DS

DATASKAI

Фреймворк для AI/ML-проектов

(R&D-версия)

Руководство пользователя Middle Data Scientist

Skoltech

Веб-страница проекта: http://dataskai.com

Содержание

1 Роли пользователей 3

2 Сборка док-контейнера для Junior DS 4

3 Настройка и запуск Metric Service 5

4 Настройка и запуск Submits Web App 6

5 Получение модели из сабмита и проверка воспроизводимости 7

6 Упаковка модели и проверка воспроизводимости 9

6.1 Общие положения 9

6.2 Пример обертывания и упаковки 10

6.3 Валидация модели 12

7 Model Wrapper и Models Player 13

7.1 Настройка и запуск Models Player 14

7.2 Отправка модели в Models Player 14

7.3 Останов модели в Models Player 16

8 Создание Feature Extractor 17

8.1 Соглашения о режимах 18

8.2 Соглашения о файлах конфигурации 18

8.3 Соглашения о входных и выходных файлах 20

1 Роли пользователей

В настоящем руководстве мы придерживаемся разделения всех дата саентистов (DS далее в этом документе), связанных с проектом проектом, на три группы:

Junior DS

Типичные задачи:

Выполнить исследовательский анализ данных (EDA),

создать и представить работу над моделями для задач DS.

Middle DS

Типичные задачи:

– проверка моделей,

– создание и проверка экстракторов функций,

– развертывание служб DATASKAI.

Senior DS

Типичные задачи:

– управление определениями задач DS,

– создание и проверка моделей предметной области,

– отслеживание результатов задач.

2 Сборка док-контейнера для Junior DS

Настройка среды для каждого дата саентиста в отдельности может занять очень много времени. Чтобы сократить время, вы можете создать один контейнер Docker со всеми необходимыми библиотеками и предоставить его Junior DS. Различные окружения также могут повлиять на воспроизводимость результатов. На одном сервере легко запускать несколько контейнеров, что позволяет масштабировать команду. Конкретные инструменты, установленные в контейнере, зависят от задачи.

Мы подготовили контейнер ds_base_docker с базовым набором инструментов. Перед тем, как использовать, вам нужно собрать его командой:

```
./build
```

Затем его можно запустить локально или на сервере:

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

Вы также можете указать разные порт и имя для запуска контейнера на сервере для разных дата саентистов:

```
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
```

3 Настройка и запуск Metric Service

Чтобы автоматически рассчитывать метрики для сабмитов, вам необходимо настроить и запустить Metric Service. Подробно процесс описан в Руководстве администратора. Опции Metric Service позволяют настроить его таким образом, чтобы он рассчитывал набор различных метрик сразу для нескольких задач. Таким образом, вам не нужно запускать несколько экземпляров серсиса для решения нескольких задач.

4 Настройка и запуск Submits Web App

Чтобы просмотреть рассчитанные метрики для сабмитов, необходимо запустить Submits Web App. Процесс настройки и запуска подробно описан в Руководстве администратора.

Параметры Submit Web App позволяют отображать таблицу результатов для нескольких проектов и различных задач в рамках этих проектов. Таким образом, один экземпляр сервиса может использоваться для всех проектов.

После запуска Submit Web App вы можете просматривать отправленные материалы, сортировать их по разным столбцам, а также выполнять поиск по этим столбцам:

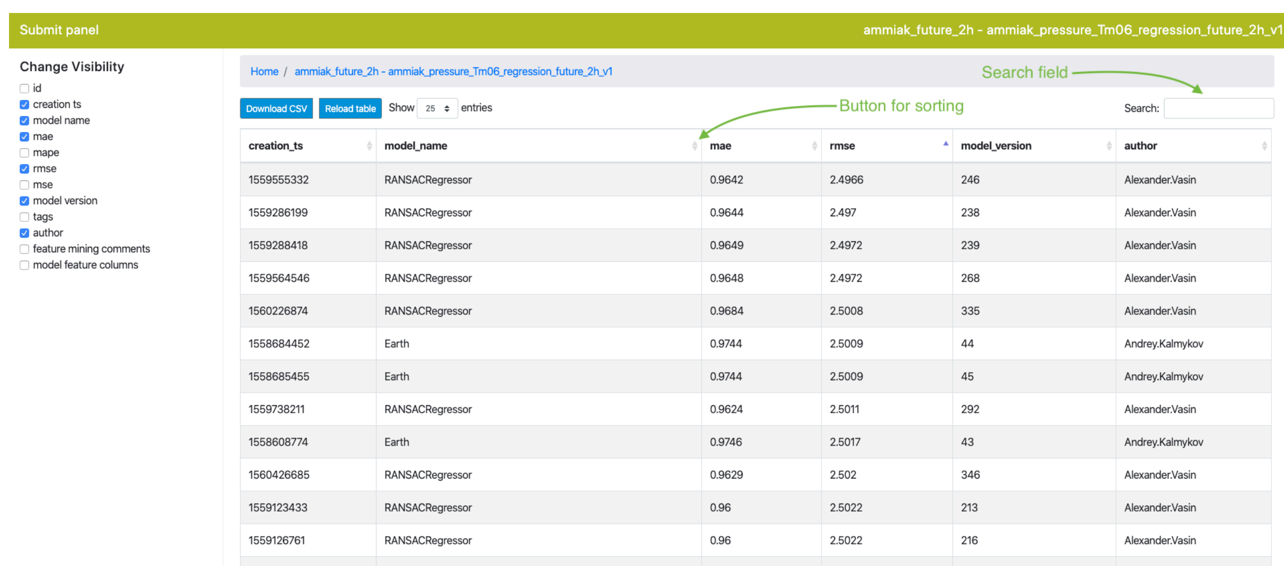

Рис. 1. Интерфейс Submits Web App

5 Получение модели из сабмита и проверка воспроизводимости

Прежде чем переносить модель в сервис, восстановим ее из данных submit и проверим воспроизводимость, то есть сравним выходные результаты.

Для восстановления модели из submit потребуются следующие поля:

— model_feature_columns – используемые функции;

— model_fill_na_values – для заполнения пропущенных значений;

— data_scaler_gridfs_id – преобразователь предварительной обработки данных;

— model_gridfs_id – сама модель;

— result_gridfs_id – прогнозы для тестовых данных.

Первые два из них – простые списки, остальные – объекты, которые нужно запросить у MongoDB GridFS и распаковать из двоичного формата 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']
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()
```

После чего появилась возможность сравнения результатов:

Submit panel | ammiak_future_2h - ammiak_pressure_Tm06_regression_future_2h_v1

Change Visibility: id, creation ts, model name, mae, mape, rmse, mse, model version, tags, author, feature mining comments, model feature columns

Home / ammiak_future_2h - ammiak_pressure_Tm06_regression_future_2h_v1

Search field — Button for sorting

Download CSV | Reload table | Show 25 entries | Search:

creation_ts	model_name	mae	rmse	model_version	author
1559555332	RANSACRegressor	0.9642	2.4966	246	Alexander.Vasin
1559286199	RANSACRegressor	0.9644	2.497	238	Alexander.Vasin
1559288418	RANSACRegressor	0.9649	2.4972	239	Alexander.Vasin
1559564546	RANSACRegressor	0.9648	2.4972	268	Alexander.Vasin
1560226874	RANSACRegressor	0.9684	2.5008	335	Alexander.Vasin
1558684452	Earth	0.9744	2.5009	44	Andrey.Kalmykov
1558685455	Earth	0.9744	2.5009	45	Andrey.Kalmykov
1559738211	RANSACRegressor	0.9624	2.5011	292	Alexander.Vasin
1558608774	Earth	0.9746	2.5017	43	Andrey.Kalmykov
1560426685	RANSACRegressor	0.9629	2.502	346	Alexander.Vasin
1559123433	RANSACRegressor	0.96	2.5022	213	Alexander.Vasin
1559126761	RANSACRegressor	0.96	2.5022	216	Alexander.Vasin

Рис. 2. Проверка воспроизводимости после восстановления представленной модели

6 Упаковка модели и проверка воспроизводимости

6.1 Общие положения

Для упаковки модели необходимо:

— обернуть модель – подготовить файл с классом модели с помощью компонента Model Wrapper;

— подготовить конфигурацию модели, которая должна находиться в файле <models_dir> /configs/config.json;

— выделить необходимые объекты и изменить требуемые значения;

— подготовить каталог конкретной модели со всеми необходимыми файлами.

Каталог модели должен иметь следующую структуру:

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

Где:

— configs – папка с файлом конфигурации модели; — data – необязательный каталог, дополнительные данные (например, файл pickles), необходимые для инициализации модели; — model_wrapper – папка с содержимым компонента model_wrapper; — model.py – модуль Python, описывает методы инициализации, прогнозирования и выключения модели.

Примечание

Имя каталога модели используется в Models Player в качестве идентификатора модели (параметр model_id).

Конфигурация обернутой модели записывается в виде json-файла и помещается в директорию configs. Пример конфигурации модели показан ниже:

```
{
    "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"
}
```

Файл конфигурации содержит следующие обязательные поля:

— model_name – название модели, например Ridge, AllSumModel и др.; обратите внимание, что это не идентификатор модели;

— host – адрес (хост и порт) для сервера gRPC с упакованной моделью;

— model_filename – имя файла Python, который содержит класс с методами инициализации, прогнозирования и завершения модели;

— model_class_name – имя класса Python, который используется для описания модели в модуле <model_filename>;

— message – название текущей сформулированной проблемы;

— model_features – список функций, используемых моделью;

— feature_name_aliases – словарь с псевдонимов функций в формате {true_name: alias}; после применения псевдонимов старые имена теряются;

— tags – ключевые слова для выбора способа работы с моделью, например тег is_prod_model означает, что модель должна использоваться в производстве;

— logging_level – уровень ведения журнала, может быть «информация», «предупреждать», «ошибка» и т. д.

Эти поля обязательны, но возможно внесение любых дополнений в этот конфигурационный файл.

6.2 Пример обертывания и упаковки

Ниже приводится пример обертывания и упаковки модели.

```
import pickle

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

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)
```

```
In [17]:
1 !mkdir my_model_wrapped/
2 !git clone git@srv-iot-git.skoltech.ru:IDCP/model_wrapper.git my_model_wrapped/model_wrapper
3 !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. 3. Клонирование репозитория Model Wrapper и подготовка структуры каталогов

Подготовка класса модели (с удаленными комментариями):

```
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)
```

Конфигурация модели:

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

```
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)
```

Последний шаг – архивация каталога модели:

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

В результате получили упакованную и готовую к отправке в сервис Models Player модель.

6.3 Валидация модели

Перед отправкой проверим упакованную модель. Существует сценарий send_lines_to_model.py, предназначенный для получения нескольких строк прогнозов (только один раз после запуска сценария). Нам нужно подготовить небольшой фрагмент данных, запустить сценарий и сравнить прогнозы с фактическими.

```
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
```

Как видно, получены идентичные результаты (рис. 4).

```
In [17]: 1 !mkdir my_model_wrapped/
         2 !git clone git@srv-iot-git.skoltech.ru:IDCP/model_wrapper.git my_model_wrapped/model_wrapper
         3 !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. Использование скрипта send_lines_to_model и сравнение результатов.

7 Model Wrapper и Models Player

Использование моделей машинного обучения в качестве веб-сервиса важно в случае, когда необходима потовая передача данных для вывода (и получения прогнозов на основе новых данных). Таким образом, модель запускается однократно, а не каждый раз перед выводом. Также важно, чтобы одновременно было доступно много разных моделей машинного обучения.

Models Player – веб-сервис контроллера моделей. Он предоставляет HTTP API, который позволяет пользователю управлять моделями машинного обучения: запускать и останавливать их, делать прогнозы, получать рабочую статистику и т. д.

Model Wrapper используется для упаковки модели перед архивацией и отправкой для использования в Models Player.

7.1 Настройка и запуск Models Player

Свойства из model_player /../ config / default config.json используются по умолчанию, если не передан аргумент config_path. Вы можете изменить хост, порт, связанные с моделью пути и параметры запуска. Также существует возможность передавать параметры через переменные среды. См. Руководство администратора для получения полного списка свойств и команд для запуска Models Player.

Чтобы запустить Models Player (также можно использовать образ Docker):

```
import subprocess
import time

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

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

Для упрощения запросов GET / POST существует Python-оболочка models_player_requests:

```
import models_player_requests

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

7.2 Отправка модели в Models Player

Каждая модель в Models Player имеет собственное независимое состояние: ее можно архивировать, запускать или останавливать. Для переходов между этими состояниями, добавления и удаления моделей, получения прогнозов и различной статистики предоставляется HTTP API (рис. 5).

См. Описание интерфейса для управления жизненным циклом моделей с помощью Models Player API.

Отправка упакованной модели — см. рис. 6.

Запуск обернутой модели — см. рис. 7.

Получение прогнозов — см. рис. 8.

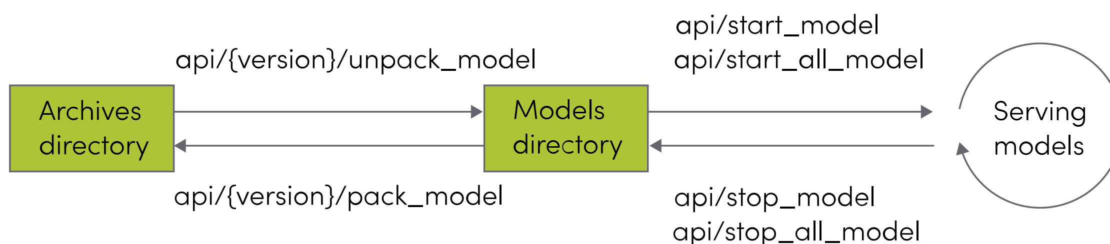

Рис. 5. Состояния модели в Models Player

```
In [42]:
1  models_player.upload_model('my_model_wrapped.tar.gz')
2  print(models_player.list_archives_dir())
3
4  models_player.unpack_model('my_model_wrapped.tar.gz')
5  print(models_player.list_models_dir())

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

Рис. 6. Результат загрузки и распаковки модели

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

['my_model_wrapped']
```

Рис. 7. Результат запуска модели

```
In [44]:
1  model_settings = models_player.get_models_settings(model_id='my_model_wrapped')
2  used_features = model_settings['model_config']['model_features']
3  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. 8. Прогностические данные от модели на Models Player

7.3 Останов модели в Models Player

Используйте stop_model, чтобы остановить определенную модель:

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

Для полного удаления модели требуются дополнительные действия (см. рис. 9).

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

[]
[]
[]
```

Рис. 9. Текущее состояние Models Player

8 Создание Feature Extractor

Feature Extractor (экстрактор признаков) – программа, запускаемая службой Feature Builder для извлечения признаков из необработанных данных. Feature Extractor использует контейнер докеров службы Feature Builder, поэтому в этом контейнере должны учитываться все зависимости. Обычно они переносятся из контейнера DS в контейнер службы Feature Builder.

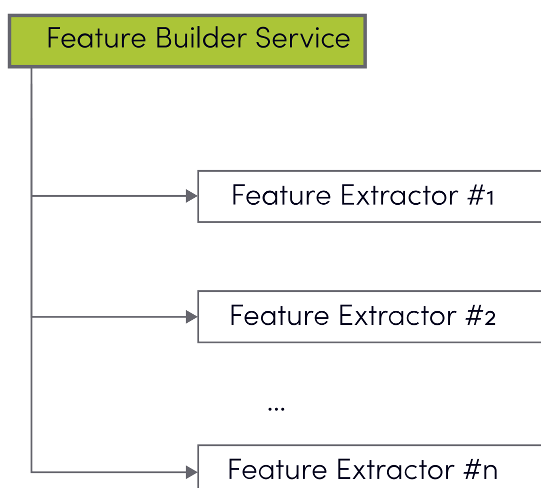

Рис. 10. Сервис Feature Builder и экстрактор признаков

Архитектура экстрактора функций предназначена для быстрого внедрения, разработки и отладки в локальной среде. В случае сбоя служба Feature Builder оставляет входные и выходные файлы в каталоге, поэтому процесс сборки можно изолировать и отладить.

Каталог типичного экстрактора функций показан ниже:

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

└── make_features.sh

Экстрактор признаков состоит из двух необходимых частей:

1) папки config с файлами конфигурации в формате json;

2) make_features.sh – скрипта, поддерживающего следующие флаги:

– -c или –config со следующим путем к файлу конфигурации; – -i или –input со следующей входной папкой для обработки; – -o или –output со следующей выходной папкой для хранения результатов обработки; – -r или –rebuild_needed_data - специальный логический флаг для включения команды переустановки процессора функций

Если вам нужно реализовать собственный экстрактор признаков, необходимо знать некоторые соглашения по умолчанию:

— соглашения о режимах; — соглашения о файлах конфигурации; — соглашения о входных и выходных файлах.

8.1 Соглашения о режимах

Режим преобразования – основной для экстрактора признаков, поэтому, если экстрактор вызывается без какого-либо флага, он должен принять файл default.json в качестве конфигурации, принять input как входной каталог, output как выходной каталог и произвести запуск.

Экстрактор признаков может использовать дополнительную папку данных для сохранения состояний внутренних переменных, необходимых для обработки данных.

Режим восстановления – специальный режим для экстрактора, поэтому он должен использовать файлы из ввода, вычислять некоторое внутреннее состояние и параметры на основе этих входных данных и сохранять все необходимое для последующих вызовов преобразования.

8.2 Соглашения о файлах конфигурации

Файл конфигурации экстрактора функций представляет собой json. Он должен содержать некоторые обязательные поля, а также любую произвольную информацию. Сервис Feature Builder считывает обязательные поля и использует их для различных аспектов системы.

Обязательные поля:

— feature_extractor_name – строковое поле с именем экстрактора признаков. Это имя будет использоваться в качестве ключа, по которому служба конструктора функций идентифицирует текущую конфигурацию экстрактора функций;

— features_sources – список документов с настроенными типами данных. Каждый документ состоит из двух полей:

– filename – строка с именем файла, которая будет содержать входные данные для этого источника данных;

– datatype – строка с именем типа данных, настроенная в java Data Service;

— sources_minimum_time_lag_seconds – задержка, которая будет использоваться сервисом конструктора признаков для получения данных для расчета признаков;

— input_folder – строка с относительным путем к файлу, папка для входных файлов. Если указан параметр -i или –input, это значение будет перезаписано переданным параметром командной строки;

— output_folder – строка с относительным путем к файлу, папка для выходных файлов. Если указан параметр -o или –output, то это значение будет перезаписано переданным параметром командной строки;

— data_folder – строка с относительным путем к файлу, папка, которая должна использоваться как хранилище вычисляемых параметров для экстрактора признаков.

Необязательное поле:

— feature_extractor_settings – документ, который (в случае экстрактора функций Python) передается методу преобразования экстрактора функций, его можно использовать в качестве хранилища для любых необходимых дополнительных параметров, применяемых в алгоритмах извлечения функций.

Пример конфигурации экстрактора функций (с некоторыми дополнительными необязательными полями) приведен ниже:

```
{ "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",
  "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"
  }
}
```

8.3 Соглашения о входных и выходных файлах

Feature Builder предоставляет необходимые файлы для ввода путем чтения файлов конфигурации, хранящихся в экстракторах функций. Один экстрактор функций может предоставлять один или несколько файлов конфигурации, поэтому один код Feature Extractor может предоставлять более одного вывода для одних и тех же входных данных. Выходные файлы хранятся как csv (для простой отладки) или parquet (для быстрой работы, будет доступно в следующих версиях DATASKAI) и должны содержать обязательные поля индекса в соответствии с конфигурацией задачи и конфигурацией сервиса Feature Storage. Выходные файлы должны быть записаны процессом в переданную выходную папку.

Внимание

Экстрактор функций – программа для преобразования данных, и во время преобразования она не должна вызываться. Другими словами, программа не должна сохранять, загружать данные, либо каким-либо образом подключаться к другому внешнему хранилищу состояний во время работы в режиме преобразования. Это ограничение необходимо из-за внешнего ограничения службы Feature Builder: он может предоставлять файлы в папке input в произвольном порядке и без каких-либо гарантий сортировки данных.