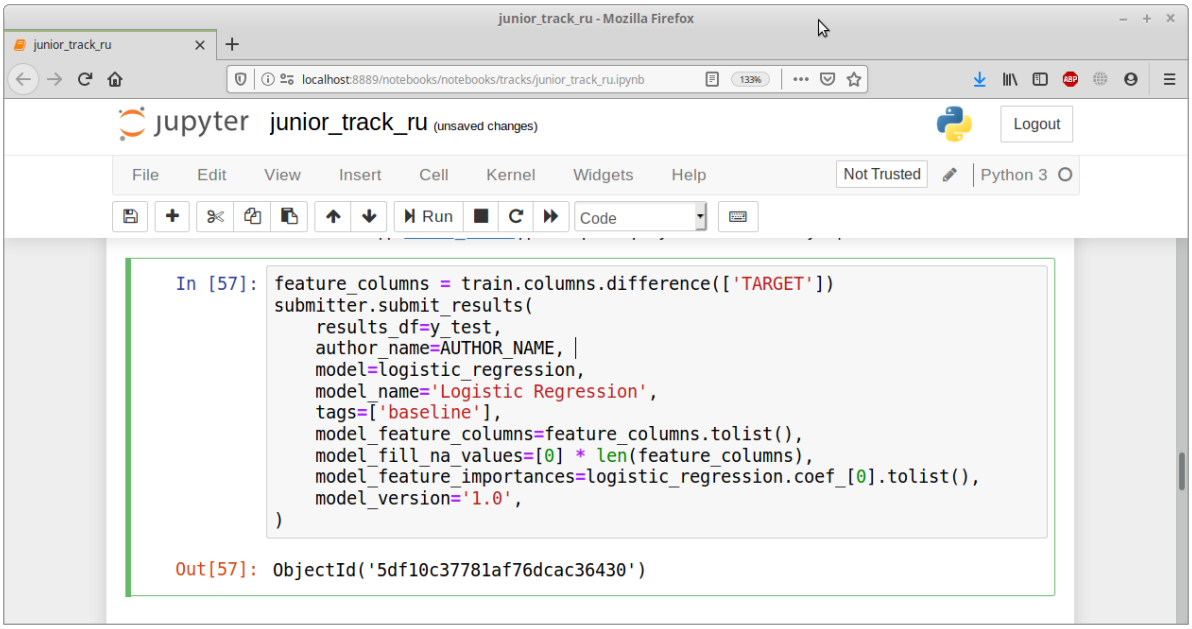


Рис.3.25: Отправка результатов в Submitter

*Внимание**

Помните, что важно безошибочно заполнять данные при отправке результата: это сэкономит много усилий и времени для вас и всей команды на следующих этапах жизненного цикла модели (например при проверке модели, упаковке модели и т. д.).

3.12 Проверка результата на тестовой выборке через графический интерфейс таблицы результатов

После отправки сабмита вы можете проверить результат через графический интерфейс таблицы результатов. Чтобы получить ссылку на него, обратитесь к вашему старшему коллеге:

Графический интерфейс таблицы результатов позволяет посмотреть значения метрик всех сабмитов для текущей ML задачи, отсортировать их по возрастанию/убыванию, производить быстрый полнотекстовый поиск, скрывать некоторые из информационных полей, и т. д. Кроме того, элемент «хлебные крошки» в верхней части окна позволяет перейти к списку всех поставленных ML задач и посмотреть них значения метрик всех сабмитов:

*Внимание**

Помните, что здоровая соревновательная атмосфера очень важна в проектах по анализу данных, поэтому оформляйте ваши результаты полноценно, заполняя все необходимые поля и поддерживая таблицу результатов в удобном для просмотра виде.

3.13 Загрузка Jupyter-ноутбука из ранее отправленных результатов

Любой из отправленных Jupyter-ноутбуков можно скачать обратно из интерфейса таблицы результатов. Это может быть полезно джуниор дата-сайентисту для ознакомления с решениями его коллег, или в случае потери оригинального файла. А миддл дата-сайентист использует это для проверки воспроизводимости работы модели машинного обучения перед ее запаковкой для конечного использования. Такую возможность предоставляет класс NotebookDownloader.

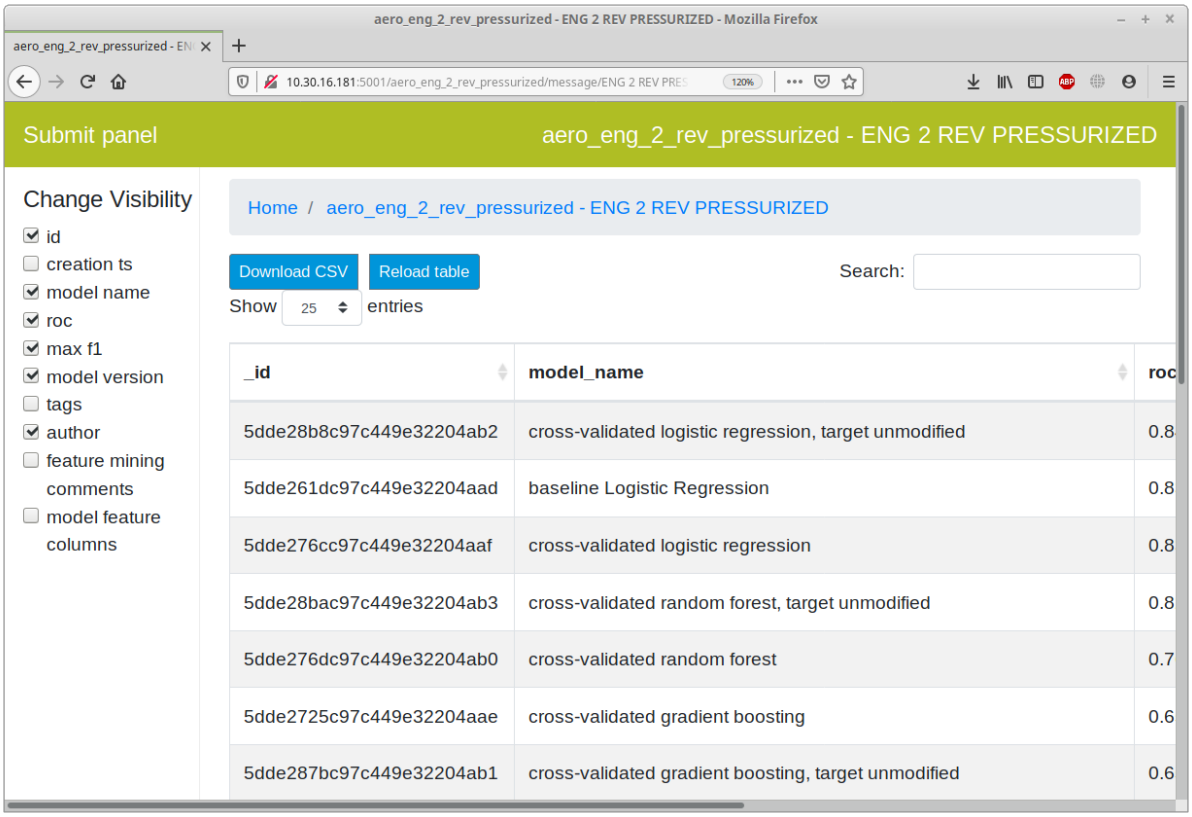


Рис.3.26: Проверка метрик для сабмита

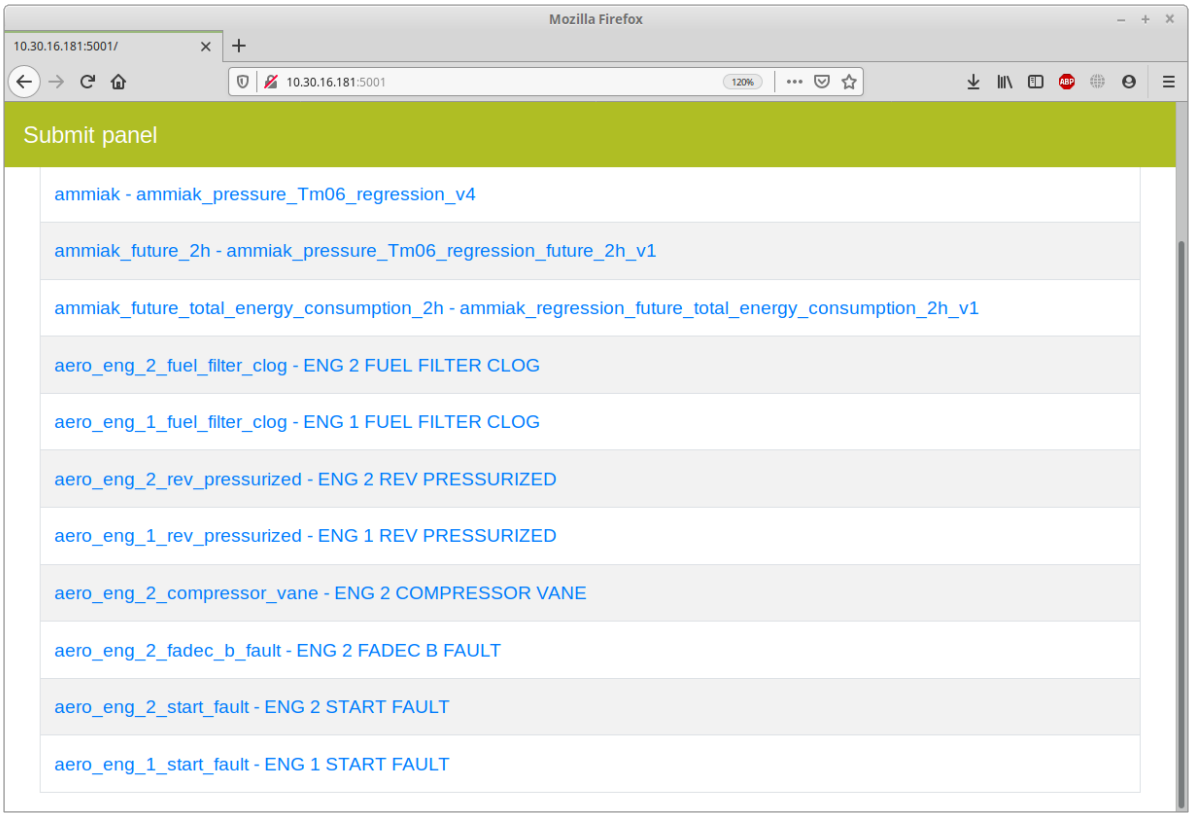


Рис.3.27: Проверка метрик для сабмита

Пусть у нас есть идентификационный номер интересующего нас сабмита из таблицы результатов лидерборда (см. раздел «Проверка результата на тестовой выборке через графический интерфейс таблицы результатов»). Тогда мы можем использовать класс NotebookDownloader для загрузки Jupyter-ноутбуков и всей сопутствующей информации:

```
from notebook_downloader import NotebookDownloader

notebook_id = '5df7731cf8427b16f81b4dbb'

nd = NotebookDownloader(TASK_CONFIG['mongo_config'])
filename, submit = nd.download_notebook_to_file(notebook_id=notebook_id,
                                                filename='downloaded_notebook.ipynb',
                                                overwrite=True)
```

Загруженный Jupyter-ноутбук появится в текущей рабочей директории.

3.14 Использование инструментов из модуля utils

3.14.1 AdversarialTrainTestValidator

Перед непосредственным построением модели машинного обучения может быть полезным воспользоваться AdversarialTrainTestValidator из модуля “utils‘.

Это метод отбора признаков, основанный на степени сходства распределений обучающей/тестовой выборок. Основная идея метода — попытаться обучить бинарный классификатор различать данные обучающей выборки и тестовой. Если такую задачу не удастся хорошо решить, значит обучающие и тестовые данные похожи, взяты из одного и того же распределения. AdversarialTrainTestValidator итеративно удаляет сильнейший признак (признак с самым большим коэффициентом важности) до тех пор, пока не будет достигнут указанный приемлемый порог по значению метрики.

Например, возьмем в качестве классификатора алгоритм случайного леса, выберем для кросс-валидации число фолдов равное 5, а приемлемым значением для метрики укажем 0.7 ROC AUC. Как мы видим, на первых итерациях значение метрики превосходит 0.7 ROC AUC — значит обучающая/тестовая выборки легко различимы даже для такой слабой модели (количество деревьев мы указали равным 5, а максимальную глубину дерева равную 2). Для достижения указанного приемлемого значения метрики потребовалось 5 итераций по удалению признаков и дополнительных stopping_rounds=5 итераций для уверенности. В конце, чтобы узнать какие признаки были удалены, воспользуемся атрибутом removed_features. Вывод: стоит попробовать удалить эти признаки из обучающей и тестовой выборок.

3.14.2 TargetBinaryEncoder

3.14.3 BinaryGenetics

In [34]:

```python
from utils import AdversarialTrainTestValidator
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import KFold

clf = RandomForestClassifier(
    n_estimators=5,
    max_depth=2,
    random_state=17,
)

kf = KFold(
    n_splits=5,
    shuffle=True,
    random_state=17,
)

validator = AdversarialTrainTestValidator(
    classifier=clf,
    scoring='roc_auc',
    cv=kf,
    threshold=0.7,
    stopping_rounds=5,
    verbose=2,
)

validator.validate(x_train, x_test)

print(validator.removed_features)
```

```
Step: 1, Score: 0.7256764465201794, removed feature: cur_val__aids_rep_01__alt
0
Step: 2, Score: 0.7292390885492711, removed feature: cur_val__aids_rep_01__n1c_2
0
Step: 3, Score: 0.711758268472457, removed feature: cur_val__aids_rep_01__n1c_1
0
Step: 4, Score: 0.7081490764252288, removed feature: cur_val__aids_rep_01__tn_2
0
Step: 5, Score: 0.6893923395665462, removed feature: cur_val__aids_rep_01__pha_sc_1
1
Step: 6, Score: 0.6899058776320618, removed feature: cur_val__aids_rep_01__gle_2
2
Step: 7, Score: 0.6787174804398723, removed feature: cur_val__aids_rep_01__t25_2
3
Step: 8, Score: 0.6854036827438927, removed feature: cur_val__aids_rep_01__n2_2
4
Step: 9, Score: 0.6664119107243437, removed feature: cur_val__aids_rep_01__vl_sd_1
5
['cur_val__aids_rep_01__alt', 'cur_val__aids_rep_01__n1c_2', 'cur_val__aids_rep_01__n1c_
1', 'cur_val__aids_rep_01__tn_2', 'cur_val__aids_rep_01__pha_sc_1']
```

Рис.3.28: Пример использования AdversarialTrainTestValidator.

Глава 4

Типовые задачи мидл дата-сайентиста

4.1 Создание Docker-контейнера для джуниор дата-сайентистов

Если настраивать рабочее окружение для каждого дата-сайентиста в отдельности, то это может занять много времени. Чтобы избежать этого, можно создать один Docker-контейнер со всеми необходимыми библиотеками, и предоставить его всем дата-сайентистам вашей команды. Различие в рабочих окружениях также может привести к проблемам с воспроизводимостью результатов. Несколько контейнеров могут быть легко запущены на одном сервере, что позоляет масштабировать команду. Конкретный набор инструментов, содержащийся в контейнере, зависит от поставленной ML задачи. Подготовленный нами контейнер „ds_base_docker“ содержит базовый набор инструментов. Чтобы его использовать, сначала нужно собрать его командой:

```
./build
```

после чего он может быть запущен локально или на сервере:

```
docker run --rm -it idcp/jupyter:$YOUR_TAG /bin/bash
```

Чтобы разграничить контейнеры для разных дата-сайентистов, вы можете запускать их на сервере с различными портами и именами:

```
docker run --name data_scientist_1 -p 10001:8888 --rm -it idcp/jupyter:$YOUR_TAG /bin/bash
docker run --name data_scientist_2 -p 10002:8888 --rm -it idcp/jupyter:$YOUR_TAG /bin/bash
```

4.2 Настройка и запуск Metric Service

Чтобы автоматически рассчитать метрики для каждого сабмита, вам нужно настроить и запустить Metric Service. Этот процесс подробно описан в ref:документации. Конфигурация Metric Service позволяют настроить его так, что он будет рассчитывать несколько различных метрик для нескольких ML задач одновременно. Так что вам не нужно запускать несколько экземпляров этого сервиса чтобы решать несколько ML задач.

4.3 Настройка и запуск Submits Web App

Чтобы просмотреть посчитанные метрики для сабмитов, вам нужно запустить Submits Web App. Процесс настройки и запуска Submits Web App подробно описан в документации. Submits Web App позволяет выводить таблицу результатов для нескольких проектов и нескольких ML задач в рамках этих проектов. Так что вы можете иметь один экземпляр этого сервиса для всех ваших проектов.

После запуска Submits Web App, вы можете просматривать сабмиты, сортировать их по различным столбцам, а также осуществлять текстовый поиск по столбцам:

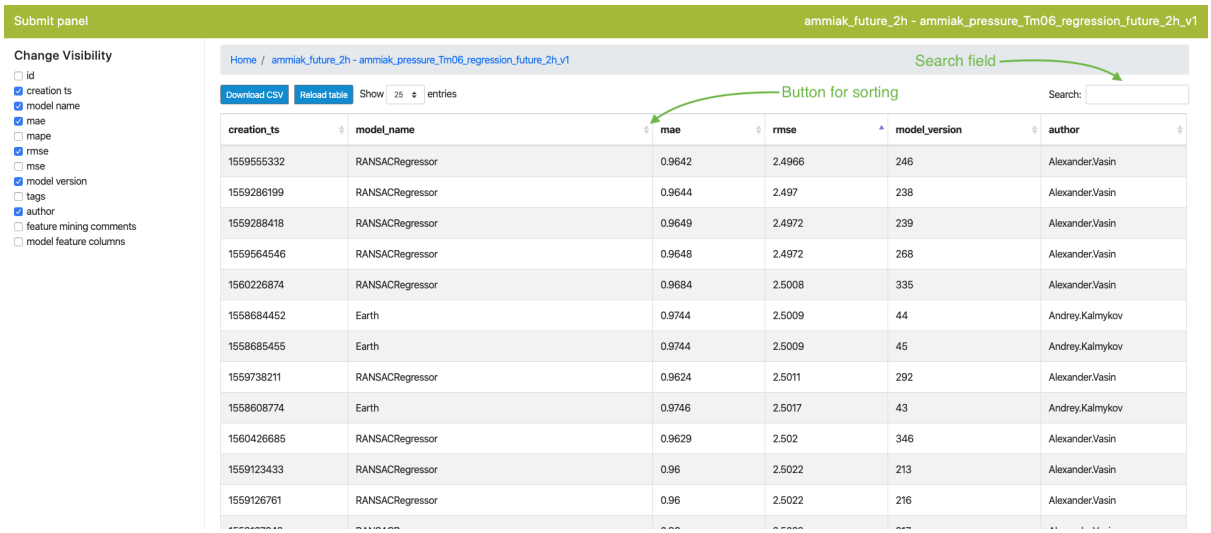


Рис.4.1: Устройство интерфейса Submits Web App.

4.4 Проверка воспроизводимости после восстановления отправленной модели

Перед запаковкой модели для сервиса давайте восстановим модель из сабмита (см. *Загрузка Jupyter-ноутбука из ранее отправленных результатов*) для проверки воспроизводимости, т. е. сравним получившиеся результаты.

Чтобы восстановить модель из сабмита, нам понадобятся следующие поля:

- model_feature_columns — имена признаков модели
- model_fill_na_values — значения для заполнения пропущенных значений
- data_scaler_gridfs_id — идентификатор трансформера по масштабированию данных
- model_gridfs_id — идентификатор модели
- result_gridfs_id — идентификатор предсказаний для тестовой выборки

Первые два из них это простые списки, другие это объекты требующие последующего запроса в MongoDB GridFS и распаковки из бинарного типа с помощью класса Fileworker.

```
feature_cols = submit['model_feature_columns']
fillna_values = submit['model_fill_na_values']

scaler_in_gridfs = submit['data_scaler_gridfs_id']
model_in_gridfs = submit['model_gridfs_id']
result_in_gridfs = submit['result_gridfs_id']
```

```
import gridfs
import pymongo

from submitter import DEFAULT_SUBMITS_COLLECTION

db_config = TASK_CONFIG['mongo_config']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
client = pymongo.MongoClient(db_config['host'], port=db_config['port'])
db = client[db_config['db']]
gridfs_instance = gridfs.GridFS(db, collection=DEFAULT_SUBMITS_COLLECTION)

scaler_blob = gridfs_instance.get(scaler_in_gridfs).read()
model_blob = gridfs_instance.get(model_in_gridfs).read()
result_blob = gridfs_instance.get(result_in_gridfs).read()
```

```
from fileworker import FileWorker

fileworker = FileWorker()

scaler = fileworker.convert_binary_to_object(scaler_blob)
model = fileworker.convert_binary_to_object(model_blob)
result = fileworker.convert_binary_to_df(result_blob)
```

```
X_test_ = X_test[feature_cols]
fillna_values_for_df = dict(zip(feature_cols, fillna_values))
filled_X_test = X_test_.fillna(fillna_values_for_df)
scaled_X_test = scaler.transform(filled_X_test)

reproduced_result = model.predict_proba(scaled_X_test)[:, 1].tolist()
```

Теперь можно сравнить полученные результаты:

```
In [15]: 1  print(all(np.isclose(result['TARGET'].tolist(), reproduced_result)))
         True
```

Рис.4.2: Проверка воспроизводимости после восстановления отправленной модели.

4.5 Model packing and validating reproducibility

4.5.1 Rules

For model packing we need to:

- wrap the model - prepare the file with model class using Model Wrapper component;
- prepare model's configuration which must be located in file <models_dir>/configs/config.json;
- pickle needed objects and jsonify needed values;
- prepare a specific model directory with all necessary files.

Model directory must have the next structure:

```
├── model_directory
│   ├── configs
│   │   └── config.json
│   ├── data
│   ├── model_wrapper
│   └── model.py
```

Where:

- **configs** - a folder with model configuration file;

- **data** - optional catalog, an additional data (e.g. file pickles), required for model initialization;
- **model_wrapper** - a folder with content of model_wrapper component;
- **model.py** - a Python module, describes init, predict and shutdown model methods.

Примечание: The name of model directory used in Models Player as a model identifier (model_id parameter).

Configuration for wrapped model is written as json-file and placed in a **configs** directory. The example of the model configuration is shown below:

```
{
    "model_name": "sum_model",
    "host": "localhost:10005",
    "model_filename": "sum_model.py",
    "model_class_name": "AllSumModel",
    "message": "Name of formulated problem",
    "model_features": ["feature_1","feature_9","feature_10"],
    "feature_name_aliases":
    {
        "feature_9": "main_feature",
        "feature_10": "feature_3",
    }
    "tags":["is_prod_model"],
    "logging_level": "info"
}
```

Configuration file contains the following required fields:

- **model_name** - a name of model, e.g. Ridge, AllSumModel, etc; note that this is not an ID of model;
- **host** - an address (host and port) for gRPC server with wrapped model;
- **model_filename** - name of Python file, which contains class with model **init**, **predict** and **shutdown** methods;
- **model_class_name** - name of Python class which is used for model description in <model_filename> module;
- **message** - name of current formulated problem;
- **model_features** - list of features used by model;
- **feature_name_aliases** - a dictionary with features aliases in format {true_name: alias}; after applying aliases old names are lost;
- **tags** - key words for choosing a way to work with the model, e.g. «is_prod_model» tag means that model should be used in production;
- **logging_level** - the level of logging, can be «info», «warn», «error», etc.

This fields are required, but you always can add any additions to this configuration file.

4.5.2 Wrapping and packing example

You can see a complete example of model wrapping and packing below.

```
import pickle

with open('my_model_wrapped/data/fillna_values.json', 'w') as f:
    json.dump(fillna_values_for_df, f)
```

(continues on next page)

```
In [17]:
!mkdir my_model_wrapped/
!git clone git@srv-iot-git.skoltech.ru:IDCP/model_wrapper.git my_model_wrapped/model_wrapper
!cp -R my_model_wrapped/model_wrapper/template/* my_model_wrapped/
```

```
Cloning into 'my_model_wrapped/model_wrapper'...
remote: Enumerating objects: 328, done.
remote: Counting objects: 100% (328/328), done.
remote: Compressing objects: 100% (140/140), done.
remote: Total 328 (delta 185), reused 319 (delta 176)
Receiving objects: 100% (328/328), 65.93 KiB | 16.48 MiB/s, done.
Resolving deltas: 100% (185/185), done.
```

Рис.4.3: Cloning Model Wrapper repository and preparing directory structure.

(продолжение с предыдущей страницы)

```
with open('my_model_wrapped/data/scaler.pkl', 'wb') as f:
    pickle.dump(scaler, f)

with open('my_model_wrapped/data/model.pkl', 'wb') as f:
    pickle.dump(model, f)
```

Подготовка класс модели (опущены строки документирования):

```
my_model_wrapped_py_content = \
'''
import json
import os
import pickle
import sys

import numpy as np

sys.path.append(os.path.dirname(os.path.realpath(__file__)))
from model_wrapper.src.model_abstract_wrapper import ModelAbstractWrapper

class MyModelWrapped(ModelAbstractWrapper):

    def model_init(self, model_parameters=dict()):
        with open('data/model.pkl', 'rb') as f:
            self.model = pickle.load(f)
        with open('data/fillna_values.json', 'r') as f:
            self.fillna_values = json.load(f)
        with open('data/scaler.pkl', 'rb') as f:
            self.scaler = pickle.load(f)

    def predict(self, X):
        inds = np.where(np.isnan(X))
        X[inds] = np.take(list(self.fillna_values.values()), inds[1])
        X = self.scaler.transform(X)
        return self.model.predict_proba(X)[:, 1]

    def shutdown(self):
        pass
'''

with open('my_model_wrapped/model.py', 'w') as f:
    f.write(my_model_wrapped_py_content)
```

Задание конфигурации модели:

```
with open('my_model_wrapped/configs/template_config.json', 'r') as f:
    config = json.load(f)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
config['model_name'] = 'my_model'               # model name
config['model_filename'] = './model.py'         # main model filename
config['message'] = 'ENG 2 FUEL FILTER CLOG'    # task message
config['model_class_name'] = 'MyModelWrapped'   # model class name
config['model_features'] = feature_cols         # used features
config['window'] = 21                           # prediction horizon

with open('my_model_wrapped/configs/config.json', 'w') as f:
    json.dump(config, f, indent=2)
```

Последним шагом создадим архив директории с моделью:

```
!tar -czf my_model_wrapped.tar.gz my_model_wrapped/
```

The model is packed and ready to be sent to Models Player service.

4.5.3 Model validation

Before sending let's validate packed model. There is send_lines_to_model.py script designed to get several lines predictions (only once after the script has started). We need to prepare small data chunk, run the script and compare predictions with the actual ones.

```
df = X_test[feature_cols].head(5)
df.to_csv('my_model_wrapped/partial_test_input.tsv', header=None, sep='\t', index=None)
```

```
import subprocess
import time

command = (
    'cd my_model_wrapped/ && '
    'python3 model_wrapper/src/model_runner.py --config ./configs/config.json --port 10005 '
)

subprocess.Popen([command], shell=True)

time.sleep(3)  # model starting will take some time
```

```
In [23]:
! cd my_model_wrapped/ && cat "partial_test_input.tsv" \
  | python model_wrapper/src/send_lines_to_model.py | grep "Model batch reply"

print('Original model reply:', end=' ')
for x in model.predict_proba(scaler.transform(df.fillna(fillna_values_for_df)))[:, 1]:
    print(x, end=' ')

Model batch reply: 0.0013861564747873428 0.0013861564747873428 0.0010746891469505262 0.0006832766778644246 0.00068
32766778644246
Original model reply: 0.0013861564747873428 0.0013861564747873428 0.0010746891469505262 0.0006832766778644246 0.00
06832766778644246
```

Рис.4.4: Использование send_lines_to_model скрипта.

Как мы видим, результаты совпадают.

4.6 Model Wrapper и Models Player

Using ML models as a web-service is essential when we aimed at streaming data for inference (getting predictions on new data). Thus model starts only once, and not every time for inference. It is also important that many different ML models should be available at the same time.

Models Player is models controller web-service. It provides HTTP API that allows user to control ML models: start and stop them, make predictions, get working statistics, etc.

Model Wrapper is used for wrapping model before packing and sending for usage to Models Player.

4.7 Настройка и запуск Models Player

Properties from model_player/../config/default_config.json are used by default unless config_path argument is not passed. You can change host, port, model-related paths and startup settings. Also you can pass parameters through environment variables. See documentation for the complete list of properties, as well as commands for starting Models Player.

To start Models Player (it is also possible to use Docker image):

```
import subprocess
import time

command = (
    'cd ../../../model_player/scripts/ && '
    './start.sh'
)

subprocess.Popen([command], shell=True)
```

Для упрощения GET и POST запросов давайте использовать Python оберткой models_player_requests:

```
import models_player_requests

models_player = models_player_requests.Requests(
    host='0.0.0.0',
    port='9066',
)
```

4.8 Отправка модели в Models Player

Каждая модель в Models Player может находиться в нескольких состояниях: она может быть заархивирована и разархивирована, запущена или остановлена. Для добавления и удаления моделей, управления их состоянием, получением от них предсказаний и статистики использования, предоставляется HTTP API:

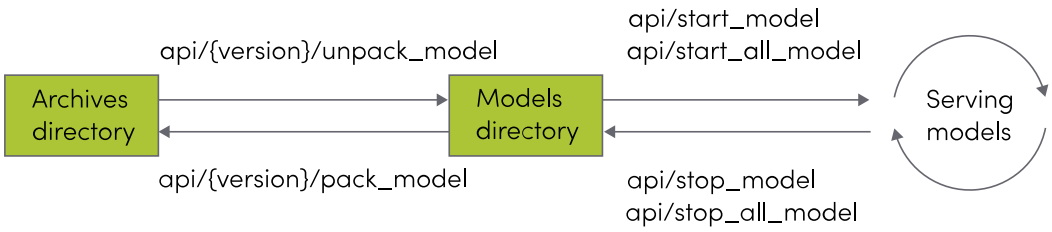


Рис.4.5: Различные состояния модели в Models Player.

See documentation to manage the lifecycle of your models with Models Player API.

Отправка запакованной модели:

Запуск обернутой модели:

```
In [42]:
models_player.upload_model('my_model_wrapped.tar.gz')
print(models_player.list_archives_dir())

models_player.unpack_model('my_model_wrapped.tar.gz')
print(models_player.list_models_dir())

['my_model_wrapped.tar.gz']
['my_model_wrapped']
```

Рис.4.6: Вывод после загрузки и разархивирования модели.

```
In [43]:
models_player.start_model('my_model_wrapped')
print(models_player.list_running_models())

['my_model_wrapped']
```

Рис.4.7: Вывод после запуска модели.

Для получения предсказаний:

```
In [44]:
model_settings = models_player.get_models_settings(model_id='my_model_wrapped')
used_features = model_settings['model_config']['model_features']
models_player.predict_batch('my_model_wrapped', df[used_features].values.tolist())

Out[44]: [{'y_pred': 0.0013861564747873428},
          {'y_pred': 0.0013861564747873428},
          {'y_pred': 0.0010746891469505262},
          {'y_pred': 0.0006832766778644246},
          {'y_pred': 0.0006832766778644246}]
```

Рис.4.8: Predicts from the model on Models Player.

4.9 Остановка модели в Models Player

Используйте метод stop_model для остановки конкретной модели:

```
models_player.stop_model('my_model_wrapped')
```

Для полного удаления модели из всех состояний нужно дополнительно выполнить:

4.10 Создание построителя признаков

Feature extractor is program run by Feature Builder service to extract features from raw data. Feature extractor utilizes Feature Builder service docker container, so all dependencies should be accounted in that container. Typically, all the dependencies from DS container carried over to Feature Builder service container.

Architecture of feature extractor is made for fast implementing, development and debugging in local environment. In case of failure Feature Builder service leaves input and output files in directory, so build process could be isolated and debugged. Directory of typical feature extractor is shown below:

```
.
├── config
│   └── default.json
├── data
│   └── some_stored_data_for_calculations.json
├── input
│   └── ...
├── output
│   └── ...
```

(continues on next page)

```
In [47]:
1  print(models_player.list_archives_dir())
2  print(models_player.list_models_dir())
3  print(models_player.list_running_models())
```

```
[]
[]
[]
```

Рис.4.9: Получение текущего состояния моделей в Models Player.

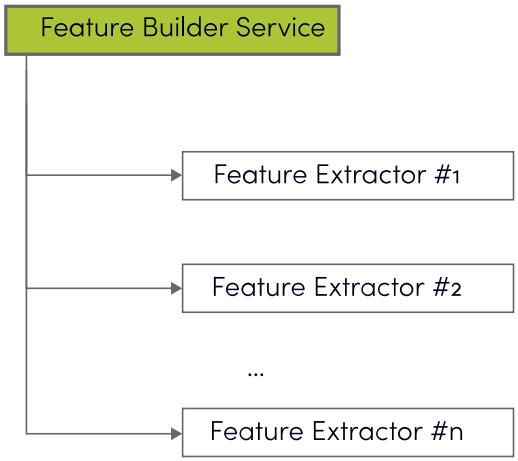


Рис.4.10: Feature Builder service and feature extractor

(продолжение с предыдущей страницы)

```
├── some_feature_extractor.py
└── make_features.sh
```

Построитель признаков должен обязательно включать в себя:

1) поддиректорию config с конфигурационным файлом в формате JSON
2) make_features.sh скрипт со следующими флагами:
   - -c или --config с указанием пути к конфигурационному файлу
   - -i или --input с указанием пути к директории с входными данными
   - -o или --output с указанием пути к директории для хранения результатов построения признаков
   - -r или --rebuild_necessary_data — специальный двоичный флаг указывающий на то, нужно ли перезапускать обучение во время обработки признаков

In case you need to implement your own feature extractor you need to know some default conventions:

1) Соглашение о выборе режима работы
2) Соглашение об использовании конфигурационного файла
3) Соглашение об организации входных и выходных файлов

4.10.1 Соглашение о выборе режима работы

Основным режимом работы построителя признаков является режим преобразования данных, поэтому если построитель признаков вызван без каких-либо флагов, то он запускается, по умолчанию ожидая конфигурационный файл default.json, поддиректорию с входными файлами input, поддиректорию для выходных

файлов output. Построитель признаков может при необходимости использовать также поддиректорию data для хранения внутреннего состояния.

Особенным режимом работы построителя признаков является режим с перезапуском обучения, при котором он берет входные данные, вычисляет внутреннее состояние, и сохраняет для последующего применения преобразования.

4.10.2 Соглашение об использовании конфигурационного файла

Configuration file of feature extractor is written as json document. It should contain some mandatory fields and could also contain any arbitrary information. Feature Builder service reads mandatory fields and uses them for different system aspects. Mandatory fields are:

- feature_extractor_name - строковое поле с именем построителя признаков. Это имя будет использоваться как ключ, с помощью которого Feature builder будет идентифицировать соответствующий конфигурационный файл построителя признаков
- features_sources - список документов с указанием на тип данных. Каждый документ состоит из двух полей:
  - filename - строковое поле с указанием на имя файла с исходными данные для этого источника данных
  - datatype - строковое поле с указанием на тип данных, сконфигурированный в Data Service (Java часть)
- sources_minimum_time_lag_seconds - задержка, которая будет использоваться Feature builder при получении данных для построения признаков
- input_folder - строковое поле с указанием на относительный путь к директории с входными файлами. Будет использоваться, если не задан параметр -i или --input
- output_folder - строковое поле с указанием на относительный путь к директории с выходными файлами. Будет использоваться, если не задан параметр -o или --output
- data_folder - строковое поле с с указанием на относительный путь к директории, в которой будут храниться вычисляемые параметры для построителя признаков

Необязательные поля:

- feature_extractor_settings - документ, который (в случае с построителем признаков на Python) передается при применении преобразования данных, используется для хранения всех необходимых необязательных параметров построителя признаков.

Пример конфигурационного файла построителя признаков (с некоторые дополнительными необязательными полями):

```
{ "feature_extractor_name":"some_feats_feats_90_94_97",
  "features_sources":[
    {
      "filename":"flights.tsv",
      "datatype": "flights"
    },
    {
      "filename":"aids_reports_01.tsv",
      "datatype": "aids_reports_01"
    }
  ],

  "sources_minimum_time_lag_seconds":31415926,
  "logging_filename":"./logs/some_feats_feats_90_94_97.log",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  "logging_level":"INFO",
  "copy_logs_to_stdout":true,
  "input_folder":"./input",
  "output_folder":"./output",
  "data_folder":"./data",
  "feature_extractor_settings":
  {
    "postprocessing":{
      "ecw5_t5":"force_convert_to_float",
      "y1":"force_convert_to_float",
      "y2":"force_convert_to_float",
      "y1_1":"force_convert_to_float",
      "y2_1":"force_convert_to_float",
      "psel_1":"force_convert_to_float",
      "psel_2":"force_convert_to_float"
    },
    "input_schedule_file":"flights.tsv",
    "output_temporary_dir":"./tmp",
    "quantiles_for_quantiles":"transform_data.json",
    "seasonalities_models": "transform_models.pkl"
  }
}
```

4.10.3 Соглашение об организации входных и выходных файлов

Feature Builder service provides necessary files for input by reading configuration files stored in feature extractors. One feature extractor could provide one or more configuration files, so one feature extractor code could provide more than one output on same input data. Output files stored as csv (for simple debugging), or parquet (for fast work, in future releases) and should contain obligatory index fields according to task configuration and Feature Storage service configuration. Output files should be written by process in passed output folder.

Внимание: Feature extractor is a program to transform data and while it do transform it should be stateless within one call in transform regime. In other words it should not save and load nor anyhow connect to other external state storage while it operates in transform regime. This limitation is necessary due to external limitation of Feature Builder service: it could provide files in input folder with arbitrary order and without any guaranties for data to be sorted.

Глава 5

Типовые задачи сеньор дата-сайентиста

5.1 Что такое предметная область?

Предметная область не включена в релиз предварительной альфа-версии. Пожалуйста, используйте построитель признаков, чтобы тщательно обработать исходные данные и извлекать признаки.

5.2 Построение простой предметной области

Subject domain are not included in Pre Alpha release. Some basic information you can find in *Defining target and train/test sets for task* part of user guide. Please use feature extractor to carefully process raw data and extract features.

5.3 Типичные архитектуры предметной области

Предметная область не включена в релиз предварительной альфа-версии. Пожалуйста, используйте построитель признаков, чтобы тщательно обработать исходные данные и извлекать признаки.

5.4 Создание проекта и ML задачи с помощью мастера

DATASKAI project can be configured and deployed with Project Wizard component. Let's see how it works on the following example.

First step, install wizard and check if it is available:

```
dataskai --help
```

Make a new project:

```
dataskai project new
```

Wizard will ask you:

- **What is the name of your project?** - text name which will be used as a name for project directory, added to services Docker containers names, etc.
- **Which IP address to use?** - choose a host IP for Docker daemon.

- **Use ‘sudo‘ for Docker commands?** - if yes, all docker and docker-compose commands will be preceded by sudo.

For example,

```
? What is the name of your project? my_project
? Which IP address to use?  127.0.0.1
? Use `sudo` for Docker commands?  No
? Let's start the installation?  Yes
```

After confirmation, a new project directory my_project will be generated in a working dir. It contains services configuration files, .env with project environment variables, DATASKAI Python toolkits and etc.

Now we need to up DATASKAI services. Wizard do it with a start command:

```
cd my_project
dataskai project start
```

All services will be deployed in separate Docker containers. On success you must see the next statuses in CLI:

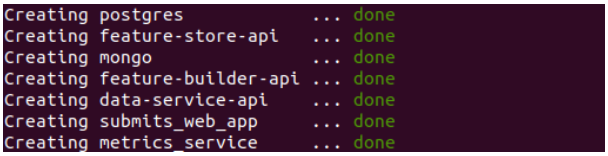


Рис.5.1: Wizard log after starting services

5.5 Configuration database structure and manage rules

Tasks for DS are stored in special database. Database named according to project name and consists of collections:

Таблица5.1: Project database

collection name	collection meaning
feature_loader_configs	configurations for Feature Loader component
metric_service_configs	configurations for Metric Service component
raw_data_loader_configs	configurations for Raw Data Loader component
submits	main submits part
submits.chunks	additional part of submits files stored in GridFS
submits.files	additional part of submits files stored in GridFS
submitter_configs	configurations for Submitter component
target_loader_configs	configurations for Target Loader component
tasks_loader_configs	configurations for Task Loader component

These collections holds configurations for all of the components from Evaluation Tools and submits of DS with notebooks and metadata. As DS project grows, you need to manage configurations in this collections according to these rules:

1. If you need to change some viable aspect of task (train/test sets, target, task name, subjectdomain etc) please create new task for this in tasks_loader_configs
2. If you add new config to any of these collections, please use new unique name for it
3. If you change some non-viable configuration of components it is highly recommented to create copy of configuration in according collection and switch for this new config in Task Loader configuration once it is done

4. Delete stored configurations only in case if you strongly sure that this would not be required in future.

5. Don't delete configurations which was used by Task Loader when users are working on tasks

5.6 Defining tasks

Configuration for tasks are stored in mongodb database according to project in collection named tasks_loader_configs. Each one document in this collection holds one DS task definition. Task definition includes:

1. Configurations links for components used in task. Links are formed by configuration names. You could use several components from one category in one task simultaneously. Note that it is possible to use another Task Loader as a component.

2. Settings for default fill-in constructor parameters used in Evaluation Tools.

Example of task configuration is listed below:

```
{
    "config_name" : "task name",
    "mongo_config" : {
        "host" : "192.168.1.1",
        "port" : 27017,
        "db" : "test_db"
    },
    "description_as_markdown" : {
        "en" : "some markdown task description in english",
        "ru" : "some markdown task description in russian"
    },
    "autofill_parameters" : [
        {
            "function" : "self.submitter.submit_results",
            "parameter_name" : "task_name",
            "parameter_value" : "default task name"
        }
    ],
    "used_config_names" : {
        "submitter.Submitter" : {
            "field_names": ["submitter1", "submitter2", "submitter3"]
            "config_names" : ["submitter_config_name1", "submitter_config_name2", "submitter_
↪config_name3"]
        },
        "target_loader.TargetLoader" : {
            "field_names": ["target_loader1", "target_loader2"]
            "config_names" : ["target_loader_config_name1", "target_loader_config_name2"]
        },
        "feature_loader.FeatureLoader" : {
            "field_names": ["feature_loader"]
            "config_names" : ["feature_loader_config_name"]
        },
        "raw_data_loader.RawDataLoader" : {
            "field_names": ["raw_data_loader1", "raw_data_loader2"]
            "config_names" : ["raw_data_loader_config_name1", "raw_data_loader_config_name2"]
        }
        "task_loader.TaskLoader" : {
            "field_names": ["task_loader2"]
            "config_names" : ["task_loader_config_name2"]
        }

    }
}
```

Config below is deprecated. It will be not supported soon. Old config doesn't provide an ability to define several components in one task:

```
{
    "config_name" : "task name",
    "mongo_config" : {
        "host" : "192.168.1.1",
        "port" : 27017,
        "db" : "test_db"
    },
    "description_as_markdown" : {
        "en" : "some markdown task description in english",
        "ru" : "some markdown task description in russian"
    },
    "autofill_parameters" : [
        {
            "function" : "self.submitter.submit_results",
            "parameter_name" : "task_name",
            "parameter_value" : "default task name"
        }
    ],
    "used_config_names" : {
        "submitter.Submitter" : {
            "config_name" : "submitter_config_name"
        },
        "target_loader.TargetLoader" : {
            "config_name" : "target_loader_config_name"
        },
        "feature_loader.FeatureLoader" : {
            "config_name" : "feature_loader_config_name"
        },
        "raw_data_loader.RawDataLoader" : {
            "config_name" : "raw_data_loader_config_name"
        }
    }
}
```

All fields in this document are obligatory:

- config_name - name of task configuration
- mongo_config - parameters of database that holds configuration
- description_as_markdown - task descriptions in two main languages: English and Russian. Field description_as_markdown may contain templates which will be replaced with corresponding values from task configuration json. Write templates as follows: {fieldname.fieldname_1[0].fieldname_2}, where fieldname_1 is an array. If field name contains dot (.), it should be escaped by «», example: {field\.name.fieldname_1}.
- autofill_parameters - parameters for default fill-in in used tools
- used_config_names - instruments and according configurations names that would be used by Task Loader tool while loading other tools.

This config would be interpreted and processed in two steps:

1. Each tool listed in used_config_names would be initialized (see below)
2. Some methods/functions in loaded tools would be decorated with default parameter values

5.6.1 Initialization of tools in Task Loader

Field used_config_names filled with instruments used for data science in current task. Concretely, names of documents corresponds to module_name.tool_class_name from Evaluation Tools, and document value is just a kwargs parameters that passed to tool on Task Loader init. Then put result object to Task Loader object in field named exacly as module from which it was taken.

For example: If you take configuration listed above, Task Loader would interpret first document from used_config_names in this way:

- to load task, first init all of tools in it, first init submitter.Submitter tool:
  - import submitter module
  - import Submitter class from it
  - construct object of this class with kwargs {"config_name" : "submitter_config_name"}
  - put result object into submitter field of Task Loader object
- repeat for all documents listed in used_config_names

Внимание

There is no guaranteed order of tools initialization by Task Loader.

5.6.2 Decoration with default parameters

For each document listed in autofill_parameters, Task Loader would take method/function listed in function field and decorate it (using partial and docstring carried over to new function). Resulted decorated function is exacly the same, but with one parameter filled in and absent in function signature (because it is already filled in).

This method is actually opens up a way to implement a lot of functionality through Task Loader, for example, you can put some specific tools in git project and init them right away to be available through Task Loader object.

Внимание

Fill-in of parameters are done in exact order as it is listed in autofill_parameters, order is guaranteed.

5.7 Создание записей с признаками для ML задачи

MongoDB config for ML task include reference to FeatureLoader config (alongside with configs for RawDataLoader, TargetLoader and Submitter). FeatureLoader allows to get features (input variables for ML algorithms) from a feature storage. It also heavily uses types for data: types casting is applied to data when FeatureLoader returns pandas.DataFrame object.

Example of MongoDB collection for FeatureLoader usage (partially suppressed):

```
{
    "config_name" : "aero__fw_classification_v1_features",
    "local_temp_dir" : "/tmp",
    "feature_manager_config" : {
        "feature_records" : [
            {
                "name" : "cur_val__aids_rep_04",
                "features_type" : "network_api",
                "info" : "AIDS report 01.",
                "feature_loader_args" : {
                    "files" : [
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    "http://10.30.16.181:8290/api/feature_builder/export?
↪build=1568896505205"
                ],
                "dtypes" : {
                    "cur_val__aids_rep_04__alt" : "np.float32",
                    "cur_val__aids_rep_04__egt_1" : "np.float32",
                    "cur_val__aids_rep_04__egt_2" : "np.float32",
                    "cur_val__aids_rep_04__flight_phase" : "str"
                },
                "default_dtype" : "np.float32",
                "index_columns" : [
                    "AC",
                    "CYCLE_END_TS"
                ],
                "sep" : ","
            },
            "features_full_list" : [
                [
                    "cur_val__aids_rep_04__alt",
                    "cur_val__aids_rep_04__egt_1",
                    "cur_val__aids_rep_04__egt_2",
                    "cur_val__aids_rep_04__flight_phase"
                ]
            ]
        },
    ]
}
}
```

In [8]:
```
feature_loader = task_loader.feature_loader
df = feature_loader.load_features(feature_selector=lambda x: 'cur_val__aids_rep_04' in x)
display(df.sample(5))
print(df.dtypes)
```

AC	CYCLE_END_TS	cur_val__aids_rep_04__alt	cur_val__aids_rep_04__egt_1	cur_val__aids_rep_04__egt_2	cur_val__aids_rep_04__flight_phase
VP-BTN	1453650660	2701.0	7310.0	7700.0	5
VP-BHG	1367005500	1751.0	7050.0	7220.0	5
VP-BHP	1401245460	1804.0	7170.0	7220.0	5
VP-BHL	1384588440	1923.0	6510.0	6560.0	5
VP-BTX	1445611080	1912.0	6710.0	6940.0	5

```
cur_val__aids_rep_04__alt            float32
cur_val__aids_rep_04__egt_1          float32
cur_val__aids_rep_04__egt_2          float32
cur_val__aids_rep_04__flight_phase    object
dtype: object
```

Рис.5.2: The resulting dataframe with features.

First level keys:

- config_name, will be used in TaskLoader config;
- local_temp_dir, path to directory for storing temporary users files, for example to cache responses;
- feature_manager_config, stores feature_records with list of different features records.

Each feature record (feature_records) should include:

- name of features record;
- features_type, states is it local worker or network worker usage (local_disk_config or network_api);
- info, text description of what this record is about;

- feature_loader_args:
  - files, path or link to file with data (Feature Storage);
  - dtypes, dictionary with features names as keys and data types as values;
  - default_dtype, used for features which are not specified in dtypes dict;
  - index_columns, list of index columns names for Pandas;
  - sep, delimiter if used for reading file content, usually for comma or tab separated files;
- features_full_list (optional), if specified then it is possible to get all features names and their types without downloading all features data.

5.8 Создание записей с исходными данными для ML задачи

RawDataLoader allows to get raw data for exploratory data analysis. The structure of the MongoDB collection for RawDataLoader is identical to one for FeatureLoader (see *Defining feature records for task* section) and shares same keys and values. The usage is slightly differs:

- it is possible to load only one raw data record at once;
- files points to Data Service;
- features names should not necessarily follow *feature naming conventions*;
- index_columns always empty list for raw data.

```
{
    "config_name" : "aero__raw_data",
    "local_temp_dir" : "/tmp",
    "feature_manager_config" : {
        "feature_records" : [
            {
                "name" : "flights",
                "features_type" : "network_api",
                "info" : "Flight schedule.",
                "feature_loader_args" : {
                    "files" : [
                        "http://10.30.16.181:8190/datasets/aerophm/flights/data?
↪namespace=aerophm&dataset=flights"
                    ],
                    "dtypes" : {

                        "aircraft_id" : "str",
                        "airport_from" : "str",
                        "airport_to" : "str",
                        "actual_departure_time" : "np.int64",
                        "actual_arrival_time" : "np.int64"
                    },
                    "default_dtype" : "str",
                    "index_columns" : [],
                    "sep" : "\t"
                },
                "features_full_list" : [
                    [
                        "aircraft_id",
                        "airport_from",
                        "airport_to",
                        "actual_departure_time",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                        "actual_arrival_time"
                    ]
                ]
            },
        ]
    }
}
```

In [6]:
```
raw_data_loader = task_loader.raw_data_loader
df = raw_data_loader.load_data_one_record(record_to_use='flights')
display(df.head())
print(df.dtypes)
```

	aircraft_id	airport_from	airport_to	actual_departure_time	actual_arrival_time
0	VP-BTU	OMS	DME	1322698260	1322709600
1	VP-BTN	TJM	DME	1322699160	1322707560
2	VP-BTW	KJA	DME	1322700060	1322716860
3	VP-BHV	SVX	DME	1322700120	1322707800
4	VP-BHL	NOZ	DME	1322700240	1322716560

```
aircraft_id              object
airport_from             object
airport_to               object
actual_departure_time     int64
actual_arrival_time       int64
dtype: object
```

Рис.5.3: The resulting dataframe with raw data.

5.9 Defining target and train/test sets for task

Target definition along with train/test sets definition are one of the most important part in task setup. Target definition is made through these basic steps:

1. Subject model programming
2. Configuring Target Loader component to use certain subject model field

Detailed description about subject model programming process would be provided in future versions of DATASKAI, for now lets just count basic steps you need to implement:

1. In your project git create subdirectory inside ./modules/subject_domain with name of new model
2. Decide which field from which object would be your target field
3. Implement objects and relations between them in python using OOP
4. Create factory with method to construct target object. Method should take kwargs which would become indexes for future datasets
5. On init, factory should use data downloaded with Raw Data Loader tool to fill objects content

Outcome of subject model creation process should be factory, which would init objects of subject domain with target field. You should be able to init and run this factory alone in your jupyter notebook and reproduce creation of objects through construction method and kwargs with indexes.

Train/test sets are defined through Target Loader component configuration. Simple target loader configuration (you can find it in unit tests) is listed below:

```
{
    'config_name' : 'terminator_mass_prediction_v1',
    'config_version' : '0.1',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    'use_mongo_cache' : true,
    'cache_mongodb_address' : '10.30.16.181:27017',

    'config_parameters':
        {
            'subject_model':
                {'directory_path': "../../modules/subject_domain/terminators_v1/",},
            'central_object':
                {
                    'module_and_class_name':'t_600.T600',
                    'id_fields':['robot_index',],
                    'constructor_fields':[''],
                    'target_fields':['robot_mass',],
                    'use_factory':True,

                },
            'factory_for_central_object':
                {
                    'factory_module_and_class_name': 't_factory.TerminatorFactory',
                    'factory_default_init_kwargs' : {
                                    'using_normalization_mass' : 21,
                                    'construct_only_target' : true
                                },
                    'factory_method_for_central_object':'construct_terminator',
                    'factory_method_fields':['t_prod_index',],

                },
            'validation_type':'train_test',
            'folds':[[(1,),(2,),(3,)],[(4,),(5,),(6,)]]
        },
    'override_parameters':
        {
            'tmp_dir':'/tmp',
            'subject_area_dir':'./',
        }
}
```

Let's figure out all the first-level parameters purpose from listed above:

- config_name - string, name of configuration
- config_version - string, version of configuration
- use_mongo_cache - bool, setting for saving result train/test indexes and corresponding target values into mongodb. Should be set to true in order to make Metric Service component count metrics
- cache_mongodb_address - cache mongodb address
- config_parameters - document with main setting for Target Loader component
- override_parameters - internal fields of Target Loader component which should be overwritten

As you can see all interesting parameters are concentrated inside config_parameters document:

- subject_model - subject model code description:
  - directory_path - path to subject model code
- central_object - document with description of object used as target source for task:
  - module_and_class_name - string, module and class of object which would be used for target construction

  - id_fields - list of strings, fields used as index fields in case if object construction is possible without factory
  - constructor_fields - list of strings, constructor fields used in case if object construction is possible without factory
  - target_fields - list of strings, fields to extract from object and use as task target
  - use_factory - bool, whether or not use factory on object construction
- factory_for_central_object - document with description of factory used to obtain object:
  - factory_module_and_class_name - string, name of module and class used as factory
  - factory_default_init_kwargs - kwargs which is passed to factory on factory init
  - factory_method_for_central_object - string, method used for construction of subject domain object
  - factory_method_fields - list of strings, names for args passed to method constructing objects
- validation_type - string, type of validation applied to task (currently only „tran_test“ is supported)
- folds - list, contains lists with train and tests arguments sets to init objects for train and test set respectively

Pay attention for folds field in config_parameters document, this is the place which should be filled in to fix train/test set for task.

As configuration for Task Loader component is written, you can link it to Task Loader component by name and init Task Loader. On first init, Target Loader component would create all objects stated in train/test set and extract target fields. Once this process done, it'll place indexes and corresponding values to cache database, so next call to Target Loader would be much faster and not include subject domain objects creation.

5.10 Создание метрик для ML задачи

Metrics for project are stored in subdirectory ./modules/metrics. For now, these modules should contain function which take two pandas dataframes and outputs scalar. This function would be used by Metrics Service to produce and save results on test set of current task. These functions could use libraries available from DS container because almost every dependency from DS container is carried over to Metric Service container.

Currently metrics are defined in Metrics Service source code, so to change it you should add it to container on build. Modules with metrics should be carried over into ./modules/metrics_provider.py file and implement function provide_metrics. This function should return two lists, first one is metrics names, other one - metrics functions. To add new metrics in metrics list, which is being returned by provide_metrics, you should register the metrics function in the METRIC_STORAGE object via the method register or via decorator @metric(name="„) from ./modules/metrics/metrics_provider_tools.py (see Metrics Service API. To apply different metrics for different tasks you should edit config files in metric_service/configs and specify which metrics should be used in the field tasks_to_use_metrics (for «null» value all available metrics will be applied):

```
"tasks_to_use_metrics": {
        "some_task1": ["roc", "sss"],
        "some_task2": null,
        "some_task3": ["max_f1", "mae"]
    }
```

You could predefine templates to reuse metric lists in tasks below:

```
"tasks_templates":
{
    "template1":
    {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "metrics": ["roc", "sss"]
    },
    "template2":
    {
        "metrics": null
    },
    "template3":
    {
        "metrics": ["max_f1", "mae"]
    }
},
"tasks_to_use_metrics": {
    "some_task1": {"task_template":"template1"},
    "some_task2": {"task_template":"template2"},
    "some_task3": {
      "task_template":"template3",
      "metrics": ["roc_custom"]}
},
```

Metrics names are used by Metrics Service to store according values into submits collection. Example of such implemented interface are shown below:

```
from sklearn.metrics import precision_recall_curve
from modules.metrics.metrics_provider_tools import metric, METRIC_STORAGE


@metric(name="max_f1")
def max_f1_metric(true_result_df, predicted_result_df):
    max_f1_value = None
    precision, recall, thresholds = precision_recall_curve(true_result_df['TARGET'], predicted_
↪result_df['TARGET'])

    for pr, rec, in zip(precision, recall):
        cur_max_f1 = 2 * pr * rec / (pr + rec)

        if max_f1_value is None:
            max_f1_value = cur_max_f1

        if cur_max_f1> max_f1_value:
            max_f1_value = cur_max_f1

    return max_f1_value

def provide_metrics(task_name=None):
    return METRIC_STORAGE.list_metrics(task_name)
```

The method METRIC_STORAGE.list_metrics(task_name) is used to list metrics names and metrics functions in two different lists.

Внимание: Interface of metric function use Pandas dataframe for true test answers and predicted test answers, derived from user. Metric function could use additional information from indexes to calculate metrics. For example, you could weight some of the predictions for important instance with bigger weights than for other.

In future versions task definition would contain detailed configuration of used metrics from git project, including git hash and branch from git repository for each task.

Глава 6

Устранение неполадок

Глава 7

Дополнительная информация

Построено Sphinx, git commit: 01f31f4bd8134d549f4e6afab4830fdf3d9ed15b.