Дата последней сборки документа:
15 марта 2021 г. в 00:10

Заметки по машинному обучению и анализу данных

Подвойский А.О.

Здесь приводятся заметки по некоторым вопросам, касающимся машинного обучения, анализа данных, программирования на языках `Python`, `R` и прочим сопряженным вопросам так или иначе, затрагивающим работу с данными.

Краткое содержание

Содержание

1. Основные термины

Квантиль – значение, которое заданная случайная величина не превышает с фиксированной вероятностью. Если вероятность задана в процентах, то квантиль называют процентилем. Пример: фраза «90-й процентиль массы тела у новорожденных мальчиков составляет 4 кг», что означает 90% мальчиков рождаются с массой тела, меньшей или в частном случае равной 4 кг, а 10% соответственно – с массой большей 4 кг. Если распределение непрерывно, то α-квантиль однозначно задается уравнением

$$F_X(x_\alpha) = \alpha.$$

Для непрерывных распределений справедливо следующее широко использующееся при построении доверительных интервалов равенство

$$\mathbb{P}\big(x_{\frac{1-\alpha}{2}} \leqslant X \leqslant x_{\frac{1+\alpha}{2}}\big) = \alpha.$$

Интерквартильный размах – разность между третьим и первым квартилями, то есть $x_{0,75} - x_{0,25}$. Интерквартильный размах является характеристикой разброса и является робастным аналогом дисперсии. Вместе, медиана и интерквартильный размах могут быть использованы вместо математического ожидания и дисперсии в случае распределений с большими выбросами.

Web-сокет – это технология, позволяющая создавать интерактивное соединение для обмена сообщениями в режиме реального времени. Web-сокет в отличие от HTTP не нуждается в повторяющихся запросах к серверу. Сокет работает таким образом, что достаточно лишь один раз выполнить запрос, а потом ждать отклика. То есть можно спокойно слушать сервер, который отправит сообщения по мере готовности. Сокеты применяют приложениях, обрабатывающих информацию в «реальном времени» (IoT-приложения, чаты и пр.)

2. Теория алгоритмов и структуры данных

В *теории сложности вычислений* широкое распространение получило обозначение «O-большое». Типичный результат выглядит следующим образом: «данный алгоритм работает за время $O(n^2 \log n)$», и его следует понимать как «существует такая константа $C > 0$, что *время работы* алгоритма в *наихудшем* случае не превышает $C\, n^2 \log n$, начиная с некоторого n».

Практическая ценность асимптотических результатов такого рода зависит от того, насколько мала неявно подразумеваемая константа c. Как мы уже отмечали выше, для подавляющего большинства известных алгоритмов она находится в разумных пределах, поэтому, как правило, имеет место следующий тезис: алгоритмы, более эффективные с точки зрения их асимптотического поведения, оказываются также более эффективными и при тех сравнительно небольших размерах входных данных, для которых они реально используются на практике. Другими словами, *асимптотические оценки эффективности* достаточно полно отражают реальное положение вещей.

Теория сложности вычислений по определению считает, что алгоритм, работающий за время $O(n^2 \log n)$ лучше алгоритма с временем работы $O(n^3)$, и в подавляющем большинстве случаев это отражает реально существующую на практике ситуацию.

3. Настройка непрерывной интеграции (CI) на GitHub Actions

3.1. Порядок работы с pull-request

Кратко о pull-request. Pull-request (запрос на изменения) – запрос к управляющему каким-либо репозиторием (человеку, группе или вообще роботу) на выполнение изменений из вашего репозитория (и указанной ветки).

Порядок выполнения pull-request на свой проект:

- Делаем форк репозитория,
- Через свой собственный GitHub-профиль клонируем репозиторий на локальную машину,
- Создаем новую ветку под изменения,
- Вносим изменения из-под новой ветки,
- Делаем `git push origin my_branch`,
- На странице GitHub репозитория выбираем новую ветку и нажимаем кнопку «Compare & pull request»; появится форма, в которой можно описать коммит и и добавить комментарий; затем нажимаем «Create pull request»,
- На вкладке «Pull requests» появится диалоговая форма с возможностью сделать «Merge pull request», просмотреть коммит и пр.

Можно выполнить «Merge pull request» через командную строку

```
# Step 1: From your project repository, bring in the changes and test.
git fetch origin
git checkout -b correct-description-readme origin/correct-description-readme
git merge main

# Step 2: Merge the changes and update on GitHub.
git checkout main
git merge --no-ff correct-description-readme
git push origin main
```

После удачного «Merge pull request» лучше удалить соответствующие локальную и удаленную (remote) ветки

```
# удаление локальной ветки
git branch -d <branch_name>
# удаление удаленной ветки
git push origin --delete <branch_name>
```

3.2. Конфигурационные файлы для непрерывной интеграции

Существуют различные инструменты непрерывной интеграции (continuous integration, CI). Вот некоторые из них

- GitHub CI,
- CircleCI,
- Travis CI,
- Buildkite и т.д.

В качестве пример рассмотрим настройку непрерывной интеграции с помощью GitHub Actions.

Простой пример `ci.yaml`

.github/workflows/ci.yaml

```yaml
# This workflow will install Python dependencies, run tests and lint with a variety of Python
    versions
# For more information see: https://help.github.com/actions/language-and-framework-guides/using-
    python-with-github-actions

name: Python package

on:
push:
branches: [ master ]
pull_request:
branches: [ master ]
jobs:
build:
runs-on: ubuntu-latest
strategy:
matrix:
python-version: [3.6, 3.7, 3.8]
steps:
- uses: actions/checkout@v2
- name: Set up Python
uses: actions/setup-python@v2
with:
python-version: ${{ matrix.python-version }}
- name: Cache pip
uses: actions/cache@v1
with:
path: ~/.cache/pip # This path is specific to Ubuntu
# Look to see if there is a cache hit for the corresponding requirements file
key: ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}
restore-keys: |
${{ runner.os }}-pip-
${{ runner.os }}-
# You can test your matrix by printing the current Python version
- name: Display Python version
run: python -c "import sys; print(sys.version)"
- name: Install dependencies
run: |
python -m pip install --upgrade pip
pip install Cython
pip install -r requirements.txt
pip install pycocotools
pip install shapely
pip install black flake8 mypy pytest hypothesis isort pylint
- name: Run black
run:
black --check .
- name: Run flake8
run: flake8
- name: Run pylint
run: pylint iglovikov_helper_functions
- name: Run Mypy
run: mypy iglovikov_helper_functions
- name: Run isort
run: isort --profile black iglovikov_helper_functions
```

```
- name: tests
run: |
pip install .[tests]
pytest
```

Пример CI-файла посложнее

.github/workflows/ci.yaml

```
name: CI
on: [push, pull_request]  # триггеры

jobs:
  test_and_lint:
    name: Test and lint
    runs-on: ${{ matrix.operating-system }}
    strategy:
      matrix:
        operating-system: [ubuntu-latest, windows-latest, macos-latest]
        python-version: [3.6, 3.7, 3.8]
      fail-fast: false
    steps:
    - name: Checkout
      uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}
    - name: Update pip
      run: python -m pip install --upgrade pip
    - name: Install PyTorch on Linux and Windows
      if: >
        matrix.operating-system == 'ubuntu-latest' ||
        matrix.operating-system == 'windows-latest'
      run: >
        pip install torch==1.4.0+cpu torchvision==0.5.0+cpu
        -f https://download.pytorch.org/whl/torch_stable.html
    - name: Install PyTorch on MacOS
      if: matrix.operating-system == 'macos-latest'
      run: pip install torch==1.4.0 torchvision==0.5.0
    - name: Install dependencies
      run: pip install .[tests]
    - name: Install linters
      run: pip install "pydocstyle<4.0.0" flake8 flake8-docstrings mypy
    - name: Run PyTest
      run: pytest
    - name: Run Flake8
      run: flake8
    - name: Run mypy
      run: mypy .

  check_code_formatting:
    name: Check code formatting with Black
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8]
    steps:
    - name: Checkout
      uses: actions/checkout@v2
    - name: Set up Python
```

```
    uses: actions/setup-python@v2
    with:
      python-version: ${{ matrix.python-version }}
  - name: Update pip
    run: python -m pip install --upgrade pip
  - name: Install Black
    run: pip install black==19.3b0
  - name: Run Black
    run: black --config=black.toml --check .

check_sphinx_build:
  name: Check Sphinx build for docs
  runs-on: ubuntu-latest
  strategy:
    matrix:
      python-version: [3.8]
  steps:
  - name: Checkout
    uses: actions/checkout@v2
  - name: Set up Python
    uses: actions/setup-python@v2
    with:
      python-version: ${{ matrix.python-version }}
  - name: Update pip
    run: python -m pip install --upgrade pip
  - name: Install dependencies
    run: pip install -r docs/requirements.txt
  - name: Run Sphinx
    run: sphinx-build -b html docs /tmp/_docs_build

check_transforms_docs:
  name: Check that transforms docs are not outdated
  runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8]
    steps:
    - name: Checkout
      uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}
    - name: Update pip
      run: python -m pip install --upgrade pip
    - name: Install dependencies
      run: pip install .
    - name: Run checks
      run: python tools/make_transforms_docs.py check README.md
```

4. Настройка непрерывной доставки (CD) с помощью Codefresh

5. Разработка собственных Python-пакетов для PyPI

5.1. Семантическое управление версиями

Номер релиза обычно дается в формате MAJOR.MINOR.PATCH:

- MAJOR: изменяется, когда разработчики вносят обратно несовместимые изменения API,
- MINOR: изменяется, когда разработчики вносят обратно совместимые изменения; например, если какая-нибудь функция публичного API признается устаревшей или появляется обратно совместимая новая функциональность,
- PATCH: изменяется, когда разработчики исправляют обратно совместимые ошибки (x.y.Z | x $>$ 0).

Предрелизная версия имеет более низкий приоритет, чем связанные с ней обычные версии. Предрелизная версия указывает, что версия нестабильна и может не удовлетворять предполагаемым требованиям совместимости. Примеры: `1.0.0-alpha`, `1.0.0-alpha.1`, `1.0.0-0.3.7`.

Можно указывать метаданные сборки (метаданные не входят в приоритет): `1.0.0-alpha+001`, `1.0.0+201303`, `1.0.0-beta+exp.sha.5114f85`.

Пример разрешения приоритетов: `1.0.0-alpha` $<$ `1.0.0-alpha.1` $<$ `1.0.0-alpha.beta` $<$ `1.0.0-beta` $<$ `1.0.0-beta.2` $<$ `1.0.0-beta.11` $<$ `1.0.0-rc.1` $<$ `1.0.0`.

5.2. Краткая дорожная карта разработки собственного пакета

Дорожная карта по разработке пользовательского python-пакета для PyPI:

1. Создать рабочую директорию пакета, например, с именем `advancedstatistic`: другими словами, это директория локального git-репозитория, в которой будут расположены поддиректории с python-модулями, реализующими основной функционал пакета, конфигурационные файлы, `README.md`, лицензия и пр.,
2. Создать сконфигурированное виртуальное окружение,
3. Инициализировать git-репозиторий с помощью `git init`
4. Создать поддиректорию пакета с тем же именем что и корневая директория: то есть в локальном репозитории с именем `advancedstatistic` будет находиться поддиректория `advancedstatistic` с модулями, реализующими функционал пакета; в поддиректории обязательно должен находиться файл `__init__.py` (без него будет неверно определена структура пакета)

```
advancedstatistic/ <-- git-репозиторий
    README.md
    LICENSE
    .git/
    .gitignore
    ...
    advancedstatistic/ <-- директория пакета с модулями
        __init__.py <-- специальный файл, который помогает определить структуру пакета
        ... <-- разные py-модули
```

5. Создать файл зависимостей для разработки пакета `requirements_dev.txt`

requirements_dev.txt

```
pip==20.2.4
pytest==6.2.1
pytest-cov==2.10.1
wheel==0.35.1
twine==3.2.0
```

6. Установить зависимости в виртуальное окружение

```
pip install -r requirements_dev.txt
```

7. Запустить процедуру сборки «классического» архива и whl-архива

```
python setup.py sdist bdist_wheel
```

В локальном репозитории будут созданы следующие директории:

- `dist`,
- `build`,
- `advancedstatistic.egg-info`.

8. Теперь можно опубликовать пакет на TestPyPI

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

9. Для проверки можно установить пакет с TestPyPI на локальную машину

```
pip install --index-url https://test.pypi.org/simple/ advancedstatistic
# или если у пакета есть зависимости
pip install \
  --index-url https://test.pypi.org/simple/ \
  --extra-index-url https://pypi.org/simple advancedstatistic
```

В сеансе Python

```
>>> from advancedstatistic.Gauss import source_outliers
...
```

10. Если все прошло без проблем, то теперь можно опубликовать пакет на PyPI

```
twine upload dist/*
```

11. Теперь можно отправить материалы на удаленный git-репозиторий с помощью `git push origin my_branch`

5.3. Инструменты и приемы разработки пакетов

Очень неплохое введение в процедуру оформления проекта от Игловикова https://ternaus.blog/tutorial/2020/08/28/Trained-model-what-is-next.html.

Полезные статьи по оформлению пакета

- https://towardsdatascience.com/10-steps-to-set-up-your-python-project-for-success-14ff88b5
- https://towardsdatascience.com/build-your-first-open-source-python-project-53471c9942a7.

Для того чтобы подчеркнуть, что определенный набор пакетов устанавливается только разработчиками настоящего пакета, имя файла с зависимостями задают не как `requirements.txt`, а как `requirements_dev.txt`.

Пример файла зависимостей

requirements_dev.txt

```
pip==20.2.4
pytest==6.2.1
pytest-cov==2.10.1
wheel==0.35.1
black==20.8b1
```

ВАЖНО: здесь указываются точные версии пакетов в формате «`major.minor.micro`».

Установить пакеты из файла зависимостей можно так

```
pip install -r requirements_dev.txt
```

Файл `setup.py` – это сценарий сборки пакета. Функция `setuptools.setup()` создаст иерархию пакета для загрузки на PyPI. Эта функция содержит информацию о пакете, номере версии и о том какие пакеты требуются пользователям

setup.py

```
from setuptools import setup, find_packages

with open("README.md", "r") as readme_file:
    readme = readme_file.read()

# этот список должен быть как можно менее ограничительным
requirements = ["ipython>=6", "nbformat>=4", "nbconvert>=5", "requests>=2"]

setup(
    name="advancedstatistic",
    version="0.0.1",
    author="Leor Finkelberg",
    author_email="leor.finkelberg@yandex.ru",
    description="A package to convert your Jupyter Notebook",
    long_description=readme,
    long_description_content_type="text/markdown",
    url="https://github.com/LeorFinkelberg/advancedstatistic.git",
    packages=find_packages(),
    install_requires=requirements,  # <-- NB
    classifiers=[
        "Programming Language :: Python :: 3.7",
        "License :: OSI Approved :: GNU General Public License v3 (GPLv3)",
    ],
)
```

Замечание

Иногда в `setup.py` можно встретить аргумент `package_dir`. Этот параметр нужен только тогда, когда устанавливаемые пакеты находятся в папке с другим именем

В отличие от списка зависимостей для разработки `requirements_dev.txt`, список `requirements` в `install_requires=...` должен быть как можно менее ограничительным.

Подробнее об этом в Stack Overflow.

В `install_requires=...` следует включать только те пакеты, которые необходимы для работы приложения.

Замечание

В случае когда пакет уже установлен и требуется его обновить до последней доступной версии, следует использовать конструкцию `pip install -U package_name`

Twine – это набор утилит для безопасной публикации Python-пакетов на PyPI. `twine` нужно добавить в `requirements_dev.txt`

requirements_dev.txt

```
pip==20.2.4
```

```
pytest==6.2.1
pytest-cov==2.10.1
wheel==0.35.1
twine==3.2.0
```

Теперь в корневой директории проекта следует запустить сборку

```
python setup.py sdist bdist_wheel
```

Будет создано несколько директорий – `dist`, `build` и `package_name.egg-info`.

В директории `dist` будут лежать:

- `package_name-0.0.1-py3-none-any.whl`: whl-архив
- `advancedstatistic-0.0.1.tar.gz`: архив исходников:
  - `package_name-0.0.1/.gitignore`,
  - `package_name-0.0.1/package_name/outlier_detection.py`
  - и т.д.

На машине пользователя `pip` будет устанавливать пакет из whl-архива (всякий раз, когда это возможно). Такие whl-архивы быстрее устанавливаются. В том случае, если `pip` не может установить пакет из whl-архива, то он будет пытаться установить пакет из архива исходников.

Теперь необходимо создать учетную запись на тестовом сервере TestPyPI https://test.pypi.org/account/register/.

ВАЖНО: пароли для тестового сервера TestPyPI и официального PyPI должны различаться.

Для безопасной публикации пакета на TestPyPi будем использовать `twine` (при публикации пакета будет предложено ввести логин и пароль)

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Если возникнут ошибки, то нужно обновить версию пакета и удалить артефакты старой сборки (директории `build`, `dist` и `egg`). Затем перестроить пакет с помощью `python setup.py sdist bdist_wheel` и перезагрузить пакет с помощью Twine. Номера версий на платформе TestPyPI не имеет большого значения.

Чтобы каждый раз при публикации пакета не нужно было вводить логин и пароль, можно в домашней директории подготовить конфигурационный файл (`twine` будет просматривать этот файл)

~/.pypirc

```
[distutils]
index-servers =
pypi
testpypi

[testpypi]
repository: https://test.pypi.org/legacy
username = your_username
password = your_pypitest_password

[pypi]
username = your_username
password = your_pypi_password
```

Теперь можно опубликовать свой пакет

```
twine upload -r testpypi dist/*  # на TestPyPI
twine upload dist/*  # на PyPI
```

Если бы пакет был установлен на PyPI, то его можно было бы установить на локальную машину как обычно `pip install package_name`.

В случае с TestPyPI придется использовать модифицированную команду

```
pip install --index-url https://test.pypi.org/simple my_package
```

Если нужно разрешить `pip` извлекать другие пакеты из PyPI, то нужно использовать флаг `--extra-index-url`, чтобы указать на PyPI. Это полезно, когда тестируемый пакет имеет зависимости

```
pip install \
  --index-url https://test.pypi.org/simple/ \
  --extra-index-url https://pypi.org/simple my_package
```

Если у пакета есть зависимости, то нужно использовать второй вариант.

Посмотреть информацию о пакете (размещение пакета и пр.) можно с помощью так

```
pip show my_package
```

Например, на macOs пакеты в виртуальных окружениях устанавливаются в `HOME‣opt‣anaconda3‣envs‣env_name‣lib‣python3.7‣site-packages`.

После того как пакет, например, `my_package`, будет установлен с помощью `pip install my_package`, в директории `... > site-packages` будет создано две поддиректории

- `my_package`,
- `my_package-0.0.2.dist-info`.

5.4. Примеры файлов `setup.py`

Полезные ссылки на материалы, рассказывающие о тонкостях написания `setup.py`:

- https://github.com/navdeep-G/setup.py,
- https://packaging.python.org/guides/distributing-packages-using-setuptools/

Пример простого файла `setup.py` из документации Python https://packaging.python.org/tutorials/packaging-projects/

setup.py

```
import setuptools

with open("README.md", "r", encoding="utf-8") as fh:
    long_description = fh.read()

setuptools.setup(
    name="example-pkg-YOUR-USERNAME-HERE", # станет именем пакета
    version="0.0.1",
    author="Example Author",
    author_email="author@example.com",
    description="A small example package",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/pypa/sampleproject",
    packages=setuptools.find_packages(),
    classifiers=[
```

```
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    python_requires='>=3.6',
)
```

Директория проекта выглядит так

```
packaging_tutorial
 |-- LICENSE
 |-- README.md
 |-- example_pkg  # будет пакетом!!!
 |   |-- __init__.py  # <-- NB
 |-- setup.py
 |-- tests
```

В этом примере:

- `name` – имя дистрибутива пакета,
- `version` – версия пакета (см. PEP 440),
- `author`/`author_email` – автор пакета,
- `description` – одностроковое описание пакета,
- `long_description` – подробное описание пакета (читается из `README.md`),
- `url` – URL домашней страницы пакета (как правило это ссылка на GitHub, GitLab или еще какой-нибудь сервис хостинга),
- `packages` – это список пакетов, которые должны быть включены в дистрибутив; вместо того, чтобы перечислять каждый пакет, можно для автоматического обнаружения пакетов и подпактов использовать функцию `find_packages()` (в данном случае функция найдет только пакет `example_pkg`),
- `classifier` – дополнительные метаданные о пакете.

Для создания дистрибутива пакета следует запустить следующую команду в той же директории, где лежит файл `setup.py`

```
python setup.py sdist bdist_wheel
```

Эта команда должна в директории `dist` сгенерировать два файла

```
dist/
  example_pkg_YOUR_USERNAME_HERE-0.0.1-py3-none-any.whl  # whl-архив
  example_pkg_YOUR_USERNAME_HERE-0.0.1.tar.gz  # tar-архив исходников
```

Для распространения пакетов рекомендуется использовать `twine`.

Другой пример файла сборки позаимствован из репозитория https://github.com/navdeep-G/setup.py

setup.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Note: To use the 'upload' functionality of this file, you must:
#   $ pipenv install twine --dev

import io
import os
```

```python
import sys
from shutil import rmtree

from setuptools import find_packages, setup, Command

# Package meta-data.
NAME = 'mypackage'
DESCRIPTION = 'My short description for my project.'
URL = 'https://github.com/me/myproject'
EMAIL = 'me@example.com'
AUTHOR = 'Awesome Soul'
REQUIRES_PYTHON = '>=3.6.0'
VERSION = '0.1.0'

# What packages are required for this module to be executed?
# Этот список должен быть как можно менее ограничительным
REQUIRED = [
    # 'requests', 'maya', 'records',
]

# What packages are optional?
EXTRAS = {
  # 'fancy feature': ['django'],
}

# The rest you shouldn't have to touch too much :)
# ------------------------------------------------
# Except, perhaps the License and Trove Classifiers!
# If you do change the License, remember to change the Trove Classifier for that!

here = os.path.abspath(os.path.dirname(__file__))

# Import the README and use it as the long-description.
# Note: this will only work if 'README.md' is present in your MANIFEST.in file!
try:
    with io.open(os.path.join(here, 'README.md'), encoding='utf-8') as f:
        long_description = '\n' + f.read()
    except FileNotFoundError:
        long_description = DESCRIPTION

# Load the package's __version__.py module as a dictionary.
about = {}
if not VERSION:
    project_slug = NAME.lower().replace("-", "_").replace(" ", "_")
    with open(os.path.join(here, project_slug, '__version__.py')) as f:
        exec(f.read(), about)  # ???
else:
    about['__version__'] = VERSION


class UploadCommand(Command):
    """Support setup.py upload."""

    description = 'Build and publish the package.'
    user_options = []

    @staticmethod
    def status(s):
        """Prints things in bold."""
        print('\033[1m{0}\033[0m'.format(s))
```

```
    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        try:
            self.status('Removing previous builds...')
            rmtree(os.path.join(here, 'dist'))
        except OSError:
            pass

        self.status('Building Source and Wheel (universal) distribution...')
        os.system('{0} setup.py sdist bdist_wheel --universal'.format(sys.executable))

        self.status('Uploading the package to PyPI via Twine...')
        os.system('twine upload dist/*')

        self.status('Pushing git tags...')
        os.system('git tag v{0}'.format(about['__version__']))
        os.system('git push --tags')

        sys.exit()


# Where the magic happens:
setup(
    name=NAME,
    version=about['__version__'],
    description=DESCRIPTION,
    long_description=long_description,
    long_description_content_type='text/markdown',
    author=AUTHOR,
    author_email=EMAIL,
    python_requires=REQUIRES_PYTHON,
    url=URL,
    packages=find_packages(exclude=["tests", "*.tests", "*.tests.*", "tests.*"]),
    # If your package is a single module, use this instead of 'packages':
    # py_modules=['mypackage'],

    # entry_points={
  #     'console_scripts': ['mycli=mymodule:cli'],
  # },
    install_requires=REQUIRED,
    extras_require=EXTRAS,
    include_package_data=True,
    license='MIT',
    classifiers=[
        # Trove classifiers
        # Full list: https://pypi.python.org/pypi?%3Aaction=list_classifiers
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.6',
        'Programming Language :: Python :: Implementation :: CPython',
        'Programming Language :: Python :: Implementation :: PyPy'
    ],
    # $ setup.py publish support.
```

```
    cmdclass={
        'upload': UploadCommand,
    },
)
```

Шикарный `setup.py` файл из репозитория `aiomisc`

setup.py (эталон)

```
import os

from setuptools import setup, find_packages
from importlib.machinery import SourceFileLoader


module_name = 'aiomisc'

try:
    version = SourceFileLoader(
        module_name,
        os.path.join(module_name, 'version.py') # читаем файл version.py (см. ниже)
    ).load_module()  # вернет что-то вроде  <module 'aiomisc' from 'aiomisc/version.py'>

    version_info = version.version_info  # (11, 1, 0, 'asdfasd')
except FileNotFoundError:
    version_info = (0, 0, 0)


__version__ = '{}.{}.{}'.format(*version_info) # задаем версию на базе version.py


def load_requirements(fname):
    """ load requirements from a pip requirements file """
    with open(fname) as f:
        line_iter = (line.strip() for line in f.readlines()) # генераторное выражение
        return [line for line in line_iter if line and line[0] != '#']


setup(
    name=module_name,
    version=__version__,
    author='Dmitry Orlov',
    author_email='me@mosquito.su',
    license='MIT',
    description='aiomisc - miscellaneous utils for asyncio',
    long_description=open("README.rst").read(),
    platforms="all",
    classifiers=[
        "Framework :: Pytest",
        'Intended Audience :: Developers',
        'Natural Language :: Russian',
        'Operating System :: MacOS',
        'Operating System :: POSIX',
        'Programming Language :: Python',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.5',
        'Programming Language :: Python :: 3.6',
        'Programming Language :: Python :: 3.7',
        'Programming Language :: Python :: 3.8',
        'Programming Language :: Python :: 3.9',
        'Programming Language :: Python :: Implementation :: CPython',
```

```
    ],
    packages=find_packages(exclude=['tests']),
    package_data={"aiomisc": ["py.typed"]},
    install_requires=load_requirements('requirements.txt'),
    extras_require={
      'aiohttp': ['aiohttp'],
      'asgi': ['aiohttp-asgi'],
      'carbon': ['aiocarbon~=0.15'],
      'contextvars': ['contextvars~=2.4'],
      'develop': load_requirements('requirements.dev.txt'),
      'raven': ['raven-aiohttp'],
      'uvloop': ['uvloop>=0.14,<1'],
      'cron': ['croniter~=0.3.34'],
      ':python_version < "3.7"': 'async-generator',
    },
    entry_points={ # точка входа
      "pytest11": ["aiomisc = aiomisc_pytest.pytest_plugin"]
    },
    url='https://github.com/mosquito/aiomisc'
)
```

version.py

```
""" This file is automatically generated by distutils. """

# Follow PEP-0396 rationale
version_info = (11, 1, 0, 'g4edc986')
__version__ = '11.1.0'
```

В `SourceFileLoader` передаем имя и полный путь до модуля, который нужно загрузить, например, так

```
from importlib.machinery import SourceFileLoader

version = SourceFileLoader(
    "version.py",
    os.path.join("fakedir", "version.py")
).load_module()
version.version_info
...
```

5.5. Точки входа в `setup.py` файлах

Точки входа – это методы, с помощью которых другие программы на Python могут обнаруживать динамические свойства, обеспеченные пакетом.

Рассмотрим простой пример. Пусть рабочая директория выглядит так

```
snake_package/ # рабочая директория
  |-- snake_package/ # пакет
  |     |-- snake.py
  |-- setup.py
```

Модуль `setup.py` выглядит следующим образом

setup.py

```
from setuptools import setup

setup(
```

```
    name = "snake_package",
    entry_points = {
        "console_scripts" : [ # группа точек
            "snake = snake_package.snake:main",
        ],
    }
)
```

Здесь `snake` – это имя утилиты командной строки (и одновременно имя точки входа), `snake_package.snake` – это путь до модуля `snake` от корня директории, а `main` – функция модуля `snake`.

Теперь как выглядит модуль `snake.py`

snake.py

```
simple_snake = "simple snake"

def main():
  print(simple_snake)


if __name__ == "__main__":
  main()
```

Теперь, если запустить

```
python setup.py develop
```

то в виртуальном окружении (например, `C▸Users▸ADM▸Anaconda3▸envs▸env_for_tests▸Scripts`) будут созданы следующие файлы

- `snake-script.py`,
- `snake.exe`.

Упрощенно можно считать так: при вызове в командной строке `snake` будет вызвана одноименная точка входа из группы `console_scripts`, которая вызовет ассоциированную с точкой входа `snake` функцию (в данном случае `main`).

Автоматически созданный сценарий `snake-script.py` будет иметь следующее содержание

snake-script.py

```
#!C:\Users\ADM\Anaconda3\envs\env_for_tests\python.exe
# EASY-INSTALL-ENTRY-SCRIPT: 'snake-package','console_scripts','snake'
import re
import sys

# for compatibility with easy_install; see #2198
__requires__ = 'snake-package'

try:
    from importlib.metadata import distribution
except ImportError:
    try:
        from importlib_metadata import distribution
    except ImportError:
        from pkg_resources import load_entry_point


def importlib_load_entry_point(spec, group, name):
    dist_name, _, _ = spec.partition('==')
```

```
20      matches = (
21          entry_point
22          for entry_point in distribution(dist_name).entry_points
23          if entry_point.group == group and entry_point.name == name
24      )
25      return next(matches).load()
26
27
28  globals().setdefault('load_entry_point', importlib_load_entry_point)
29
30
31  if __name__ == '__main__':
32      sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$', '', sys.argv[0])
33      sys.exit(load_entry_point('snake-package', 'console_scripts', 'snake')())
```

Здесь мы сначала пытаемся различными способами включить в пространство имен модуля функцию `distribution`, затем если что-то пошло не так – функцию `load_entry_point` из библиотеки `pkg_resources`.

Если удалось импортировать функцию `distribution`, то функции `load_entry_point` нет в словаре `globals()` и с помощью метода `setdefault` мы создаем в этом словаре пару из отсутствующего ключа `distribution` и ссылки на функцию `importlib_load_entry_point` в качестве значения по умолчанию, а затем возвращаем это значение, так сказать, в никуда. Теперь, обращаясь к функции `load_entry_point` будет вызываться функция `importlib_load_entry_point`.

Если же функция `load_entry_point` была импортирована, то она будет присутствовать в словаре `globals()` и мы просто вернем значение по ключу в никуда.

Модуль `snake-script.py` запускается на прямую, поэтому условие выполняется и мы переходим к строке 32. Здесь, используя регулярное выражение

```
(-script.\pyw?|\.exe)?$
```

которое ищет выражения вида `-script.py` или `-script.pyw` или `.exe` или ничего не ищет в конце строки, замещаем найденную группу пустой строкой в строке, которую возвращает `sys.argv[0]` (это имя запущенного сценария).

Переопределяем первый элемент списка аргументов командной строки и, наконец, вызываем функцию `load_entry_point`:

```
# эта функция возвращает ссылку на функцию, ассоциированную с ключом snake
load_entry_point(
    "snake-package",    # имя пакета
    "console_scripts", # имя группы точек входа
    "snake"             # имя точки входа
)  # <function snake_package.snake.main()>
```

Эта функция вернет ссылку на функцию `main` из модуля `snake.py` пакета `snake_package`, поэтому в строке 33 функция `load_entry_point` вызывается как

```
load_entry_point(...)() # main() -> "simple snake"
```

В том случае, если была загружена не непосредственно функция `pkg_resources.load_entry_point`, а функция по умолчанию `importlib_load_entry_point`, то произойдет следующее.

Сначала имя пакета будет разбито по строке `"=="` на три части с помощью строкового метода `partition`. При распаковке в переменную `dist_name` попадет первая часть строки (до подстроки `"=="`). Затем с помощью функции `distribution` от имени пакета будут извлечены точки входа

```
distribution("snake_package").entry_points
# [EntryPoint(name='snake', value='snake_package.snake:main', group='console_scripts')]
```

Далее с помощью генераторного выражения отбираем только точки входа из группы `console_scripts` с именем `snake`.

Конструкция `next(matches).load()` возвращает ссылку на функцию `main`. Дальше все как и раньше.

Если обобщить, то в данном случае скрипт `snake-script.py` просто вызывает функцию `main()`, ассоциированную с точкой входа `snake`.

Замечание

Для простоты можно считать, что, когда мы говорим в командной строке `snake`, Python ищет точку входа с этим именем и вызывает ассоциированную с этой точкой входа функцию

С помощью утилиты `epi` (Entry Point Inspector) можно визуализировать точки входа

```
# выведет список имен групп точек входа
epi group list | grep console_scripts
```

Продолжим этот пример, но теперь рассмотрим вариант пакета с зарегистрированными точками входа. Пусть `setup.py` выглядит так

setup.py главного пакета

```
from setuptools import setup

setup(
    name = "snake_package",
    entry_points = {
      "console_scripts" : [ # группа точек входа командных сценариев
          "snake = snake_package.snake:main", # вызывает функцию main() из snake_package/snake.
    py
      ],
      "snake_types" : [ # группа точек входа
          "normal = snake_package.snake:normal_snake", # точка входа
          "simple = snake_package.snake:simple_snake", # точка входа
      ],
    }
)
```

а основной модуль так

snake_package/snake.py

```
import pkg_resources

simple_snake = "simple_snake"

normal_snake = "normal snake"

def main():
    for entry_point in pkg_resources.iter_entry_points("snake_types"):
        print(f"{entry_point.name} --> {entry_point.load().upper()}")


if __name__ == "__main__":
    main()
```

В случае, если точка входа связанна с функцией, `entry_point.load()` вернет ссылку на эту функцию (а не вызовет функцию!!!), а в случае переменной `entry_point.load()` вернет значение этой переменной.

Устанавливаем пакет

```
python setup.py develop
```

Теперь при вызове в командной строке консольного сценария `snake` будет вызвана одноименная точка из группы `console_scripts`, которая вызовет связанную с этой точкой функцию `main` из модуля `snake_package/snake.py`.

Функция `main` с помощью `pkg_resources.iter_entry_points` просканирует окружение на предмет наличия групп точек входа с именем `snake_types`, и извлечет из ее точек имя и ссылку на связанную функцию.

Что особенно интересно `pkg_resources.iter_entry_points("snake_types")` будет сканировать *все* окружение, то есть, если в каком-то другом пакете будет объявлена группа точек входа `snake_types` (в `setup.py`), то функция `main` нашего пакета об этом узнает

setup.py из какого-то другого пакета

```
...
setup(
    name = "foobar",
    ...
    packages = ["foobar"],
    entry_points = {
        "console_scripts" : [
            "foobar-server = foobar.server:main",
            ...
        ]
        "snake_types" : [ # регистрация группы точек входа
            "pretty = add_module:pretty_snake", # вызовет переменную из модуля add_module.py в к
    орне пакета foobar
        ],
    }
)
```

Другой пример

setup.py

```
from setuptools import setup

setup(
    name="foobar",
    entry_points={ # точки входа
        "console_scripts"  : [ # группа точек входа
            "foobar-server = foobar.server:main", # вызовет функцию main() из foobar/server.py
            "foobar-client = foobar.client:main", # вызовет функию main() из foobar/client.py
        ],
    }
)
```

Здесь `foobar-server` и `foobar-client` – это сценарии командной оболочки. Модуль `setuptools` читает `foobar-server = foobar.server:main` как

```
<консольный_скрипт> = <путь к питоновскому модулю:функция>
```

создавая для каждого элемента консольную утилиту при установке пакета.

Теперь если запустить

```
python setup.py develop
```

то в виртуальном окружении (например, `C▸Users▸ADM▸Anaconda3▸envs▸env_for_tests▸Scripts`) будут созданы следующие файлы

- `foobar-client.exe`,
- `foobar-client-script.py`,
- `foobar-server.exe`,
- `foobar-server-script.py`.

Например, файл `foobar-client-script.py` имеет следующее содержание

foobar-client-script.py

```
#!C:\Users\ADM\Anaconda3\envs\env_for_tests\python.exe
# EASY-INSTALL-ENTRY-SCRIPT: 'foobar','console_scripts','foobar-client'
import re
import sys

# for compatibility with easy_install; see #2198
__requires__ = 'foobar'

try:
    from importlib.metadata import distribution
except ImportError:
    try:
        from importlib_metadata import distribution
    except ImportError:
        from pkg_resources import load_entry_point


def importlib_load_entry_point(spec, group, name):
    dist_name, _, _ = spec.partition('==')
    matches = (
        entry_point
        for entry_point in distribution(dist_name).entry_points
        if entry_point.group == group and entry_point.name == name
    )
    return next(matches).load()


globals().setdefault('load_entry_point', importlib_load_entry_point)


if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$', '', sys.argv[0])
    sys.exit(load_entry_point('foobar', 'console_scripts', 'foobar-client')())
```

Точки входа организованны в группы: каждая группа – это список пар ключ/значение. Пары ключ/значение используют формат `path.to.module:variable_name`.

Простейший путь для визуализации точек входа, доступных в пакете, – это использование пакета Entry Point Inspector. Его можно установить, запустив `pip install entry-point-inspector`. После установки доступна утилита командной строки `epi`

```
$ epi group list
+----------------------------------------+
```

```
| Name                                    |
+-----------------------------------------+
| apscheduler.executors                   |
| apscheduler.jobstores                   |
| apscheduler.triggers                    |
| asdf_extensions                         |
| babel.checkers                          |
```

Здесь каждый элемент таблицы – это имя группы точек входа.

Точка входа может быть использована `setuptools` для установки маленькой программы в системное окружение, которая вызывает определенную функцию одного из модулей. Используя `setuptools`, можно указать вызов функции, с которой *начнется программа*, установив пару ключ / значение в точку входа группы:

- ключ – это имя устанавливаемого скрипта,
- а значение – это путь функции (что-то вроде `my_module.main`).

Для того чтобы работала связка с группами точек входа, требуется, чтобы главная функция пакета (как в данном случае, функция `main`) умела сканировать группу точек входа, а сами группы точек входа могут объявляться как в файле `setup.py` главного пакета, так и в файлах `setup.py` других пакетов.

Создадим cron-подобного демона `pycrond`, который позволит любой программе Python регистрировать команду и выполнять ее каждые несколько секунд путем регистрации точки входа в группе `pytimed`.

Проект выглядит так

```
base_directory/
  |-- pytimed.py
  |-- setup.py
  |-- hello
        |-- hello.py
```

Вот реализация `pycrond` с применением `pkg_resources` для поиска точек входа в программе под названием `pytimed.py`

pytimed.py

```python
import pkg_resources
import time


def main():
    seconds_passed = 0
    while True:
        for entry_point in pkg_resources.iter_entry_points("pytimed"): # сканирует окружение на
    предмет наличия группы pytimed
            try:
                seconds, callable = entry_point.load()() # потому что entry_point.load() возвращ
    ает лишь ссылку на функцию, а не вызывает ее
            except:
                pass
            else:
                if seconds_passed % seconds == 0:
                    callable()
        time.sleep(1)
        seconds_passed += 1
```

hello/hello.py

```
def print_hello():
    print("Hello, world")

def say_hello():
    return 2, print_hello
```

Наконец, файл `setup.py`

setup.py

```
from setuptools import setup

setup(
    name = "hello",
    version = "1",
    packages = ["hello"],
    entry_points = {
      "pytimed" : [ # группа точек входа
          "hello = hello.hello:say_hello", # вызывается функция say_hello() модуля hello.py паке
    та hello
      ],
    }
)
```

Теперь

```
python setup.py develop
```

Скрипт `setup.py` регистрирует точку входа в группе `pytimed` с ключом `hello` и значением, обращенным к функции `hello.hello.say_hello`.

Как только с помощью `setup.py` устанавливается пакет – например, через `pip install`, – скрипт `pytimed` обнаруживает вновь добавленную точку входа.

При импорте `pytimed` отсканирует группу `pytimed` и найдет ключ `hello`. Далее он вызовет функцию `hello.hello.say_hello`, получив два значения: количество секунд для паузы между каждым вызовом и функцию для вызова

```
>>> import pytimed
>>> pytimed.main()
Hello, world
Hello, world
...
```

Возможности, предоставляемые этим механизмом, обширны: можно без проблем создавать драйверы, устанавливать хуки и расширения.

Точки входа делают процесс поиска и динамической загрузки кода легче для развертываемого пакета, но это не единственное их применение. Любое приложение может предлагать и регистрировать точки входа или их группы для использования по своему усмотрению.

Библиотека `stevedore` обеспечивает поддержку динамических плагинов на основе ранее продемонстрированного механизма. Рассмотренный пример уже очень прост, но его можно еще упростить в следующем скрипте

pytimed_stevedore.py

```
from stevedore.extension import ExtensionManager
import time
```

```
def main():
    seconds_passed = 0
    extensions = ExtensionManager("pytimed", invoke_on_load=True)
    while True:
        for extension in extensions:
            try:
                seconds, callable = extension.obj
            except:
                pass
            else:
                if seconds_passed % seconds == 0:
                    callable()
        time.sleep(1)
        seconds_passed += 1
```

Класс `ExtensionManager`, принадлежащий `stevedore`, обеспечивает простой путь для загрузки всех расширений группы точек входа. Имя передается в качестве аргумента. Аргумент `invoke_on_load=Tru` обеспечивает, при ее нахождении, вызов каждой функции группы. Это делает результаты доступными напрямую из атрибута `obj`, принадлежащего расширению.

Если нужно загрузить и запустить только одно расширение из группы точек входа, то это можно сделать с помощью класса `stevedore.driver.DriverManager`

Применение stevedore для запуска одного расширения из точки входа

```
from stevedore.driver import DriverManager
import time

def main(name):
    seconds_passed = 0
    seconds, callable = DriverManager("pytimed", name, invoke_on_load=True).driver
    while True:
        if seconds_passed % seconds == 0:
            callable()
        time.sleep(1)
        seconds_passed += 1

main("hello")
```

В этом случае только одно расширение загружается и выбирается по имени. Это позволяет быстро создать систему драйверов, в которой программа загружает и использует только одно расширение.

6. Обработка исключений при чтении данных в Pandas

При загрузке данных в Pandas необходимо обрабатывать исключения, связанные с тем, что файл может не существовать или быть занят другим приложением и пр.

Обрабатывать такие исключения удобнее всего с помощью *менеджера контекста*

```
def read_data(data_filepath: str) -> pd.DataFrame:
    with open(data_filepath, "r") as f:
        print(f) # <_io.TextIOWrapper name='../filename.csv' mode='r' encoding='cp1251'>
        data = pd.read_csv(f, delimiter=";")
    return data
```

То есть в данном случае функции `read_csv` передается не путь до файла или имя файла в формате строки, а *файловый объект*.

Известно, что программа завершится от любого необработанного исключения, а не только от SystemExit. Таким образом, если в вашем коде используются какие-то ресурсы, которые требуется правильным образом закрывать перед завершением работы, нужно оборачивать работу с ними в блоки `try...finally...`.

Однако, при использовании конструкции `with` это *оборачивание происходит автоматически*, и все ресурсы закрываются корректно.

Так как выход из программы – это всего лишь брошенное исключение, то и в случае использования функции `sys.exit` закрытие открытых в операторе `with` ресурсов произойдёт корректно

```
import contextlib

class Closeable:
    def close(self):
        print("closed")

with contextlib.closing(Closeable()): # для классов, которые не приспособлены для работы с with
    sys.exit() # напечатает closed
```

7. Выход из приложения при перехвате исключения ветками try-except

Быстрый способ выйти из приложения при ошибке сводится к использованию `sys.exit()`. По умолчанию статус выхода 0. При `sys.exit(0)` не будет выводится дополнительная информация об ошибке. При статусе завершения `>0` вывод будет обогащен дополнительными сведениями.

Вызов `sys.exit()` возбуждает исключение `SystemExit`. Если, к примеру, требуется завершить работу приложения, если файл с данными не найден, то можно в конце блока `except` добавить строку `sys.exit()`

```
def preprocessing_paths():
    ...
    try:
        if not data_filepath.exists():
            raise DataFilepathNotExists("Ошибка! Файл с данными не существует")
    except DataFilepathNotExists as err:
        print(f"{err}")
        sys.exit() # завершит работу приложения и выбросит из сессии
    else:
        ...
        return data_filepath, output_fig_filepath
```

8. Логгирование в Python

Для того чтобы записи журнала уровня INFO не игнорировались логгером, следует установить базовый уровень логгирования в INFO или DEBUG (уровень логгирования по умолчанию WARNING)

```
file_log = logging.FileHandler("logs.log") # файловый хендлер
console_out = logging.StreamHandler(sys.stdout) # потоковый хендлер

logging.basicConfig(
    handlers=(file_log, console_out),
```

```
    format=("[%(asctime)s | %(levelname)s]: %(message)s"),
    datefmt='%Y-%m-%d %H:%M:%S',
    level=logging.INFO, # базовый уровень логгирования
)
...
logging.info(...)
logging.error(...)
```

9. Приемы работы с библиотекой argparse

Для того чтобы логику работы python-сценария можно было менять «на лету», используются различные инструменты разработки приложений с интерфейсом командной строки. Например, можно использовать библиотеку `argparse`

```
parser = argparse.ArgumentParser()
parser.add_argument("--config-path", type = str)
args = parser.parse_args()

CONFIG_FILENAME = args.config_path # вернет значение, переданное флагу --config-path
```

В командной оболочке вызов должен выглядеть так

```
python file_name.py --config-path config.yaml
```

10. Конфигурационные файлы как интерфейс доступа к Python-сценарию

Конфигурационный файл может использоваться как интерфейс доступа к Python-приложению. Пример файла конфигурации

config.yaml

```
  input_data_path: "ml_example/data/raw/train.csv" # относительно корня проекта
  output_model_path: "models/model.pkl"
  metric_path: "models/metrics.json"
  splitting_params:
  val_size: 0.1
  random_state: 3
  train_params:
  model_type: "RandomForestRegressor"
  feature_params: # этому элементу соответствует класс FeatureParams
    categorical_features:
        - "MSZoning"
      - "Neighborhood"
      - "RoofStyle"
      ...
```

Чтобы использовать значения из конфигурационного файла в приложении, следует прочитать этот файл с помощью библиотеки `yaml`

```
import yaml

with open("config.yaml", "r") as f:
  config = yaml.safe_load(f) # config -- это обычный словарь
```

Однако в данном случае не проверяются типы переданных значений. Для организации проверки типов можно использовать модуль `dataclasses`

train_pipeline_params.py

```
from dataclasses import dataclass
from .split_params import SplittingParams
from .feature_params import FeatureParams
from .train_params import TrainingParams
from marshmallow_dataclass import class_schema
import yaml

@dataclass()
class TrainingPipelineParams:
    """
    Главный класс для построения схемы конфигурационного файла
    """
    input_data_path: str
    output_model_path: str
    metric_path: str
    splitting_params: SplittingParams
    feature_params: FeatureParams
    train_params: TrainingParams

TrainingPipelineParamsSchema = class_schema(TrainingPipelineParams)

def read_training_pipeline_params(path: str) -> TrainingPipelineParams:
    with open(path, "r") as input_stream:
        schema = TrainingPipelineParamsSchema()
    return schema.load(yaml.safe_load(input_stream)) # возвращает объект, к полям которого можно
     обращаться с помощью точечной нотации
```

Пример класса для построения схемы подэлемента конфигурационного файла

feature_params.py

```
from dataclasses import dataclass, field
from typing import List, Optional

@dataclass()
class FeatureParams:
    """
    Класс для построения схемы подэлемента конфигурационного файла.
    Соответствует элементу feature_params файла config.yaml
    """
    categorical_features: List[str]
    numerical_features: List[str]
    features_to_drop: List[str]
    target_col: Optional[str]
    use_log_trick: bool = field(default=True) # <-
```

Например, если

```
params = read_training_pipeline_params("config.yaml")
```

то вызвать значение, скажем, `use_log_trick` можно так

```
params.feature_params.use_log_trick # вернет True (так как это значение по умолчанию в классе
    FeatureParams)
```

11. Тестирование в Python

Хранить тесты нужно в поддиректории `tests` пакета, приложения или библиотеки, к которым они относятся. Использование иерархии в дереве теста, которая повторяет иерархию модуля, сделает тесты более управляемыми. Это значит, что тест для кода `mylib/foobar.py` необходимо размещать в `mylib/tests/test_foobar.py`.

Имена тестов должны совпадать с именами тестируемых модулей, но с добавлением префикса или постфикса `test`, т.е. `test_modulename` или `modulename_test`. Это поможет `pytest` находить тесты.

`pytest` запущенный без аргументов или с указанием каталога ищет функции для тестирования рекурсивно в подкаталогах в соответствии с соглашением по именам:

- файлы называются `test_().py` или `()_test.py`,
- тестовые функции называются `test_()`,
- тестовые классы `Test()`.

Тесты бывают:

- позитивные: пишутся просто через проверку `assert condition`,
- негативные: пишутся через `with pytest.raises(TypeError)`; чтобы убедиться, что Python возбудил правильное исключение,
- граничные (подходит Hypothises).

Для оценки полноты покрытия кода тестами можно использовать плагин `pytest-cov` (показывает какие строки не исполнялись во время тестирования).

Для маленьких проектов с простым применением – пакет `pytest`. Пакет `pytest` предоставляет команду `pytest`, которая загружает каждый файл, имя которого начинается на `test_`, и затем выполняет все функции внутри каждого файла, если они тоже начинаются на `test_`.

Запустить тест можно так

```
pytest -v test_true.py
```

Флаг `-v` выводит имя каждого теста в отдельной строке. Если тест не пройден, вывод меняется и отображается информация об ошибке.

Тест не проходит, как только возникает исключение `AssertationError`.

11.1. Что можно тестировать в задачах анализа данных

Концепция *разработки через тестирование* (TDD) не работает в задачах машинного обучения и анализа данных.

Что тестировать:

- Функциональные тесты: результат, а не реализация
- Нефункциональные тесты: быстродействие
- Тесты производительности модели: если какой-то признак может поламать модель, тесты должны это показать
- Тесты собственно данных: имена колонок, границы, дубли и пр.

11.2. Пример организации директории под тесты

Если в директории, которая позиционируется как пакет, есть модуль `__init__.py`, то при импорте пакета (`import package`, `from my_cool_lib import package`) будет выполнено все, что написано в модуле `__init__.py`. И если модуль `__init__.py`, например, импортирует какие-то функции из другого модуля, то эти функции станут частью пространства имен пакета, ассоциированного с модулем `__init__.py`.

Пример директории проекта

```
myproject/
  -- main_part/
    -- __init__.py # from .simple_module import my_summa\n\n__all__ = ["my_summa"]
    -- simple_module.py # функция my_summa
  -- tests/
    -- __init__.py # пустой файл
    -- test_summa.py # from main_part import my_summa
```

main_part/__init__.py

```
from .simple_module import my_summa # <- NB: относительный импорт модуля!!!

__all__ = ["my_summa"] # my_summa станет частью пространства имен пакета main_part
```

simple_module.py

```
def my_summa(a, b):
    return a + b
```

test_summa.py

```
from main_part import my_summa

def test_summa():
    a = 10
    b = 20
    res = my_summa(a, b)
    assert res == 30
```

В директории (пакете) `main_part` лежит модуль `__init__.py`, который с помощью относительного импорта (но можно было использовать и технику абсолютного импорта) извлекает из модуля `simple_module` функцию `my_summa`. Поэтому при импорте пакета `main_part` функия `my_summa` станет частью пространства имен пакета.

СВОДКА: таким образом, из-под директории `tests` используя технику абсолютного импорта (от корня проекта) можно импортировать *нужные рабочие* модули проекта. В директории `__init__.py` и пакетах обязательно должны находиться модули `__init__.py` (не смотря на то, что в последних версиях Python это и не обязательно). При импорте подмодулей текущего пакета инструкции импорта в модуле `__init__.py` могут использовать либо технику абсолютного, либо относительного импорта.

ПРАВИЛА импортов пакетов и модулей обычного проекта:

- При импорте пакетов или модулей проекта используется *техника абсолютного импорта*, т.е. прописывается путь до интересующего пакета или модуля относительно корня проекта.
- Если в директории/пакете присутствует файл `__init__.py`, внутри которого выполняется импорт функций, классов и пр. локальных модулей, то эти фукнции и классы становятся частью пространства имен текущего пакета.

ПРАВИЛА импортов пакетов и модулей из-под директории `tests`:

- в модулях, предназначенных для тестирования, используется *техника абсолютного импорта*, т.е. прописывается путь до интересующего пакета или модуля относительно корня проекта, например, если в проекте есть пакет `ml_example`, в котором, в свою очередь, есть пакет `entities` (в этом пакете доступен класс TrainingPipelineParams), то импорт из-под тестового модуля директории `tests` может выглядеть так

tests/test_end2end_training.py

```
# относительно корня проекта
from ml_example.entities import TrainingPipelineParams
# модуль train_pipeline лежит в корне проекта
from train_pipeline import train_pipeline # из модуля train_pipeline импортируется соответст
    вующая функция
```

11.3. Пропуск тестов

Чтобы пропустить тест удобно пользоваться декоратором `pytest.mark.skip()`. Этот декоратор безоговорочно пропускает декорированную функцию, поэтому его можно использовать всегда, когда нужно пропустить тест

Пропуск тестов

```
imoprt pytest

try:
    import mylib
except ImportError:
    mylib = None

@pytest.mark.skip("Do not run this")
def test_fail():
    assert False

@pytest.mark.skipif(mylib is None, reason="mylib is not available")
def test_mylib():
    assert mylib.foobar() == 42

def test_skip_at_runtime():
    if True:
        pytest.skip("Finally I don't want to run it")
```

11.4. Запуск определенных тестов

При использовании `pytest` часто возникает необходимость запустить только определенные тесты. Можно выбрать какие запустить, передав их директорию или файлы в качестве аргументов в командную строку. Например, вызов `pytest test_one.py` запускает только `test_one.py`. Пакет `pytest` также принимает директорию в качестве агрумента, и в этом случае он рекурсивно просмотрит папки и запустит все файлы, соответствующие шаблону `test_*.py`.

Обычно разработчик группирует тесты по типам и функциональности, а не по именам. Библиотека `pytest` обеспечивает динамическую систему меток, позволяющую маркировать тесты с помощью ключевого слова, которое затем может быть использовано в фильтре. Для маркировок тестов таким способом следует использовать опцию `-m`. Если настроить пару тестов вроде этих

test_mark.py

```
import pytest

@pytest.mark.skip("Do not run this") # если нужно пропустить тест
@pytest.mark.mymark # метка
def test_something():
    a = ["a", "b"]
    assert a == a


def test_something_else():
    assert False
```

то можно использовать аргумент `-m` с `pytest` для запуска только одного из них

```
pytest -v test_mark.py -m mymark
```

Чтобы при запуске `pytest -v` не выводились ненужные предупреждения о якобы опечатках в именах меток «*Unknown pytest.mark.params - is this a typo?*», нужно зарегистрировать пользовательскую метку.

Это можно сделать с помощью конфигурационных файлов https://docs.pytest.org/en/stable/mark.html. Например, с помощью `pyproject.toml`

pyproject.toml

```
# после двоеточия указывается необязательное описание
[tool.pytest.ini_options]
markers = [
    "mymark: marks test (deselect with '-m \"not mymark\"')",
    "DB",
    "disttest",
    # "params"
]
...
```

Метка `-m` принимает и более сложные выражения, поэтому можно, например, запустить все тесты, которые *не* имеют метки

```
pytest test_mark.py -m 'not mymark'
```

В примере `pytest` выполнит каждый тест, не отмеченный `mymark`. `pytest` принимает сложные выражения, состоящие из `or`, `and` и `not`, что позволяет производить сложную фильтрацию.

11.5. Параллельный запуск тестов

Запуск тестовых наборов может отнимать много времени. По умолчанию `pytest` запускает тесты последовательно, в определенном порядке. Так как большинство компьютеров имеют многоядерные процессоры, можно ускориться, если произвести разделение тестов для запуска на нескольких ядрах.

Для этого в `pytest` есть плагин `pytest-xdist`. Этот плагин расширяет командную строку `pytest` аргументом `--numprocesses` (сокращенно `-n`), принимающим в качестве аргумента количество используемых ядер.

Запуск `pytest -n 4` запустит тестовый набор в четырех параллельных процессах, сохраняя баланс между загруженностью доступных ядер.

Плагин также принимает ключевое слово `auto`. В этом случае количество доступных ядер будет возвращено автоматически

```
pytest -v test_true.py -n auto
```

Вот еще несколько полезных опций `pytest`:

- `--tb=[auto/long/short/line/native/no]`: управляет стилем трассировки,

```
# с отключенной трассировкой (--tb=no)
pytest --tb=no -q test_true.py
```

- `-l / --showlocals`: отображает локальные переменные рядом с трассировкой стека,
- `--lf / --last-failed`: запускает только те тесты, которые завершились неудачей,
- `-x / --exitfirst`: останавливает тестовую сессию при первом сбое,
- `--pdb`: запускает интерактивный сеанс отладки в точке сбоя

```
# выводим не более 3 ошибок с подробным описанием
pytest --tb=no -v --lf --maxfial=3
# запустить сеанс отладки
pytest -v --lf -x --pdb
```

В сессии `pdb` можно использовать следующие команды[1]

- `p expr`: вывести значение `expr`,
- `l / list`: вывести строку точки сбоя и строки окружения (5 сверху и 5 снизу),
- `a / args`: вывести аргументы текущей функции,
- `u / up`: переместиться на один уровень вверх по трассе стека,
- `d / down`: переместиться на один уровень вниз по трассе стека,
- `q / quit`: завершить сеанс.

Еще один полезный флаг `--collect-only` бывает полезен в ситуациях, когда перед запуском тестов нужно убедиться в группу запускаемых попали именно те тесты, которые нужны.

Посмотреть доступные фикстуры можно с помощью флага `--fixtures`.

11.6. Создание объектов, используемых в тестах, с помощью фикстур

В модульном тестировании часто придется выполнять набор стандартных операций до и после запуска теста, и эти инструкции задействуют определенные компоненты. Например, может понадобиться объект, который будет выражать состояние конфигурации приложения, и он должен инициализироваться перед каждым тестированием, а потом сбрасываться до начальных значений после выполнения. Аналогично, если тест зависит от временного файла, этот файл должен создаваться перед тестом и удаляться после. Такие компоненты называются *фикстурами*. Они устанавливаются перед тестированием и удаляются после его выполнения.

В `pytest` фикстуры объявляются как простые функции. Функция фикстуры должна возвращать желаемый объект, чтобы в тестировании, где она используется, мог использоваться этот объект.

У фикстур есть область действия `scope`: `session`, `module`, `class`, функция (по умолчанию). Фикстура запускается для области действия один раз и действует в рамках этой области видимости.

[1] Навигационные команды `step` и `next` не очень полезны, так как мы находимся прямо в операторе `assert`

Хорошей практикой считается размещать фикстуры в отдельном файле `conftest.py` в корневой директории тестов, оттуда они будут рекурсивно доступны во всех подкаталогах. Это не модуль, а плагин. Его не надо импортировать в тестовых модулях.

Есть встроенные фикстуры:

- `tmpdir`: создает временную директорию,
- `tmpdir-factory`: тоже создает временную директорию, но для сессии,
- `capsys`: захватывает стандартные потоки вывода, стандартный поток вывода ошибок.

Если тестируются pandas-данные, то для проверки условий `assert` лучше использовать специальное ключевое слово `assert_frame_equal`. Аналогично для `numpy` и `matplotlib`

```
from pandas.testing import assert_frame_equal, assert_series_equal
from numpy.testing import assert_array_equal
from matplotlib.testing.exceptions import ImageComparisonFailure
```

Фикстуры передаются просто как аргументы (импортировать их не нужно).

Еще один простой пример фикстуры

```
import pytest
import psycopg2


@pytest.fixture # фикстура
def database():
    conn = psycopg2.connect("postgresql://postgres@localhost:5432/demo")
    return conn

@pytest.mark.DB  # <-- NB
def test_insert(database):
    cur = database.cursor()
    cur.execute("TABLE tickets LIMIT 5;")
    res = cur.fetchall()
    assert res[0][1] == "06B046"
```

Вызов

```
pytest -v test_true.py -m DB
```

Фикстура базы данных автоматически используется любым тестом, который имеет аргумент `database` в своем списке. Функция `test_insert()` получит результат функции `database()` в качестве первого аргумента и будет использовать этот результат по своему усмотрению. При таком использовании фикстуры не нужно повторять код инициализации базы данных несколько раз.

Еще одна распространенная особенность тестирования кода – это возможность удалять лишнее после работы фикстуры. Например, закрыть соединение с базой данных. Реализация фикстуры в качестве генератора добавит функциональность по очистке проверенных объектов

test_true.py

```
import psycopg2

@pytest.fixture
def get_cursor(): # функция-фикстура-генератор
    conn = psycopg2.connect("postgresql://postgres@localhost:5432/demo")
    cur = conn.cursor()
    yield cur    # отдать объект курсора
    cur.close()  # закрыть после теста объект курсора
    conn.close() # закрыть после теста объект соединения
```

```
@pytest.mark.DB
def test_fetch(get_cursor):
    # cur = database.cursor()
    get_cursor.execute("table tickets limit 5;")
    res = get_cursor.fetchall()
    assert res[0][1] == "06B046"
```

Вызов `pytest -v test_true.py -m DB`. Здесь код после утверждения `yield` выполнится только в конце теста.

Закрытие соединения с базой данных для каждого теста может вызывать неоправданные траты вычислительных мощностей, так как другие тесты могут использовать уже открытое соединение. В этом случае можно передать аргумент `scope` в декоратор фикстуры, *указывая область ее видимости*

```
import pytest

@pytest.fixture(scope="module") # <-- NB
def database():
    conn = psycopg2.connect("postgresql://postgres@localhost:5432/demo")
    cur = conn.cursor()
    yield cur
    cur.close()
    conn.close()

def test_fetch():
    ...
```

Указав параметр `scope="module"`, мы инициализировали фикстуру единожды для всего модуля, и теперь открытое соединение с базой данных будет доступно для всех тестовых функций, запрашивающих его.

11.7. Параметрические фикстуры и тестовые функции

Параметрические фикстуры запускаются несколько раз все тесты, где они используются единожды для каждого указанного параметра.

Запустим одного теста дважды, но с разными параметрами

```
import pytest
import psycopg2


@pytest.fixture(params=["mysql", "postgresql"])
def get_cursor(request):
    conn = psycopg2.connect(f"{request.param}://postgres@localhost:5432/demo")
    cur = conn.cursor()
    yield cur
    cur.close()
    conn.close()

@pytest.mark.DB
def test_fetch(get_cursor):
    get_cursor.execute("table tickets limit 5;")
    res = get_cursor.fetchall()
    assert res[0][1] == "06B046"
```

ВАЖНО: `request` – это специальное ключевое слово.

А есть еще *параметрические тестовые функции*, которые могут принимать набор имен переменных и набор значений для этих переменных

test_true.py

```
...
@pytest.mark.params
@pytest.mark.parametrize(
    [
        "alpha", "expected_multiplier" # параметры
    ],
    [
        (1.5, 192), (3, 255) # значения параметров (два расчетных случая)
    ]
)
def test_random_brightness(alpha, expected_multiplier):
    assert (1.5, 192) == (alpha, expected_multiplier)
...
```

Вызов `pytest -v test_true.py -m params`

11.8. Управляемые тесты с объектами-пустышками

Объекты-пустышки (или заглушки, mock objects) – это объекты, которые имитируют поведение реальных объектов приложения, но в особенном, управляемом состоянии. Они наиболее полезны в создании окружений, которые досконально описывают условия проведения теста. Можно заменить все объекты, кроме тестируемого, на объекты-пустышки и изолировать его, а также создать окружение для тестирования кода.

Один из случаев их использования – создание HTTP-клиента. Практически невозможно (или точнее, невероятно сложно) создать HTTP-сервер, на котором можно прогнать все варианты ситуаций и сценарии для каждого возможного значения. HTTP-клиенты особенно сложно тестировать на сценарии ошибок.

Начиная с версии Python 3.3 `mock` объединен с библиотекой `unittest.mock`. Поэтому можно использовать фрагмент кода, приведенный ниже, для обеспечения обратной совместимости между Python 3.3 и более ранними версиями

```
try:
    from unittest import mock
except ImportError:
    import mock
```

Библиотека `mock` очень проста в использовании. Любой атрибут, доступный для объекта `mock.Mock`, создается динамически во время выполнения программы. Такому атрибуту может быть присвоено любое значение.

Можно также динамически создавать *метод* для изменяемого объекта

```
import numpy as np
from unittest import mock

m = mock.Mock()
m.compute_effect.return_value = np.random.RandomState(42).randn()
m.compute_effect() # 0.4967141530112327
m.compute_effect("blah", "blah") # 0.4967141530112327
```

Объект `mock.Mock` теперь имеет метод `compute_effect()`, который возвращает псевдослучайное число. Он принимает любой тип аргумента, пока проверка того, что это за аргумент, отсутствует.

Библиотека `mock` также может быть использована для замены функции, метода или объекта из внешнего модуля

```
from unittest import mock
import os

def fake_os_unlink(path):
    raise IOError("Testing")

with mock.patch("os.unlink", fake_os_unlink): # заменяте функцию os.unlink в нижеприведенной стр
    оке на fake_os_unlink
    os.unlink("foobar") # то есть будет fake_os_unlink("foobar")
```

С методом `mock.patch()` можно изменить любую часть внешнего кода, заставив его вести себя так, чтобы протестировать все условия приложения

mock_sample.py

```
import pytest
import requests
from unittest import mock


class WhereIsPythonError(Exception):
    pass


def is_python_still_a_programming_language():
    try:
        r = requests.get("http://python.org")
    except IOError:
        pass
    else:
        if r.status_code == 200:
            return "Python is a programming language" in r.content
    raise WhereIsPythonError("Somthing bad happened")


def get_fake_get(status_code, content):
    m = mock.Mock()  # создаем объект-заглушку
    m.status_code = status_code  # создаем атрибут
    m.content = content          # создаем атрибут

    def fake_get(url):
        return m

    return fake_get


def raise_get(url):
    raise IOError(f"Unable to fetch url {url}")


@mock.patch("requests.get", get_fake_get(
    200, "Python is a programming language for sure"))
def test_python_is():
```

```
    assert is_python_still_a_programming_language() is True


@mock.patch("requests.get", get_fake_get(
    200, "Python is no more a programming language"))
def test_python_is_not():
    assert is_python_still_a_programming_language() is False


@mock.patch("requests.get", get_fake_get(404, "Whatever"))
def test_bad_status_code():
    with pytest.raises(WhereIsPythonError):
        is_python_still_a_programming_language()


@mock.patch("requests.get", raise_get)
def test_ioerror():
    with pytest.raises(WhereIsPythonError):
        is_python_still_a_programming_language()
```

Этот листинг реализует тестовый случай, который ищет все экземпляры строки «*Python is a programming language*» на сайте `http://python.org`. Не существует варианта, при котором тест не найдет ни одной заданной строки на выбранной веб-странице. Чтобы получить отрицательный результат, необходимо изменить страницу, а этого сделать нельзя. Но с помощью `mock` можно пойти на хитрость и изменить поведение запроса так, чтобы он возвращал ответ-пустышку с выдуманной страницей, не содержащей заданной строки. Это позволит протестировать отрицательный сценарий, в котором `http://python.org` не содержит заданной строки, и убедиться, что программа обрабатывает такой случай корректно.

Чуть подробнее об этом примере. Когда мы в командной строке запускаем `pytest -v mock_sample.py`, происходит следующее: вызывается функция `test_python_is()`, «разворачивается» код функции `is_python_still_a_programming_language()` как строка и `mock.patch` в этой функции замещает подстроку `requests.get` ссылкой на функцию `fake_get`, вызов которой вернет объект-заглушку с атрибутами `status_code` и `content`; поскольку в данном случае `status_code` имеет значение 200, а `content` – «*Python is a programming language for sure*», условие выполняется и функция `is_python_still_a_programming_language()` возвращает `True` (т.е. тест пройден).

Далее вызывается функция `test_python_is_not()`. Снова код функции `is_python_still_a_programming_language()` разворачивается как строка, в которой `mock.patch` замещает `requests.get` ссылкой на `get_fake`, которая возвращает объект-заглушку с атрибутами `status_code=200` и `content` «*Python is no more a programming language*».
Функция `is_python_still_a_programming_language()` возвращает `False`. Условие выполняется, тест пройден.

В случае функции `test_bad_status_code()` возбуждается исключение `WhereIsPythonError`, которое обрабатывается соответствующим менеджером контекста. Тест пройден.

И, наконец, четвертая тестовая функция `test_ioerror()`. Здесь, как и раньше, код функции `is_python_still_a_programming_language()` читается в строку и `mock.patch` замещает подстроку `requests.get` ссылкой на функцию `raise_get`, котрая возбуждает исключение `IOError`. Когда это происходит в функции `is_python_still_a_programming_language()` возбуждается исключение `WhereIsPythonError`, которое перехватывается менеджером контекста. Тест пройден.

ВАЖНО: декоратор `@mock.patch("f1", f2)` заменяет одну функцию, представленную в виде строки, ссылкой на другую функцию, рассматривая код задекорированной функции как строку.

11.9. Выявление непротестированного кода с помощью `coverage`

Отличным дополнением для модульного тестирования является инструмент `coverage`, который находит непротестированные части кода. Он использует инструменты анализа и отслеживания кода для выявления тех строк, которые не были выполнены. В модульном тестировании он может выявить, какие части кода были задействованы многократно, а какие вообще не использовались.

Покрытие кода является показателем того, какой процент тестируемого кода тестируется (покрывается) набором тестов.

Инструменты покрытия кода отлично подходят для того, чтобы сообщить, какие части системы полностью пропущены тестами.

Инструмент оценки покрытия кода тестами `coverage` удобнее всего вызывать из-под `pytest`. Поскольку `coverage` является одной из зависимостей `pytest-cov`, достаточно установить `pytest-cov` и он притянет за собой `coverage.py`.

Опция `--cov pytest` включает вывод отчета `coverage` в конце тестирования. Необходимо передать имя пакета в качестве аргумента, чтобы плагин должным образом отфильтровал отчет. Вывод будет содержать стоки кода, которые не были выполнены, а значит, не тестировались. Все, что останется, – открыть редактор и написать тест для этого кода

```
pytest -v --cov=.
```

Еще можно добавить к `pytest` флаг `--cov-report=html`. Тогда в директории, из-под которой запускается команда

```
pytest -v --cov=. --cov-report=html
```

будет создана директория `htmlcov`, содержащая html-страницы. Каждая страница покажет, какие части исходного кода были или не были запущены.

Команда `pytest --cov=src` (при условии, что тестируемый код находится в `src`) создает отчет о покрытии только для указанной директории.

Пример вывода

```
$ cd /path/to/code/ch7/tasks_proj_v2
$ pytest --cov=src

===================== test session starts ======================

plugins: mock-1.6.2, cov-2.5.1
collected 62 items
tests/func/test_add.py ...
tests/func/test_add_variety.py ............................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .........
tests/func/test_unique_id.py .
tests/unit/test_cli.py .....
tests/unit/test_task.py ....

---------- coverage: platform darwin, python 3.6.2-final-0 -----------

Name                          Stmts   Miss  Cover
```

```
----------------------------------------------
src\tasks\__init__.py                2      0   100%
src\tasks\api.py                    79     22    72%
src\tasks\cli.py                    45     14    69%
src\tasks\config.py                 18     12    33%
src\tasks\tasksdb_pymongo.py        74     74     0%
src\tasks\tasksdb_tinydb.py         32      4    88%
----------------------------------------------
TOTAL                              250    126    50%

================== 62 passed in 0.47 seconds ==================
```

В этом примере некоторые файлы имеют довольно низкий процент покрытия. Лучший способ посмотреть чего не хватает для полного покрытия тестами это html-отчеты (см. рис. 1)

```
pytest --cov=src --cov-report=html
```

```
31      def list_tasks(self, owner=None):  # type (str) -> list[dict]
32          """Return list of tasks."""
33          if owner is None:
34              return self._db.all()
35          else:
36              return self._db.search(tinydb.Query().owner == owner)
37
38      def count(self):  # type () -> int
39          """Return number of tasks in db."""
40          return len(self._db)
41
42      def update(self, task_id, task):  # type (int, dict) -> ()
43          """Modify task in db with given task_id."""
44          self._db.update(task, eids=[task_id])
45
46      def delete(self, task_id):  # type (int) -> ()
47          """Remove a task from db with given task_id."""
48          self._db.remove(eids=[task_id])
49
50      def delete_all(self):
51          """Remove all tasks from db."""
52          self._db.purge()
```

Рис. 1. Страница html-отчета о покрытии кода тестами

Из рис. 1 можно заключить, чт:

- функция `list_tasks` не тестируется для случая, когда задано имя владельца (эта строка отмечена красным),
- не тестируются функции `update` и `delete`.

Теперь можно эти функции включить в список TO-DO по тестированию вместе с тестированием системы конфигурации.

Хотя большой процент покрытия – это хорошая цель, а инструменты тестирования полезны для получения информации о состоянии тестового покрытия, сама по себе величина процента не особо информативна.

Например, покрытие кода тестами на 100% – достойная цель, но это не обязательно означает, что код тестируется полностью. Эта величина лишь показывает, что все строки кода в программе выполнены, но не сообщает, что были протестированы все условия.

Кроме того, когда есть, который не тестируется, это может означать. что необходим тест. Но это также может означать, что есть некотрые функции системы, которые не нужны и могут быть удалены.

Стоит использовать информацию о покрытии с целью расширения набора тестов и создания их для кода, который не запускается. Это упрощает поддержку проекта и повышает общее качество кода.

11.10. Виртуальные окружения

Одно из главных применений виртуального окружения – обеспечение чистого окружения для запуска модульных тестов.

Для тестирования приложений в различных средах удобно использовать `tox`. `tox` – утилита командной строки, которая позволяет запускать полный набор тестов в нескольких средах (например, с различными версиями Python, или различными конфигурациями для различных операционных систем).

В общих чертах `tox` работает так

1. Создает виртуальную среду в каталоге `.tox`,
2. Устанавливает некоторые зависимости,
3. Устанавливает пакет из `sdist`,
4. Запускает тесты,
5. Создает отчет с результатами.

Для того чтобы включить в проект поддержку `tox`, нужно подготовить файл `tox.ini` на том же уровне, что и `setup.py`

```
project/
  |-- setup.py
  |-- tox.ini
  ...
```

Конфигурационный файл `tox.ini` может выглядеть так

tox.ini

```
[tox]
envlist = py27,py36

[testenv]
deps=pytest   # если pytest нет в системе, то tox его установит
commands=pytest # запустит pytest внутри тестового окружения

[pytest]
addopts = -rsxX -l --tb=short --strict
markers =
    smoke: Run the smoke test functions
    get: Run the test functions that test tasks.get()
```

Если запустить код сейчас, `tox` создаст окружение, установит новую зависимость и запустит команду `pytest`, котрая выполнит все модульные тесты. Для добавления новых зависимостей можно либо добавить их в опцию `deps` конфигурации, как в примере, либо воспользоваться `-rfile` для чтения из файла.

Здесь строка `envlist = py27,py36` позволяет запускать тесты с использованием Python 2.7 и Python 3.6.

Строка `deps=pytest` заставляет `tox` проверить установлен ли `pytest`. Строка `commands=pytest` говорит `tox`, что нужно запускать `pytest` в каждой среде.

Строка `addopts = -rsxX` включает дополнительную сводную информацию для пропусков, а `-l` включает отображение локальных переменных в трассировке стека. Он также по умолчанию использует сокращенные трассировки стека (`--tb=short`) и гарантирует, что все маркеры, используемые в тестах, будут объявлены первыми (`--strict`).

Можно запустить тесты и для другой версии Python, если передать метку `-e` в `tox`, например

```
tox -e py26
```

По умолчанию `tox` имитирует любое окружение, которое совпадает со следующими версиями Python: `py24`, `py25`, `py26`, `py27`, `py30`, `py31`, `py32`, `py33`, `py34`, `py35`, `py36`, `py37`, `jython` и `pypy`.

Можно определить свои собственные окружения. Для этого достаточно добавить секцию с именем `[testenv:envname]`

```
[testenv]
deps=pytest
commands=pytest

[testenv:py36-coverage]
deps=
    {[testenv]deps} # значение переменной deps из раздела [testenv]
    pytest-cov
commands=pytest --cov=myproject
```

Используя `pytest --cov=myproject` на секции `py36-coverage`, как показано в примере, мы переопределили команды для окружения `py36-coverage`. Когда мы запустим `tox -e py36-coverage`, то установим `pytest` как одну из зависимостей, а сама команда `pytest` запуститься с опцией `coverage`.

Здесь мы заменили значение `deps` на аналогичное значение из `testenv` и добавили зависимость с `pytest-cov`.

В `tox` поддерживается интерполяция переменных, поэтому можно обращаться к любому полю из файла `tox.ini` и использовать его как переменную с помощью синтаксиса

```
{[env_name]variable_name}
```

12. Автоматическое тестирование в Python

Для автоматического тестирования удобно использовать библиотеку `Hypothesis` https://hypothesis.readthedocs.io/en/latest/. Подходит для граничного тестирования

Hypothesis:

- Работает перебором, поиск минимального примера,
- "Знает"об особенностях данных в Python,
- Поддерживает map,
- Подходит для тестирования кода, который работает с пользователем напрямую.

Можно тестировать модели.

13. Инструменты автоматического форматирования, инспектирования и анализа кода

Список обсуждаемых инструментов:

- Deepsource, Deepcode, Codacy,
- flake8,
- black,
- pre-commit,

Выполнить автоматический анализ кода можно с помощью следующих инструментов на базе AI:

- Deepsource https://deepsource.io/: нужно просто зарегистрироваться, например, через git-репозиторий, а затем подключить свой репозиторий для анализа; после того, как будет сделан коммит, запуститься процедура анализа на платформе DeepSource (результаты); в репозитории будет создан специальный конфигурационный файл `.deepsource.toml`,
- Deepcode https://www.deepcode.ai/,
- Codacy https://www.codacy.com/

Пример конфигурационного файла для автоматического анализа кода на базе Deepsource

.deepsource.toml

```
version = 1

test_patterns = [
  'tests/**'
]

exclude_patterns = [

]

[[analyzers]]
name = "python"
enabled = true
runtime_version = "3.x.x"

  [analyzers.meta]
  max_line_length = 79
```

Наиболее общий инструмент инспектирования кода (так называемые линтеры) – `flake8`. Flake8 умеет работать не только с PEP8, но и с другими правилами (кроме того поддерживате пользовательские плагины).

Для автоматического форматирования кода из командной строки или как pre-commit hook удобно использовать `black` https://black.readthedocs.io/en/stable/installation_and_usage.html.

Инструмент `black` поддерживается `vim` https://black.readthedocs.io/en/stable/editor_integration.html#vim.

Проще всего установить `black` в Vim с помощью менеджера плагинов `Vundle`.

Сначала нужно прописать в конфигурационном файле `~/.vimrc` следующую строку

/.vimrc

```
Plugin 'psf/black'
```

А затем в сеансе Vim нужно набрать `:PluginInstall`.

Если возникнет ошибка конца строки «E492: Not an editor command: `^M`», то в директории плагина (например, `Users‣leor.finkelberg‣.vim‣bundle‣black`) можно воспользоваться конструкцией

```
# переконвертировать все vim-файлы в unix-формат
find . -name '*.vim' | xargs dos2unix -f
```

Утилита командной строки (и pre-commit hook) `yesqa` https://pypi.org/project/yesqa/ используется для автоматического удаления ненужных комментариев вида `# noqa`.

Для того чтобы перед коммитом `yesqa` проверяла код требуется включить в конфигурационный файл `.pre-commit-config.yaml` следующие строки

.pre-commit-config.yaml

```
...
-   repo: https://github.com/asottile/yesqa
    rev: v1.2.2
    hooks:
-   id: yesqa
...
```

Вот сложный пример `.pre-commit-config.yaml`

.pre-commit-config.yaml

```
exclude: _pb2\.py$
repos:
- repo: https://github.com/pre-commit/mirrors-isort
rev: f0001b2  # Use the revision sha / tag you want to point at
hooks:
- id: isort
args: ["--profile", "black"]
- repo: https://github.com/psf/black
rev: 20.8b1
hooks:
- id: black
- repo: https://github.com/asottile/yesqa
rev: v1.1.0
hooks:
- id: yesqa
additional_dependencies:
- flake8-bugbear==20.1.4
- flake8-builtins==1.5.2
- flake8-comprehensions==3.2.2
- flake8-tidy-imports==4.1.0
- flake8==3.7.9
-   repo: https://github.com/pre-commit/pre-commit-hooks
rev: v2.3.0
hooks:
- id: check-docstring-first
- id: check-json
- id: check-merge-conflict
- id: check-yaml
- id: debug-statements
- id: end-of-file-fixer
- id: trailing-whitespace
- id: flake8
- id: requirements-txt-fixer
-   repo: https://github.com/pre-commit/mirrors-pylint
rev: d230ffd
hooks:
-   id: pylint
args:
```

```
- --max-line-length=119
- --ignore-imports=yes
- -d duplicate-code
- repo: https://github.com/asottile/pyupgrade
rev: v2.7.3
hooks:
- id: pyupgrade
args: ['--py37-plus']
- repo: https://github.com/pre-commit/pygrep-hooks
rev: v1.5.1
hooks:
- id: python-check-mock-methods
- id: python-use-type-annotations
- repo: https://github.com/pre-commit/mirrors-mypy
rev: 9feadeb
hooks:
- id: mypy
args: [--ignore-missing-imports, --warn-no-return, --warn-redundant-casts, --disallow-incomplete
    -defs]
```

Перед коммитом полезно выполнять команду

```
pre-commit run --all-files
```

ВАЖНО: если виртуальное окружение конструируется с помощью `conda` может возникнуть ошибка «FileNotFoundError: [Errno 2] No such file or directory...»
при запуске `pre-commit run --all-files`.

Поэтому следует использовать `virtualenv` версии 20.0.33 (!)

```
$ pip install virtualenv==20.0.33
$ virtualenv env
$ source env/bin/actiavte # для Linux/MacOs etc
$ env\Scripts\activate.bat # для Windows
```

14. Тонкости импортирования модулей и пакетов в Python

ОЧЕНЬ ВАЖНО: при импорте *модуля* Python выполняет *весь* код в этом модуле

```
import something_module  # весь код модуля выполнился
```

ОЧЕНЬ ВАЖНО: при импорте *пакета* Python выполняет модуль `__init__.py` этого пакета и импортированные имена становятся частью пространства имен пакета

```
import something_package  # выполнился модуль __init__.py этого пакета
```

Пример. Пусть структура пакетов выглядит так

```
root_dir/
|-- packageA/
    |-- __init__.py  # from packageA import moduleA
    |                # from packageA import packageB
    |                # from packageA import foo
    |-- moduleA.py   # здесь объявлена функция mainA()
    |-- packageB/
    |   |-- __init__.py  # from packageA.packageB import moduleB
    |   |-- moduleB.py   # здесь объявлена функция mainB()
    |-- foo.py
```

То есть в модуле `__init__.py` пакета `packageA` лежит такой код

packageA/__init__.py

```
# используются абсолютные пути от корня директории, из-под которой запускается
# интерактивная Python-сессия или базовый Python-сценарий
from packageA import moduleA
from packageA import packageB
from packageA import foo
```

Другими словами, при импорте пакета `packageA` будет выполнен модуль `__init__.py`, который добавит в пространство имен пакета имена `moduleA`, `packageB` и `foo`. Это означает, что эти модули и пакет будут доступны через обращение к пакету `packageA`

```
import packageA

packageA.foo
packageA.moduleA
packageA.packageB
```

Убедится в том, что при запуске главного сценария из-под корневой директории будут доступны для импорта через имя пакета `packageA` имена `moduleA`, `packageB` и `foo` можно так

вызов из-под корневой директории

```
import pkgutil

list(pkgutil.iter_modules(["./packageA"]))
# [ModuleInfo(module_finder=FileFinder('./packageA'), name='foo', ispkg=False),
# ModuleInfo(module_finder=FileFinder('./packageA'), name='moduleA', ispkg=False),
# ModuleInfo(module_finder=FileFinder('./packageA'), name='packageB', ispkg=True)]
```

Функцию `pkgutil.iter_modules(path=search_path)` можно использовать, чтобы получить список всех модулей/пакетов, которые можно импортировать по заданному пути.

При импорте пакета `packageB` аналогично будет выполнен модуль `__init__.py` этого пакета

packageA/packageB/__init__.py

```
# используются абсолютные пути от корня директории, из-под которой запускается
# интерактивная Python-сессия или базовый Python-сценарий
from packageA.packageB import moduleB
```

Таким образом, модуль `moduleB` становится частью пространства имен пакета `packageB`. То есть при импорте пакета `packageB` к модулю `moduleB` можно будет обраться так

```
import pkgutil

import packageA.packageB as pkB

pkB.moduleB
list(pkgutil.iter_modules(["./packageA/packageB"]))
# [ModuleInfo(module_finder=FileFinder('./packageA/packageB'), name='moduleB', ispkg=False)]
```

Первым элементом списка путей `sys.path` будет директория, в которой лежит выполняемый Python-сценарий.

Python ищет модули в следующем порядке

- Модули стандартной библиотеки (например, `math`, `os` etc.),
- Модули/пакеты, указанные в `sys.path`:

  - Если интерпретатор запущен в интерактивном режиме:
    * `sys.path[0]` – пустая строка, `""`. Это значит, что Python будет искать в текущей директории, из-под которой был запущен интерпретатор,
  - Если сценарий был запущен командой `python <script.py>`:
    * `sys.path[0]` – это путь к `<script.py>`.
- Директории, указанные в переменной среды `PYTHONPATH`,
- Директория по умолчанию, которая зависит от дистрибутива Python.

Замечание

При запуске сценария для `sys.path` важна не директория, из-под которой запускается сценарий, а *путь к самому сценарию*

У модуля `__init__.py` есть две функции:

- Превратить папку со скриптами в импортируемый пакет модулей (до версии Python 3.3, в версиях Python 3.3+ в этом нет необходимости, так как все директории по умолчанию считаются пакетами),
- Выполнить код инициализации пакета.

Чтобы импортировать модуль/пакет из директории, которая находится не в директории запущенного сценария, этот модуль/пакет должен быть в пакете.

В момент, когда пакет или один из его модулей импортируется в первый раз, Python выполняет `__init__.py` в корне пакета, если такой файл существует. Все объекты, определенные в `__init__.py`, считаются частью пространства имен пакета.

Замечание

Если Python-сценарий запускается из-под пакета (директории, содержащей или не содержащей модуля `__init__.py`), то модуль `__init__.py` этого пакета не вызывается, так как директория, в которой находится сценарий *пакетом не считается*

При *абсолютном* импорте используется полный путь от начала корневой папки проекта к нужному модулю.

При *относительном* импорте используется относительный путь, начиная с местоположения текущего модуля и заканчивая местоположением интересующего модуля.

Замечание

Как правило, рекомендуется использовать абсолютный импорт

---

15. Логистическая функция потерь

Логистическую функцию потерь (logloss) еще называют *перекрестной энтропией* и часто используют в задачах классификации.

Функция ошибки называется *скоринговой* (proper scoring rules), если

$$p = \arg\min \mathbf{E}_y \, L(y, a), \;\; y \sim Bernoulli(p),$$

то есть оптимальный ответ на каждом объекте – его вероятность принадлежности к классу 1. Например, логистическая функция потерь является скоринговой. Сама теория скоринговых

функций является самостоятельным направлением в теории вероятностей и математической статистики.

Функцию ошибки MSE в задачах классификации называют «ошибкой Брайера» (Brier score), именно под таким названием она реализована в `scikit-learn`

```
from sklearn.metrics import brier_score_loss
brier_score_loss(y_true, y_prob)
```

16. Автоматический анализ кода с платформой DeepsSource

Платформа DeepSource https://deepsource.io/ позволяет проводить автоматический анализ кода на анти-паттерны, проблемы с производительностью и поиск уязвимостей при каждом коммите или pull-request. Кроме того платформа отслеживает количество зависимостей, покрытие документацией и пр.

С документацией можно ознакомиться здесь https://deepsource.io/docs/.

Управлять поведением DeepSource можно с помощью конфигурационного файла `.deepsource.toml`

.deepsource.toml

```
version = 1

test_patterns = [
    "tests/**",
    "test_*.py"
]

exclude_patterns = [
    "migrations/**",
    "**/examples/**"
]

[[analyzers]]
name = "python"  # анализатор для Python
enabled = true
dependency_file_paths = ["requirements/development.txt"] # список внешних зависимостей

    [analyzers.meta]  # метаданные для анализатора
    runtime_version = "3.x.x" # версия языка
    type_checker = "mypy" # анализатор проверки типов
    max_line_length = 79  # максимально допустимая длина строки
    skip_doc_coverage = ["module", "magic", "init"] # артифакты, которые следует пропустить при
    расчете покрытия документации
    additional_builtins = ["_", "pretty_output"] # дополнительные модули

[[analyzers]]
name = "shell"  # анализатор для сценариев командной оболочки
enabled = true

    [analyzers.meta]
    dialect = "zsh"

[[analyzers]]  # анализатор для SQL
name = "sql"
enabled = true
```

```
    [analyzers.meta]
    max_line_length = 100
    tab_space_size = 4
    indent_unit = "tab"
    comma_style = "trailing"
    capitalisation_policy = "consistent"
    allow_scalar = true
    single_table_references = "consistent"
```

По умолчанию, если в репозитории обнаруживаются ниже перечисленные файлы, то они проверяются на наличие зависимостей

- `Pipfile`,
- `Pipfile.lock`,
- `poetry.lock`,
- `pyproject.toml` (если содержит разделы `[tool.poetry]` или `[tool.fit]`),
- `requirements.txt`,
- `setup.py`.

Файл `.deepsource.toml` располагается в корне проекта

```
.
|-- .deepsource.toml
|-- README.md
|-- bar
|   |-- baz.py
|-- foo.py
```

Вот исчерпывающий список возможных вариантов конфигурации

- `version`: обязательное свойство; на текущий момент поддерживается только значение «1»

```
version = 1
```

- `exclude_patterns`: список шаблонов файлов и директорий, которые не должны учитываться при выполнении анализа; шаблоны строятся относительно корня проекта

```
exclude_patterns = [
    "bin/**",
    "**/node_modules/",
    "js/*/*/min.js"
]
```

- `test_patterns`: список шаблонов файлов и директорий, содержащих тестовые файлы; шаблоны строятся относительно корня проекта,

```
test_patterns = [
    "tests/**",
    "test_*.py"
]
```

- `analyzers`: список анализаторов (в конфигурации должен быть хотя бы один анализатор),

```
[[analyzers]]
name = "python"
enabled = true
dependency_file_paths = [
    "requirements.txt",
    "Pipfile"
]
```

- `transformers`: список трансформеров

```
[[transformers]]
name = "black"
enabled = true
```

16.1. Связка DeepSource и Travis CI

Для того чтобы платформа DeepSource имела возможность извлекать метрики покрытия кода тестами из Travis CI https://www.travis-ci.com/ следует в конфигурационный файл `.deepsource.toml` добавить анализатор «Test Coverage» (см. https://deepsource.io/docs/how-to/add-python-cov-ci.html)

.deepsource.toml

```
...
[[analyzers]]
name = "test-coverage"
enabled = true
...
```

а в конфигурационный файл `.travis.yml` следующие строки

.travis.yml

```
# Travis CI config
...
after_success:  # this is for DeepSource.io
  # Generate coverage report in xml format
  - coverage xml

  # Install deepsource CLI
  - curl https://deepsource.io/cli | sh

  # From the root directory, run the report coverage command
  - ./bin/deepsource report --analyzer test-coverage --key python --value-file ./coverage.xml
```

Затем на странице Travis CI, на вкладке Settings, например, https://www.travis-ci.com/github/LeorFinkelberg/termostablizator/settings, необходимо создать переменную окружения `DEEPSOURCE_DSN` со значением Data Source Name (DNS), которое можно узнать на странице DeepSource по пути `Settings‣Reporting`.

Получится что-то вроде

```
DEEPSOURCE\_DSN=https://0a69347b13674f13a8ca39b9464cb682@deepsource.io
```

17. Python и LaTeX

Для компиляции LaTeX-документов прямо из-под `Python` можно использовать библиотеку `pylatex`[2] https://jeltef.github.io/PyLaTeX/current/.

[2] Устанавливается как обычно `pip install pylatex`

18. Градиентный бустинг

18.1. Общие сведения

18.2. Особенности реализации в пакете `sklearn`

18.3. Особенности реализации в пакете `XGBoost`

Алгоритм экстремального градиентного бустинга XGBoost добавляет больше масштабируемых методов, которые задействуют многопаточность на одиночной машине и параллельную обработку на кластерах из многочисленных серверов (используя сегментирование).

Самое важное усовершенствование, вносимое алгоритмом XGBoost, по сравнению с градиентным бустингом, состоит в возможности первого управлять разреженными данными. Алгоритм XGBoost принимает разреженные данные автоматически, не храня нулевых значений в памяти. Второе преимущество XGBoost заключается в том, каким образом вычисляются значения наилучшего расщепления узлов при ветвлении дерева, при этом используется метод, который называется квантильной схемой. Этот метод преобразует данные алгоритмов взвешивания, в результате которого потенциальные расщепления сортируются на основе определенного уровня точности.

Экстремальный градиентый бустинг XGBoost является по-настоящему масштабируемым решением с различных точек зрения. XGBoost – это новое поколение алгоритмов градиентного бустинга с серьезной доводкой исходного алгоритма бустинга деревьев. Алгоритм XGBoost обеспечивает параллельную обработку. Предлагаемая алгоритмом масштабируемость реализуется благодаря доработанным авторами несколькими параметрическими настройками и добавлениями:

- алгоритм принимает *разреженные данные*, в которых могут задействоваться разреженные матрицы, экономя оперативную память (отсутствует потребность в плотных матрицах) и продолжительность вычисления (нулевые значения обрабатываются особым образом),
- обучение приближенному дереву (взвешенные метод квантильной схемы), которое показывает аналогичные результаты, но за гораздо меньшее время, чем классический исчерпывающий просмотр возможных точек ветвления,
- *параллельные вычисления* на одиночной машине (используя многопоточность в фазе поиска лучшего расщепления) и аналогичным образом *распределенные вычисления* на нескольких машинах,
- *внеядерные вычисления на одиночной машине* с привлечением решения для хранения данных под названием «постолбцовый блок», которое располагает данные на диске столбцами, тем самым экономя вермя – данные с диска поступают в том виде, в котором их ожидает алгоритм оптимизации (который оперирует векторами-столбцами).

XGBoost довольно неплохо обрабатывает *пропущенные данные*. Другие древовидные ансамбли, основанные на стандартных деревьях решений, требуют сначала *импутировать*[3] пропущенные данные, используя внешкальные значения (в частности, *большое отрицательное число*, например, -999999), чтобы выработать надлежащее ветвление дерева в случае пропущенных значений.

[3]Импутация – процесс замещения пропущенных, некорректных или несостоятельных значений другими значениями

В отличие от них, алгоритм XGBoost сначала выполняет подгонку всех непропущенных значений и после создания ветвления для переменной затем решает, какая ветвь лучше всего подходит для пропущенных значений с целью уменьшения ошибки прогнозирования. Такой подход приводит к более компактным деревьям, а эффективная стратегия импутации – к большей прогнозирующей способности.

Самые важные параметры алгоритма XGBoost:

- `learning_rate`: скорость (темп) обучения,
- `min_child_weight`: более высокие значения предотвращают переподгонку и вычислительную сложность,
- `max_depth`: максимальная глубина дерева базовых учеников,
- `subsample`: доля подвыборок обучающих экземпляров, которые берутся на каждой итерации,
- `colsample_bytree`: доля признаков, которые используются при построении каждого дерева,
- `reg_lambda`: L_2-регуляризация.

Алгоритм экстремального градиентного бустинга XGBoost по умолчанию параллелизирует алгоритм по всем доступным ядрами. С помощью библиотеки `joblib` можно сохранять натренированные модели и затем использовать их для прогноза

```
import joblib
import xgboost as xgb
from sklearn.model_selection import GridSearchCV

...

params = {
    'max_depth' : [4, 6, 8],
    'n_estimators' : [100],
    'min_child_weight' : range(1, 3),
    'learning_rate' : [0.1, 0.01, 0.001],
    'colsample_bytree' : [0.8, 0.9, 1.0],
    'gamma' : [0, 1]
}

xgbr = xgb.XGBRegressor(gamma=0, objective='reg:squarederror', n_jobs=-1)
gscv = GridSearchCV(
            estimator=gscv,
            param_grid=params,
            n_jobs=-1,
            scoring='neg_mean_absolute_error',
            verbose=True
)
gscv.fit(X_train, y_train)
y_pred = gscv.predict(X_test)
joblib.dump(gscv.best_estimator_, 'grid_search_cv_best.pkl')  # в текущей директории появится
    pkl-файл
```

18.3.1. Установка пакета `xgboost` на `Windows`

Устанавливать пакет `xgboost` рекомендуется с помощью следующей команды

```
conda install -c anaconda py-xgboost
```

Существует альтернативный способ установки пакета `xgboost` (разумеется он работает и для других пакетов). Для начала требуется вывести список доступных каналов (см. рис. 2), по кото-

рым будет проводиться поиск интересующего пакета (в данном случае пакета `xgboost`), а затем можно воспользоваться конструкцией

```
anaconda search -t conda xgboost
```

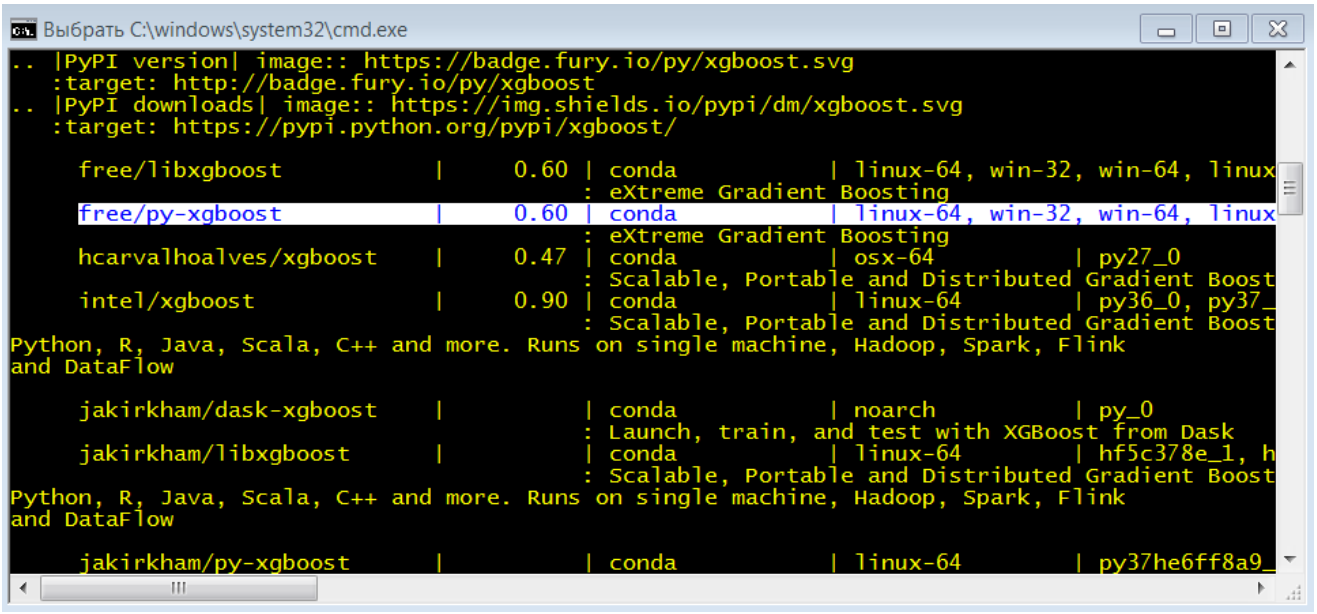


Рис. 2. Окно командной оболочки `cmd.exe` со списком доступных каналов, по которым будет проводиться поиск пакета `xgboost`

После, выбрав канал, можно приступать к установке пакета

```
conda install -c free py-xgboost
```

18.3.2. Простой пример работы с `xgboost` и `shap`

Решается задача бинарной классификации. Требуется построить модель, предсказывающую годовой доход заявителя по порогу $50'000 (то есть больше или меньше $50'000 зарабатывает заявитель в год). Используется набор данных UCI Adult income

```
import xgboost
import shap  # для оценки важности признаков вычисляются значения Шепли (Shapley value)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

shap.initjs()

X, y = shap.datasets.adult()
X_display, y_display = shap.datasets.adult(display=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=7)
d_train = xgboost.DMatrix(X_train, label=y_train)
d_test = xgboost.DMatrix(X_test, label=y_test)

params = {
    'eta' : 0.01,
    'objective' : 'binary:logistic',
    'subsample' : 0.5,
    'base_score' : np.mean(y_train),
```

```
    'eval_metric' : 'logloss'
}
model = xgboost.train(params, d_train,
                      num_boost_round = 5000,  # число итераций бустинга
                      evals = [(d_test, 'test')],
                      verbose_eval=100,  # выводит результат на каждой 100-ой итерации бустинга
                      early_stopping_rounds=20)

xgboost.plot_importance(model)
```

На рис. 3, рис. 4 и рис. 5 изображены графики важности признаков.

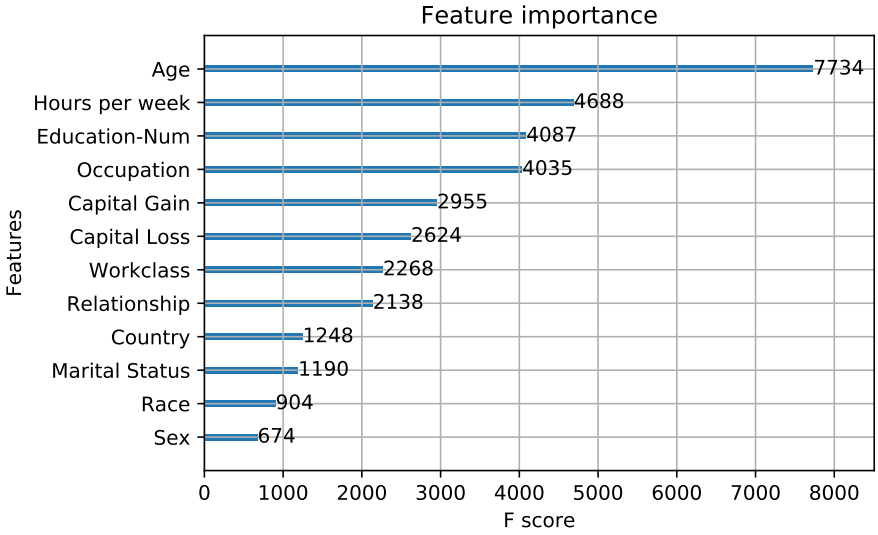


Рис. 3. График важности признаков `xgboost.plot_importance(model)`, построенный с помощью пакета `xgboost`

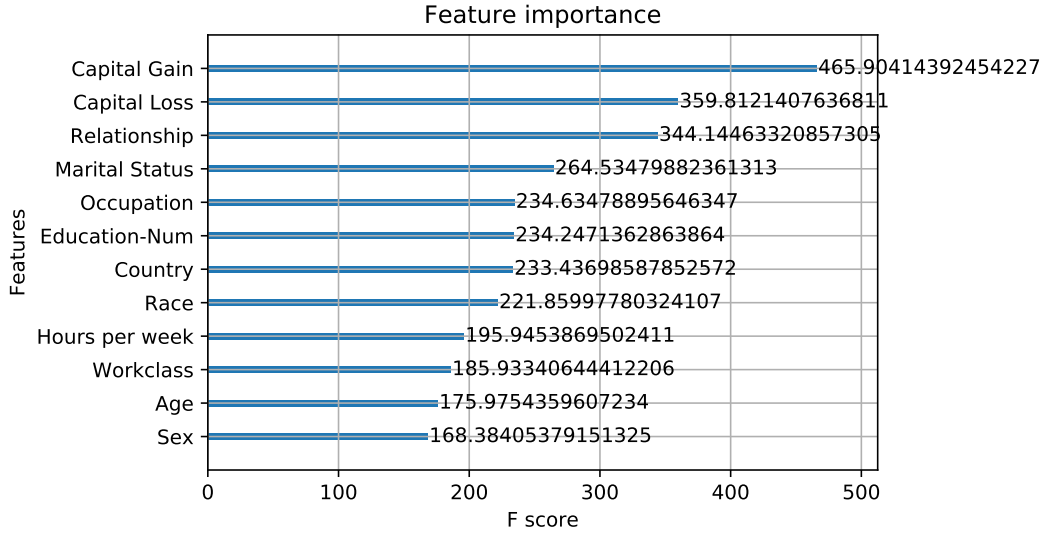


Рис. 4. График важности признаков `xgboost.plot_importance(model, importance_type='cover')`, построенный с помощью пакета `xgboost`

Следует иметь в виду, что в библиотеке `xgboost` поддерживается три варианта вычисления важности признаков (см. Interpretable Machine Learning with XGBoost):

- `weight`: общее число сценариев по всем деревьям, когда i-ый признак используется для расщепления обучающего набора данных,

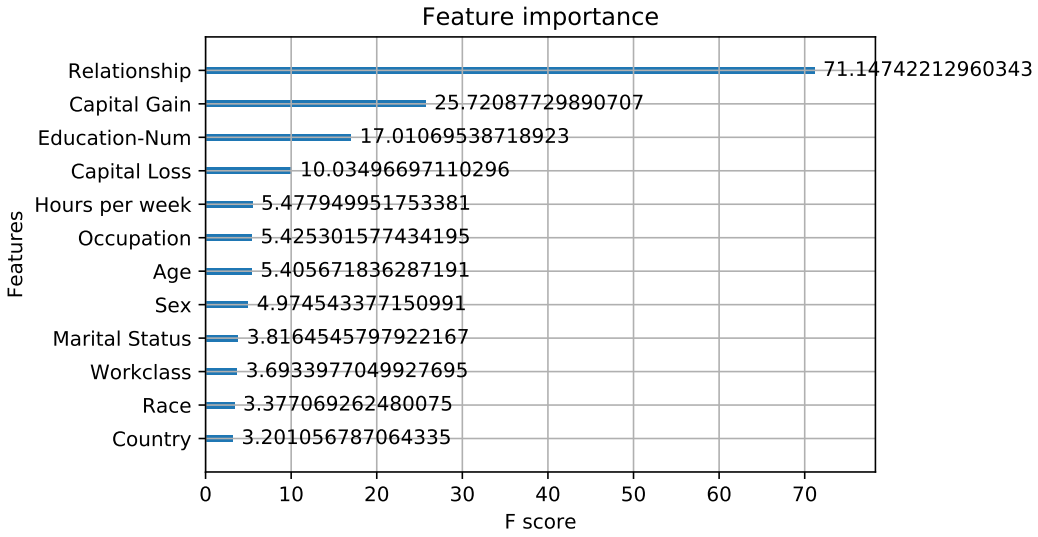


Рис. 5. График важности признаков `xgboost.plot_importance(model, importance_type='gain')`, построенный с помощью пакета `xgboost`

- `cover`: общее число сценариев по всем деревьям, когда i-ый признак используется для расщепления набора данных, взвешенное по числу точек обучающего набора данных, которые проходят через эти расщепления,
- `gain`: среднее снижение потерь на обучающем наборе данных, полученное при использовании i-ого признака.

Еще один простой типовой пример использования библиотеки `xgboost`

```
import xgboost
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import mean_squared_error

boston = load_boston()
X, y = boston.data, boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

xgbr = xgboost.XGBRegressor(verbosity=0)
xgbr.fit(X_train, y_train)

score = xgbr.score(X_train, y_train)
scores = cross_val_score(xgbr, X_train, y_train, cv=5)
kfold = KFold(n_splits10, shuffle=True)
kf_cv_scores = cross_val_score(xgbr, X_train, y_train, cv=kfold)

y_pred = xgbr.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
```

18.4. Особенности реализации в пакете `LightGBM`

18.5. Особенности реализации в пакете `CatBoost`

19. Потоки и процессы. Глобальная блокировка интерпретатора

Процесс – просто программа в ходе ее выполнения. *Потоки* подобны процессам, за исключением того, что все они выполняются в пределах одного и того же процесса, следовательно используют один и тот же контекст.

Все потоки, организованные в одном процессе, используют общее пространство данных с основным процессом, поэтому могут обмениваться информацией или взаимодействовать друг с другом с меньшими сложностями по сравнению с отдельными процессами. Потоки, как правило, выполняются параллельно.

Если два или несколько потоков получают доступ к одному и тому же фрагменту данных, то в зависимости от того, в какой последовательности происходит доступ, могут возникать несогласованные результаты [11, стр. 184].

Выполнением кода Python управляет виртуальная машина Python (называемая также *главным циклом интерпретатора*). Язык Python разработан таким образом, чтобы в этом главном цикле мог выполняться *только один поток управления*.

В интерпретаторе Python могут эксплуатироваться *несколько потоков*, но в каждый отдельный момент времени интерпретатором выполняется *строго один поток* [11, стр. 185].

Для управления доступом к виртуальном машине Python применяется *глобальная блокировка интерпретатора* (GIL). Именно эта блокировка обеспечивает то, что выполняется *один и только один поток*.

Вообще говоря, применение нескольких потоков в программе может способствовать ее улучшению. Однако в интерпретаторе Python применяется глобальная блокировка, которая накладывает свои ограничения, поэтому *многопоточная* организация является более подходящей для приложений, ограничиваемых пропускной способностью ввода-вывода[4] (при вводе-выводе происходит освобождение глобальной блокировки интерпретатора, что способствует повышению степени распараллеливания), а не приложений, ограничиваемых пропускной способностью процессора[5]. В последнем случае для достижения более высокой степени распараллеливания необходим иметь возможность *параллельного выполнения процессов* несколькими ядрами или *процессорами* [11, стр. 229].

Глобальная блокировка GIL защищает состояние интерпретатора от вытесняющей (приоритетной многопоточности), когда один поток пытается взять на себя управление программой, прерывая другой поток. Если такое прерывание происходит в неподходящий момент времени, то это может разрушить состояние интерпретатора.

Выводы:

- для приложений, ограниченных пропускной способностью ввода-вывода следует использовать
  - многопоточность,
  - приемы асинхронного программирования,

[4] Так называемые IO-bound задачи
[5] CPU-bound задачи

- для приложений, ограниченных пропускной способностью процессора следует иметь возможность выполнения процессов несколькими процессорами.

20. Форматирование строк в языке Python

Пример форматирования строк в Python

```
'{:*>+12.3f}, {:#^+17.5G}, {!r}'.format(
                                    math.pi,
                                    -math.exp(1)*10**(+6),
                                    type(list)  # для этого объекта будет
                                                # использована функция repr()
                                )
# "*******+3.142, ###-2.7183E+06###, <class 'type'>"
```

Часть, стоящая после двоеточия, называется *спецификатором формата* [4, стр. 283]. Полезные приемы форматирования можно найти в [6].

В Python f-строки поддерживают вложенные элементы `{...}`. Например, выведем числа n_i в едином формате, вычисляемые по формуле

$$n_i = (-1)^i \, \pi^{(-1)^i \, B}, \quad i = (1, \ldots, m).$$

```
import math

B = 15
for i in range(1, 5+1):
    n = (-1)**i*math.pi**((-1)**i*B)
    print(f'This is pi with {i} decimal places: {n:#>+15.{i}e}.')
    # вложенный элемент {...} может находится только в части спецификатора формата, после ':'
# вывод
This is pi with 1 decimal places: #######-3.5e-08.
This is pi with 2 decimal places: ######+2.87e+07.
This is pi with 3 decimal places: #####-3.489e-08.
This is pi with 4 decimal places: ####+2.8658e+07.
This is pi with 5 decimal places: ###-3.48941e-08.
```

21. SSH-клиент в браузере

В работе клиенты используют протокол SSH (Secure Shell) – сетевой протокол, позволяющий осуществлять удаленное управление различными операционными системами. Поддерживает туннелирование TCP-соединений для передачи файлов и различные алгоритмы шифрования, благодаря чему возможна безопасная передача других протоколов через SSH-туннели.

Приложение Secure Shell App представляет собой эмулятор терминала, совместимый с xterm и SSH-клиент для Chrome. Он работает путем соединения SSH-команд, портированных в Google Native Client с эмулятором терминала hterm, что позволяет приложению предоставить клиенту Secure Shell прямо в браузере, не полагаясь на внешние прокси.

Установить SSH-клиент еще можно с помощью приложения Termius. Поддреживает Windows, MacOS, Linux.

22. Большие данные в Hadoop

Hadoop это платформа для распределенного хранения и распределенной обработки больших данных.

`Hadoop` лучше всего подходит для:

- Для хранения и обработки *неструктурированных данных* объемом от 1 терабайта – такие массивы сложно и дорого хранить в локальном хранилище,
- Для компонуемых вычислений – когда нужно собрать множество схожих разрозненных данных в одно целое. Также подходит для выделения полезной информации из массива лишней информации,
- Для пакетной обработки, обогащения данных и ETL – извлечения информации из внешних источников, ее переработки и очистки под потребности компании, последующей загрузки в базу данных.

23. Теорема Байеса

Пусть X – случайная величина, ее возможное значение (или реализацию) будем обозначать через x. Если $\vec{X} = (X_1, \ldots, X_n)$ – случайный вектор, то его реализация – $\vec{x} = (x_1, \ldots, x_n)$.

Для того чтобы охарактеризовать случайную величину, необходимо задать распределение вероятностей по ее возможным значениям. Для осуществления этого используется понятие *функции распределения* вероятностей, которое является универсальным инструментом, пригодным для изучения любой случайной величины, одномерной или многомерной, и непрерывного, дискретного или смешанного типа.

Если X – одномерная случайная величина непрерывного типа с бесконечным числом возможных значений на действительной оси $\mathbb{R}^1 = \{x : -\infty < x < +\infty\}$, то она характеризуется функцией распределения вероятностей, которая определяется в виде

$$F_X(x) = \mathbf{P}(X \leqslant x),\ x \in \mathbb{R}^1.$$

Другими словами, функция распределения непрерывной случайной величины это вероятность события, состоящая в том, что случайная величина X примет значение меньшее или в частном случае равное некоторому значению x, т.е. вероятность события, что случайная величина окажется левее.

Иногда удобнее описывать случайную величину X одномерной *плотностью распределения вероятностей* $f_X(x) = F_X^{'}(x),\ x \in \mathbb{R}^1$.

Для описания случайного вектора $\vec{X} = (X_1, \ldots, X_n)$ используют функцию n переменных, которая в точке $(x_1, \ldots, x_n) \in \mathbb{R}^n$, где $\mathbb{R}^n$ обозначает n-мерное евклидово пространство, определяется с помощью вероятности совместного осуществления событий в квадратных скобках, то есть функция распределения случайного вектора $\vec{X} = (X_1, \ldots, X_n)$ определяется в виде

$$F_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = F_{\vec{X}}(x_1, \ldots, x_n) = \mathbf{P}[\, X_1 \leqslant x_1; X_n \leqslant x_n\,],\ (x_1, \ldots, x_n) \in \mathbb{R}^n.$$

Итак, для того чтобы охарактеризовать случайную величину X, необходимо задать ее функцию распределения вероятностей.

Как известно, различают дискретные и непрерывные случайные величины. Две случайные величины называются *независимыми*, если

$$p(x, y) = p(x)p(y),$$

где $p(x)$ и $p(y)$ – плотности распределения непрерывных случайных величин.

Чтобы получить обратно из совместной вероятности вероятность того или иного исхода одной из случайных величин, нужно просуммировать по другой (этот процесс часто называют маргинализацией)

$$p(x) = \sum_y p(x, y).$$

В случае непрерывных случайных величин получается, что мы фактически проецируем двумерное распределение – поверхность в трехмерном пространстве – на одну из осей, получая функцию от одной переменной

$$p(x) = \int_Y p(x, y)dy.$$

Условная вероятность $p(x|y)$ – вероятность наступления одного события, если известно, что произошло другое. Формально ее обычно определяют так

$$p(x|y) = \frac{p(x, y)}{p(y)}.$$

Аналогично можно определить *условную независимость*: x и y условно независимы при условии z, если

$$p(x, y|z) = p(x|z)p(y|z).$$

23.1. Регистрация пользовательских функций выхода из приложения

Удобно использовать встроенный модуль `atexit` для регистрации пользовательских функций, которые вызываются при выходе из приложения. Например

```
import atexit


def hello_fun():
    print('Hello message')

def exit_fun():
    print('New exit message')

atexit.register(exit_fun)
hello_fun()
```

24. Глубокое обучение

24.1. Функции активации

Без функции активации (например, такой как ReLU) полносвязный слой `keras.layers.Dense` сможет обучаться только на *линейных* (аффинных) преобразованиях входных данных: про-

странство гипотез было бы совокупностью всех возможных линейных преобразований входных данных.

Такое пространство гипотез слишком ограничено, и наложение нескольких слоев представлений друг на друга не приносило бы никакой выгоды, потому что *глубокий стек линейных слоев* все равно реализует *линейную операцию*: добавление новых слоев не расширяет пространство гипотез.

Чтобы получить доступ к более обширному пространству гипотез, дающему дополнительные выгоды от увеличения глубины представлений, необходимо применить *нелинейную*, или функцию активации.

Наконец, нужно выбрать *функцию потерь* и *оптимизатор*. Пусть для определенности перед нами стоит задача бинарной классификации и результатом работы сети является вероятность (сеть заканчивается одномодульным слоем с сигмоидной функцией активации). В этом случае предпочтительнее использовать функцию потерь `binary_crossentropy`. Перекрестная энтропия обычно дает более качественные результаты, когда результатами работы модели являются вероятности.

Перекрестная энтропия – мера расстояния между распределением вероятностей, или между фактическими данными и предсказаниями.

$$H(p,q) \stackrel{\text{def}}{=} H(p) + D_{KL}(p||q),$$

где $H(p)$ – энтропия[6] p, $D_{KL}(p||q)$ – дивергенция Кульбака-Лейблера[7] от p и q (она же относительная энтропия).

Для дискретных p и q

$$H(p,q) = -\sum_x p(x)\log q(x).$$

Для непрерывного распределения

$$H(p,q) = -\int_X p(x)\log q(x)dx.$$

Нужно учесть, что, не смотря на формальную аналогию функционалов для непрерывного и дискретного случаев, они обладают разными свойствами и имеют разный смысл. Непрерывный случай имеет ту же специфику, что и понятие дифференциальной энтропии.

Настраиваем модель оптимизатором `rmsprop` и функцией потерь `binary_crossentropy`

```
model.compile(optimizator='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Или так, если нужно передать дополнительные параметры настройки

```
from keras import optimizers
from keras import metrics
from keras import losses
```

[6]Мера неопределенности некоторой системы

[7]Дивергенция Кульбака-Лейблера – неотрицательнозначный фукнционал, являющийся несимметричной мерой удаленности друг от друга двух вероятностных распределений, определенных на общем пространстве элементарных событий

```
# настройка оптимизатора
model.compile(
    optimizer=optimizers.RMSprop(lr=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# использование нестандартных функций потерь и метрик
model.compile(
    optimizer=optimizers.RMSprop(lr=0.001),
    loss=losses.binary_crossentropy,
    metrics=[metrics.binary_crossentropy]
)
```

Чтобы проконтролировать точность модели во время обучения на данных, которые она прежде не видела, создадим проверочный набор данных, выбрав 10000 образцов из оригинального набора обучающих данных.

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Теперь проведем обучение модели в течение 20 эпох (выполнив 20 итераций по всем образцам в тензорах `x_train`, `y_train`) пакета по 512 образцов. В тоже время будем следить за потерями и точностью на 10000 отложенных образцах. Для этого достаточно передать проверочные данные в аргументе `validation_data`

```
model.compile(
    optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['acc']
)
history = model.fit(
    partial_x_train,
    partial_y_trian,
    epochs=20,  # 20 пробегов по обучающему набору данных
    batch_size=512,
    validation_data=(x_val, y_val) # вычисляем потерю и точность в конце каждой эпохи
)
```

В конце каждой эпохи обучение приостанавливается, потому что модель вычисляет *потерю* и *точность* на 10000 образцах проверочных данных.

24.2. Стохастический градиентный спуск

Стохастический градиентный спуск это метод поиска *локального* экстремума. Идея состоит в следующем:

- Извлекается пакет обучающих экземпляров `x` и соответствующих целей `y`,
- Сеть обрабатывает пакет `x` и получает пакет предсказаний `y_pred`,
- Вычисляются потери сети на пакете, дающие оценку несовпадания между `y_pred` и `y`,
- Вычисляется градиент потерь для параметров сети (обратных проход),
- Параметры корректируются на небольшую величину в направлении антиградиента, и тем самым снижают потери.

Сколько ни придумывай хитрых способов ускорить градиентный спуск, обойти небольшие локальные минимумы, выбраться из ущелий, мы все равно не сможем изменить тот факт, что градиентный спуск – это метод местного значения, и ищет он только *локальный* минимум/максимум.

Строго говоря, это реализация *минипакетного стохастического градиентного спуска*. А истинный стохастический градиентный спуск на каждой итерации использует единственный образец и цель, а не весь пакет данных.

Здесь термин стохастический относится к тому, что *каждый пакет* данных выбирается *случайно*.

Обучим новую сеть с нуля

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)

# сделать прогноз
model.predict(x_test)
```

В задачах бинарной классификации в конце нейросети должен находиться полносвязанный слой `Dense` с одним нейроном и функцией активации `sigmoid`: результатом работы сети должно быть скалярное значение в диапазоне между 0 и 1, представляющее собой вероятность.

С таким скалярным результатом, получаемым с помощью сигмоидной функции, в задачах бинарной классификации следует использовать функцию потерь `binary_crossentropy`.

В общем случае оптимизатор `rmsprop` является наиболее подходящим выбором для любого типа задач.

25. Хэшируемые пользовательские классы в языке Python

Чтобы класс был хэшируемым[8], следует реализовать метод `__hash__`. Нужно также, чтобы векторы были *неизменяемыми*. И этого можно добиться, сделав компоненты `x` и `y` свойствами, доступными только для чтения.

Пример неизменяемого, но нехэшируемого класса

```
import array
import math

class Vector2d:
    '''
    Неизменяемый, но еще нехэшируемый класс
    '''
```

[8] Обычно говорят, что объект называется хэшируемым если i) у него есть хэш-значение, которое не изменяется пока объект существует, и ii) объект поддерживает сравнение с другими объектами. Однако на мой взгляд лучше сказать, что объект является хэшируемым, если его структура не может изменяется и он поддерживает сравнение с другими объектами

```
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = x  # закрытый атрибут экземпляра класса
        self.__y = y  # закрытый атрибут экземпляра класса

    # открытое свойство; прочитать значение `x` можно, но нельзя передать новое значение
    @property
    def x(self):
        return self.__x

    # открытое свойство; прочитать значение `y` можно, но нельзя передать новое значение
    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)

    def __str__(self):
        return str(tuple(self))

    def angle(self):
        return math.atan2(self.y, self.x)

    def __format__(self, fmt_spec = ''):  # пользовательский формат
        if fmt_spec.endswith('p'):  # если спецификатор формата заканчивается на 'p',
                                    # то координаты выводятся в полярном формате
            fmt_spec = fmt_spec[:-1]
            coords = (abs(self), self.angle())
            outer_fmt = '<{}, {}>'
        else:
            coords = self
            outer_fmt = '({}, {})'
        components = (format(c, fmt_spec) for c in coords)
        return outer_fmt.format(*components)

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) + bytes(array(self.typecode, self)))

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))
```

То есть здесь декоратор `@property` помечает метод чтения свойств, который возвращает значение закрытого атрибута экземпляра класса `self.__x` или `self.__y`.

Так как в реализации класса есть метод `__format__`, можно печатать класс управляя форматом, например,

Пример использования класса с реализованным методом `__format__`

```
>>> v1 = Vector2d(10, 5)
>>> '{:*^+12.3gp}'.format(v1)  # '<***+11.2****, ***+0.464***>'
>>> '{:.3f}'.format(v1)  # '(10.000, 5.000)'
```

Наконец, можно реализовать метод `__hash__`. Он должен возвращать `int` и в идеале учитывать хэши объектов-атрибутов, потому что у равных объектов хэши также должны быть одинаковыми.

В документации по специальному методу `__hash__` рекомендуется объединять хэши компонентов с помощью побитового оператора[9] *исключающего ИЛИ* (`^`) [4, стр. 287]

```
...
def __hash__(self):
    return hash(self.__x) ^ hash(self.__y)  # побитовое исключающее ИЛИ
```

Теперь класс `Vector2d` стал *хэшируемым*.

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)  # (7, 384307168202284039)
>>> set([v1, v2])  # {Vector2d(3, 4), Vector2d(3.1, 4.2)}
```

Замечание

Строго говоря, для создания хэшируемого типа необязательно вводить свойства или как-то иначе защищать атрибуты экземпляра класса от изменения. Требуется только корректно реализовать методы `__hash__` и `__eq__`. Но хэш-значения экземпляра никогда не должно изменяться [4, стр. 288]

---

26. Как интерпретировать связь между именем функции и объектом функции в Python

Рассмотрим класс, который печатает выводимые в терминал строки в обратном порядке

```
1  class LookingGlass:
2      def __enter__(self):
3          import sys
4          # атрибут экземпляра класса self.original_write -> объект функции sys.stdout.write
5          self.original_write = sys.stdout.write
6          # переменная sys.stdout.write -> объект функции self.reverse_write
7          sys.stdout.write = self.reverse_write
8          return 'jabberwocky'.upper()
9
10     def reverse_write(self, text):
11         self.original_write(text[::-1])
12
13
14     def __exit__(self, exc_type, exc_value, traceback):
15         import sys
16         # переменная sys.stdout.write "через" атрибут экземпляра self.original_write
17         # ссылается на объект функции sys.stdout.write
18         sys.stdout.write = self.original_write
```

В методе `__enter__` есть несколько неочевидных нюансов. В строке 4 атрибут экземпляра класса `self.original_write` получает ссылку на метод `write` стандартного потока вывода, а

[9] Побитовые операторы рассматривают операнды как бинарные последовательности

в строке 5 «как бы метод» `sys.stdout.write` получает ссылку на метод экземпляра класса `self.reverse_write` и кажется, что должен был бы образоваться рекурсивный вызов, но на самом деле это не так. Дело в том, что значение имеет с какой стороны от оператора `=` стоит имя функции: если слева, то это *имя переменной*, а если справа, то это *объект функции*.

Итак, по порядку: в строке 4 атрибут экземпляра класса `self.original_write` получает ссылку на *объект функции* `sys.stdout.write`, а в 5-ой строке *переменная* `sys.stdout.write` получает ссылку на *объект функции* (метод экземпляра класса) `self.reverse_write`, который «через» атрибут экземпляра `self.original_write` вызывает *объект функции* `sys.stdout.write`.

А в строке 18, мы возвращаем все как было, т.е. *переменная* `sys.stdout.write` получает ссылку на *объект функции* `sys.stdout.write`.

Рассмотрим более простой пример (см. рис. 6)

```
>>> def f(): pass  # переменная f -> объект функции f()
>>> def g(): pass  # переменная g -> объект функции g()
# модель: переменная -> объект
>>> a = f  # переменная a -> объект функции f()
>>> f = g  # переменная f -> объект фукнции g()
# НИКАКОЙ ТРАНЗИТИВНОСТИ!
>>> a  # <function __main__.f()>
>>> f  # <function __main__.g()>
```

То есть, когда объявляется функция, например, `def f(): pass`, то создается *переменная* `f`, которая получает ссылку на *объект функции* `f()`.

Замечание

Даже если используется одно и тоже имя `f`: слева от оператора присваивания `f` – это *переменная*, а справа от оператора `f` – это *объект* (например, объект функции), так как в `Python` переменные ссылаются только на объекты!

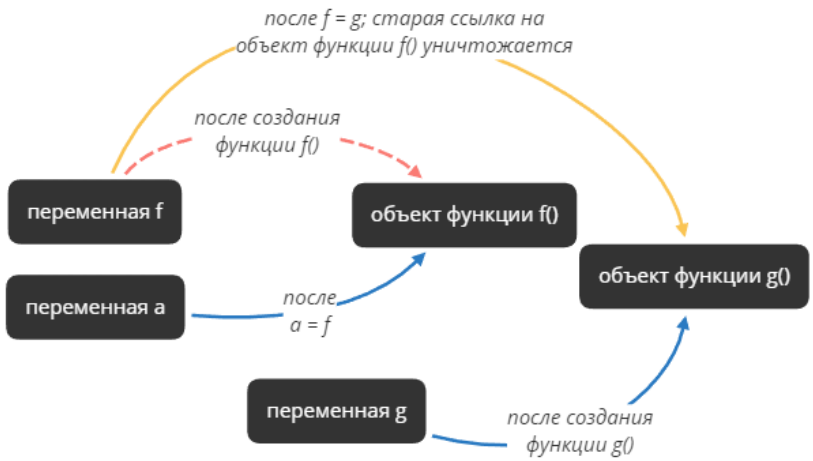


Рис. 6. Схема, описывающая связи между именами функций и их объектами

27. Использование `@contextmanager`

Если *генератор* снабжен декоратором `@contextmanager`, то `yield` разбивает тело функции на две части:

- все, что находится до `yield`, исполняется в начале блока `with`, когда интерпретатор вызывает метод `__enter__` ,
- а все, что находится после `yield`, выполняется при вызове метода `__exit__` в конце блока.

Например,

неудачный пример

```
# mirror_gen.py
import contextlib

@contextlib.contextmanager  # декорируем генераторную функцию
def looking_glass():  # генераторная функция
    import sys
    original_write = sys.stdout.write  # (1)

    def reverse_write(text):  # замыкание
        original_write(text[::-1])  # здесь original_write -- свободная переменная

    sys.stdout.write = reverse_write  # (2)
    # все что выше 'yield' выполняется в начале блока with
    yield 'jabberwocky'.upper()  # (3)
    # все что ниже 'yield' выполняется в конце блока with
    sys.stdout.write = original_write  # (4)
```

Комментарии к коду:

- (1) – локальная *переменная* `original_write` получает ссылку на *объект функции* (вернее на объект метода) стандартного потока вывода; теперь вызывая `original_write` мы будет вызывать `sys.stdout.write`,
- (2) – переменная `write` из подмодуля `stdout` модуля `sys` получает ссылку на *замыкание* `reverse_write` (функцию с расширенной областью видимости, которая включает все неглобальные переменные); теперь, когда мы вызываем `sys.stdout.write` будет вызываться `reverse_write`, который в свою очередь будет вызывать `original_write`, вызывающий метод `sys.stdout.write` и передавать ему обращенную строку,
- (3) – здесь функция приостанавливается на время выполнения блока `with`,
- (4) – когда поток выполнения покидает блок `with` любым способом, выполнение функции возобновляется с места, следующего за `yield`; в данном случае восстанавливается исходный метод `sys.stdout.write`

Пример работы функции

```
>>> from mirror_gen import looking_glass

>>> with looking_glass() as what:
        print('Alice, Kitty and Snowdrop')  # pordwonS dna yttiK ,ecilA
        print(what)                         # YKCOWREBBAJ
```

По существу декоратор `@contextlib.contextmanager` обертывает функцию классом, который реализует методы `__enter__` и `__exit__`[10].

Метод `__enter__` этого класса выполняет следующие действия [4, стр. 488]:

1. Вызывает *генераторную функцию* `looking_glass()`[11] и запоминает объект-генератор (пусть называется `gen`),

[10] Этот класс называется `_GeneratorContextManager`
[11] При вызове генераторной функции возвращается объект-генератор

2. Вызывает `next(gen)`, чтобы заставить генератор выполнить код до предложения `yield`,
3. Возвращает значение, отданное `next(gen)`, чтобы его можно было связать с переменной в части `as` блока `with`, т.е. строка, отданная инструкцией `yield` связывается с переменной `what`.

По завершении блока `with` метод `__next__` выполняет следующие действия:

1. Смотрит, было ли передано исключение в параметре `exc_type`; если да, вызывает `gen.throw(exception)`, в результате чего строка в теле генераторной функции, содержащая `yield`, возбуждает исключение,
2. В противном случае вызывает `next(gen)`, что приводит к выполнению части генераторной функции после `yield`.

В рассмотренном примере есть очень серьезный дефект: если в теле блока `with` возникает исключение, то интерпретатор перехватывает его и повторно возбуждает в выражении `yield` внутри `looking_glass`. Но здесь нет никакой обработки исключений, поэтому функция аварийно завершается, оставив систему в некорректном состоянии.

Более аккуратный вариант генераторной функции приведен ниже

Правильный вариант

```
# mirror_gen_exc.py
import contextlib

@contextlib.contextmanager
def looking_glass():  # здесь генераторная функция работает скорее как сопрограмма
    import sys
    original_write = sys.stdout.write  # переменная получает -> на объект функции write

    def reverse_write(text):  # замыкание
        original_write(text[::-1])

    sys.stdout.write = reverse_write  # переменная write получает -> на замыкание reverse_write
    msg = ''
    try:
        yield 'jabberwocky'.upper()  # отдает строку и переключается на блок with
    except ZeroDivisionError:
        msg = 'Пожалуйста не делите на ноль!'
    finally:  # выполняется в любом случае
        sys.stdout.write = original_write  # переменная write получает -> на объект функции
    write
        if msg:  # if msg != ''
            print(msg)
```

Пример выполнения

```
>>> from mirror_gen_exc import looking_glass
>>> with looking_glass() as what:
        print('aaaabb')  # bbaaaa
        print(5/0)       # Пожалуйста не делите на ноль!
```

Замечание

Отметим, что использование слова `yield` в генераторе, который используется совместно с декоратором `@contextmanager`, не имеет ничего общего с итерированием. В рассмотренных примерах генераторная функция работает скорее, как *сопрограмма*: процедура, которая доходит до определенной точки, затем приостанавливается и дает возможность поработать клиентскому коду до тех пор, пока он не захочет возобновить выполнение процедуры с прерванного места

28. Перегрузка операторов в языке Python

Перегрузка операторов позволяет экземплярам классов участвовать в обычных операциях [6].

Основы перегрузки операторов:

- запрещается перегружать операторы для встроенных типов,
- запрещается создавать новые операторы, можно перегружать существующие,
- несколько операторов нельзя перегружать вовсе: `is`, `and`, `or`, `not` (на побитовые операторы это не распространяется)

Фундаментальное правило: инфиксный оператор всегда возвращает *новый объект*, т.е. создает новый экземпляр (составные операторы изменяемых объектов возвращают `self`, т.е. изменяют левый операнд на месте).

Иначе говоря, в случае инфиксных операторов нельзя модифицировать `self`, а нужно создавать и возвращать новый экземпляр подходящего типа [4, стр. 405].

Замечание

Инфиксные операторы (`*`, `+` и т.д.) независимо от типа данных всегда возвращают *новый объект*. *Составные* операторы (`+=`, `*=` и пр.) для объектов *неизменяемого* типа данных (кортежи, строки и пр.) возвращают новый объект, но в случае объектов *изменяемого* типа данных (списки) – изменяют объект на месте

Сравнение работы инфиксных и составных операторов

```
# изменяемый объект
>>> lst = [100]
>>> id(lst)  # 179426376
>>> lst = lst*2  # инфиксный оператор возвращает новый объект, поэтому id будет другим
>>> id(lst)  # 117159368 -- изменился
>>> lst  # [100, 100]
>>> lst *= 2  # но составной оператор для изменяемого объекта изменяет левый операнд на месте
>>> lst  # [100, 100, 100, 100]
>>> id(lst)  # 117159368 -- не изменился
# неизменяемый объект
>>> tpl = (100,)
>>> id(tpl)  # 114189896
>>> tpl = tpl*2  # инфиксный оператор вернет новый объект
>>> tpl  # (100, 100)
>>> id(tpl)  # 82350344 -- изменился
>>> tpl *= 2  # составной оператор создаст новый объект и перепривяжет его к tpl
>>> tpl # (100, 100, 100, 100)
>>> id(tpl)  # 93229768 -- изменился
```

При умножении *последовательности* (списки, кортежи, строки) на *целое число* создается копия последовательности заданное число раз, а затем копии склеиваются.

Как читать выражения с математическими операторами:

- Смотрим к какому классу относится оператор: *инфиксному* или *составному*,
- Если оператор инфиксный, то независимо от того являются операнды изменяемыми или нет будет возвращен новый объект[12],
- Если оператор составной, то нужно выяснить является левый операнд изменяемым или нет,
  - левый операнд изменяемый: составной оператор изменит левый операнд на месте (идентификатор не изменится),
  - левый операнд неизменяемый: составной оператор создаст новый объект и перепривяжет его к переменной (изменится идентификатор).

28.1. Перегрузка оператора сложения

Для поддержки операций с объектами *разных типов* в `Python` имеется особый механизм диспетчеризации для специальных методов, ассоциированных с инфиксными операторами.

Видя выражение `a + b`, интерпретатор выполняет следующие шаги:

- Если у `a` есть метод `__add__`, вызвать `a.__add__(b)` и вернуть результат, если только он не равен `NotImplemented`[13] (т.е. оператор не знает как обрабатывать данный операнд),
- Если у левого операнда `a` нет метода `__add__` или его вызов вернул `NotImplemented`, проверить, есть ли у правого операнда `b` «правый» метод `__radd__`[14], и, если да, вызвать `b.__radd__(a)` и вернуть результат, если только он не равен `NotImplemented`,
- Если у `b` нет метода `__radd__` или его вызов вернул `NotImplemented`, возбудить исключение `TypeError`.

Рассмотрим реализацию методов сложения для объектов

```
import itertools
import reprlib

class VectorUser:
    def __init__(self, seq):
        self._seq = array('d', seq)

    def __iter__(self):
        return iter(self._seq)

    def __repr__(self):
        components = reprlib.repr(self._seq)
        components = components[components.find('['):-1]
        return f'Vector({components})'

    def __add__(self, other):
        try:
            pairs = itertools.zip_longest(self, other, fillvalue=0.0)
            return VectorUser(a + b for a, b in pairs)  # возвращает новый экземпляр класса
        except TypeError:
            return NotImplemented

    def __radd__(self, other):
```

[12] При условии, что оператор в случае данных операндов имеет смысл

[13] `NotImplemented` – это значение-синглтон, которое должен возвращать специальный метод инфиксного оператора, чтобы сообщить интерпретатору, что не умеет обрабатывать данный операнд

[14] Иногда такие методы называют «инверсными» методами, но лучше их представлять как *правые* методы, так как они вызываются от имени правого операнда

```
        return self + other
```

Как работает этот код. Рассмотрим случай, когда экземпляр класса `Vector` находится слева от оператора `+`

```
>>> v1 = VectorUser([3, 4, 5])
>>> v1 + (10, 20, 30)  # Vector([13.0, 24.0, 35.0])
# v1.__add__((10, 20, 30))
# удобно представлять VectorUser.__add__(v1, (10, 20, 30))
```

Первым делом интерпретатор пытается выяснить есть ли у левого операнда метод `__add__`. В данном случае у объекта `v1` есть такой метод, поэтому ничто не мешает вызвать его напрямую. Аргумент `self` метода `__add__` получает ссылку на `v1` (экземпляр класса `Vector`), а `other` – ссылку на кортеж. Далее с помощью `zip_longest` конструируется генератор кортежей, который в следующей строке используется в генераторном выражении при создании нового экземпляра класса `Vector` (оператор должен возвращать новый объект).

Теперь рассмотрим случай, когда экземпляр класса `VectorUser` находится справа от оператора `+`

```
>>> (10, 20, 30) + v1
```

И снова интерпретатор пытается выяснить есть ли у левого операнда метод `__add__`. У кортежа есть такой метод, но он не умеет работать с объектом `VectorUser` (возвращает `NotImplemented`).

Теперь интерпретатор проверяет есть ли у правого операнда «правый» метод `__radd__`. Правый операнд это экземпляр класса `VectorUser`, поэтому `v1.__radd__((10, 20, 30))` это то же самое что и `VectorUser.__radd__(v1, (10, 20, 30))`.

Другими словами, аргумент `self` метода `__radd__` получает ссылку на объект `v1`, а аргумент `other` – ссылку на кортеж. И тогда в выражении `self + other`, которое возвращается методом `__radd__`, экземпляр класса `VectorUser` окажется слева от оператора `+`. Интерпретатор, встретив выражение `self + other`, начинает с поиска метода `__add__` у левого операнда и, найдя его, возвращает новый экземпляр класса `VectorUser(...)`.

Замечание

Еще раз: чтобы поддержать операции с *разными типами*, мы возвращаем специальное значение `NotImplemented` – не исключение, – давая интерпретатору возможность попробовать еще раз: поменять операнды местами и вызывать специальный инверсный (правый) метод, соответствующий тому же оператору (например, `__radd__`)

28.2. Перегрузка оператора умножения на скаляр

Рассмотрим в качестве примера умножение вектора `VectorUser` на скаляр

```
import numbers

# внутри класса VectorUser
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):  # сравнение с абстрактным базовым классом
        return VectorUser(n*scalar for n in self)
    else:
        return NotImplemented

def __rmul__(self, scalar):
```

```
    return self*scalar
```

```
>>> v1 = VectorUser([3, 4, 5])
>>> v1*4  # Vector([12.0, 16.0, 20.0])
>>> 10*v1 # Vector([30.0, 40.0, 50.0])
```

В первом случае интерпретатор начинает с поиска метода `__mul__` у левого операнда. Метод найден, объект справа (число 4) действительно является экземпляром подкласса абстрактного базового класса `numbers.Real`. Значит теперь можно вернуть экземпляр `VectorUser`.

Во втором случае интерпретатор так же начинает с поиска метода `__mul__` у левого операнда и не находит его. Поэтому на следующем шаге ищется правый метод `__rmul__` у правого операнда. Теперь объект `v1` в выражении `self*scalar` стоит слева и потому в методе `__rmul__` аргумент `self` ссылается на `v1`, а `scalar` – на `4`. Видя выражение `self*scalar` интерпретатор вызывает метод `__mul__`, который на этот раз выполняется без проблем.

Замечание

В общем случае, если прямой инфиксный метод (например, `__mul__`) предназначен для работы только с операндами того же типа, что и `self`, бесполезно реализовывать соответствующий инверсный метод (например, `__rmul__`), потому что он, по определению, вызывается, только когда второй операнд имеет другой тип [4, стр. 425]

28.3. Операторы сравнения

Обработка операторов сравнения (`==`, `!=`, `>`, `<=` и т.д.) интерпретатором `Python` похожа на обработку инфиксных операторов, но есть два важных отличия [4, стр. 417]:

- для прямых и инверсных (правых) методов служит один и тот же набор методов; например, в случае оператора `==` как прямой, так и правый вызов обращаются к методу `__eq__`, но изменяется порядок аргументов.
- в случае `==` и `!=`, если инверсный (правый) вызов завершается ошибкой, то `Python` сравнивает идентификаторы объектов, а не возбуждает исключение (см. табл. 1).

Таблица 1. *Операторы сравнения. Инверсные (правые) методы вызываются, когда прямой вызов вернул* `NotImplemented`

Группа	Инфиксный оператор	Прямой вызов метода	Инверсный вызов метода	Запасной вариант
Равенство	`a == b`	`a.__eq__(b)`	`b.__eq__(a)`	`return id(a) == id(b)`
	`a != b`	`a.__ne__(b)`	`b.__ne__(a)`	`return not (a == b)`
Порядок	`a > b`	`a.__gt__(b)`	`a.__lt__(b)`	`raise TypeError`
	`a < b`	`a.__lt__(b)`	`a.__gt__(b)`	`raise TypeError`
	`a >= b`	`a.__ge__(b)`	`a.__le__(b)`	`raise TypeError`
	`a <= b`	`a.__le__(b)`	`a.__ge__(b)`	`raise TypeError`

Однако поведение оператора `==` пользовательских классов зависит от реализации метода `__eq__`. Например, пусть есть класс `Vector`

```
# в классе Vector
def __eq__(self, other):
    if isinstance(other, Vector):
        return (len(self) == len(other) and all(a == b for a, b in zip(self, other)))
    else:
```

```
        return NotImpemented
```

и какой-то другой класс `Vector2d`

```
# в классе Vector2d
def __eq__(self, other):
    retrun tuple(self) == tuple(other)
```

Если теперь сравнить экземпляры этих классов

```
>>> v1 = Vector([1, 2])
>>> v2 = Vector2d(1, 2)
>>> v1 == v2  # True
```

то порядок действий будет следующим:

- для вычисления `v1 == v2` интерпретатор вызовет `Vector.__eq__(v1, v2)`,
- метод `Vector.__eq__(v1, v2)` видет, что `v2` не является экземпляром класса `Vector` и возвращает `NotImplemented`,
- получив значение `NotImplemented`, интерпретатор вызывает метод `__eq__` правого операнда, т.е. `v2`: `Vector2d.__eq__(v2, v1)`,
- `Vector2d.__eq__(v2, v1)` преобразует оба операнда в кортежи и сравнивает их, результат оказывается равен `True`.

Теперь рассмотрим сравнение с кортежем

```
>>> t = (1, 2)
>>> v1 == t  # False
```

В этом случае:

- для вычисления `v1 == t` `Python` вызывает `Vector.__eq__(v1, t)`,
- метод `Vector.__eq__(v1, t)` видит, что кортеж `t` не является экземпляром класса `Vector` и возвращает `NotImplemented`,
- получив результат `NotImplemented`, интерпретатор вызывает метод `__eq__` правого объекта, т.е. `tuple.__eq__(t, v1)`
- но `tuple.__eq__(t, v1)` ничего не знает о классе `Vector`, и поэтому возвращает `NotImplemented`,
- если правый вызов вернул `NotImplemented`, то `Python` в качестве последнего средства сравнивает идентификаторы объектов, что в данном случае возвращает `False`

29. Области видимости в языке Python

Когда мы говорим о поиске значения имени применительно к программному коду, под термином *область видимости* подразумевается *пространство имен* – то есть место в программном коде, где имени было присвоено значение [1].

В любом случае область видимости переменной (где она может использоваться) всегда определяется местом, где ей было присвоено значение.

Замечание

Термины «*область видимости*» и «*пространство имен*» можно использовать как синонимичные

При каждом вызове функции создается новое *локальное пространство имен*. Это пространство имен представляет локальное окружение, содержащее имена параметров функции, а также имена переменных, которым были присвоены значения в теле функции.

По умолчанию операция присваивания создает локальные имена (это поведение можно изменить с помощью `global` или `local`).

Схема разрешения имен в языке `Python` иногда называется *правилом LEGB*[15] [1, стр. 477]:

- Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости – в *локальной*, затем в *локальной области любой объемлющей функции* или в выражении `lambda`, затем в *глобальной* и, наконец, во *встроенной*. Поиск завершается, как только будет найдено первое подходящее имя.
- Когда внутри функции выполняется операция присваивания `a=10` (а не обращения к имени внутри выражения), интерпретатор всегда создает или изменяет имя в *локальной области видимости*, если в этой функции оно не было объявлено глобальным или нелокальным.

Пример

```
# глобальная область видимости
X = 99

def func(Y):  # Y и Z локальные переменные
    # локальная область видимости
    Z = X + Y # X - глобальная переменная
    return Z

func(1)  # Y = 1
```

Переменные `Y` и `Z` являются *локальными* (и существуют только во время выполнения функции), потому что присваивание значений обоим именам осуществляется внутри определения функции: присваивание переменной `Z` производится с помощью инструкции `=`, а `Y` – потому что аргументы всегда передаются через операцию присваивания.

Когда внутри функции выполняется операция присваивания значения переменной, она всегда выполняется в *локальном пространстве имен функции*

```
a = 10  # глобальная область видимости

def f():
    a = 100  # локальная область видимости
    return a
```

В результате переменная `a` в теле функции ссылается на совершенно другой объект, содержащий значение 100, а не тот, на который ссылается внешняя переменная.

Переменные во вложенных функциях привязаны к *лексической области видимости*. То есть поиск имени переменной начинается в *локальной области видимости* и затем последовательно продолжается во всех *объемлющих областях видимости внешних функций*, в направлении от внутренних к внешним.

Если и в этих *пространствах имен* искомое имя не будет найдено, поиск будет продолжен в *глобальном пространстве имен*, а затем во *встроенном пространстве имен*, как и прежде.

При обращении к локальной переменной до того, как ей будет присвоено значение, возбуждается исключение `UnboundLocalError`. Следующий пример демонстрирует один из возможных сценариев, когда такое исключение может возникнуть

```
i = 0
def foo():
    i = i + 1  # приведет к исключению UnboundLocalError
    print(i)
```

[15] Local, Enclosing, Global, Built-in

В этой функции переменная `i` определяется как *локальная* (потому что внутри функции ей присваивается некоторое значение и отсутствует инструкция `global`).

При этом инструкция присваивания `i = i + 1` пытается прочитать значение переменной `i` еще до того, как ей будет присвоено значение.

Хотя в этом примере существует глобальная переменная `i`, она не используется для получения значения. Переменные в функциях могут быть либо *локальными*, либо *глобальными* и не могут произвольно изменять *область видимости* в середине функции.

Замечание

Оператор `global` делает локальную переменную в теле функции *глобальной* и говорит интерпретатору чтобы тот не искал переменную в локальной области видимости текущей функции

Например, нельзя считать, что переменная `i` в выражении `i + 1` в предыдущем фрагменте обращается к глобальной переменной `i`; при этом переменная `i` в вызове `print(i)` подразумевает локальную переменную `i`, созданную в предыдущей инструкции.

Обобщение по вопросу

Когда интерпретатор, построчно сканируя тело функции `def`, натыкается на строку `i = i + 1`, он заключает что переменная `i` является *локальной*, так как ей присваивается значение именно в теле функции. А когда функция вызывается на выполнение и интерпретатор снова доходит до строки `i = i + 1`, выясняется, что переменная `i`, стоящая в правой части, не имет ссылок на какой-либо объект и потому возникает ошибка `UnboundLocalError`

---

30. Декораторы в Python

Декораторы выполняются *сразу после* загрузки или импорта модуля, однако увидеть какие-либо изменения можно только в том случае, если декоратор явно взаимодействует с пользователем на «верхнем уровне»[16], например, печатает строку в терминале. *Задекорированные* же функции выполняются строго в результате явного вызова [4, стр. 217].

30.1. Реализация простого декоратора

Рассмотрим простой декоратор, который хронометритует каждый вызов задекорированной функции и печатает затраченное время

clockdeco.py, не очень удачный пример декоратора

```
import time

def clock(func):
    print('test string from `clock`')  # <- строка будет выведена в терминал
                                       # сразу после загрузки модуля, который
                                       # импортирует данный декоратор
    def clocked(*args):  # замыкание
        t0 = time.perf_counter()  # запомнить начальный момент времени
        result = func(*args)  # вызвать функцию
        elapsed = time.perf_counter() - t0  # вычислить сколько прошло времени
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
```

[16] Если декоратор простой одноуровневый, то под верхним уровнем понимается его локальная область видимости, а если декоратор содержит замыкание, то – понимается область видимости объемлющей функции

```
        print(f'{elapsed}, {name}({arg_str}) -> {result}')
        return result  # вернуть результат
return clocked
```

Использование декоратора выглядит так

clockdeco_demo.py

```
import time
from clockdeco import clock

def simple_deco_1(f):
    '''
    Декоратор с замыканием
    '''
    def inner():
        print('test string from `simple_deco_1`')  # <- строка НЕ будет выведена
                                                   # после загрузке модуля
    return inner

def simple_deco_2(f):
    '''
    Простой одноуровневый декоратор
    '''
    print('test string from `simple_deco_2`')  # <- строка будет выведена в терминал
                                               # сразу после загрузки модуля
    return f

@simple_deco_1  # simple_func_1 = simple_deco_1(f=simple_func_1) -> inner
def simple_func_1():
    print('test string from `simple_func_1`')

@simple_deco_2  # simple_func_2 = simple_deco_2(f=simple_func_2) -> simple_func_2
def simple_func_2():
    print('test string from `simple_func_2`')

@clock  # snooze = clock(func=snooze) -> clocked
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)


if __name__ == '__main__':
    print('*'*10, 'Calling snooze(.123)')
    print('snooze_result = {}'.format(snooze(.123)))
    print('*'*10, 'Calling factorial(6)')
    print('6! = ', factorial(6))
    print(f'This is result from `simple_func_1`: {simple_func_1()}')
    print(f'This is result from `simple_func_2`: {simple_func_2()}')
```

Вывод clockdeco_demo.py

```
test string from `simple_deco_2`
test string from `clock`
test string from `clock`
********** Calling snooze(.123)
0.1261, snooze(0.123) -> None
snooze_result = None
```

```
********** Calling factorial(6)
1.866e-06, factorial(1) -> 1
7.589e-05, factorial(2) -> 2
0.0001266, factorial(3) -> 6
0.0001732, factorial(4) -> 24
0.0002224, factorial(5) -> 120
0.0002715, factorial(6) -> 720
6! =  720
test string from ‘simple_deco_1‘
this is result from ‘simple_func_1‘: None
test string from ‘simple_func_2‘
this is result from ‘simple_func_2‘: None
```

Замечание

Приведенный выше пример декоратора `clock` из модуля `clockdeco.py` не удачен в том смысле, что если нам, например, потребуется вывести значение атрибута `__name__` задекорированной функции `snooze`, т.е. `snooze.__name__`, то будет возвращена строка `'clocked'`, а не `'snooze'`.

Чтобы декоратор «не портил» значения атрибута `__name__`, следует задекорировать замыкание декоратора с помощью `@functools.wraps(func)`

При разгрузке модуля `clockdeco_demo.py` будут выполнены все декораторы, но только декораторы `simple_deco_2` и `clock` выведут в терминал строки, потому как эти строки расположены на верхнем уровне декораторов (т.е. находятся не внутри вложенных функций). Декоратор `simple_deco_1` ничего не выводит, так как строка находится в области видимости вложенной функции.

Важно отметить следующее: после загрузки модуля, как уже говорилось выше, будут выведены в терминал строки, расположенные на верхнем уровне декораторов, но самое главное заключается в том, что после выполнения декоратора `clock` объект `snooze` уже будет ссылаться на внутреннюю функцию `clocked` декоратора `clock`, а после выполнения декоратора `simple_deco_1` объект `simple_func_1` будет ссылаться на внутреннюю функцию `inner`. Что же касается декоратора `simple_deco_2`, то объект `simple_func_2` будет ссылаться на `simple_func_2`.

По этой причине при вызове функции `simple_func_1()` печатается строка из внутренней функции `inner`, а при вызове функции `simple_func_2()` – строка из этой же функции.

Еще один пример декоратора с замыканием

```
def deco(f):
    def inner(*args, **kwargs):
        print(f'from ‘deco-inner‘: args={args}, kwargs={kwargs}')
        return f  # f - свободная переменная
    return inner

@deco  # target = deco(f=target) -> inner :: target -> inner :: target=inner
def target(a, b=10):
    return (f'from ‘target‘: a={a}, b={b}')

print(target(20, b=500)(250))  # сначала вызывается inner(20, b=500), а потом target(250)
```

Выведет

```
from ‘deco-inner‘: args=(20,), kwargs={'b': 500}
from ‘target‘: a=250, b=10
```

30.2. Кэширование с помощью `functools.lru_cache`

Декоратор `functools.lru_cache` очень полезен на практике. Он реализует запоминание: прием оптимизации, смысл которого заключается в сохранении результатов предыдущих дорогостоящих вызовов функции, что позволяет избежать повторного вычисления с теми же аргументами, что и раньше [4, стр. 230].

Например

```
import functools
from clockdeco import clock

@functools.lru_cache
@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__ == '__main__':
    print(fibonacci(6))
```

Замечание

`lru_cache` хранит результаты в словаре, ключи которого составлены из позиционных и именованных аргументов вызовов, а это значит, что все аргументы, принимаемые декорируемой функции должны быть *хешируемыми*

30.3. Одиночная диспетчеризация и обобщенные функции

Декоратор `functools.singledispatch` позволяет каждому модулю вносить свой вклад в общее решение. Обычная функция, декорированная `@singledispatch` становится *обобщенной функцией*: групповой функцией, выполняющей одну и ту же логическую операцию по-разному в зависимости от типа первого аргумента [4, стр. 234]. Именно это и называется *одиночной диспетчеризацией*. Если бы для выбора конкретных функций использовалось больше аргументов, то мы имели бы дело с *множественной диспетчеризацией*.

Например

```
from functools import singledispatch
from collections import abc
import numbers
import html

@singledispatch  # делает функцию обобщенной
def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)


@htmlize.register(str)  # будет вызываться для объектов строкового типа данных
def _(text):
    content = html.escape(text).replace('\n', '<br>\n')
    return '<p>{}</p>'.format(content)


@htmlize.register(numbers.Integral)  # будет вызваться для объектов целочисленного типа данных
```

```
def _(n):
    return '<pre>{} (0x{:x})</pre>'.format(n)


@htmlize.register(tuple)
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'
```

Замечание

По возможности следует стараться регистрировать специализированные функции для обработки абстрактных базовых классов, например, `numbers.Integral` или `abc.MutableSequence`, а не конкретные реализации типа `int` или `list`

Замечательное свойство механизма `singledispatch` состоит в том, что специализированные функции можно зарегистрировать в любом месте системы, в любом модуле [4].

30.4. Композиции декораторов

Когда два декоратора `@d1` и `@d2` применяются к одной и той же функции `f` в указанном порядке, получается то же самое, что в результате композиции `f = d1(d2(f))`.

Иными словами

```
@d1
@d2
def f():
    print('f')
```

эквивалентен следующему

```
def f():
    print('f')

f = d1(d2(f))
```

Рассмотрим еще один пример композиции декораторов

```
def deco1(f):  # выполняется вторым
    print('deco-1')  # # будет выведена в терминал
    def inner1():
        print('string from `deco1-inner`')
    return inner1

def deco2(f):  # выполняется первым
    print('deco-2')  # будет выведена в терминал
    def inner2():
        print('string from `deco2-inner')
    return inner2

@deco1  # 2) inner2 = deco1(f=inner2) -> inner1 :: inner2 -> inner1 :: inner2 = inner1
@deco2  # 1) target = deco2(f=target) -> inner2 :: target -> inner2 :: target = inner2
def target():  # 3) target -> inner1
    print('string from `target`')


if __name__ == '__main__':
    target()  # выведем string from `deco1-inner`
```

Выведет

```
deco-2
deco-1
string from ‘deco1-inner‘
```

Замечание

Первым выполняется тот декоратор, который ближе расположен к декорируемой функции

То есть при загрузке или импорте модуля будут выполнены декораторы `deco1` и `deco2`: сначала `deco2`, а затем `deco1`, потому как `deco2` ближе к декорируемой функции. Декоратор `deco1` применяется к той функции, которую возвращает `deco2`.

30.5. Параметризованные декораторы

Параметризованные декораторы часто называют *фабриками декораторов*. Фабрики декораторов возвращают настоящие декораторы, которые применяются к декорируемой функции.

Пример

```
registry = set()

def register(activate=True):  # фабрика декораторов
    def decorate(func):  # декоратор
        print(f'running register(activate={activate})->decorate({func})')
        if activate:
            registry.add(func)
        else:
            registry.discard(func)
        return func
    return decorate

@register(activate=False)  # f1 = decorate(func=f1) -> f1 :: f1 -> f1
def f1():
    print('running f1()')

@register()  # f2 = decorate(func=f2) -> f2 :: f2 -> f2
def f2():
    print('running f2()')

def f3():
print('running f3()')
```

Идея в том, что функция `register()` возвращает декоратор `decorate`, который затем применяется к декорируемой функции [4].

Замечание

Фабрика декораторов возвращает *декоратор*, который применяется к декорируемой функции

Чуть подробнее: сразу после загрузки или импорта модуля выполняется фабрика декораторов `register`, которая возвращает декоратор `decorate`, который и применяется к функциям. Можно представлять, что фабрика декораторов нужна только для того, чтобы собрать значения каких-то дополнительных переменных, которые потребуются позже. В данном примере можно представить, что строка `@register()` заменяется на строку `@decorate`. То есть декоратор применяется к функции, расположенной на следующей строке, и работает как обычно.

Как можно работать с этой фабрикой декораторов

```
register()(f3)  # добавить ссылку на функцию f3 во множество registry
register(activate=False)(f2)  # удалить ссылку на функцию f2
```

Конструкция `register()` возвращает декоратор, который затем применяется к переменной (например, к `f3`), ассоциированной с декорируемой функцией, и работает так, как если бы изначально был только он (без фабрики декораторов) [4].

Если бы у декоратора был еще один уровень вложенности, т.е. было бы определено еще и замыкание, то это изменило бы только ссылку на функцию, которую возвращает замыкание

```
def fabricdeco():  # фабрика декораторов
    def deco(f):  # декоратор
        def inner():  # замыкание
            print(f'from inner: {f}')
        return inner
    return deco

@fabicdeco()  # target = deco(f=target) -> inner :: target -> inner :: target=inner
def target():
    print('from target')

target()  # на самом деле вызывается inner() -> from inner: <function target at 0x0...08B05318>
```

Рассмотрим еще один пример параметризованного декоратора

```
import time

DEFAULT_FMT = '[{elapsed}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT):  # фабрика декораторов
    def decorate(func):  # декоратор
        count = 0
        def clocked(*_args):  # замыкание
            nonlocal count  # делает переменную свободной
            count += 1 # без nonlocal здесь была создана новая локальная переменная count
            print(f'args-{count}: {_args}')
            t0 = time.time()
            _result = func(*_args)
            elapsed = time.time() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)
            result = repr(_result)
            print(fmt.format(**locals())) # использование **locals() позволяет ссылаться
                                          # на любую локальную переменную clocked
            return _result
        return clocked
    return decorate


if __name__ == '__main__':
    @clock()  # snooze = decorate(func=snooze) -> clocked :: snooze -> clocked
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(0.123)
```

В этом примере необходима строка `nonlocal count`, так как во вложенной функции `clocked` создается новая локальная переменная `count`, которая «затирает» переменную `count` из области

видимости объемлющей функции `decorate`. Ключевое слова `nonlocal` говорит интерпретатору, что при поиске значения переменной `count` не следует ограничиваться локальной областью видимости функции `clocked`.

Теперь фабрику декораторов можно вызывать, например, так:

```
@clock('log::{name}({args}), dt={elapsed:.5g}s')
def snooze(seconds):
    time.sleep(seconds)
```

Объяснение: сразу после загрузки модуля (когда модуль загружается как скрипт), интерпретатор наталкивается на строку `@clock()` после чего вызывает *фабрику декораторов* `clock`, которая возвращает ссылку на *декоратор* `decorate`, который в свою очередь начинает работать как и в описанных выше случаях, т.е. аргумент `func` декоратора получает ссылку на `snooze`, а сам декоратор возвращает ссылку на *замыкание* `clocked`.

Замечание

Интерпретатор вызывает *декоратор* или *фабрику декораторов* из той строки, в которой находится конструкция `@deco`, поэтому если, как в данном примере, `@clock()` разместить в блоке проверки значения атрибута `__name__`, а сам модуль *импортировать* (а не выполнять как сценарий), то фабрика декораторов не будет вызвана, потому что не будет выполнено условие `if __name__ == '__main__'` и фрагмент модуля со строкой `@clock()` останется скрытым от интерпретатора

Однако здесь есть любопытный момент. Переменные `fmt`, `func` и `count` вообще говоря являются *свободными переменными*, поэтому их значения можно читать из-под замыкания (находясь в области видимости замыкания) даже после того, как *локальная область видимости объемлющей функции (декоратора) будет уничтожена.*

Но, присваивая значение переменной `count` на уровне замыкания `clocked`, мы делаем эту переменную локальной и привязываем к области видимости функции `clocked`. Таким образом, интерпретатор «думает», что переменная `count` локальная для функции `clocked` и следовательно значение этой переменной должно быть в пределах функции `clocked`. При вызове функции `clocked` вычисления `count = count + 1` начинаются с правой части и когда интерпретатор не находит значения переменой `count` в области видимости функции `clocked` возникает ошибка `UnboundLocalError`.

Замечание

Если переменная локальная, то интерпретатор в поисках значения этой переменной не может покинуть соответствующую локальную области видимости

Еще раз. *Свободные переменные* по умолчанию можно только читать из-под замыкания. Когда мы присваиваем новое значение переменной `count` в теле замыкания, мы делаем эту переменную *локальной* для замыкания `clocked`.

Чтобы объяснить интерпретатору, что переменная `count` должна рассматриваться как *свободная* даже если ей присваивается значение в области видимости замыкания (что делает переменную локальной), следует использовать оператор `nonlocal`.

Замечание

Можно сказать, что оператор `nonlocal` разрешает интерпретатору искать значение указанных переменных в области видимости *объемлющей функции*, а оператор `global` – в глобальной области видимости, т.е. на уровне модуля

Пример

```
a = 10

def f():
    '''
    Разрешает искать в
    области видимости объемлющей функции
    '''
    a = 100
    def inner():
        nonlocal a  # <-- NB
        a += 1
        print(a)
    return inner

f()()  # 101
```

```
a = 10

def f():
    '''
    Разрешает искать
    в глобальной области видимости
    '''
    a = 100
    def inner():
        global a  # <-- NB
        a += 1
        print(a)
    return inner

f()()  # 11
```

30.6. Цепочка параметрических декораторов

Рассмотрим пример параметрических декораторов

deco.py

```
def check_user_is_not(username): # фабрика декораторов -> декоратор
    def user_check_decorator(f):
        def wrapper(*args, **kwargs):
            if kwargs.get("username") == username:
                raise Exception(f"User {username} is not allowed to get food")
            return f(*args, **kwargs)
        return wrapper
    return user_check_decorator


class Store:
    def __init__(self):
        self.storage = {
          "meat" : 10,
          "eggs" : 20,
        }


     @check_user_is_not("admin") # wrapper#1 = user_check_decorator(f=wrapper#1) -> wrapper#2
     @check_user_is_not("alex") # get_food = user_check_decorator(f=get_food) -> wrapper#1
     def get_food(self, *, username=None, food=None): # get_food = wrapper#2
         return self.storage.get(food)


store = Store()
print(store.get_food(username="leor", food="meat"))
# Вывод
# admin
# alex
```

```
# 10  # из-под wrapper уровня "admin"
```

Сразу после загрузки модуля `deco.py` запускается цепочка параметрических декораторов, начиная с того декоратора, который расположен ближе к декорируемой функции.

О параметрическом декораторе (т.е. параметрической фабрике декораторов – функции, которая возвращает декоратор) можно думать так, как, если бы никакой фабрики там не было, а был только обычный декоратор, который возвращается фабрикой, и передает переменной `username` какое-то значение, т.е.

deco.py

```
...
@check_user_is_not("admin") # @user_check_decorator
@check_user_is_not("alex")  # @user_check_decorator
def get_food(self, *, username=None, food=None): # get_food = wrapper#2
    ...
```

Другими словами, параметрический декоратор `check_user_is_not()` возвращает ссылку на вложенную функцию (настоящий декоратор) `user_check_decorator` и на этом останавливается. Затем включается «настоящий» декоратор `user_check_decorator`, который, как обычно получает ссылку на декорируемую функцию и возвращает ссылку на вложенную функцию `wrapper` и на этом останавливается

```
get_food = user_check_decorator(f=get_food) -> wrapper#1
```

Теперь эта вложенная функция `wrapper` становится декорируемой. И ее декориует параметрический декоратор `check_user_is_not("admin")`, который снова возвращает ссылку на `user_check_decorator`, но уже с другим значением переменной `username`. А декоратор `user_check_decorator` возвращает ссылку на вложенную функцию `wrapper`

```
wrapper#1 = user_check_decorator(f=wrapper#1) -> wrapper#2
```

В итоге декорируемая функция `get_food` заменяется на функцию `wrapper` с `username="admin"`. И потому при вызове метода `get_food` вызывается функция `wrapper#2` со значением переменной `username="admin"` и собственными параметрами `username="leor", food="meat"`. В результате чего условие

```
if kwargs.get("username") == username: # "leor" == "admin"
```

не выполняется и функция `wrapper#2` (со значением `"admin"`) вызывает функцию `f(*args, **kwargs)` (т.е. функцию `wrapper#1(username="leor", food="meat")`, для которой также не выполняется условие

```
if kwargs.get("username") == username: # "leor" == "alex"
```

и вызывается функция `f(*args, **kwargs)` (т.е. функция `get_food(username="leor", food="meat")`), которая уже, наконец, возвращает по ключу `"meat"` значение 10 из словаря `self.storage`. И это значение возвращается из-под `wrapper#2`.

Поэтому сначала в терминал выводится стока `admin`, а уже затем `alex`.

ВАЖНО: Сначала выполняется тот параметрический декоратор в цепочке, который ближе всех располагается к декорируемой функции. Затем та функция, которая возвращается этим «ближайшим» декоратором декорируется декоратором, который расположен выше в цепочке декораторов и т.д.

Самое главное заключается в том, что функция, которая возвращается последним декоратором (т.е. первым в цепочке/списке декораторов) связывается с декорируемой функцией, как в пример выше `get_food -> wrapper#2`.

30.7. Обобщение по механизму работы декораторов

Если обобщить сказанное выше, то получается, что задекорированная функция ссылается на ту функцию, которую возвращает декоратор, аргумент которого получил ссылку на данную функцию. И происходит это *сразу после* загрузки или импорта модуля. А затем остается только вызвать задекорированную функцию, которая вообще говоря уже ссылается на какую-то другую функцию, которую возвращает декоратор, т.е. если

```
def deco(f):
    def inner():  # замыкание
        print('inner')
    return inner

@deco  # выполняется при загрузке/импорте модуля
def target():
    print('target')
```

то `target = deco(f=target) -> inner`
и, следовательно, `target -> inner` (можно считать, что `target=inner`);
поэтому при вызове `target()` на самом деле вызывается `inner()` и будет выведена строка `'inner'` (см. рис. 7).

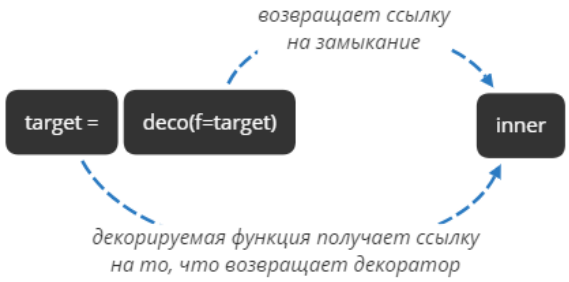


Рис. 7. К вопросу о механизме работы декоратора с вложенной функцией

30.8. Написание декораторов класса

Есть возможность реализовать декораторы класса, правда, они используются гораздо реже. Декораторы класса работают так же, как и декораторы функций, но с классами. Следующий фрагмент кода – это пример декоратора класса, который устанавливает атрибуты для двух классов

```
import uuid

def set_class_name_add_id(klass):
    klass.name = str(klass)         # станет атрибутом класса
    klass.random_id = uuid.uuid4() # станет атрибутом класса
    return klass

@set_class_name_and_id
class SomeClass:
    pass
```

Когда класс будет объявлен и загружен, он установит атрибуты класса `name` и `random_id`

```
>>> SomeClass.name # <class '__main__.SomeClass'>
>>> SomeClass.random_id # UUID('9b75e122-f52a-468b-9796-311d394de2bf')
```

Еще можно таким способом регистрировать объявленные классы

```
import numpy as np


registry = {}


def register_class(cl):
    name = str(cl.__name__)
    random_id = np.random.randn()
    registry[name] = random_id
    return cl # возвращает ссылку на класс


@register_class
class TestClass:
    pass


@register_class
class SimpleClass:
    pass


for key, value in registry.items():
    print(key, value)
```

Иногда бывает удобно применять декораторы класса для декорирования функций или других классов

```
class CountClass:
    def __init__(self, f):
        self.f = f
        self.called = 0


    def __call__(self, *args, **kwargs):
        self.called += 1
        return self.f(*args, **kwargs)


@CountClass
def print_hello():
    print("hello")


print(print_hello.called) # 0
print_hello()
print(print_hello.called) # 1
print_hello()
print(print_hello.called) # 2
```

Здесь при загрузке модуля аргумент `f` метода `__init__()` получает ссылку на декорируемую функцию, а переменная `self.called` инициализируется нулем.

Теперь при каждом вызове функции `print_hello()` будет вызываться метод `__call__()`. При этом значение переменной `self.called` будет увеличиваться на единицу, а затем будет вызываться собственно задекорированная функция.

31. Методы в Python

31.1. Статические методы

Статические методы принадлежат классу, а не его экземпляру, поэтому они фактически не работают и не влияют на экземпляры класса. Вместо этого статический метод оперирует параметрами, которые принимает. Статические методы обычно используются для создания функций полезности, потому что не зависят от состояния классов или объектов.

Использование статических методов предоставляет несколько преимуществ:

- Скорость. Python не должен устанавливать связанный метод для каждого связанного объекта. Связанный метод – это тоже объект, а создание нового объекта отнимает системные ресурсы, хоть и незначительные,
- Удобочитаемость кода. Видя декоратор `@staticmethod`, мы знаем, что метод не зависит от состояния объекта.

К сожалению, Python не всегда способен определить, является ил метод статическим, – это издержки дизайна языка.

31.2. Классовые методы

Классовые методы связаны с классом, а не с экземпляром. Классовые методы очень полезны при динамическом создании экземпляров класса

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def from_fridge(cls, fridge):
        return cls(fridge.get_cheese() + fridge._get_vegetables())
```

32. Замыкания/фабричные функции в Python

Под термином *замыкание* или *фабричная функция* подразумевается объект функции, который сохраняет значения в *объемлющих областях видимости*, даже когда эти области могут прекратить свое существование [1, стр. 488].

В источнике [4, стр. 222] приводится несколько отличное определение[17]: *замыкание* – это вложенная функция с расширенной областью видимости, которая охватывает все *неглобальные* переменные, объявленные в области видимости объемлющей функции, и способная работать с этими переменными даже после того как локальная область видимости объемлющей функции будет уничтожена.

Замыкания и вложенные функции особенно удобны, когда требуется реализовать концепцию отложенных вычислений [2].

[17] Определение содержит авторские правки

Замечание

Все же правильнее «фабрикой функций» называть всю конструкцию из объемлющей и вложенной функций, а «замыканием» – только вложенную функцию

Рассмотрим в качестве примера следующую функцию

```
def maker(N):
    def action(X):
        return X**N  # функция action запоминает значение N в объемлющей области видимости
    return action
```

Здесь определяется внешняя функция, которая просто создает и возвращает вложенную функцию, не вызывая ее. Если вызвать внешнюю функцию

```
>>> f = maker(2)  # запишет 2 в N
>>> f  # <function action at 0x0147280>
```

она вернет ссылку на созданную ею вложенную функцию, созданную при выполнении вложенной инструкции `def`. Если теперь вызвать то, что было получено от внешней функции

```
>>> f(3)  # запишет 3 в X, в N по-прежнему хранится число 2
>>> f(4)  # 4**2
```

будет вызвана вложенная функция, с именем `action` внутри функции `maker`. Самое необычное здесь то, что вложенная функция продолжает хранить число 2, значение переменной `N` в функции `maker` даже при том, что к моменту вызова функции `action` функция `maker` уже *завершила свою работу и вернула управление*.

Когда функция используется как вложенная, в замыкание включается все ее окружение, необходимое для работы внутренней функции [2, стр. 137].

32.1. Области видимости и значения по умолчанию применительно к переменным цикла

Существует одна известная особенность для функций или lambda-выражений: если lambda-выражение или инструкция `def` вложены в цикл внутри другой функции и вложенная функция ссылается на переменную из объемлющей области видимости, которая изменяется в цикле, все функции, созданные в этом цикле, будут иметь одно и то же значение – значение, которое имела переменная на последней итерации [1, стр. 492].

Например, ниже предпринята попытка создать список функций, каждая из которых запоминает текущее значение переменной `i` из объемлющей области видимости

Эта реализация работать НЕ будет

```
def makeActions():
    acts = []
    for i in range(5):  # область видимости объемлющей функции
        acts.append(
            lambda x: i**x  # локальная область видимости вложенной анонимной функции
        )
    return acts

acts = makeActions()
print(acts[0](2))  # вернет 4**2, последнее значение i
print(acts[3](2))  # вернет 4**2, последнее значение i
```

Такой подход не дает желаемого результата, потому что поиск переменной в объемлющей области видимости производится позднее, *при вызове вложенных функций*, в результате все они получат одно и то же значение (значение, которое имела переменная цикла на последней итерации).

Это один из случаев, когда необходимо явно сохранять значение из объемлющей области видимости в виде аргумента со значением по умолчанию вместо использования ссылки на переменную из объемлющей области видимости.

То есть, чтобы фрагмент заработал, необходимо передать текущее значение переменной из объемлющей области видимости в виде значения по умолчанию. Значения по умолчанию вычисляются в момент *создания вложенной функции* (а не когда она *вызывается*), поэтому каждая из них сохранит свое собственное значение `i`

Правильная реализация

```
def makeActions():
    acts = []
    for i in range(5):
        acts.append(
            lambda x, i=i: i**x  # сохранить текущее значение i
        )
    return acts

acts = makeActions()
print(acts[0](2))  # вернет 0**2
print(acts[2](2))  # вернет 2**2
```

Обобщение по вопросу

Значения аргументов по умолчанию вложенных функций, динамически создаваемых в цикле на уровне области видимости объемлющей функции, вычисляются в момент *создания* этих вложенных функций, а не в момент их вызова, поэтому `lambda x, i=i: ...` работает корректно

---

33. Значения по умолчанию изменяемого типа данных в Python

Если у функции есть аргумент, который получает ссылку на *объект изменяемого типа данных* как на значение по умолчанию, то *все вызовы функций* будут ссылаться на один и тот же изменяемый объект[18] (идентификационный номер объекта не изменится).

Это удивляет. И когда говорят об аномальном поведении функции, аргумент которой ссылается на объект изменяемого типа данных, то обычно такое поведение объясняют следующим образом: значения аргументов по умолчанию вычисляются только один раз при загрузке модуля [5, стр. 77]. Однако такое объяснение не вскрывает механизм «разделения» ссылки между вызовами.

Лучше сказать так: если у функции есть аргумент, который ссылается на объект изменяемого типа данных, и в теле функции выполняется какая-то работа с этим изменяемым объектом (т.е. вносятся изменения в объект), то новые вызовы такой функции не сбрасывают значения по умолчанию до тех, которые были вычислены при загрузке модуля. Другими словами, если аргумент функции ссылается на объект изменяемого типа данных и над этим объектом выполняется

[18] По этой причине, как правило, только *объекты неизменяемого типа данных* могут быть значениями по умолчанию. Если значение аргумента функции должно иметь возможность изменяться динамически, то этот аргумент функции инициализируют с помощью `None`, а затем передают ссылку на объект по условию

какая-то работа в теле функции, то каждый новый вызов функции будет изменять этот изменяемый объект в *определении* функции и потому каждый следующий вызов будет оперировать с уже измененным объектом изменяемого типа данных.

Замечание

Значения аргументов по умолчанию для избежания странного поведения функции должны ссылаться на *объекты неизменяемого типа данных*

---

34. Генераторы, сопрограммы в Python

Цепочки вычислений лучше строить на базе генераторов или сопрограмм. Например простую цепочку преобразований можно построить и с помощью функции `reduce` (но лучше так не делать!)

Так лучше не делать!

```
from functools import reduce

test_str = 'python fortran matlab'
pipe_func = (
    str.split,  # разбивает переданную строку по пробелу и возвращает список
    lambda lst: (elem.upper() for elem in lst)  # каждую выделенную на предыдущем этапе
                                                # строку приводит к верхнему регистру
)

main_red = reduce(
    lambda args, f: f(args),
    pipe_func,
    test_str
)

for elem in main_red:
    print(elem)
# PYTHON
# FORTRAN
# MATLAB
```

Для данной задачи конвейер преобразований на базе генераторов может выглядеть так

Правильный вариант решения задачи о конвейерах преобразований

```
def gen_split(s: str):
    '''
    Функция-генератор
    '''
    for elem in s.split():
        yield elem

def gen_upper(sub_str: str):
    '''
    Функция-генератор
    '''
    for elem in sub_str:
        yield elem.upper()

main_gen = gen_upper(gen_split(test_str))  # объект-генератор
for elem in main_gen:
```

```
    print(elem)
# PYTHON
# FORTRAN
# MATLAB
```

35. Библиотека functools

35.1. Каррированные функции с помощью functools.partial

```
from functools import partial

# инициализируем частичную функцию списком
part_f = partial(lambda lst, transform: [transform(elem) for elem in lst],
                 lst=['python', 'fortran'])
# передает преобразование
part_f(transform=str.upper)  # ['PYTHON', 'FORTRAN']
```

Иногда бывает удобно использовать метод partialmethod

```
from functools import partialmethod

class Live:
    def __init__(self):
        self._live = False

    def set_live(self, state: bool):
        self._live = state

    def get_live(self):
        return self._live

    def __call__(self):
        return self.get_live()

    # атрибуты класса
    set_alive = partialmethod(set_live, True) # замораживает метод set_live() со значением True
    set_dead = partialmethod(set_live, False) # замораживает метод set_live() со значением False

live = Live()
live.set_alive()  # изменит значение атрибута self._live на True
                  # метод set_alive() вызовет метод set_live(True)
print(live())  # True: вызовет __call__, а тот в свою очередь get_live()
# разумеется можно вызвать метод set_live(False) и напрямую
```

36. Калибровка классификаторов

Подробности в статье А. Дьяконова «Проблема калибровки уверенности».

Ниже описываются способы оценить качество калибровки алгоритма. Надо сравнить *уверенность* (confidence) и *долю верных ответов* (accuracy) на тестовой выборке.

Если классификатор «хорошо откалиброван» и для большой группы объектов этот классификатор возвращает вероятность принадлежности к положительному классу 0.8, то среди этих объектов будет приблизительно 80% объектов, которые в действительности принадлежат положительному классу. То есть, если для группы точек данных общим числом 100 классификатор

возвращает вероятность положительного класса 0.8, то приблизительно 80 точек на самом деле будут принадлежать положительному классу и доля верных ответов тогда составит 0.8.

36.1. Непараметрический метод гистограммной калибровки (Histogram Binning)

Изначально в методе использовались бины одинаковой ширины, но можно использовать и равномощные бины.

Недостатки подхода:

- число бинов задается наперед,
- функция деформации не непрерывна,
- в «равноширинном варианте» в некоторых бинах может содержаться недостаточное число точек.

Метод был предложен Zadrozny B. и Elkan C. Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers.

36.2. Непараметрический метод изотонической регрессии (Isotonic Regression)

Строится монотонно неубывающая функция деформации оценок алгоритма.

Метод был предложен Zadrozny B. и Elkan C. Transforming classifier scores into accurate multiclass probability estimates.

Функция деформации по-прежнему не является непрерывной.

36.3. Параметрическая калибровка Платта (Platt calibration)

Изначально этот метод калибровки разрабатывался только для метода опорных векторов, оценки которого лежат на вещественной оси (по сути, это расстояния до оптимальной разделяющей классы прямой, взятые с нужным знаком). Считается, что этот метод не очень подходит для других моделей.

Предложен Platt J. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods.

36.4. Логистическая регрессия в пространстве логитов

36.5. Деревья калибровки

Стандартный алгоритм строит строит суперпозицию дерева решений на исходных признаках и логистических регрессий (каждая в своем листе) над оценками алгоритма:

- Построить на исходных признаках решающее дерево (не очень глубокое),
- В каждом листе – обучить логистическую регрессию на одном признаке,
- Подрезать дерево, минимизируя ошибку.

36.6. Температурное шкалирование (Temperature Scaling)

Этот метод относится к классу DL-методов калибровки, так как он был разработан именно для калибровки нейронных сетей. Метод представляет собой простое многомерное обобщение шкалирования Платта.

37. Приемы работы с менеджером пакетов conda

37.1. Создание виртуального окружения

Создать виртуальное окружение dashenv

```
conda create --name dashenv
```

Создать виртуальное окружение с указанием версии Python

```
conda create --name testenv python=3.6
```

Создать виртуальное окружение с указанием пакета

```
conda create --name testenv scipy
```

Создать виртуальное окружение с указанием версии Python и нескольких пакетов

```
conda create --name testenv python=3.6 scipy=0.15.0 astroid babel
```

Замечание

Рекомендуется устанавливать сразу несколько пакетов, чтобы избежать конфликта зависимостей

Для того чтобы при создании нового виртуального окружения не требовалось каждый раз устанавливать базовые пакеты, которые обычно используются в работе, можно привести их список в конфигурационном файле .condarc в разделе create_default_packages

.condarc

```
ssl_verify: true
channels:
 - conda-forge
 - defaults
report_errors: true
default_python:
create_default_packages:
 - matplotlib
 - numpy
 - scipy
 - pandas
 - seaborn
```

Если для текущего виртуального окружения не требуется устанавливать пакеты из набора по умолчанию, то при создании виртуального окружения следует указать специальный флаг --no-default-packages

```
conda create --no-default-packages --name testenv python
```

Создать виртуальное окружение можно и из файла environment.yml (первая строка этого файла станет именем виртуального окружения)

environment.yml

```
name: stats2
channels:
 - conda-forge
 - defaults
dependencies:
 - python=3.6   # or 2.7
```

```
 - bokeh=0.9.2
 - numpy=1.9.*
 - nodejs=0.10.*
 - flask
 - pip:
  - Flask-Testing
```

```
conda env create -f environment.yml
```

При создании виртуального окружения можно указать путь до целевой директории, где будут размещаться файлы окружения. Следующая команда создаст виртуальное окружение в поддиректории текущей рабочей директории `envs`[19]

```
conda create --prefix ./envs jupyterlab matplotlib
```

С помощью файла спецификации можно создать *идентичное виртуальное окружение* (i) на той же платформе операционной системы, (ii) на той же машине, (iii) на какой-либо другой машине (перенести настройки окружения).

Для этого предварительно требуется создать собственно файл спецификации

```
conda list --explicit > spec-file.txt
```

Имя файла спецификации может быть любым. Файл спецификации обычно не является кросс-платформенным и поэтому имеет комментарий в верхней части файла (`#platform: osx-64`), указывающий платформу, на которой он был создан.

Теперь для того чтобы *создать* окружение достаточно воспользоваться командой

```
conda create --name myenv --file spec-file.txt
```

Файл спецификации можно использовать для установки пакетов в существующее окружение

```
conda install --name myenv --file spec-file.txt
```

37.2. Активация/деактивация виртуального окружения

Активировать виртуальное окружение `dashenv`

```
conda activate dashenv
```

Активировать виртуальное окружение в случае, когда оно создавалось с `--prefix`, можно указав полный путь до окружения

```
conda activate E:\[WorkDirectory]\[Python_projects]\directory_for_experiments\envs
```

В этом случае в строке приглашения командной оболочки по умолчанию будет отображаться полный путь до окружения. Чтобы заменить длинный префикс в имени окружения на более удобный псевдоним достаточно использовать конструкцию

```
conda config --set env_prompt ({name})
```

которая добавит в конфигурационный файл `.condarc` следующую строку

.condarc

```
...
env_prompt: ({name})
```

[19] В данном случае чтобы удалить виртуальную среду достачно просто удалить директорию `envs`

и теперь имя окружения будет `(envs)`.

Деактивировать виртуальное окружение

```
conda deactivate
```

37.3. Обновление виртуального окружения

Обновить виртуальное окружение может потребоваться в следующих случаях:

- обновилась одна из ключевых зависимостей,
- требуется добавить пакет (добавление зависимости),
- требуется добавить один пакет и удалить другой.

В любом из этих случаев все что нужно для того чтобы обновить виртуальное окружение это просто обновить файл `environment.yml`[20], а затем запустить команду

```
conda env update --prefix ./envs --file environment.yml --prune
```

Опция `--prune` приводит к тому, что `conda` удаляет все зависимости, которые больше не нужны для окружения.

37.4. Вывод информации о виртуальном окружении

Вывести список доступных виртуальных окружений

```
conda env list
```

Вывести список пакетов, установленных в указанном окружении

```
conda list --name myenv
```

Вывести информацию по конкретному пакету указанного окружения

```
conda list --name dashenv matplotlib
```

37.5. Удаление виртуального окружения

Удалить виртуальное окружение `heroku_env`

```
conda env remove --name heroku_env
```

37.6. Экспорт виртуального окружения в `environment.yml`

Экспортировать активное виртуальное окружение в yml-файл

```
conda env export > environment.yml
```

[20] Этот файл должен находится в той же директории что и директория окружения `envs`

38. Инструмент автоматического построения дерева проекта под задачи машинного обучения

Для автоматизации построения типового (или кастомизированного) дерева проекта по машинному обучению и анализу данных удобно использовать cookicutter.

На операционную систему под управлением Windows cookicutter можно установить с помощью менеджера пакетов pip

```
pip install cookiecutter
```

а на операционную систему под управлением MacOS X с помощью менеджера brew

```
brew install cookiecutter
```

В самом простом случае cookicutter можно использовать как утилиту командной строки. Например для того чтобы создать проект по шаблону для задач машинного обучения достаточно сделать следующее

```
cookiecutter https://github.com/drivendata/cookiecutter-data-science
```

Утилита предложит ответить на несколько вопросов (название репозитория, имя автора и т.д.), а затем создаст дерево проекта.

39. Управление локальными переменными окружения проекта

Для того чтобы создать *локальные переменные проекта*[21] достаточно разместить пары вида «ключ=значение» в файле .env, а затем прочитать его с помощью специальной библиотеки dotenv https://pypi.org/project/python-dotenv/. Например

```
#.env в текущей директории проекта
EMAIL = leor.finkelberg@yandex.ru
POSTGRESQL_PASSWORD = Evdimonia
```

```
import os
from pathlib import Path
from dotenv import load_dotenv

dotenv_path = Path(__file__).resolve().parents[0].joinpath('.env')
print(f'[INFO] path: {dotenv_path}')  # [INFO] path: E:\[WorkDirectory]\[Python_projects]\
    directory_for_experiments\.env

load_dotenv(dotenv_path)  # загрузить .env

# извлекать значения локальных переменных окружения проекта можно с помощью `os.getenv(key)`
# или `os.environ.get(key)`
for key in (s.upper() for s in ('email', 'postgresql_password')):
    print('[INFO] from file `.env`({}) -> {}'.format(key, os.getenv(key)))
```

40. Приемы работы с модулем subprocess

Ниже приводится пример использования модуля subprocess для отыскания самого большого файла в git-репозитории

[21] То есть переменные, привязанные к текущему проекту

```
import os
import subprocess
import pathlib
from subprocess import Popen, PIPE, STDOUT

# --- объявление функций: begin
def popen_2_str(cmd: str, shell=True, universal_newlines=True, stdout=PIPE) -> str:
    return Popen(cmd, shell=shell,
                 universal_newlines=universal_newlines,
                 stdout=stdout).stdout.read().strip()

def stat(filename):
    res = popen_2_str(f"stat {filename}")
    print(f'>>> Statistic:\n{res}')

def summary(commits):
    print(f'### Summary ({__file__}) ###:\n>>> idx-file name: {idx_file}'
          f'\n>>> SHA blob: {shablob}\n>>> Commits:')
    print(commits)
# --- объявление функций: end

GIT_PATH = pathlib.Path('.git/objects/pack/')

# тоже самое что и `git gc &> /dev/null`
exit_code = subprocess.call("git gc", shell=True,
                            stdout=open(os.devnull, 'w'), stderr=STDOUT)

if not exit_code:
    # возвращает имя idx-файла
    idx_file = popen_2_str(f"ls -l {GIT_PATH} | grep -iE '*.idx' "
                           f"| awk -F ' ' '{{ print $9 }}'")
    # возвращает абсолютный путь до idx-файла
    abs_path_idx_file = pathlib.Path.joinpath(GIT_PATH, idx_file)
    if os.path.exists(abs_path_idx_file):
    # возвращает SHA <<большого>> файла
        shablob = popen_2_str(f"git verify-pack -v {abs_path_idx_file} | sort -k 3 -n "
                              f"| tail -n 1 | awk -F ' ' '{{ print $1 }}'")
        # возвращает имя файла по его SHA
        filename = popen_2_str(f"git rev-list --objects --all | grep {shablob} "
                               f"| awk -F ' ' '{{ print $2 }}'")
        # возвращает коммиты, связанные с данным файлом
        commits = popen_2_str(f"git log --oneline -- {filename}")
        summary(commits)
        stat(filename)
    else:
        print(f"File {abs_path_idx_file} not found...")
else:
    print('Something went wrong.')
```

41. Решающие деревья и сопряженные вопросы

41.1. Коэффициент Джини

Коэффициент Джини[22] (Gini impurity) это просто вероятность неверной маркировки в узле случайно выбранного образца (для чистых листьев коэффициент Джини равен 0)

$$I_G(n) = 1 - \sum_{i=1}^{J} p_i^2, \tag{1}$$

где p_i – частоты представителей разных классов в листе дерева.

К примеру, если решается задача бинарной классификации ($J = 2$) на выборке из 6 объектов и в данном расщеплении в один класс попали 2 объекта, а в другой 4, то индекс Джини будет равен

$$I_G(n) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = 0,444. \tag{2}$$

41.2. Случайный лес

Случайный лес – это модель, представляющая ансамбль решающих деревьев, дополненная двумя концепциями:

- концепцией бутрстапированных выборок,
- концепцией случайных подпространств.

Хотя каждое решающее дерево может иметь большой разброс по отношению к определенному набору тренировочных данных, обучение деревьев на разных наборах образцов позволяет снизить общий разброс леса.

42. Анализ временных рядов

42.1. Признаки на временных рядах

Можно выделить следующие несколько групп признаков, которые можно вычислить на временных рядах:

- признаки на основе коэффициентов автокорреляции и частных коэффицентов автокорреляции,
- оптимальное значение параметра λ преобразования Бокса-Кокса,
- коэффициент Херста,
- количество раз, когда временной ряд пересекает свою собственную медиану,
- признаки, рассчитываемые на основе STL-компонент разложения временного ряда,
- дисперсия дисперсии, рассчитанных по наблюдениям из непересекающихся временных отрезков,
- дивергенция Кульбака-Лейблера в следующих друг за другом отрезках,
- спектральная энтропия ряда,
- дисперсия средних значений,
- минимальное число дифференциирований временного ряда, необходимое для достижения его стационарности.

[22] Еще говорят индекс Джини или загрязненность Джини

42.2. Прогнозирование временных рядов. Метод имитированных исторических прогнозов

При разбиении данных на обучающую и проверочную выборки важно помнить о том, как модель в итоге будет использоваться на практике. Так, при выполнении предсказаний для той же генеральной совокупности, из которой получены исходные данные (*интерполяция*), достаточным может оказаться простое случайное разбиение данных. В случаях же, когда модель предназначена для прогнозирования будущего (*экстраполяция*), более точную оценку ее предсказательных свойств можно получить только если проверочная выборка содержит данные из будущего (например, если исходные данные охватывают период в два года, то модель можно было бы обучить на данных первого года, а затем проверить ее обобщающую способность на данных второго года).

Стандартным методом оценки качества нескольких альтернативных моделей является перекрестная проверка. Суть этого метода сводится к тому, что исходные обучающие данные случайным образом разбиваются на k блоков, после чего модель k раз обучается на $k-1$ блоках, а оставшийся блок каждый раз используется для проверки качества предсказаний на основе той или иной подходящей случаю метрики. Полученная таким образом средняя метрика будет хорошей оценкой качества предсказаний модели на новых данных.

К сожалению, в случае с моделями временных рядов такой способ выполнения перекрестной проверки будет бессмысленным и не отвечающим стоящей задаче. Поскольку во временных рядах, как правило, имеет место тесная корреляция между близко расположенными наблюдениями, мы не можем просто разбить такой ряд случайным образом на k частей – это приведет к потере указанной корреляции. Более того, в результате случайного разбиения данных на несколько блоков может получиться так, что в какой-то из итераций мы построим модель преимущественно по недавним наблюдениям, а затем оценим ее качество на блоке из давних наблюдений. Другими словами, мы построим модель, которая будет предсказывать прошлое, что не имеет никакого смысла – ведь мы пытаемся решить задачу по предсказаниям будущего.

Для решения описанной проблемы при работе с временными рядами применяют несколько модификаций перекрестной проверки. Например, в пакете Prophet, реализован так называемый метод «имитированных исторических прогнозов» (simulated historical forecast).

Метод имитированных исторических прогнозов https://r-analytics.blogspot.com/2019/10/prophet-shf.html. Для создания модели временного ряда мы используем данные за определенный исторический отрезок времени. Далее по полученной модели рассчитываются прогнозные значения для некоторого интересующего нас промежутка времени (горизонта прогноза) в будущем. Такая процедура повторяется каждый раз, когда необходимо сделать новый прогноз.

В пределах отрезка с исходными обучающими данными выбирают k точек отсчета (в терминологии Prophet), на основе которых формируются блоки данных для выполнения перекрестной проверки: все исторические наблюдения, предшествующие k-ой точке отсчета (а также сама эта точка), образуют обучающие данные для подгонки соответствующей модели, а H исторических наблюдений, следующих за точкой отсчета, образуют *прогнозный горизонт*. Расстояние между точками отсчета называется периодом и по умолчанию составляет $H/2$. Обучающие наблюдения в первом из k блоков образуют так называемый начальный отрезок. В Prophet длина этого отрезка по умолчанию составляет $3\,H$, однако этот параметр можно изменить.

Каждый раз после подгонки модели на обучающих данных из k-ого блока рассчитываются предсказания для прогнозного горизонта того же блока, что позволяет оценить качество прогноза с помощью подходящей метрики. Значения этой метрики, усредненные по каждой дате

прогнозных горизонтов каждого блока, в итоге дают оценку качества предсказаний, которую можно ожидать от модели, построенной *по всем исходным обучающим данным.*

42.3. Обнаружение аномалий во временных рядах

Обнаружение аномалий относится к поиску непредвиденных значений (паттернов) в потоках данных. Аномалия (выброс, ошибка, отклонение или исключение) – это отклонение поведение системы от стандартного (ожидаемого).

Аномалии могут возникать в данных самой различной природы и структуры в результате технических сбоев, аварий, преднамеренных взломов и т.д.

Аномалии в данных могут быть отнесены к одному из трех основных типов [7]:

- *Точечные аномалии*: возникают в ситуации, когда отдельный экземпляр данных может рассматриваться как аномальный по отношению к остальным данным; большинство существующих методов создано для распознавания точечных аномалий,
- *Контекстуальные аномалии*: наблюдаются, если экземпляр данных является аномальным лишь в определенном контексте (данный вид аномалий также называется условным)
  - контекстуальные атрибуты используются для определения контекста (или окружения) для каждого экземпляра; во временных рядах контекстуальным атрибутом является время, которое определяет положение экземпляра в целой последовательности; контекстуальным атрибутом также может быть положение в пространстве или более сложные комбинации свойств,
  - поведенческие атрибуты определяют не контекстуальные характеристики, относящиеся к конкретному экземпляру данных,
- *Коллективные аномалии*: возникают, когда последовательность связанных экземпляров данных (например, фрагмент временного ряда) является аномальной по отношению к целому набору данных. Отдельный экземпляр данных в такой последовательности может не являться отклонением, однако совместное появление таких экземляров является коллективной аномалией; кроме того, если точечные или контекстуальные аномалии могут наблюдаться в любом наборе данных, то коллективные наблюдаются только в тех, где данные связанны между собой.

Часто для решения задачи поиска аномалий требуется набор данных, описывающих систему. Каждый экземпляр в нем описывается меткой, указывающей, является ли он нормальным или аномальным. Таким образом, множество экземпляров с одинаковой меткой формируют соответствующий класс.

Создание подобной промаркированной выборки обычно проводится вручную и является трудоемким и дорогостоящим процессом. В некоторых случаях получить экземпляры аномального класса невозможно в силу отсутствия данных и возможных отклонениях в системе, в других могут отсутствовать метки обоих классов. В зависимости от того, какие классы данных используются для реализации алгоритма, методы поиска аномалий могут выполняться в одном из трех перечисленных режимов:

- **Режим распознавания с учителем**. Данная методика требует наличия обучающей выборки, полноценно представляющей систему и включающей экземпляры данных нормального и аномального классов. Работа алгоритма происходит в два этапа: обучение и распознование. На первом этапе строится модель, с которой в последствии сравниваются экземпляры,

не имеющие метки. В большинстве случаев предполагается, что *данные не меняют свои статистические характеристики*, иначе возникает необходимость изменять классификатор. Основной сложностью алгоритмов, работающих в режиме распознования с учителем, является формирование данных для обучения. Часто аномальный класс представлен значительно меньшим числом экземпляров, чем нормальный, что может приводить к неточностям в полученной модели. В таких случаях применяется *искусственная генерация аномалий.*

Режим распознования частично с учителем. Исходные данные при этом подходе представляют только нормальный класс. Обучившись на одном классе, система может определять принадлежность новых данных к нему, таким образом, определяя противоположенный. Алгоритмы, работающие в режиме распознования частично с учителем, не требуют информации об аномальном классе экземпляров, вследствие чего они шире применимы и позволяют распознавать отклонения в отсутствие заранее определенной информации о них.

Режим распознавания без учителя. Применяется при отсутствии априорной информации о данных. Алгоритмы распознавания в режиме без учителя базируются на предположении о том, что аномальные экземпляры встречаются гораздо реже нормальных. Данные обрабатываются, наиболее отдаленные определяются как аномалии. Для применения этой методики должен быть доступен весь набор данных, т.е. она не может применяться в режиме реального времени.

Метод опорных векторов[23] применяется для поиска аномалий в системах, где нормальное поведение представляется только одним классом. Данный метод определяет границу региона, в котором находятся экземпляры нормальных данных. Для каждого исследуемого экземпляра определяется, находится ли он в определенном регионе. Если экземпляр оказывается вне региона, он определяется как аномальный.

Пример использования одноклассового метода опорных векторов

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import OneClassSVM


def transform_to_zero_minus_one(arr):
    return np.where(arr < 0, arr, 0)

N = 250  # длина временного ряда
scale = 50  # масштаб меток для графика аномалий

# подготавливаем тренировочный и тестовый набор данных
data_train = np.random.RandomState(42).randn(N)
data_test = np.random.RandomState(2).randn(int(0.1*N))
data_train[[40, 50, 80]] *= 100
data_test[[2, 5]] *= 50

# обучаем классификатор и готовим предсказания
clf = OneClassSVM(nu=0.03).fit(data_train.reshape(-1, 1))
predicted_anomalies = clf.predict(data_test.reshape(-1, 1))

plt.plot(data_test,
    marker = '.',
    markersize = 12,
    markerfacecolor = 'w',
```

[23] В `sklearn` есть реализация одноклассового метда опорных векторов `OneClassSVM` (позволяет задать долю аномальных объектов в выборке с помощью параметра `nu`)

```
    color = 'k',
    label='тестовый набор данных')

plt.bar(np.arange(0, data_test.shape[0]),
    transform_to_zero_minus_one(predicted_anomalies)*scale,
    alpha = 0.5,
    color = 'b',
    label='аномалии')
plt.legend()
```

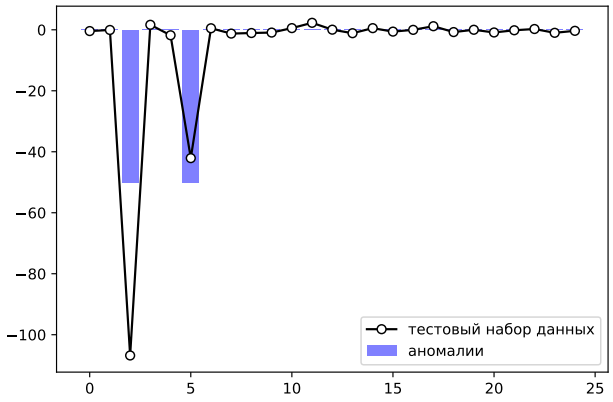


Рис. 8. Пример детектирования аномалий на тестовой наборе данных

Кластеризация. Данная методика предполагает группировку похожих экземпляров в кластеры и не требует знаний о свойствах возможных отклонений: нормальные данные образуют большие плотные кластеры, а аномальные – маленькие и разрозненные. Одной из простейших реализацией подхода на основе кластеризации является алгоритм метода k-средних.

При использовании методов статистического анализа исследуется процесс, строится его профиль (статистическая модель), которые затем сравнивается с реальным поведением. Если разница в реальном и предполагаемом поведении системы, определяется заданной функцией аномальности, выше установленного порога, делается вывод о наличии отклонений. Применяется предположении о том, что нормальное поведение системы будет находиться в зоне высокой вероятности, в то время как выбросы – в зоне низкой.

Данный класс методов удобен тем, что не требует заранее определенных знаний о виде аномалии. Однако сложности могут возникать в определении точного статистического распределения и порога.

Методы статистического анализа подразделяются на две группы:

- *Параметрические методы.* Предполагают, что нормальные данные генерируются параметрическим распределением с параметрами θ и функцией плотности вероятности $\mathbb{P}(x, \theta)$, где x – наблюдение. Аномалия является обратной функцией распределения. Эти методы часто основываются на Гауссовской или регрессионной модели, а также их комбинации.
- Непараметрические методы. Предполагается, что структура модели не определена априорно, вместо этого она определяется из предоставленных данных. Включает методы на основе гистограмм или функции ядра.

Базовый алгоритм поиска аномалий с применением гистограмм включает два этапа. На первом этапе происходит построение гистограммы на основе различных значений выбранной характеристики для экземпляров тренировочных данных. На втором этапе для каждого из ис-

следуемых экземпляров определяется принадлежность к одному из столбцов гистограммы. Не принадлежащие ни к одному из столбцов экземпляры помечаются как аномальные.

Алгоритм ближайшего соседа. Для использования данной методики необходимо определить понятие расстояния (меры похожести) между объектами. Примером может быть евклидово расстояние.

Два основных подхода основаваются на следующих предположениях:

- Расстояние до k-ого ближайшего соседа. Для реализации этого подхода расстояние до ближайшего объекта определяется для каждого тестируемого экземпляра класса. Экземпляр, являющийся выбросом, наиболее отдален от ближайшего соседа.
- Использование относительной плотности основано на оценке плотности окрестности каждого экземпляра данных. Экземпляр, который находится в окрестности с низкой плотностью, оценивается как аномальный, в то время как экземпляр в окрестности с высокой плотностью оценивается как нормальный. Для данного экземпляра данных расстояние до его k-ого ближайшего соседа эквивалентно радиусу гиперсферы с центром в данном экземпляре и содержащей k остальных экземпляров.

Выявление аномалий в режиме реального времени может потребовать дополнительной модификации методов. Наиболее простым в реализации является *алгоритм скользящего окна.*

Данная методика используется для временных рядов, которые разбиваются на некоторое число последовательностей – окон. Необходимо выбрать окно фиксированной длины, меньшей чем длина самого временного ряда, чтобы захватить аномалию в процессе скольжения. Поиск аномальной последовательности осуществляется при помощи скольжения окна по всему ряду с шагом, меньшим длины окна.

42.4. Приемы работы с библиотекой `Prophet`

Установить библиотеку можно с помощью менеджера пакетов `conda`

```
conda install -c conda-forge fbprophet
```

`Prophet` была разработана для прогнозирования большого числа различных бизнес-показателей и строит неплохие baseline-прогнозы.

По сути Prophet-модель представляет собой аддитивную регрессионную модель

$$y(t) = g(t) + s(t) + h(t) + \varepsilon_t,$$

где $g(t)$ – тренд (может быть представлен *кусочно-линейной* или *логистической функцией*[24]); $s(t)$ – сезонная компонента, отвечающая за периодические/квазипериодические изменения, связанные с *недельной* и *годовой сезонностью*[25]; $h(t)$ – отвечает за аномальные дни (праздники, Black Fridays и т.д.); ε – содержит информацию, которая не учтена моделью.

Подробнее о математической стороне вопроса рассказывается в статье https://peerj.com/preprints/3190/. К слову, в этой статье качество моделей оценивается с помощью MAPE и MAE. MAPE (mean absolute percentage error) – это средняя абсолютная ошибка нашего прогноза. Пусть y_i – значение целевого вектора, а $\hat{y}_i$ – это соответствующий этой величине прогноз модели. Тогда $\varepsilon_i = y_i - \hat{y}_i$ – это ошибка прогноза, а $p_i = \frac{\varepsilon_i}{y_i}$ – относительная ошибка прогноза.

[24] Логистическая функция удобна для моделирования роста с насыщением, когда при увеличении показателя снижается темп его роста

[25] Моделируется с помощью рядов Фурье

Таким образом средняя абсолютная ошибка выражается следующей формулой

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}|p_i|.$$

MAPE часто используется для оценки качества, поскольку эта величина относительная и по ней можно сравнивать качество даже на различных наборах данных.

Библиотека `Prophet` имеет интерфейс, похожий на интерфейс `sklearn`: сначала мы создаем модель, затем вызываем у нее метод `fit` и затем получаем прогноз. На вход метод `fit` получает объект DataFrame с двумя столбцами: `ds` – временная метка (поле должно иметь тип `date` или `timestamp`), и целевой показатель `y`.

Разработчики рекомендуют делать предсказания по нескольким месяцам данных (в идеале год и более).

Пример

```
import fbprophet
from fbprophet.plot import add_changepoints_to_plot
import pandas as pd
import matplotlib.pyplot as plt


data_all = pd.read_csv('AirPassengers.csv')
# в наборе данных, на котором обучается модель обязательно должны быть столбцы `ds` и `y`
data_all = data_all.rename(columns={ 'Month' : 'ds', 'Passengers' : 'y' })
data_all['ds'] = pd.to_datetime(data_all['ds'])
M = 100
data_train = data_all[:M]  # обучающий набор данных
data_test = data_all[M:]   # тестовый набор данных

model = fbprophet.Prophet(
    changepoint_prior_scale=0.035,
    weekly_seasonality=True,
    yearly_seasonality=True,
    seasonality_mode='multiplicative'
)
model.fit(data_train)  # обучение модели

future_points = data_test.shape[0]  # число точек прогнозного горизонта
# преобразование в точек в метки, имеющие смысл времени
time_points_for_predict = model.make_future_dataframe(future_points, freq='M')
forecast = model.predict(time_points_for_predict)  # прогноз

fig, ax = plt.subplots(figsize=(8, 4))
plt.plot(data_train['ds'], data_train['y'], marker='.', label='Train data')
plt.plot(data_test['ds'], data_test['y'], marker='.', color='k', label='Test data')
plt.plot(forecast['ds'][M:], forecast['yhat'][M:], marker='.', color='r', label='Predict')
plt.axvspan(forecast['ds'][M],forecast['ds'][M+43], facecolor='grey', alpha=0.25)
plt.legend()
# добавить точки перегиба
a = add_changepoints_to_plot(fig.gca(), model, forecast)
```

С помощью конструкции `model.plot_components(forecast);` можно посмотреть компоненты временного ряда (тренд, недельную и годовую сезонность).

С помощью библиотеки `Prophet` можно учитывать эффекты «праздников». Под термином «праздник» здесь понимается как «настоящие» официальные праздничные и выходные дни (на-

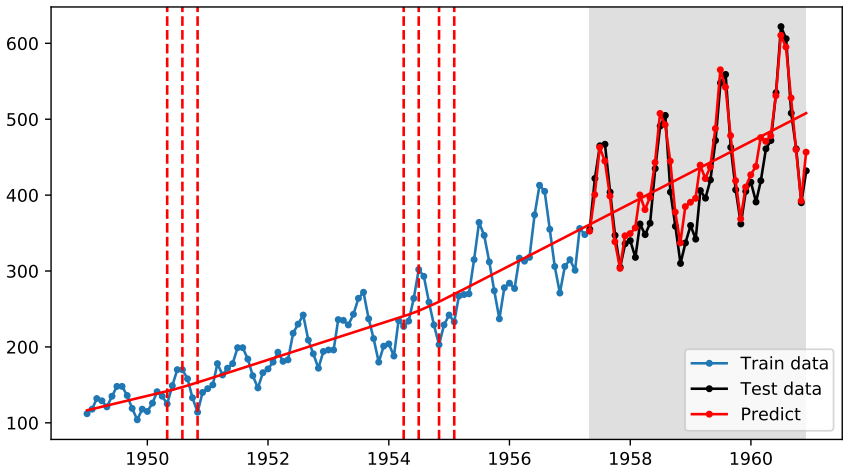


Рис. 9. Пример использования библиотеки `fbprophet`

пример, Новый Год, Рождество и пр.), так и другие события, во время которых свойства моделируемой зависимой переменной существенно изменяются (спортивные или культурные мероприятия, природные явления и пр.).

Для добавления эффектов «праздников» в Prophet-модель необходимо сначала создать отдельную таблицу, содержащую как минимум два обязательных столбца: `holiday` и `ds`. Важно, чтобы эта таблица охватывала как исторический период, на основе которого происходит обучение модели, так и период в будущем, для которого необходимо сделать прогноз. Например, если какое-то важное событие встречается в обучающих данных, то его следует указать и для прогнозного периода (при условии, конечно, что мы ожидаем повторение этого события в будущем, и что дата этого события входит в прогнозный период).

Параметры класса `Prophet`:

- `growth`: тип тренда. Принимает два возможных значения: linear и logistic,
- `changepoints`: список временных меток, соответствующих точкам излома тренда (т.е. датам, когда, как предполагается, произошли существенные изменения в тренде временного ряда). Если этот список не задан, то такие точки излома будут вычисляться автоматически,
- `n_changepoints`: предполагаемое количество, точек излома (по умолчанию 25). Если параметр `changepoints` задан, то параметр `n_changepoints` будет проигнорирован. Если же `changepoints` не задан, то `n_changepoints` потенциальных точек излома будут распределены равномерно в предалах исторического отрезка, заданного параметром `changepoint_range`,
- `changepoint_range`: доля исторических данных (начиная с самого первого наблюдения), в пределах которых будут оценены точки излома. По умолчанию составляет 0.8 (т.е. 80% наблюдений),
- `yearly_seasonality`: параметр настройки годовой сезонности (т.е. закономерных колебаний в пределах года). Принимает следующие возможные значения: auto, True, False или количество членов ряда Фурье, с помощью которого аппроксимируются компоненты годовой сезонности,
- `weekly_seasonality`: параметр настройки недельной сезонности (т.е. закономерных колебаний в пределах недели). Возможные значения те же, что и у `yearly_seasonality`,
- `daily_seasonality`: параметр настройки дневной сезонности (т.е. закономерных колебаний в пределах дня). Возможные значения те же, что и у `yearly_seasonality`,

- `holidays`: объект-DataFrame со столбцами `holiday` и `ds`. По желанию можно добавить еще два столбца – `lower_window` и `upper_window`, которые задают отрезок времени вокруг соответствующего события,
- `seasonality_mode`: режим моделирования сезонных компонент. Принимает два возможных значения: additive и multiplicative,
- `seasonality_prior_scale`: параметр, задающий «силу» сезонных компонентов модели (10 по умолчанию). Более высокие значения приведут к более «гибкой» модели, а низкие – к модели со слабо выраженными сезонными эффектами,
- `holidays_prior_scale`: параметр, задающий выраженность эффектов «праздников» и других важных событий (по умолчанию 10). Если объект-DataFrame, передаваемый в параметр `holidays`, имеет столбец `prior_scale`, то параметр `holidays_prior_scale` будет проигнорирован,
- `changepoint_prior_scale`: параметр, задающий «гибкость» автоматического механизма обнаружения «точек излома» (по умолчанию 0.05). Более высокие значения позволят иметь больше таких точек излома,
- `mcmc_samples`: целое число (по умолчанию 0). Если > 0, то параметры модели будут оценены путем *полного байесовского анализа* с использованием указанного числа итераций алгоритма MCMC. Если 0, тогда используется *оценка апостериорного максимума* (MAP),
- `interval_width`: число, определяющее ширину доверительного интервала для предсказанных моделью значений (по умолчанию 0.8, что соответствует 80%-ному интервалу),
- `uncertainty_samples`: количество итераций для оценивания доверительных интервалов (по умолчанию 1000).

Оценка максимума апостериорной вероятности (maximum aposteriori probability, MAP) тесно связана с *методом наибольшего правдоподобия* (ML), но дополнительно при оптимизации использует априорное распределение величины, которую оценивает.

Можно записать

$$\hat{\theta}_{\mathrm{MAP}}(x) = \arg\max_{\theta} f(x|\theta)g(\theta),$$

где $f(x|\theta)$ – функция правдоподобия, $g(\theta)$ – априорная плотность распределения оцениваемого параметра θ.

Пример. Предположим, что у нас есть последовательность $(x_1, \ldots, x_n)$ i.i.d (независимых и одинаково распределенных) $N(\mu, \sigma_v^2)$ случайных величин и априорное распределение μ задано $N(0, \sigma_m^2)$. Требуется найти MAP-оценку μ.

Функция, которую нужно максимизировать задана

$$\pi(\mu)L(\mu) = \frac{1}{\sqrt{2\pi\sigma_m}} \exp\left(-\frac{1}{2}\Big(\frac{\mu}{\sigma_m}\Big)^2\right) \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_v}} \exp\left(-\frac{1}{2}\Big(\frac{x_j - \mu}{\sigma_v}\Big)^2\right).$$

Теперь остается записать логарифм этой функции, затем найти производную по оцениваемому параметру, приравнять полученную производную нулю и, наконец, выразить искомый параметр. Что в итоге даст

$$\hat{\mu}_{\mathrm{MAP}} = \frac{\sigma_m^2}{n\sigma_m^2 + \sigma_v^2} \sum_{j=1}^{n} x_j.$$

42.5. Преобразование нестационарного временного ряда в стационарный

Чтобы превратить нестационарный ряд в стационарный можно использовать следующие общие приемы:

- выделить в структуре временного ряда тренд и сезонную компоненту, затем удалить их исходного временного ряда; построить прогноз на временном ряду, приведенном к стационарному, а после вернуть эти компоненты в прогноз,
- провести сглаживание (за несколько часов, за неделю и т.п.); в простейших случаях, когда период временного четко определен, можно пользоваться обычным скользящим средним, но в более сложных случаях, когда период сложно подсчитать, следует пользоваться *экспоненциально-взвешенным скользящим средним* `time_series.ewm(halflife=12).mean()`.

42.6. Стабилизация дисперсии

Для временных рядов с *монотонно* меняющейся дисперсией можно использовать стабилизирующие преобразования. Например, *логарифмирование* `np.log(ts)`.

Если исходный временной ряд не проходит тест на *гауссовость*, то можно либо воспользоваться непараметрическими методами, либо обратиться к специальным приемам, позволяющим преобразовать исходную ненормальную статистику в нормальную.

Среди множества таких методов преобразований одним из лучших (при неизвестном типе распределения) считается *преобразование Бокса-Кокса*[26], то есть это преобразование *нормализует* данные (делает их более гауссовскими)

$$\hat{y}_i = \begin{cases} \log y_i, & \lambda = 0, \\ (y_i^\lambda - 1)/\lambda, & \lambda \neq 0 \end{cases}$$

для исходной последовательности $y = \{y_1, \ldots, y_n\}$, $y_i > 0$, $i = (1, \ldots, n)$.

Пример использования преобразования Бокса-Кокса приведен на рис. 10. Такого рода преобразования полезны в ситуациях, связанных с проблемой *гетероскедостичности* (непостоянная дисперсия), или в ситуациях, где требуется *гауссовость* данных.

Параметр λ можно подбирать так, чтобы дисперсия была как можно более стабильной во времени. Прямое и обратное преобразования Бокса-Кокса реализованы в библиотеках `scipy` и `statsmodels`

```
from scipy.stats import boxcox
from statsmodels.tsa.hotwinters import (
                                        #boxcox,
                                        inv_boxcox
                                        )

# пользуемся готовым решением для обратного преобразования Бокса-Кокса
lmbda = 0.25
arr = np.array([3, 5, 10])
# можно задать значение лямбда самому или позволить вычислить его
arr_transformed = boxcox(arr, lmbda) # array([1.26429605, 1.98139512, 3.11311764])
arr_transformed, lmbda_compute = boxcox(arr)  # здесь lmbda вычисляется
                                              # с помощью максимизации логарифма правдоподобия
```

[26] Степенные преобразования – это семейство параметрических, монотонных преобразований, целью которых является отображение данных из произвольного распределения в близкое к гауссовскому распределению таким образом, чтобы *стабилизировать дисперсию* и *минимизировать ассиметрию*

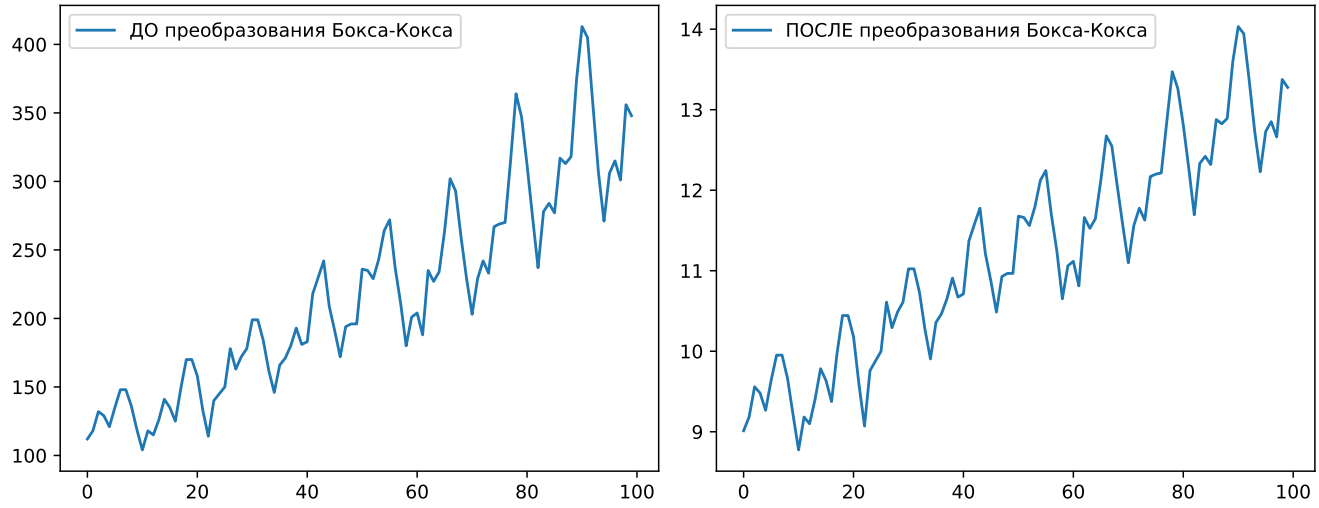


Рис. 10. Влияние преобразования Бокса-Кокса на временной ряд
с изменяющейся во времени дисперсией

```
inv_boxcox(arr_transformed, lmbda)  # array([ 3.,  5., 10.])

# пишем свою реализацию обратного преобразования Бокса-Кокса
def invboxcox(arr: np.array, lmbda: np.float) -> np.array:
    if lmbda == 0:
        return (np.exp(arr))
    else:
        return (np.exp(np.log(lmbda*arr + 1)/lmbda))
```

Так как классическое преобразование Бокса-Кокса предполагает работу только с положительными величинами, то было предложено несколько модификаций, учитывающих нулевые и отрицательные значения. Самым очевидным вариантом является сдвиг всех значений на некоторую константу α так, чтобы выполнялось условие $(y_i + \alpha) > 0$, $i = 1, \ldots, n$

$$\hat{y}_i = \begin{cases} \log(y_i + \alpha), & \lambda = 0, \\ \dfrac{(y_i + \alpha)^\lambda - 1}{\lambda}, & \lambda \neq 0. \end{cases}$$

Также для того чтобы сделать данные «более гауссовскими» можно воспользоваться *преобразованием Йео-Джонсона* (Yeo-Johnson)

$$\hat{y}_i = \begin{cases} \dfrac{(y_i + 1)^\lambda - 1}{\lambda}, & \lambda \neq 0, y_i \geqslant 0, \\ \ln(y_i + 1), & \lambda = 0, y_i \geqslant 0, \\ -\dfrac{(-y_i + 1)^{2-\lambda} - 1}{2 - \lambda}, & \lambda \neq 2, y_i < 0, \\ -\ln(-y_i + 1), & \lambda = 2, y_i < 0. \end{cases}$$

Преобразование Йео-Джонсона (как впрочем и преобразование Бокса-Кокса) реализовано в библиотеке `sklearn` (см. раздел документации Non-linear transformation)

```
import numpy as np
from sklearn.preprocessing import PowerTransformer
yj = PowerTransformer(method='yeo-johnson')
bc = PowerTransformer(method='box-cox', standardize=False)
```

```
data_log = np.random.RandomState(616).lognormal(size=(3,3))
yj.fit_transform(data_log)  # вернет новое представление данных
```

Замечание

Преобразование Бокса-Кокса требует, чтобы значения набора данных были строго положительными, в то время как преобразование Йео-Джонсона может работать как с положительными, так и с отрицательными значениями

43. Кодирование признаков

Полезная статья про кодировщики и различные стратегии валидации https://towardsdatascience.com/benchmarking-categorical-encoders-9c322bd77ee8.

Можно использовать готовые алгоритмы кодирования, реализованные в библиотеке `category_encoders` https://pypi.org/project/category-encoders/[27]

Категориальные признаки можно разделить на *порядковые признаки* (например, «медлено», «быстро», «быстрее» и т.д.) и *номинальные признаки* («кошки», «собаки» и т.д.).

Основные приемы кодирования категориальных признаков:

- для порядковых признаков
  - *пользовательское отображение*, которое учитывает определенный порядок (соотношение) категорий, например, `{"cold" : 0, "warm" : 1, "hot" : 2}`: класс `LabelEncoder` следует применять только (!) для *целевого вектора*, для которого порядок меток не имеет никакого значения; если выполнить кодировку порядкового признака с помощью `LabelEncoder`, то алгоритм будет предполагать, что между категориями существует порядковая зависимость, то есть их можно как-то отсортировать, а это неверно! [10, 151]
- для именных признаков
  - *частотное кодирование*: например, с помощью `category_encoder.CountEncoder` (затем, разумеется, нужно число элементов в категории разделить на число строк в наборе данных); можно реализовать свой собственный кодировщик

    ```
    train_nominal_freq_enc = train_nominal.copy()
    fq = train_nominal.groupby("col_nominal_name").size()/train_nominal.shape[0]
    train_nominal_freq_enc["col_nominal_name"] =
        train_nominal_freq_enc["col_nominal_name"].map(fq)
    ```

  - *кодирование с одним активным состоянием* (еще говорят унитарное кодирование): легко реализовать с помощью
    `sklearn.preprocessing.OneHotEncoder` или
    с помощью `pd.get_dummies(df, drop_first=True)` (параметр `drop_first=True` нужен для того, чтобы сократить взаимосвязь между признаками, полученными с помощью техники кодирования с одним активным состоянием, так как унитарное кодирование привносит мультиколлиниарность, что может приводить к появлению неустойчивых оценок [10, 153]),

[27] Установить можно как обычно: `pip install category-encoders`

– *M-вероятностная оценка правдоподобия* (другие варианты: аддитивное сглаживание; кодирование сглаженным средним): можно использовать
класс `category_encoders.m_estimate.MEstimateEncoder`. Каждая категория категориального признака кодируются по следующей формуле

$$e_k = \frac{s_k^t + m\dfrac{s^t}{n}}{c_k + m}, \ (k = 1, \ldots, K),$$

где s_k^t – сумма значений целевого вектора для k-ой категории; m – степень регуляризации; n – длина целевого вектора (число строк в наборе данных); c_k – размер k-ой категории (число экземпляров k-ой категории); K – число категорий (уникальных значений) категориального признака.

```
from category_encoders.m_estimate import MEstimateEncoder

# здесь m -- это степень регуляризации
me_enc = MEstimateEncoder(m=50)  # рекомендуемые значения для m = [1, 100]

data_mEst_enc = data.copy()
data_mEst_enc["col3"] = me_enc.fit_transform(
    data_mEst_enc.loc[:, "col3"],
    data_mEst_enc.loc[:, "target"]
)
```

– *взвешенное суждение* (Weight of Evidence https://contrib.scikit-learn.org/category_encoders/woe.html): чаще всего применяется в кридитном скоринге.

– *CatBoost-кодировщик*: можно использовать готовый класс `CatBoostEncoder` https://contrib.scikit-learn.org/category_encoders/catboost.html
из библиотеки `category_encoders`; этот кодировщик предназначен для преодоления проблем утечки данных, присущих кодировщику `Leave-one-out Encoder` (LOO); чтобы предотвратить переобучение, процесс кодирования повторяется несколько раз для перетасованных версий набора данных, а результаты усредняются.

Удобно использовать трансформеры столбцов

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import make_pipeline
from category_encoders.cat_boost import CatBoostEncoder

col_trans = make_column_transformer(
    (OneHotEncoder(sparse=False), ["col2", "col10"]),
    (CatBoostEncoder(), ["col20"])
)

gbm = GradientBoostingRegressor()
gbm_pipeline = make_pipeline(col_trans, gbm)

gbm_pipeline.fit(X_train, y_train)
gbm_pred = gbm_pipeline.predict(X_test)
```

Что касается стратегий валидации, то можно выделить 3 основные:

- None Validation: очень грубо,
- Single Validation: что-то среднее между None Validation и Double Validation,

- Double Validation: очень медленно.

Стратегия для одиночной валидации:

1. Разбиваем имеющийся набор данных на обучение и тест,
2. Обучающий набор данных разбиваем на несколько фолдов,
3. Каждый фолд кодируем своим кодировщиком (т.е. кодировщиков будет столько, сколько было фолдов),
4. На каждом закодированном фолде обучаем свою модель (моделей столько, сколько было фолдов),
5. Кодируем тестовый набор данных теми же кодировщиками, которые мы использовали на этапе обучения (т.е. несколько представлений тестового набора данных),
6. на каждом закодированном тестовом представлении делаем прогноз с помощью моделей, обученных на соответствующих закодированных данных обучающего набора,
7. для получения финального прогноза усредняем прогнозы моделей.

44. Машинное обучение с AutoML

На данный момент AutoML представленная следующими направлениями:

- AutoML для автоматизации подбора гиперпараметров модели,
- AutoML для неглубокого обучения,
- AutoML для глубокого обучения.

Список наиболее полезных библиотек:

- `featuretools` https://featuretools.alteryx.com/en/stable/,
- `MLBox` https://mlbox.readthedocs.io/en/latest/index.html,
- `TransmogrifAI` (Scala API) https://github.com/salesforce/TransmogrifAI.

45. Хранилища данных. DWH

Хранилище данных (Data WareHouse, DWH) – предметно-ориентированная информационная база данных, специально разработанная и предназначенная для подготовки отчетов и безнес-анализа с целью поддержки принятия решений в организации. Строится на основе систем управления базами данных и систем поддержки принятия решений. Данные, поступающие в хранилище данных, как правило, доступны только для чтения.

Замечание

DWH необходимо для проведения эффективного бизнес-анализа и построения выжных для бизнеса выводов

Данные из OLTP-систем копируются в хранилище данных таким образом, чтобы при построении отчетов и OLAP-анализе не использовать ресурсы транзакционной системы и не нарушалась ее стабильность.

В чем разница между обычными базами данных и хранилищем данных:

- Обычные СУБД хранят данные строго для определенных подсистем (другими словами базы данных привязаны к своим приложениям). Например, база данных кадровиков хранит данные по персоналу, но не товары или сделки. DWH, как правило, *хранит информацию разных подразделений* – там найдутся данные и по товарам, и по персоналу, и по сделкам,

- Обычная база данных, которая ведется в рамках стандартной деятельности компании , содержит только актуальную информацию, нужную в данный момент времени для функционирования определенной системы. В DWH пишутся не столько копии актуальных состояний, сколько *исторические данные* и *агрегированные значения*. Например, состояние запасов разных категорий товаров на конец смены за последние пять лет. Иногда в DWH пишутся и более крупные пачки данных, если они имеют критическое значение для бизнеса – например, полные данные по продажам и сделкам, то есть, по сути, это копия базы данных отдела продаж,
- Информация обычно сразу попадает в рабочие базы данных, а уже оттуда некоторые записи переползают в DWH. Склад данных, по сути, отражает состояние других баз данных и процессов в компании уже после того, как вносятся изменения в рабочих базах.

Короче говоря, DWH – это система данных, отдельная от оперативной системы обработки данных. В корпоративных хранилищах в удобном для анализа виде хранятся архивные данные из разных, иногда очень разнородных источников. Эти данные предварительно обрабатываются и загружаются в хранилище в ходе процессов извлечения, преобразования и загрузки, называемых ETL.

Хранилище данных, кроме всего прочего, упрощает процедуру сбора данных из корпоративных СУБД:

- Доступ к нужным данным. Если компания большая, на получение данных из разных источников нужно собирать разрешения и доступы. У каждого подразделения в такой ситуации, как правило, свои базы данных со своими паролями, которые надо будет запрашивать отдельно. В DWH все нужное будет под рукой в готовом виде. Можно просто сконстуировать запрос и вытащить нужную информацию,
- Сохранность нужных данных. Данные в DWH не теряются и хранятся в виде, удобном для принятия решений: есть исторические записи, есть агрегированные значения. В операционной базые данных такой информации может и не быть. Например, администраторы точно не будут хранить на складском сервере архив запасов за последние 10 лет – БД склада была бы в таком случае слишком тяжелой. А вот хранить агрегированные запасы со склада в DWH – это нормально,
- Устойчивость работы бизнес-систем. DWH оптимизируется для работы аналитиков, которые могут использовать сложные, тяжелые запросы к базе данных, способные повесить сервер с боевой базой данных, и вызвать проблемы в сопряженных системах.

Для задач, связанных с промышленным интернетом вещей (IIoT), данные с датчиков можно собирать в «*озеро данных*[28]» без фильтрации, а когда данных накопиться достаточно, можно будет их проанализировать и понять из-за чего случаются поломки. Озера данных нужны для гибкого анализа данных и построения гипотез. Они позволяют собирать как можно больше данных, чтобы потом с помощью инструментов машинного обучения и аналитики извлекать полезную для бизнеса информацию.

[28] Озеро данных – хранилище, в котором собрана неструктурированная информация любых форматов из разных источников данных. Озера данных дешевле обычных баз данных, они более гибкие и легче масштабируются. Данные можно извлекать из озера по определенным признакам или анализировать прямо внутри озера, используя системы аналитики, но важно контролировать данные, поступающие в озеро данных

46. Приемы работы с ETL-инструментом Apache NiFi

«Одиночный» экземпляр Apache NiFi https://nifi.apache.org/ можно создать с использованием Docker

```
docker run -d --name nifi -p 8080:8080 apache/nifi:latest
```

Экземпляр будет доступен через web-браузер по http://localhost:8080/nifi.

47. Приемы работы с библиотекой Vowpal Wabbit

Vowpal Wabbit – библиотека с открытым исходным кодом, ориентированная на крупно-масштабные онлайн[29]-задачи машинного обучения, в основе которых лежат так называемые алгоритмы, работающие во внешней памяти (их еще называют *внеядерными* (out-of-core) алгоритмами).

Vowpal Wabbit:

- очень качественная реализация стохастического градиентного спуска для линейных моделей,
- считывает данные с диска по одному прецеденту и делает шаг только по нему, нет необходимости хранить выборку в памяти,
- может быть запущен на кластере,
- нормализация признаков, взвешивание объектов, адаптивный градиентный шаг,
- матричное разложение, тематическое моделирование, активное обучение, обучение с подкреплением,
- разнообразие методов оптимизации: сопряженные градиенты, квазиньютоновские методы (L-DBGS).

Внеядерные алгоритмы машинного обучения не требуют загрузки всех данных в память.

Пример. Пусть выборка записана в файле train.txt. Тогда

```
# обучение
vw -d train.txt --passes 10 -c -f model.vw
```

где

-d filename – имя входного файла,

--passes n – количество проходов по выборке,

-c – включает кэширование, позволяет ускорить все проходы после первого,

-f filename – имя файла, в который сохраняется модель (здесь f от final).

```
# прогноз
vw -d test.txt -i model.vw -t -p predictions.txt
```

где

-d filename – имя входного файла,

-i filename – имя файла с моделью,

-t – режим применения существующей модели (только тестирование),

-p filename – имя файла с прогнозами.

[29] В компьютерных науках онлайновое машинное обучение (online machine learning https://en.wikipedia.org/wiki/Online_machine_learning) – это метод машинного обучения, при котором данные становятся доступными не сразу, а постепенно в определенном порядке и используются для обновления наилучшего предиктора для будущих данных на каждом шаге. В то время как пакетное машинное обучение, возвращает прогноз на основе всего набора имеющихся данных. Онлайновое машинное обучение это распространенный способ, используемый там, где обучение по всему набору данных неосуществимо с точки зрения объема вычислений. Эта техника также используется в ситуациях, когда требуется, чтобы алгоритм динамически адаптировался к новым паттернам в данных

Можно работать из-под Python

```
from vowpalwabbit import pyvw

model = pyvw.vw()

train_examples = [
    "0 | price:.23 sqft:.25 age:.05 2006",
    "1 | price:.18 sqft:.15 age:.35 1976",
    "0 | price:.53 sqft:.32 age:.87 1924",
]

for example in train_examples:
    model.learn(example)

test_example = "| price:.46 sqft:.4 age:.10 1924"

prediction = model.predict(test_example); prediction # 0.0
```

48. Приемы работы с Microsoft Machine Learning for Apache Spark

Microsoft Machine Learning for Apache Spark (MMLSpark) – это экосистема инструментов, расширяющих возможности вычислительной платформы Apache Spark в нескольких новых направлениях.

Установить MMLSpark можно разными способами (см. https://github.com/Azure/mmlspark), например, так расширяется библиотека pyspark из-под Python

```
import pyspark
spark = (pyspark.sql.SparkSession.builder.appName('test spark').
             config('spark.jars.packages', 'com.microsoft.ml.spark:mmlspark_2.11:1.0.0-rc2').
             config('spark.jars.repositories', 'https://mmlspark.azureedge.net/maven').
             getOrCreate()
import mmlspark
```

Пример использования библиотеки MMLSpark для решения задачи гиперпараметрической оптимизации

```
import pandas as pd
data = spark.read.parquet("wasbs://publicwasb@mmlspark.blob.core.windows.net/BreastCancer.
    parquet")
tune, test = data.randomSplit([0.80, 0.20])
tune.limit(10).toPandas()


from mmlspark.automl import TuneHyperparameters
from mmlspark.train import TrainClassifier
from pyspark.ml.classification import LogisticRegression, RandomForestClassifier, GBTClassifier
logReg = LogisticRegression()
randForest = RandomForestClassifier()
gbt = GBTClassifier()
smlmodels = [logReg, randForest, gbt]
mmlmodels = [TrainClassifier(model=model, labelCol="Label") for model in smlmodels]

from mmlspark.automl import *

paramBuilder = \
    HyperparamBuilder() \
```

```
        .addHyperparam(logReg, logReg.regParam, RangeHyperParam(0.1, 0.3)) \
        .addHyperparam(randForest, randForest.numTrees, DiscreteHyperParam([5,10])) \
        .addHyperparam(randForest, randForest.maxDepth, DiscreteHyperParam([3,5])) \
        .addHyperparam(gbt, gbt.maxBins, RangeHyperParam(8,16)) \
        .addHyperparam(gbt, gbt.maxDepth, DiscreteHyperParam([3,5]))
searchSpace = paramBuilder.build()
# The search space is a list of params to tuples of estimator and hyperparam
print(searchSpace)
randomSpace = RandomSpace(searchSpace)


bestModel = TuneHyperparameters(
                evaluationMetric="accuracy", models=mmlmodels, numFolds=2,
                numRuns=len(mmlmodels) * 2, parallelism=1,
                paramSpace=randomSpace.space(), seed=0).fit(tune)

print(bestModel.getBestModelInfo())
print(bestModel.getBestModel())

from mmlspark.train import ComputeModelStatistics
prediction = bestModel.transform(test)
metrics = ComputeModelStatistics().transform(prediction)
metrics.limit(10).toPandas()
```

49. Приемы работы с библиотекой BeautifulSoup

49.1. Пример использования BeautifulSoup для скрапинга сайта

В качестве простого примера извлечем имена руководителей компаний из группы компаний оборонного комплекса. Имена нужных тегов удобно искать с помощью специальных инструментов разработчика, доступных в веб-браузере. Например, в Yandex-браузере получить доступ к панели разработчика можно так Настройки » Дополнительно » Дополнительные инструменты » Инструменты разработчика.

```
import requests
import pandas as pd
import psycopg2
from pprint import pprint
from bs4 import BeautifulSoup
from pandas import DataFrame, Series

main_url = 'http://ros-oborona.ru/koncerny.html'
res = requests.get(main_url)
soup = BeautifulSoup(res.text, features='lxml')

company_list = soup.find('div',
                         {'class' : 'elementor-text-editor elementor-clearfix'})
profile_list = company_list.find_all('td')

href_list = []
for elem in profile_list:
    try:
        href_list.append(elem.find('a').get('href'))
    except AttributeError:
        continue

heads_of_company_list = []
```

```
for company_url in href_list:
    res_elem = requests.get(company_url)
    soup_elem = BeautifulSoup(res_elem.text, features='lxml')
    head_of_company = soup_elem.find('span',
                                     {'class' : 'company-info__text'}).text
    if len(head_of_company.split()) == 3:
        heads_of_company_list.append(head_of_company.split())

heads_of_company_df = DataFrame(heads_of_company_list,
                                columns=['lastname', 'firstname', 'middlename'])
heads_of_company_df.index.name = 'id'
heads_of_company_df.to_csv('heads_of_company.csv', index=True)

# -- PostgreSQL
conn = psycopg2.connect('dbname=postgres user=postgres password=evdimonia')
cursor = conn.cursor()

heads_df = pd.read_csv('heads_of_company.csv')
heads_records = heads_df.to_dict('records')

cursor.execute(
            '''CREATE TABLE IF NOT EXISTS heads_of_company(
                   id integer primary key,
                   lastname text not null,
                   firstname text not null,
                   middlename text not null)'''
)
cursor.executemany(
            '''INSERT INTO heads_of_company(id, lastname, firstname, middlename)
                   VALUES (%(id)s, %(lastname)s, %(firstname)s, %(middlename)s)
                   ON CONFLICT DO NOTHING''', heads_records
)
conn.commit()

cursor.execute('SELECT * FROM heads_of_company')
fetchall = cursor.fetchall()
pprint(fetchall)
# выведет
# [(0, 'Мясников', 'Александр', 'Алексеевич'),
# (1, 'Медовщук', 'Ирина', 'Сергеевна'),
# (2, 'Матыцын', 'Александр', 'Петрович'),
# (3, 'Смирнова', 'Оксана', 'Константиновна'),
# ...]
```

50. Приемы работы с библиотекой pandas

50.1. Определить число уникальных значений в каждом категориальном признаке

Для того чтобы выяснить число уникальных значений в каждом категориальном признаке заданного набора данных следует воспользоваться конструкцией

```
df = ...
df.select_dtypes("object").nunique()
```

50.2. Срезы в мультииндексах

Для того чтобы выбрать определенные элементы индекса определенного уровня, нужно воспользоваться констукцией

```
df = ...
df.xs("XOne", level="Platform")
```

50.3. Число уникальных значений категориальных признаков в объекте DataFrame

Для того чтобы вывести информацию по числу уникальных значений в каждом категориальном признаке некоторого объекта pandas.DataFrame можно воспользоваться конструкцией

```
X.select_dtypes('object').apply(lambda col: col.unique().shape[0])
X.select_dtypes('object').apply(lambda col: col.unique().size)
X.select_dtypes('object').nunique().values[0]
```

50.4. Прочитать файл, распарсить временную метку, назначить временную метку индексом

Иногда случается, что столбец в обрабатываемом файле, имеющий смысл временной метки, не приведен к нужному формату и поэтому простое чтение файла средствами pandas не помогает. Чтобы правильно распарсить столбец с временной меткой следует сделать так

```
#! cat test_file.csv
# date, stress
# 2020/08/18, 100
# 2020/08/19, 200

>>> import pandas as pd
>>> data = pd.read_csv('test_file.csv', index_col='date', parse_date=True)
>>> type(data.index[0]) # pandas._libs.tslibs.timestamps.Timestamp
```

50.5. Число пропущенных значений в объекте DataFrame

Информацию по числу пропущенных значений в каждом столбце можно вывести следующим образом

```
X.isna().any(axis=0)
```

50.6. Управление стилями объекта DataFrame

У объектов DataFrame есть стили и ими можно управлять, выделяя максимальные/минимальные значения в таблицы, значения, которые удовлетворяют какому-то специфическому условию и пр. Однако, эти приемы работают только в notebook'ах

```
import pandas as pd
import numpy as np
from pandas import DataFrame, Series

# определяем объект-DataFrame
m, n = 10, 4
df = DataFrame(np.random.randn(m, n),
               columns=[f'col{i}' for i in range(1, n+1)])
df.loc[[4, 6, 9], ['col1', 'col4']] = np.nan
```

```
from typing import List, TypeVar

# это способ обойти ограничения аннотаций для объектов pandas
ElemOfDataframe = TypeVar('DataFrame.iloc[int, int]')

# определяем функции для управления стилями объекта-DataFrame
def threshold_color(val: ElemOfDataframe) -> str:
    '''
    Значения большие 0.5, но меньшие 1.0 выделяет красным;
    Отрицательные значения выделяет синим;
    Все прочие значения печатаются черным
    '''
    return 'color : {}'.format('red' if ((val > 0.5) and (val < 1.0)) else
                               'blue' if val < 0. else 'black')

def background_color_max(col: Series) -> List[str]:
    '''
    Фон максимальных значений в столбце выделяется желтым.
    '''
    mask = col == col.max()  # булева маска
    return ['background-color : yellow' if bool_elem else '' for bool_elem in mask]

def background_color_min(col: Series) -> List[str]:
    '''
    Фон максимальных значений в столбце выделяется светло-зеленым.
    '''
    mask = col == col.min()  # булева маска
    return ['background-color : lightgreen' if bool_elem else '' for bool_elem in mask]
```

Работа со стилями объекта-DataFrame в ячейке выглядит следующим образом

```
(  # скобки здесь нужны для переноса строки без символа `\`
    df.style.
        applymap(threshold_color).
        apply(background_color_max).
        apply(background_color_min).
        format(
            {  # можно применять разные спецификаторы формата к разным столбцам
                'col2' : '{:.5e}',
                'col4' : '{:.3G}'
            }
        )
)
```

Результат будет выглядеть как на рис. 11.

Еще одно очень полезное применение этого приема: можно раскрашивать наиболее частые значения категориального признака

```
from typing import List

def color_code_freq_cat(col: Series) -> List[str]:
    '''
    Раскрашивает самые частые значения категориальных столбцов
    '''
    # принимает столбец-Series `col`
    freq_cat = col.value_counts().index[0]  # самое частое значение категории
    return ['color : {}'.format('red' if elem == freq_cat else 'black') for elem in col]

df = DataFrame({'col1' : list('abbbabbaaab'),
```

	col1	col2	col3	col4
0	0.18301	-8.90311e-01	-0.137676	-0.394
1	0.385463	2.93965e-01	-0.713485	2.45
2	-0.750024	1.27236e+00	0.206255	-0.263
3	-0.717099	-9.69711e-01	-0.535045	1.73
4	nan	-3.67411e-01	-0.377992	NAN
5	-1.18552	5.47732e-01	-1.04696	0.362
6	nan	-1.93330e-01	-0.737013	NAN
7	0.683556	3.94844e-01	-0.734789	-0.379
8	-0.0778395	-7.50976e-01	-1.13513	0.162
9	nan	6.34074e-02	-2.32177	NAN

Рис. 11. Отформатированный вывод `DataFrame`

```
                'col2' : list('cdcccddcscd'),
                'col3' : np.random.randn(11)})

# apply работает со столбцами или строками
df_test.iloc[:5].select_dtypes('object').style.apply(color_code_freq_cat)
```

Результат приведен на рис. 12. Вывести самое частое значение в каждом столбце можно с помощью конструкции

```
# apply работает со столбцами или строками
df.apply(lambda col: col.value_counts().index[0])
```

	col1	col2
0	a	c
1	b	d
2	b	c
3	b	c
4	a	c

Рис. 12. Результат применения функции `color_code_freq_cat`

50.7. Заполнить пропущенные значения средними по группе

Часто возникает задача заполнить отсутствующие значения групповыми средними. Сделать это можно с помощью метода `transform`

```
import pandas as pd
from pandas import DataFrame, Series

seed = 123
df = DataFrame({
  'package_name' : np.array(['Ansys',
  'Nastran',
  'Comsol',
  'Abaqus'])[np.random.RandomState(seed).randint(4, size=5)],
```

```
  'comp_effect' : np.abs(100*np.random.RandomState(seed).randn(5))
})
data.iloc[[1, 3], 1] = np.nan
# df =
#  package_name  comp_effect
#0       Comsol   108.563060
#1      Nastran          NaN
#2       Comsol    28.297850
#3       Comsol          NaN
#4        Ansys    57.860025

# следует указывать столбец, по которому будет выполняться группировка
# и столбец, для которого будут вычисляться средние по группе
data['comp_effect'] = (
    data[['package_name', 'comp_effect']].groupby('package_name').
        transform(lambda g: g.fillna(g.mean()))
)
```

Для сравнения эту же задачу можно было бы решить с помощью более общего метода `apply`, но код будет значительно сложнее

```
...
# подавить создание иерархического индекса можно с помощью `group_keys`
data['comp_effect'] = (
    data[['package_name', 'comp_effect']].groupby('key2', group_keys=False).
        apply(lambda g: g.fillna(g.mean())).
            drop(['key2'], axis=1).  # удалить столбец группировки
                sort_index()         # отсортировать
)
```

или так

```
...
# в apply можно указать интересующий столбец
data['comp_effect'] = (
    data.groupby('key2', group_keys=False).
        apply(lambda g: g['key1'].fillna(g['key1'].mean())).
            sort_index()
)
```

51. Приемы работы с библиотекой `Plotly`

Рассмотрим простой пример работы с библиотекой `plotly` в блокноте

```
import numpy as np
import chart_studio.plotly as py
import plotly.graph_objs as go
from plotly.offline import (
    download_plotlyjs,
    init_notebook_mode,
    plot,
    iplot
)
init_notebook_mode(connected=True)

# в текущей директории будет создан html-файл, а график откроется в браузере
plot(go.Figure(data=[
    go.Scatter(y=np.random.randn(100).cumsum()),
    go.Scatter(y=np.random.randn(100).cumsum())
```

```
]), filename='file_name.html')
```

52. Максимальный информационный коэффициент

Как известно, коэффициент корреляции Пирсона оценивает тесноту *только* линейной связи. Таким образом, даже если коэффициент Пирсона близок к нулю, что говорит об отсутствии линейной связи между переменными, эти переменные все равно могут быть связаны, например, *нелинейной* зависимостью.

По этой причине был предложен *максимальный информационный коэффициент* (MIC). Формально MIC эквивалентен коэффициенту детерминации R^2 и принимает значения от 0 (статистическая независимость) до 1 (бесшумная функциональная связь)

$$MIC(x,y) = \max_{x,y<B} \frac{I(x,y)}{\log_2 \min(x,y)}.$$

В числителе стоит *взаимная информация* между переменными x и y

$$I(x,y) = \sum_{x,y} p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)},$$

где $p(x,y)$ – доля данных, попавших в ячейку x,y, т.е. это совместное распределение x и y.

Взаимная информация делится на логарифм от наименьшего числа ячеек x или y, а B это в некотором смысле произвольное число ячеек.

MIC можно интерпретировать как процент одной переменной, который может быть объяснен с помощью другой переменной. MIC присваивает одну и ту же оценку одинаково шумным связям, не зависимо от типа связи.

Достоинства MIC:

- умеет выявлять широкий спектр линейных и нелинейных зависимостей (кубические, экспоненциальные, синусоидальные, суперпозиции и пр.),
- симметричный, потому что основан на взаимной информации,
- Не требует предположений относительно распределения переменных,
- Устойчив к выбросам.

Есть реализация этого коэффициента в библиотеке `mictools` (`pip install mictools`).

53. Интерпретация моделей и оценка важности признаков с библиотекой SHAP

53.1. Общие сведения о значениях Шепли

В библиотеке SHAP https://github.com/slundberg/shap для оценки *важности признаков* используются *значения Шепли*[30] (Shapley value) https://en.wikipedia.org/wiki/Shapley_value.

Или несколько точнее: при построении *локальной* интерпретации (то есть интерпретации на конкретной точке данных) значения Шепли, строго говоря, оценивают *силу влияния*[31] i-ого признака f_i на значения целевого вектора y, а вот *важность признака* в контексте модели можно

[30] Термин пришел из теории кооперативных игр

[31] Еще эту оценку можно интерпретировать как *вклад*

оценить при построении *глобальной* интерпретации с помощью значений Шепли, взятых по абсолютной величине и усредненных по имеющемуся набору данных.

Замечание

Значения Шепли объясняют как «справедливо» оценить вклад каждого признака в прогноз модели

Значения Шепли i-ого признака на *конкретном объекте* (на текущей точке данных) вычисляются следующим образом (здесь сумма распространяется на все подмножества признаков S из множества признаков N, не содержащие i-ого признака)

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n-|S|-1)!}{n!} \Big(\underbrace{v(S \cup \{i\}) - v(S)}_{f_i - contribution} \Big),$$

где n – общее число признаков; $v(S \cup \{i\})$ – прогноз модели с учетом i-ого признака; $v(S)$ – прогноз модели без i-ого признака.

Выражение $v(S \cup \{i\}) - v(S)$ – это вклад i-ого признака. Если теперь вычислить среднее вкладов по всем возможным перестановкам, то получится «честная» оценка вклада i-ого признака.

Значение Шепли для i-ого признака вычисляется для каждой точки данных (например, для каждого клиента в выборке) на всех возможных комбинациях признаков (в том числе и для пустых подмножеств S).

Замечание

Метод анализа важности признаков, реализованный в библиотеке `SHAP`, является и *согласованным*, и *точным* (см. Interpretable Machine Learning with XGBoost)

53.2. Пример построения локальной и глобальной интерпретаций

Примеры использования библиотеки `SHAP` не только для tree-base моделей можно найти по адресу https://github.com/slundberg/shap/tree/master/notebooks/tree_explainer.

Решается задача регрессии для классического набора данных `boston`. Требуется предсказать стоимость квартиры.

```
import shap
import os
import pandas as pd
import numpy as np
from pandas import DataFrame, Series
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
#%matplotlib inline  # если код оформляется в JupyterLab
#shap.initjs()  # если код оформляется в JupyterLab

boston = load_boston()
X, y = boston['data'], boston['target']  # numpy-массивы

# объекты pandas
X_full = DataFrame(X, columns=boston['feature_names'])
y_full = Series(y, name = 'PRICE')

X_train, X_test, y_train, y_test = train_test_split(X_full, y_full, random_state=42)
```

```
rf = RandomForestRegressor(n_estimators=500).fit(X_train, y_train)

explainer = shap.TreeExplainer(rf)  # <- NB
shap_values_train = explainer.shap_values(X_train)  # <- NB
```

53.2.1. Локальная интерпретация отдельной точки данных обучающего набора

Теперь можно построить локальную интерпретацию для одной точки данных из обучающего набора (см. рис. 13)

К вопросу о локальной интерпретации отдельной точки данных обучающего набора

```
row = 1
shap.force_plot(
    explainer.expected_value,  # ожидаемое значение
    shap_values_train[row, :],  # 2-ая строка в матрице значений Шепли
    X_train.iloc[row, :]  # 2-ая строка в обучающем наборе данных
)
```

Можно считать, что `explainer.expected_value` это значение, полученное усреднением целевого вектора по точкам обучающего набора данных, т.е. `y_train.mean()`.

Рис. 13. Локальная интерпретация для одной точки данных обучающего набора

Еще можно построить график частичной зависимости (рис. 14)

```
shap.dependence_plot('LSTAT', shap_values, X_train)
```

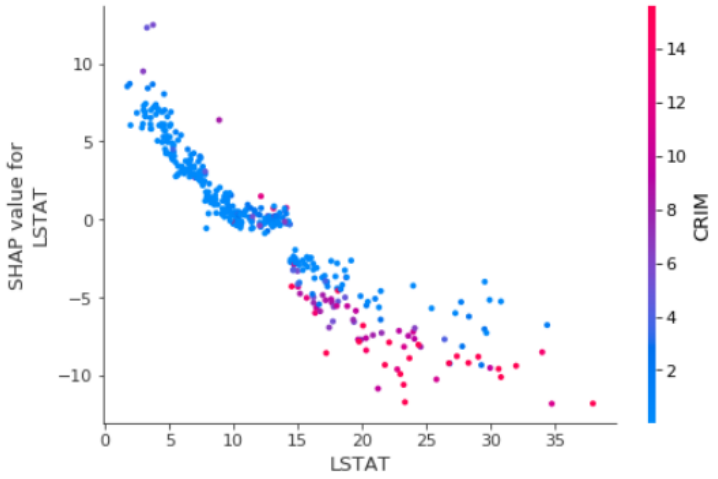
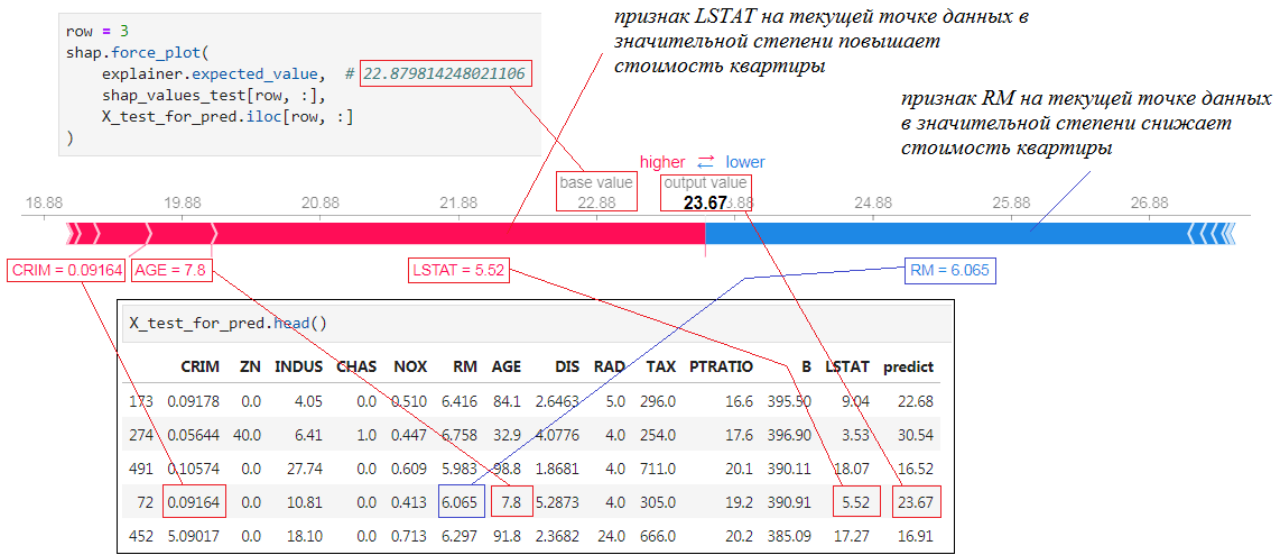


Рис. 14. График частичной зависимости признака `LSTAT` от значений Шепли с учетом влияния признака `CRIM`

53.2.2. Локальная интерпретация отдельной точки данных тестового набора

Прежде чем приступить к вычислению значений Шепли, следует создать поверхностную копию тестового набора данных



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	predict
173	0.09178	0.0	4.05	0.0	0.510	6.416	84.1	2.6463	5.0	296.0	16.6	395.50	9.04	22.68
274	0.05644	40.0	6.41	1.0	0.447	6.758	32.9	4.0776	4.0	254.0	17.6	396.90	3.53	30.54
491	0.10574	0.0	27.74	0.0	0.609	5.983	98.8	1.8681	4.0	711.0	20.1	390.11	18.07	16.52
72	0.09164	0.0	10.81	0.0	0.413	6.065	7.8	5.2873	4.0	305.0	19.2	390.91	5.52	23.67
452	5.09017	0.0	18.10	0.0	0.713	6.297	91.8	2.3682	24.0	666.0	20.2	385.09	17.27	16.91

Рис. 15. Локальная интерпретация для одной точки данных тестового набора

```
X_test_for_pred = X_test.copy()
X_test_for_pred['predict'] = np.round(rf.predict(X_test), 2)

explainer = shap.TreeExplainer(rf)
# вычисляем значения Шепли для тестового набора данных со столбцом 'predict'
shap_values_test = explainer.shap_values(X_test_for_pred)
```

Теперь можно построить локальную интерпретацию для отдельной точки данных тестового набора (рис. 15).

Из рис. 15 видно, что признаки с различной «силой»[32], которая определяется значениями Шепли, смещают предсказание модели на данной точке. Например, признак `LSTAT` (процент населения с низким социальным статусом) в значительной степени *повышает*[33] стоимость квартиры на данной точке по отношению к базовому значению `base_value`, а признак `RM` (среднее число комнат в жилом помещении) в значительной степени снижает.

К вопросу о локальной интерпретации отдельной точки данных тестового набора

```
row = 3
shap.force_plot(
    explainer.expected_value,  # 22.879814248021106
    #y_train.mean()  # 22.907915567282323
    shap_values_test[row, :],
    X_test_for_pred.iloc[row, :]
)
```

53.2.3. Глобальная интерпретация модели на тестовом наборе данных

Удобно работать с диаграммой рассеяния `shap.summary_plot` (рис. 16), на которой изображаются признаки в порядке убывания их важности, с одновременным указанием того, насколько сильно каждый из признаков влияет на целевую переменную.

```
shap.summary_plot(shap_values_test, X_test_for_pred)
```

[32] Ширина полосы

[33] Потому что значение этого признака невелико; чем меньше процент населения с низким социальным статусом проживает в округе, тем выше стоимость квартиры

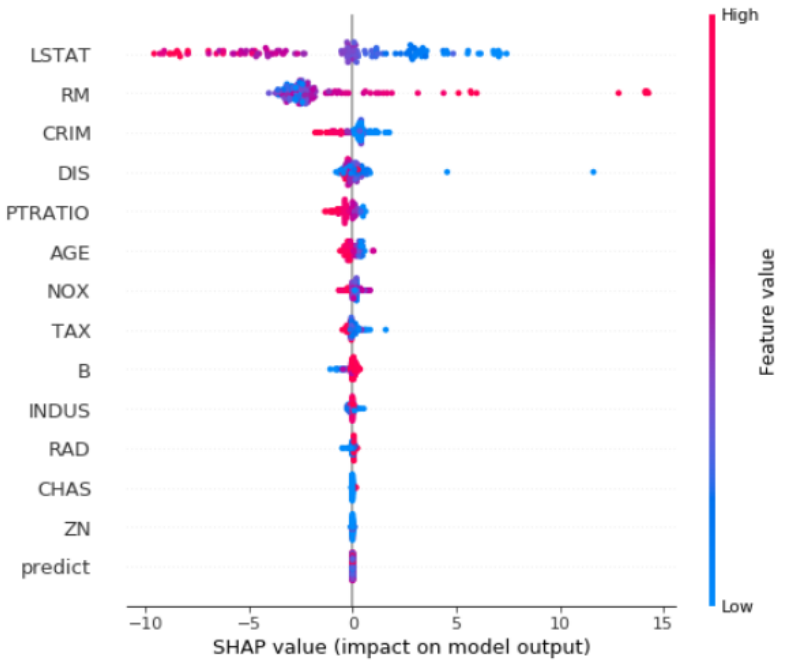


Рис. 16. Диаграмма рассеяния для точек тестового набора данных

Какие выводы можно сделать из рис. 16:

- Признаки `LSTAT`, `RM` и `CRIM` имеют высокую важность для модели в целом,
- Для признака `LSTAT` наблюдается отрицательная статистическая зависимость от целевой переменной, т.е. низкие значения этого признака отвечают высоким значениям целевой переменной (стоимости на квартиру),
- Для признака `RM` наблюдается положительная статистическая зависимость от целевой переменной: чем больше комнат в жилом помещении, тем выше стоимость квартиры.

Затем можно детальнее изучить графики частичной зависимости, построенные на тестовом наборе данных. Рассмотрим зависимость признака `CRIM` (уровень преступности в городе на душу населения) от значений Шепли, вычисленных для этого признака (рис. 17).

```
shap.dependence_plot('CRIM', shap_values_test[:, :-1], X_test_pred.iloc[:, :-1])
```

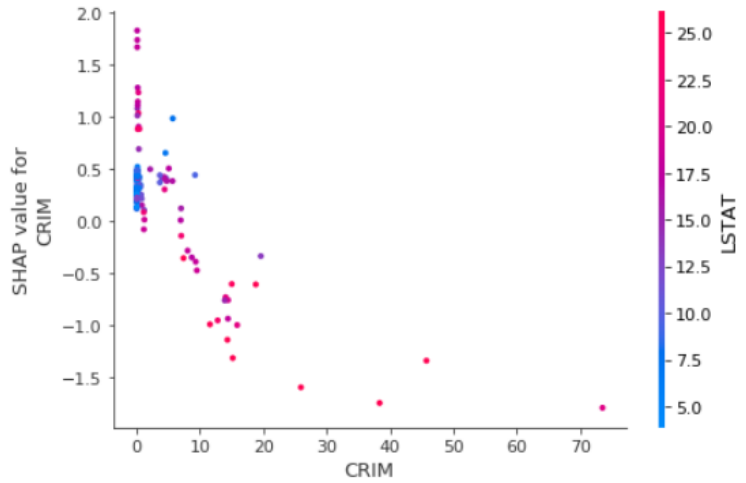


Рис. 17. График частичной зависимости признака `CRIM` от значений Шепли с учетом влияния `LSTAT`

Какие выводы можно сделать из рис. 17:

- Чем выше уровень преступности в городе, тем в большей степени снижается стоимость квартиры,
- Не везде, где проживает высокий процент населения с низким социальным статусом наблюдается высокий уровень преступности, однако в тех местах, где регистрируется высокий уровень преступности одновременно регистрируется и высокий процент населения с низким социальным статусом.

54. Перестановочная важность признаков в библиотеке `eli5`

Еще важность признаков можно оценивать с помощью так называемой *перестановочной важности* (permutation importances) https://www.kaggle.com/dansbecker/permutation-importance.

Идея проста: нужно в заранее отведенном для исследования важности признаков наборе данных (валидационном наборе) перетасовать значения признака, влияние которого изучается на данной итерации, оставив остальные признаки (столбцы) и целевой вектор без изменения.

Признак считается «важным», если метрики качества модели падают, и соответственно – «неважным», если перестановка не влияет на значения метрик. Перестановочная важность вычисляется после того как модель будет обучена.

Замечание

Перестановочная важность обладает свойством *согласованности*, но не обладает свойством *точности*

Interpretable Machine Learning with XGBoost

Рассмотрим задачу построения регрессионной модели на наборе данных `load_boston`

```
import eli5
import pandas as pd
from eli5.sklearn import PermutationImportance
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
from pandas import DataFrame, Series

boston = load_boston()

X_train, X_test, y_train, y_test = train_test_split(boston['data'],
                                                    boston['target'],
                                                    random_state=2)

X_train_sub, X_valid, y_train_sub, y_valid = train_test_split(X_train,
                                                              y_train,
                                                              random_state=0)

# модель случайного леса, как обычно, обучается на обучающей выборке
rf = RandomForestRegressor(n_estimators=500).fit(X_train_sub, y_train_sub)

# модель перестановочной важности обучается на валидационном наборе данных
perm = PermutationImportance(rf, random_state=42).fit(X_valid, y_valid)

eli5.show_weights(perm, feature_names = boston['feature_names'])  # визуализирует перестановочны
    е важности признаков
```

55. Регулярные выражения в Python

В языке Python есть несколько тонких особенностей, связанных с регулярными выражениями, а имеено с поведением жадных и нежадных квантификаторов. Рассмотрим пример с *жадным* квантификатором

```
# python
import re
re.compile('y*(\d{1,3})').search('xy1234z').groups()[0]  # '123'
```

Аналогичный результат получается и в PostgreSQL

```
-- postgresql
select substring('xy1234z', 'y*(\d{1,3})'); -- '123'
```

Но если используется *нежадный* квантификатор, то результаты будут различаться

```
# python
import re
re.compile('y*?(\d{1,3})').search('xy1234z').groups()[0]  # '123'
```

А вот в PostgreSQL

```
-- postgresql
select substring('xy1234z', 'y*?(\d{1,3})'); -- '1'
```

Совпадать результаты будут только в том случае, если в регулярном выражении Python специально указать, что {m,n} должен быть нежадным, т.е. {m,n}?

```
# python
import re
re.compile('y*?(\d{1,3}?)').search('xy1234z').groups()[0]  # '1'
```

56. Неравенство Маркова

Неравенство Маркова в теории вероятностей дает оценку вероятности, что неотрицательная случайная величина превзойдет по модулю фиксированную положительную константу, в терминах ее математического ожидания. Хотя получаемая оценка обычно груба, она позволяет получить определенное представление о распределении, когда последнее не известно явным образом.

Пусть неотрицательная случайная величина $X : \Omega \to \mathbb{R}^+$ определена на вероятностном пространстве $(\Omega, F, \mathbb{P})$, и ее математическое ожидание $\mathbb{E}X$ конечно. Тогда

$$\mathbf{P}(X \geqslant a) \leqslant \frac{\mathbb{E}X}{a},\ a > 0.$$

Пример. Пусть в среднем ученики опаздывают на 3 минуты (оценка математического ожидания), и нас интересует какова вероятность того, что ученик опоздает на 15 и более минут. Чтобы получить грубую оценку сверху

$$\mathbf{P}(|X| \geqslant 15) \leqslant \frac{3}{15} = 0,2.$$

57. Асинхронное программирование в Python

57.1. Библиотека aiomisc

Кроме библиотек asyncio, asyncpg и пр. есть еще одна очень полезная библиотека aiomisc https://github.com/aiokitchen/aiomisc. В числе прочего там есть такой полезный сервис как MemoryTracer

```
import asyncio
import os
from aiomisc import entrypoint
from aiomisc.service import MemoryTracer

async def main():
    leaking = []

    while True:
        leaking.append(os.urandom(128))
        await asyncio.sleep(0)

with entrypoint(MemoryTracer(interval=1, top_results=5)) as loop:
    loop.run_until_complete(main())
```

Еще один готовый сервис помогает профилировать приложение

```
import asyncio
import os
import time
from aiomisc import entrypoint
from aiomisc.service import Profiler


async def main():
    for i in range(100):
        time.sleep(0.01) # синхронная функция в асинхронном коде! Плохо


with entrypoint(Profiler(interval=0.1, top_results=5)) as loop:
    loop.run_until_complete(main())
```

Есть некоторые вещи, которые нельзя реализовать в асинхронном коде, поэтому приходится пользоваться *потоками*

```
>>> import asyncio, time, aiomisc

>>> @aiomisc.threaded
    def blocking_function():
        ''' Блокирующая функция. '''
        time.sleep(1)

# НО производительность решения будет зависеть от пула потоков.
# Если в пуле, скажем, 8 потоков, то значит одновременно смогут
# выполниться только 8 запросов, а остальные запросы будут
# вставать в очередь
>>> async def main():
        ''' Функции будут выполняться параррллельно. '''
        await asyncio.gather( # в двух потоках
            blocking_function(),
            blocking_function()
        )
```

```
    %%timeit # 1.02 s +/- 3.82 ms per loop (mean +/- std. dev. of 7 runs, 1 loop each)
    with aiomisc.entrypoint() as loop:
        loop.run_until_complete(main())
```

Еще удобно работать с генераторами

```
import aiomisc
import asyncio


@aiomisc.threaded_iterable(max_size=100)
def blocking_reader():
    ''' Синхронный генератор. '''
    with open('/dev/urandom', 'r+') as fp:
        mdt_hash = hashlib.md5()
        while True:
            md5_hash.update(fp.read(32))
            yield md5_hash.hexdigest().encode()


async def main():
    reader, writer = await asyncio.open_connection('127.0.0.1', 2233)
    async with blocking_reader() as gen: # асинхронный менеджер контекста нужен обязательно!!!
        async for line, digest in gen:
            writer.write(digest)
            writer.write(b'\t')
            writer.write(line)


with aiomisc.entrypoint() as loop:
    loop.run_until_complete(main())
```

Можно создать *отдельный* поток, который «умирает» вместе с декорируемой функцией (например, можно писать логи в отдельном потоке)

```
queue = Queue(max_size=100)

@aiomisc.threaded_separate
def blocking_reader(fname):
    with open('/dev/urandom', 'r+') as fp:
        while True:
            mdt_hash.update(fp.read(32))
            queue.put(md5_hash.hexdigest().encode())
```

Декораторы `@threaded` делают из обычной функции *awaitable-объект*. Нельзя выполнить функцию, обернутую `@threaded` из функции, обернутой `@threaded`.

58. Приемы работы с DVC

С основами использования инструмента DVC можно познакомиться по статье https://proglib.io/p/git-dlya-data-science-kontrol-versiy-modeley-i-datasetov-s-pomoshchyu-dvc-2020-12-02.

59. Работа с базами данных в Python

Для работы с `PostgreSQL` из-под `Python`, как правило, используется драйвер `psycopg2`. Можно использовать еще и `sqlalchemy`. Согласно спецификации DB-API 2.0, после создания объекта

соединения необходимо создать объект-курсор. Все дальнейшие запросы должны производиться через этот объект.

Пример

```
import psycopg2
import sqlalchemy

# PostgreSQL
conn_pg = psycopg2.connect('postgesql://postgres@localhost:5432/demo')
cur_pg = conn_pg.cursor()
# возвращает название источника данных в формате строки
conn_pg.dsn # 'postgesql://postgres@localhost:5432/demo'
conn_pg.get_dsn_parameters()
#{'user': 'postgres',
# 'passfile': 'C:\\Users\\ADM\\AppData\\Roaming/postgresql/pgpass.conf',
# 'dbname': 'demo',
# 'host': 'localhost',
# 'port': '5432',
# 'tty': '',
# 'options': '',
# 'sslmode': 'prefer',
# 'sslcompression': '0',
# 'krbsrvname': 'postgres',
# 'target_session_attrs': 'any'}

# вывести элементы из столбца `kv` из таблицы `test_hstore`, хранящей пары <<ключ-значение>>,
# и выбрать те строки, в которых содержится ключ `solver type`
cur_pg.execute('''
    SELECT kv->'solver type' FROM test_hstore WHERE kv ? 'solver type'
''')
cur_pg.fetchall()  # [('direct',), ('iterative',)]

# SQLAlchemy
engine_sql = sqlalchemy.create_engine('postgresql://postgres@localhost:5432')
conn_sql = engine_sql.connect()
conn_sql.execute('''
    SELECT kv->'solver type' FROM test_hstore WHERE kv ? 'solver type'
''').fetchall()  # [('direct',), ('iterative',)]
```

Еще чтобы не беспокоиться на счет статуса объекта-курсора и соединения можно пользоваться менеджером контекста

```
import psycopg2

with psycopg2.connect('postgresql://postgres@localhost:5432/demo') as conn: # соединение
    with conn.cursor() as cur: # объект-курсора
        cur.execute('select * from tickets limit %(lmt)s;', {'lmt' : 5})  # даже если передается
     объект целочисленного типа следует использовать %()s!!!
        res = cur.fetchall()

        for row in res:
            print(row)
```

Библиотека `asyncpg` https://github.com/MagicStack/asyncpg используется когда требуется реализовать *асинхронную* работу с базой данной PostgreSQL. Устанавливается библиотека как обычно с помощью менеджера пакетов `pip`: `pip install asyncpg`.

Библиотека `asyncpg` не реализует Python DB-API, так как DB-API это синхронный интерфейс программного приложения, а `asyncpg` построена вокруг асинхронной I/O-модели.

В библиотеке `psycopg2` метод `cursor.execute()` *блокирует* программу на все время выполнения запроса. Если запрос сложный, то программа будет заблокирована надолго, что не всегда желательно. Это означает, что пока запрос выполняется, программа может заниматься другими делами.

Библиотека `asyncpg` предоставляет асинхронный API, предназначенный для работы совместно с `asyncio` – библиотекой, используемой для написания конкурентного кода на Python.

Замечания: ключевое слово `async` означает, что определенная далее функция является сопрограммой, т.е. асинхронна и должна выполняться особым образом, а ключевое слово `await`[34] служит для синхронного выполнения сопрограмм.

По рекомендациям разработчиков `asyncio`

- следует использовать `asyncio.run()`, а цикл не нужен!,
- должна быть одно точка входа,
- следует использовать `async/await` везде,
- никогда не следует передавать ссылку на цикл

По рекомендациям разработчиков `asyncio` НЕ нужно использовать

- декораторы `@coroutine`,
- низкоуровневый API (`asyncio.Future`, `call_soon()`, `call_later()`, `event loop` etc)

Простой пример и сравнение

```
import asyncio

# ------- old style
def get_text_oldschool():
    '''
    Вызов функции возвращает объект-генератор
    '''
    yield 'test string (old school)'

gto = get_text_oldschool() # переменная связывается с объектом-генератором
gto.__next__() # 'test string (old school)'

# ------- new style
async def get_text_coro():
    '''
    Вызов функции возвращает объект-сопрограмму
    '''
    return 'test string (new style)'
# запустить сопрограмму можно с помощью ключевого слова await
await get_text_coro() # 'test string (new style)'
```

Еще один вариант запуска сопрограмм с помощью `asyncio.run()` (запускает цикл событий)

```
import asyncio

async def get_text(delay, text):
    await asyncio.sleep(delay)
    return text

async def say_text():
    ''' Здесь задачи будут выполняться парраллельно'''
    # создаем задачи и ставим их в очередь; они еще не выполняются
    task1 = asyncio.create_task(get_text(1, 'hello'))
    task2 = asyncio.create_task(get_text(1, 'world'))
```

[34] Запускает сопрограмму из асинхронного кода с явным переключением контекста

```
    # явно переключаем контекст и выполняем сопрограммы
    await task1
    await task2
    return ', '.join([task1.result(), task2.result()])

result = asyncio.run(say_text()); result # 'hello, world'
```

```
# задачи выполнятюсят парраллельно!!!
async def output(t):
    return await asyncio.sleep(t, 'test message')

async def main():
    tasks = [
        asyncio.create_task(output(n))
        for n in (2, 1, 3, 1)
    ]
    for task in asyncio.as_completed(tasks):
         print(await task)
```

Ограничить время ожидания awaitable-объекта можно так

```
async def eternity():  # сопрограмма
    try:
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('I was cancelled')
        raise # возбуждается исключение
    print('Finished')  # не печатается

async def main():  # сопрограмма
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('Timeout')

asyncio.run(main())
```

Подождать выполнения awaitable-объектов можно так

```
import asyncio

async def delayed_res(delay):
    return await asyncio.sleep(delay, f'test string: {delay}')

async def main():
    tasks = [ # создаем задачи
        delayed_res(i)
        for i in range(1,10+1)
    ]
    for earliest in asyncio.as_completed(tasks):
        res = await earliest # выполняем сопрограмму
        print(res)

asyncio.run(main())
```

Можно запустить синхронный код в процессе/потоке

```
import asyncio
import concurrent

async def main():
```

```
    loop = asyncio.get_running_loop()
    with concurrent.futures.ProcessPoolExecutor() as pool:
        res = await loop.run_in_executor(pool, cpu_bound)
        print('custom process pool', res)

asyncio.run(main())
```

Несколько задач сразу можно запустить так

```
import asyncio

async def main():
    coros = (
        some_async_coro(i)
        for i in range(10)
    )
    results = await asyncio.gather(*coros) # << NB

asyncio.run(main())  # у приложения должна быть одна точка входа
```

В асинхронном программировании поддерживаются еще и *асинхронные генераторы*, т.е. асинхронные функции, использующие ключевое слово `yield`

```
async def ticker(delay, to):
    ''' Асинхронный генератор'''
    for i in range(to):
        yield i
        await asyncio.sleep(delay)

async def main():
    async for i in ticker(1,10):
        print(i)

asyncio.run(main())
```

В python 3.6+ поддерживаются все comprehensions

```
# их всех можно сочетать с for и if
{ i async for i in agen() } # множество
[ i async for i in agen() ] # список
{ i : i**2 async for i in agen() } # словарь
( i**2 async for i in agen()) # генераторное выражение
```

Пример использования асинхронного генератора

```
async def ticker(delay, to):
    ''' Асинхронный генератор'''
    for i in range(to):
        yield i # <-
        await asyncio.sleep(delay)

async def main():
    results = [ # асинхронный генератор списка с двумя циклами
        (i,j)
        async for i in ticker(0.1, 5)
            async for j in ticker(0.1, 5)
        if not i % 2 and j % 2
    ]
    print(results)
```

Пример использования

```
#import asyncio
>>> import asyncpg

>>> conn = await asyncpg.connect('postgresql://postgres@localhost:5432/demo')
>>> values = await conn.fetch('''
        SELECT passenger_name, count(*)
        FROM tickets
        GROUP BY 1
        ORDER BY 2 DESC
        LIMIT 5;
    ''')
>>> type(values[0]) # asyncpg.Record
>>> values
#[<Record passenger_name='ALEKSANDR IVANOV' cnt=842>,
# <Record passenger_name='ALEKSANDR KUZNECOV' cnt=755>,
# <Record passenger_name='SERGEY IVANOV' cnt=634>,
# <Record passenger_name='SERGEY KUZNECOV' cnt=569>,
# <Record passenger_name='VLADIMIR IVANOV' cnt=551>]

>>> res = await conn.fetch('''
              SELECT passenger_name, contact_data #>> '{phone}'::text[] AS phone
              FROM tickets
              LIMIT 3;
          ''')
>>> res
# [<Record passenger_name='VALERIY TIKHONOV' phone='+70127117011'>,
#  <Record passenger_name='EVGENIYA ALEKSEEVA' phone='+70378089255'>,
#  <Record passenger_name='ARTUR GERASIMOV' phone='+70760429203'>]
>>> res[0].get('phone') # '+70127117011'
>>> for k in res[1].keys():
        print(k)
# passenger_name
# phone
>>> for v in res[2].values():
        print(v)
# ARTUR GERASIMOV
# +70760429203
>>> for i, row in enumerate(res, 1): # обход строк выдачи
        print(f'{i}: ' +
              ','.join([f'/{k}/->{v}' for k,v in row.items()])
        )
>>> await conn.close()
```

```
import asyncio
import asyncpg
import datetime

async def main():
    # Establish a connection to an existing database named "test"
    # as a "postgres" user.
    conn = await asyncpg.connect('postgresql://postgres@localhost/test')
    # Execute a statement to create a new table.
     # `execute` если не нужно ничего возвращать
    await conn.execute('''
            CREATE TABLE users(
              id serial PRIMARY KEY,
              name text,
              dob date
            )
```

```
    ''')

    # Insert a record into the created table.
    await conn.execute('''
        INSERT INTO users(name, dob) VALUES($1, $2)
    ''', 'Bob', datetime.date(1984, 3, 1))

    # Select a row from the table.
    row = await conn.fetchrow(
        'SELECT * FROM users WHERE name = $1', 'Bob')
    # *row* now contains
    # asyncpg.Record(id=1, name='Bob', dob=datetime.date(1984, 3, 1))

    # Close the connection.
    await conn.close()

asyncio.get_event_loop().run_until_complete(main())
```

Иногда бывает удобно использовать предварительно подготовленные параметризованные SQL-запросы

```
# подготовленный параметризованный SQL-запрос
>>> cmpt_stmt = await conn.prepare('select 2^$1')
>>> cmpt_stmt # <asyncpg.prepared_stmt.PreparedStatement at 0xfc4fd68>
>>> res = await cmpt_stmt.fetchval(2); res # 4.0
>>> res await cmpt_stmt.fetchval(5); res # 32.0
```

Можно вывести план выполнения запроса

```
p = await cmpt_stmt.explain(5); p
# [{'Plan': {'Node Type': 'Result',
# 'Parallel Aware': False,
# 'Startup Cost': 0.0,
# 'Total Cost': 0.01,
# 'Plan Rows': 1,
# 'Plan Width': 8,
# 'Output': ["'32'::double precision"]}}]
p = await cmpt_stmt.explain(5, analyze=True); p
# [{'Plan': {'Node Type': 'Result',
# 'Parallel Aware': False,
# 'Startup Cost': 0.0,
# 'Total Cost': 0.01,
# 'Plan Rows': 1,
# 'Plan Width': 8,
# 'Actual Startup Time': 0.001,
# 'Actual Total Time': 0.001,
# 'Actual Rows': 1,
# 'Actual Loops': 1,
# 'Output': ["'1.0715086071862673e+301'::double precision"]},
# 'Planning Time': 0.065,
# 'Triggers': [],
# 'Execution Time': 0.026}]
```

Можно использовать *транзакции*

```
>>> conn = await asyncpg.connect('...')
>>> async with conn.transaction():
        res = await conn.fetch('INSERT INTO tab VALUES (1, 2, 3)')
>>> res
```

Еще пример на транзакции

```
async with conn.transaction():
    res = await conn.fetch('''
       SELECT passenger_name, contact_data ->> 'phone' AS phone
       FROM tickets
       LIMIT $1
    ''', 3)
print(res)
```

Библиотека `asyncpg` поддерживает асинхронное итерирование с помощью `async for`

```
async def iterate(conn: Connection):
    async with conn.transaction():
        async for record in conn.cursor('SELECT generate_series(0, 100)'):
            print(record)
```

В случае когда используется связка `SQLAlchemy` и `asyncpg`, можно воспользоваться специальной библиотекой `asyncpgsa` https://asyncpgsa.readthedocs.io/en/latest/.

Для работы с аналитической СУБД `Vertica` есть своя библиотека `vertica_python`[35] https://github.com/vertica/vertica-python

```
imoprt vertica_python

conn_info = {
    'host' : '127.0.0.1',
    'port' : 5433,
    'user' : 'some_user',
    'password' : 'some_password',
    'database' : 'a_database',
    'kerberos_service_name' : 'vertica_krb',
    'kerberos_host_name' : 'vlcuster.example.com'
}

with vertica_python.conn(**conn_info) as conn:
    # do things
```

Вариант с балансировкой нагрузки

```
import vertica_python

conn_info = {
    'host' : '127.0.0.1',
    'port' : 5433,
    'user' : 'some_user',
    'password' : 'some_password',
    'database' : 'vdb',
    'connection_laod_balance' : True
}

# Server enables load balancing
with vertica_python.connect(**conn_info) as conn:
    cur = conn.cursor()
    cur.execute('SELECT NODE_NAME FROM V_MONITOR.CURRENT_SESSION')
    print('Client connects to primary node:', cur.fetchone()[0])
    cur.execute("SELECT SET_LOAD_BALANCE_POLICY('ROUNDROBIN')")

with vertica_python.connect(**conn_info) as conn:
    cur = conn.cursor()
    cur.execute('SELECT NODE_NAME FROM V_MONITOR.CURRENT_SESSION')
```

[35] Устанавливается как обычно с помощью менеджера пакетов `pip`: `pip install vertica-python`

```
    print('Client redirects to node:', cur.fetchone()[0])
```

Доступ к колоночной аналитической СУБД `ClickHouse`, позволяющей выполнять аналитические запросы в режиме реального времени на структурированных больших данных, можно получить с помощью Python-библиотеки `clickhouse_driver`[36]

```
# DP API example
from clickhouse_driver import connect

conn = connect('clickhouse://localhost')
cursor = conn.cursor()

cursor.execute('CREATE TABLE test(x Int32) ENGINE=Memory')
cursor.executemany(
    'INSERT INTO test(x) VALUES',
    [{'x' : 100}]
)
cursor.execute(
    'INSERT INTO test(x) '
    'SELECT * FROM system.numbers LIMIT %(limit)s',
    {'limit' : 3}
)
cursor.execute('SELECT sum(x) FROM test')
cursor.fetchall()  # [(303,)]
```

Также есть возможность управлять работой *графовых* баз данных, например, Neo4j[37] https://neo4j.com с помощью, например, специального языка обхода графов `Gremlin` (есть альтернативы). Есть реализация Gremlin-Python https://tinkerpop.apache.org/docs/current/reference/#gremlin-python и соответствующая библиотека `gremlin_python`[38]

```
from gremlin_python.process.anonymous_traversal_source import traversal

g = traversal().withRemote(
        DriverRemoteConnection('ws://localhost:8182/gremlin','g', headers={'Header' : 'Value'}))
```

```
# классы, функции и токены, которые обычно используются с Gremlin
from gremlin_python import statics
from gremlin_python.process.anonymous_traversal import traversal
from gremlin_python.process.graph_traversal import __
from gremlin_python.process.strategies import *
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection
from gremlin_python.process.traversal import T
from gremlin_python.process.traversal import Order
from gremlin_python.process.traversal import Cardinality
from gremlin_python.process.traversal import Column
from gremlin_python.process.traversal import Direction
from gremlin_python.process.traversal import Operator
from gremlin_python.process.traversal import P
from gremlin_python.process.traversal import Pop
from gremlin_python.process.traversal import Scope
from gremlin_python.process.traversal import Barrier
from gremlin_python.process.traversal import Bindings
from gremlin_python.process.traversal import WithOptions
```

[36] Устанавливается с помощью менеджера пакетов `pip`: `pip install clickhouse-driver`

[37] Существует соответствующая python-библиотека `neo4j` https://neo4j.com/developer/python/. Используется собственные язык запросов Cypher, но поддерживается и Gremlin

[38] Устанавливается как обычно с помощью менеджера пакетов `pip`: `pip install gremlinpython`

```
...
```

Затем в консоли можно выполнить запрос

```
>>> g.V().hasLabel('person').has('age',P.gt(30)).order().by('age',Order.desc).toList()  # [v[6],
    v[4]]
```

Приемы базовой работы с Gremlin можно изучить в разделе документации https://tinkerpop.apache.org/docs/current/reference/#basic-gremlin

```
v1 = g.addV('person').property('name','marko').next()
v2 = g.addV('person').property('name','stephen').next()
g.V(Bindings.of('id',v1)).addE('knows').to(v2).property('weight',0.75).iterate()
```

Простыми словами, обход графа – это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или *ребрами* графа. Вершины графа также называются *узлами*.

Основными алгоритмами обхода графа являются:

- поиск в глубину (depth-first search, DFS),
- поиск в ширину (breadth-first search, BFS).

60. Особенности использования менеджера пакетов `pip`

Чтобы установить пакет в редактируемом режиме в текущую директорию следует использовать флаг `-e`

```
pip install -e .
```

Это нужно, если требуется править исходники пакета, который устанавливается.

61. Приемы работы с `flake8`

`flake8` – инструмент, позволяющий выявить стилистические ошибки в коде.

Установаить `flake8` можно как обычно

```
pip install flake8
```

Можно передать путь до рабочей директории или имя конкретного файла

```
flake8 work_dir
flake8 file_name.py
```

Еще Flake8 поддерживает pre-commit хуки для Git и Mercurial. Эти хуки позволяют, например, не допускать создание коммита при нарушении каких-либо правил оформления.

Установить хук для Git

```
flake8 --install-hook git
```

Настроить сам Git, чтобы он учитывал правила Flake8

```
git config --bool flake8.strict true
```

Управлять поведением Flake8 можно с помощью конфигурационных файлов (`setup.cfg`, `tox.ini`, `.flake8`). Например

.flake8

```
[flake8]
  ignore =
    # F812: list comprehension redefines ...
    F812,
    D203
    # H101: Use TODO(NAME)
    H101,
    H301
  exclude = .git, __pycache__, build, old, dist, docs/conf.py
```

62. Особенности работы с форматером `black`

Black https://pypi.org/project/black/ – инструмент анализа и автоматического форматирования кода. Следит за файлами и форматирует только те файлы, которые были изменены.

Можно запретить форматирование отдельных блоков кода с помощью `# fmt: on` и `fmt: off`, обозначающие начало и конец блока.

Установить Black можно так

```
pip install black
```

Поведением Black удобно управлять с помощью конфигурационного toml-файла[39]

pyproject.toml

```
[tool.black]
line-length = 80
target-version = ['py37']
include = '\.pyi?$'
exclude = '''
(
    \(
          \.eggs
        | \.git
        | \.hg
        | \.mypy_cache
        | \.tox
        | \.venv
        | build
        | dist
    \)
    | foo.py
)
'''
```

Black будет искать файл `pyproject.toml`, начиная с текущего рабочего каталога и заканчивая корнем файловой системы, если придется.

Если вызвать `black` с флагом `-v` (или `--verbose`), то, при условии, что файл `pyproject.toml` найдется, в терминал будет выведен путь до конфигурационного файла.

Для работы с различными хуками удобно пользоваться специальным менеджером хуков `pre-commit`[40] https://pre-commit.com. Нужно просто указать список хуков, а `pre-commit` будет запускать хук перед каждым коммитом.

[39] TOML – формат конфигурационных файлов

[40] Установить можно как обычно `pip install pre-commit`

Остается только добавить конфигурационный файл `.pre-commit-config.yaml` в рабочий локальный git-репозиторий

.pre-commit-config.yaml

```
repos:
  - repo: https://github.com/psf/black
    rev: 19.10b0 # Replace by any tag/version: https://github.com/psf/black/tags
    hooks:
      - id: black
        language_version: python3 # Should be a command that runs python3.6+
```

А затем запустить установку хуков

```
pre-commit install
```

Теперь `pre-commit` будет автоматически запускаться каждый раз при создании коммита.

Перед фиксацией изменений можно на всякий случай вызвать

```
pre-commit run --all-files
# вывод
[INFO] Initializing environment for https://github.com/psf/black.
[INFO] Installing environment for https://github.com/psf/black.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
black.....................................................................Failed
- hook id: black
- files were modified by this hook

reformatted /Users/leor.finkelberg/Python_projects/termostablizator/css.py
reformatted /Users/leor.finkelberg/Python_projects/termostablizator/sou.py
All done!
2 files reformatted.
```

Здесь говориться, что отработал хук `black` и отформатировал 2 файла.

63. Разработка интерактивных карт с помощью библиотеки Folium

Полезная статья по разработке интерактивных карт https://habr.com/ru/company/skillfactory/blog/521840/.

Для рассматриваемой задачи можно выбрать следующие библиотеки: `Altair`, `Plotly` и `Folium`[41] https://pypi.org/project/folium/ (предпочтнее отдается этой библиотеке).

Для развертывания приложения, созданного с помощью библиотеки `folium` на облачной платформе `Streamlit` требуется использовать библиотеку `streamlit_folium`[42] https://github.com/randyzwitch/streamlit-folium.

`Folium` – полностью настраиваемая и интерактивная. Включает подсказки, всплывающие окна и пр. Для работы со специальными файлами формы (shapefile) можно использовать библиотеку `pyshp` https://github.com/GeospatialPython/pyshp/.

Интерактивная карта (еще говорят, хороплет) требует двух видов данных: геопространственные данные, географические границы для заполнения карт (обычно это векторный файл `.shp` (Shapefile) или GeoJSON), и две точки на каждом квадрате карты для цветного кодирования.

Пример простого приложения

[41] Устанваливается как обычно `pip install folium`

[42] Установить можно так: `pip install streamlit-folium` или `conda install -c conda-forge streamlit-folium`

```
import folium

# создать объект карты
m = folium.Map(location=[45.5236, -122.6750], tiles="Stamen Toner", zoom_start=13)

# добавить элемент на карту
folium.Circle(
    radius=100,
    location=[45.5244, -122.6699],
    popup="The Waterfront",
    color="crimson",
    fill=False,
).add_to(m)

folium.CircleMarker(
    location=[45.5215, -122.6261],
    radius=50,
    popup="Laurelhurst Park",
    color="#3186cc",
    fill=True,
    fill_color="#3186cc",
).add_to(m)

m
```

Для наложения, например, сети газопроводов в плане

Список иллюстраций

Список литературы

1. *Лутц М.* Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.
2. *Бизли Д.* Python. Подробный справочник. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с.
3. *Чакон С.*, *Штрауб Б.* Git для профессионального программиста. – СПб.: Питер, 2020. – 496 с.
4. *Рамальо Л.* Python. К вершинам мастерства. – М.: ДМК Пресс, 2016. – 768 с.
5. *Слаткин Б.* Секреты Python: 59 рекомендаций по написанию эффективного кода. – М.: ООО «И.Д. Вильямс», 2016. – 272 с.
6. *Прохоренок Н.А.*, *Дронов В.А.* Python 3 и PyQt 5. Разработка приложений. – СПб.: БХВ-Петербург, 2016. – 832 с.
7. *Chandola V.*, *Banerjee A.* etc. Anomaly detection: A survey, ACM Computing Surveys, vol. 41(3), 2009, pp. 1–58.
8. *Элбон К.* Машинное обучение с использованием Python. Сборник рецептов. – СПб.: БХВ-Петербург, 2019. – 384 с.
9. *Карау Х.*, *Уоррен Р.* Эффективный Spark. Масштабирование и оптимизация. – СПб. Питер, 2018. – 352 с.
10. *Рашка С.*, *Мирджалили В.* Python и машинное обучение: машинное и глубокое обучение с использованием Python, scikit-learn и TensorFlow. – СПб.: ООО «Диалектика», 2019. – 656 с.
11. *Чан У.* Python: создание приложений. Библиотека профессионала, 3-е изд. – М.: ООО «И.Д. Вильямс», 2015. – 816 с.