Дата последней сборки документа:
31 мая 2020 г. в 16:55

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.

Содержание

1. Градиентный бустинг

1.1. Общие сведения

1.2. Особенности реализации в пакете `sklearn`

1.3. Особенности реализации в пакете `XGBoost`

1.3.1. 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. 1), по которым будет проводиться поиск интересующего пакета (в данном случае пакета **xgboost**), а затем можно воспользоваться конструкцией

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

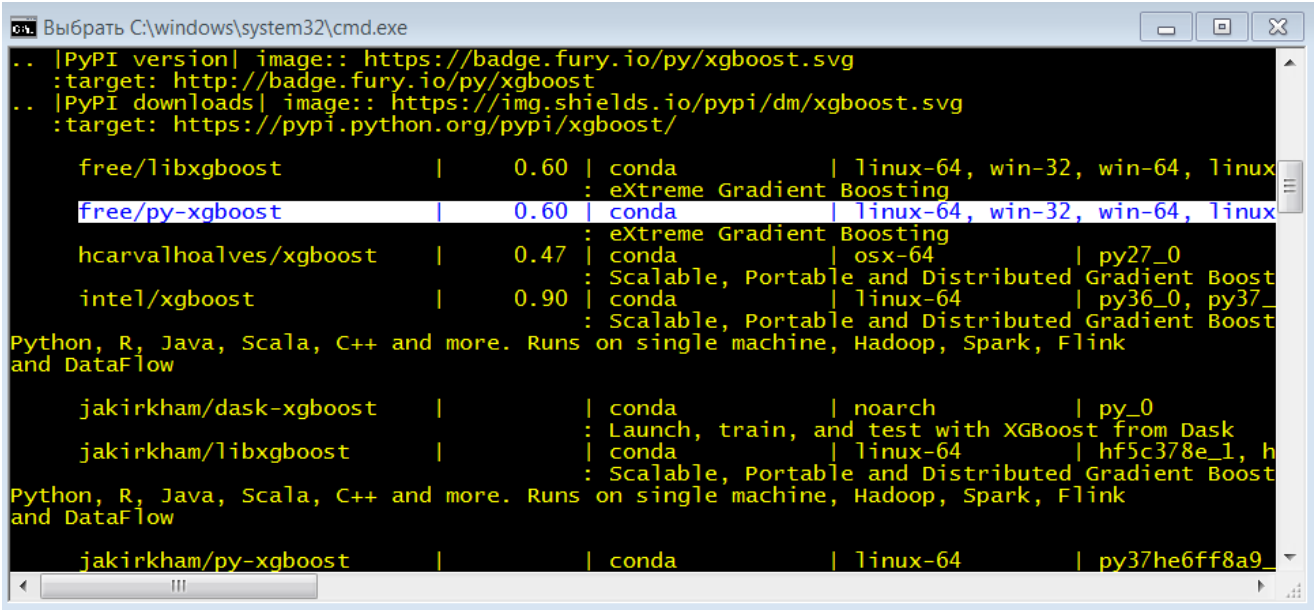

Рис. 1. Окно командной оболочки `cmd.exe` со списком доступных каналов, по которым будет проводиться поиск пакета **xgboost**

После, выбрав канал, можно приступать к установке пакета

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

1.3.2. 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. 2, рис. 3 и рис. 4 изображены графики важности признаков.

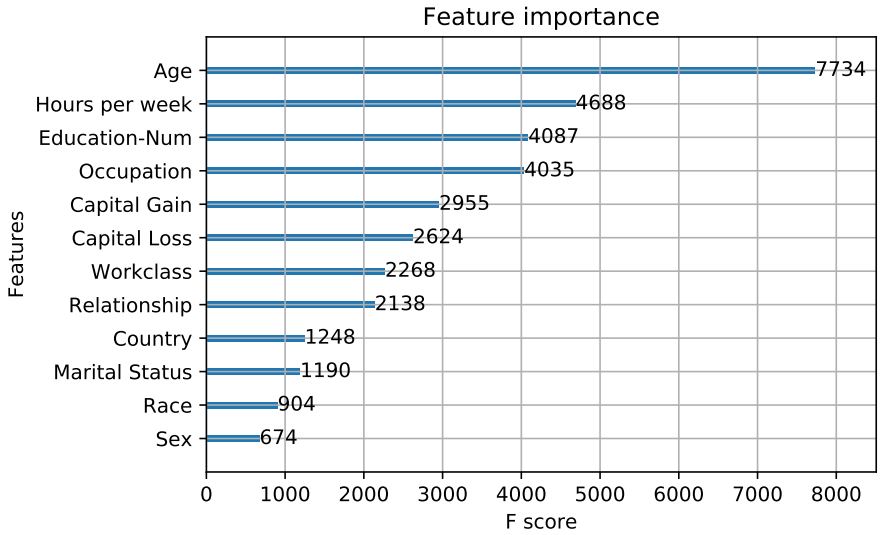

Рис. 2. График важности признаков `xgboost.plot_importance(model)`, построенный с помощью пакета `xgboost`

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,
- `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,

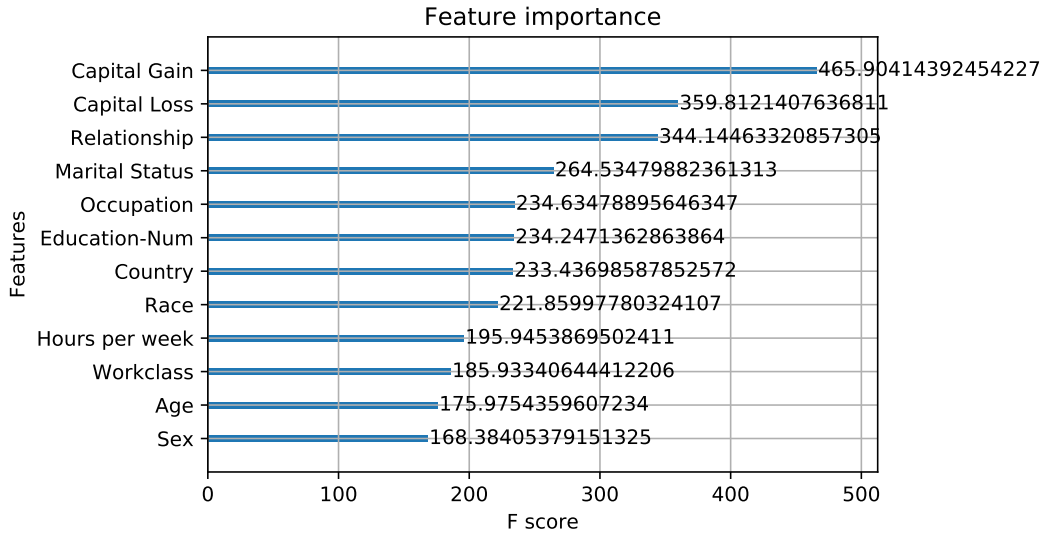

Рис. 3. График важности признаков `xgboost.plot_importance(model, importance_type='cover')`, построенный с помощью пакета `xgboost`

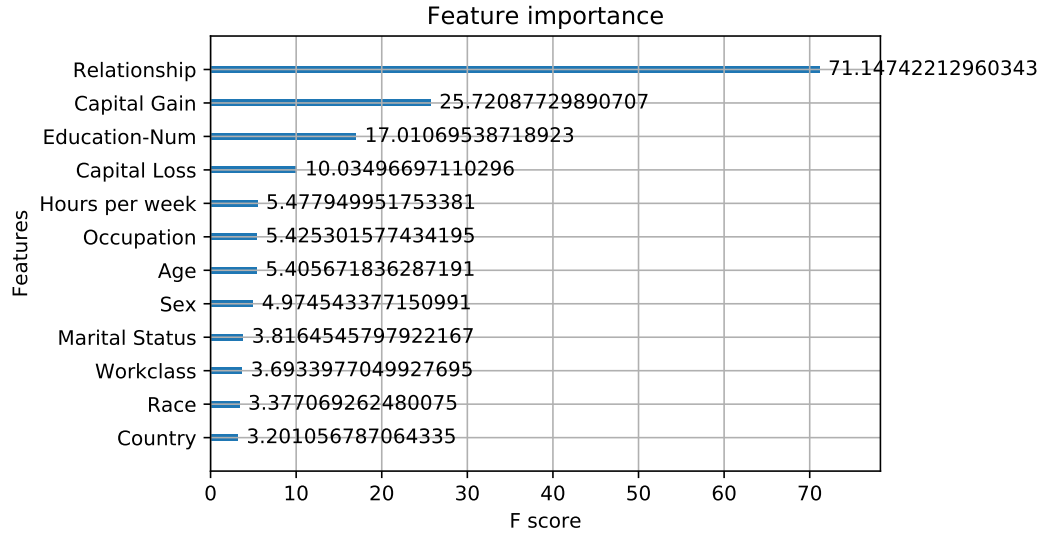

Рис. 4. График важности признаков `xgboost.plot_importance(model, importance_type='gain')`, построенный с помощью пакета `xgboost`

- `gain`: среднее снижение потерь на обучающем наборе данных, полученное при использовании i-ого признака.

1.4. Особенности реализации в пакете `LightGBM`

1.5. Особенности реализации в пакете `CatBoost`

2. 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].

3. Хэшируемые пользовательские классы в языке Python

Чтобы класс был хэшируемым[1], 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
    '''
    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]
```

[1] 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

```
            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[2] *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]

---

[2] Побитовые операторы рассматривают операнды как бинарные последовательности

4. 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 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. 5)

```
>>> 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!

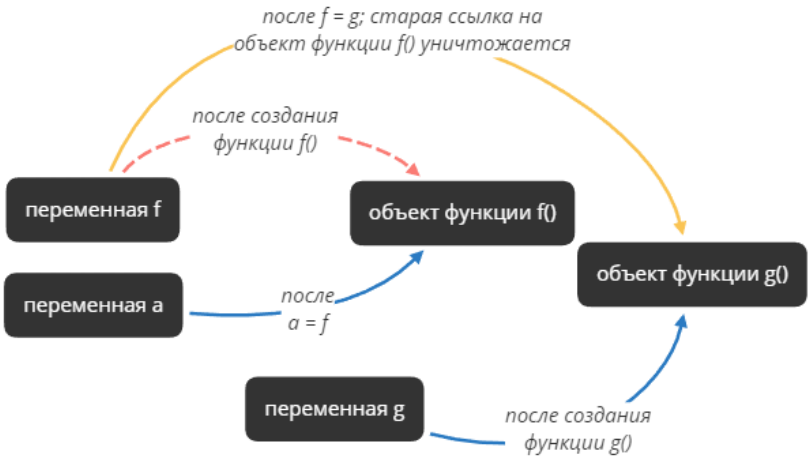

Рис. 5. Схема, описывающая связи между именами функций и их объектами

5. 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__`[3].

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

1. Вызывает *генераторную функцию* `looking_glass()`[4] и запоминает объект-генератор (пусть называется `gen`),
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:
```

[3] Этот класс называется `_GeneratorContextManager`

[4] При вызове генераторной функции возвращается объект-генератор

```
        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

---

6. 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[5],
- 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).

6.1. 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`[6] (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__`[7], 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)
```

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

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

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

```
    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):
        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__`)

6.2. 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]

6.3. 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.

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`

- в случае `==` и `!=`, если инверсный (правый) вызов завершается ошибкой, то `Python` сравнивает идентификаторы объектов, а не возбуждает исключение (см. табл. 1).

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`

7. 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*[8] [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*

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

```
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)
```

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`

---

8. 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»[9], например, печатает строку в терминале. *Задекорированные* же функции выполняются строго в результате явного вызова [4, стр. 217].

8.1. 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)
        print(f'{elapsed}, {name}({arg_str}) -> {result}')
        return result  # вернуть результат
return clocked
```

Использование декоратора выглядит так

clockdeco_demo.py

```
1  import time
2  from clockdeco import clock
3
4  def simple_deco_1(f):
5      '''
6      Декоратор с замыканием
7      '''
8      def inner():
9          print('test string from `simple_deco_1`')  # <- строка НЕ будет выведена
10                                                    # после загрузке модуля
11     return inner
12
13 def simple_deco_2(f):
14     '''
15     Простой одноуровневый декоратор
16     '''
17     print('test string from `simple_deco_2`')  # <- строка будет выведена в терминал
18                                               # сразу после загрузки модуля
19     return f
20
21 @simple_deco_1  # simple_func_1 = simple_deco_1(f=simple_func_1) -> inner
22 def simple_func_1():
```

[9] 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('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
```

8.2. 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*

8.3. 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].

8.4. 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()  # выведет 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`.

8.5. 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 множенство 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
            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)
```

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 следователь-

но 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, то мы делаем эту переменную *локальной* для замыкания `clocked`, т.е. переменная `count` перестает быть свободной.

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
```

8.6. 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. 6).

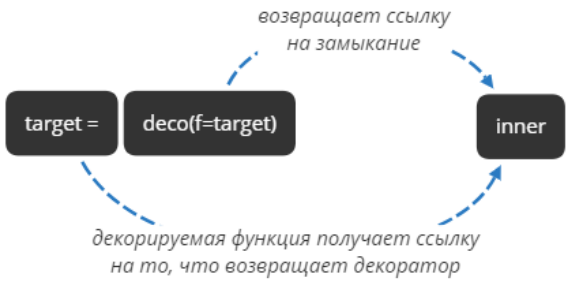

Рис. 6. К вопросу о механизме работы декоратора с вложенной функцией

9. 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[10]: *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].

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

[10] Определение содержит авторские правки

```
>>> 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].

9.1. 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

10. 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[11] (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 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*

11. 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.

[11] 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

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.

11.1. 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.

11.2. 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.

11.3. 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.

11.4. Логистическая регрессия в пространстве логитов

11.5. 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.

11.6. 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.

12. Приемы работы с менеджером пакетов `conda`

12.1. 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`[12]

```
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
```

12.2. 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

[12] В данном случае чтобы удалить виртуальную среду достачно просто удалить директорию `envs`

```
conda config --set env_prompt ({name})
```

которая добавит в конфигурационный файл `.condarc` следующую строку

.condarc

```
...
env_prompt: ({name})
```

и теперь имя окружения будет `(envs)`.

Деактивировать виртуальное окружение

```
conda deactivate
```

12.3. 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`[13], 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.

12.4. 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
```

12.5. Удаление виртуального окружения

Удалить виртуальное окружение `heroku_env`

```
conda env remove --name heroku_env
```

12.6. Экспорт виртуального окружения в `environment.yml`

Экспортировать активное виртуальное окружение в yml-файл

```
conda env export > environment.yml
```

[13] Этот файл должен находится в той же директории что и директория окружения `envs`

13. 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.

14. Управление локальными переменными окружения проекта

Для того чтобы создать *локальные переменные проекта*[14] 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)))
```

15. Приемы работы с модулем subprocess

Ниже приводится пример использования модуля subprocess для отыскания самого большого файла в git-репозитории

[14] 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.')
```

16. Приемы работы с пакетом Vowpal Wabbit

17. Приемы работы с библиотекой pandas

17.1. 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('category').apply(lambda col: col.unique().shape[0])
```

17.2. Число пропущенных значений в объекте DataFrame

Информацию по числу пропущенных значений в каждом столбце можно вывести следующим образом

```
X.isna().any(axis=0)
```

17.3. 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. 7.

	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. 7. Отформатированный вывод `DataFrame`

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'),
                '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. 8. 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. 8. Результат применения функции `color_code_freq_cat`

18. Интерпретация моделей и оценка важности признаков с библиотекой SHAP

18.1. Общие сведения о значениях Шепли

В библиотеке SHAP https://github.com/slundberg/shap для оценки *важности признаков* используются *значения Шепли*[15] (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*[16] i-ого признака f_i на значения целевого вектора y, а вот *важность признака* в контексте модели можно 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.

[15] Термин пришел из теории кооперативных игр

[16] Еще эту оценку можно интерпретировать как *вклад*

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)

18.2. 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
```

18.2.1. Локальная интерпретация отдельной точки данных обучающего набора

Теперь можно построить локальную интерпретацию для одной точки данных из обучающего набора (см. рис. 9)

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. 9. Локальная интерпретация для одной точки данных обучающего набора

Еще можно построить график частичной зависимости (рис. 10)

```
shap.dependence_plot('LSTAT', shap_values, X_train)
```

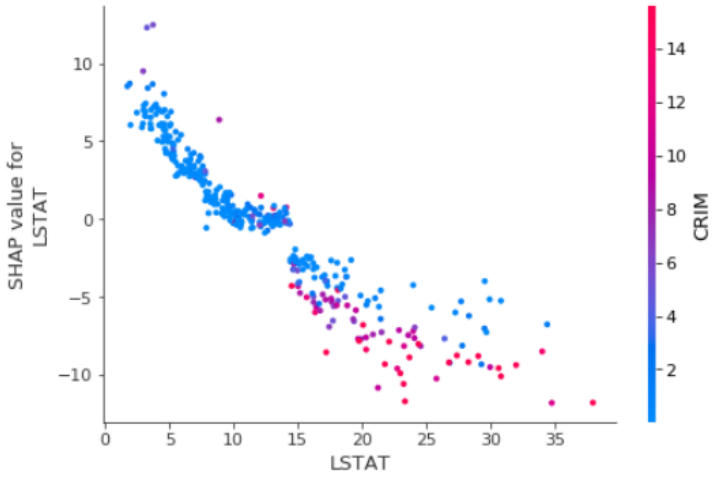

Рис. 10. График частичной зависимости признака `LSTAT` от значений Шепли с учетом влияния признака `CRIM`

18.2.2. Локальная интерпретация отдельной точки данных тестового набора

Прежде чем приступить к вычислению значений Шепли, следует создать поверхностную копию тестового набора данных

```
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. 11).

Из рис. 11 видно, что признаки с различной «силой»[17], которая определяется значениями Шепли, смещают предсказание модели на данной точке. Например, признак `LSTAT` (процент населения с низким социальным статусом) в значительной степени *повышает*[18] 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.

[17] Ширина полосы

[18] Потому что значение этого признака невелико; чем меньше процент населения с низким социальным статусом проживает в округе, тем выше стоимость квартиры

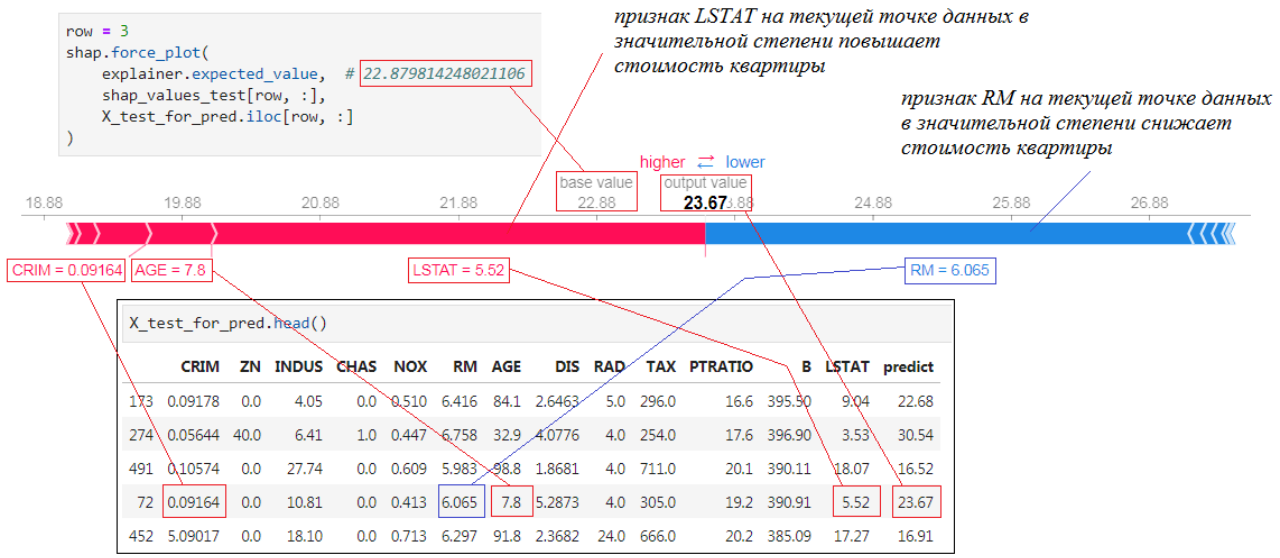

Рис. 11. Локальная интерпретация для одной точки данных тестового набора

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, :]
)
```

18.2.3. Глобальная интерпретация модели на тестовом наборе данных

Удобно работать с диаграммой рассеяния `shap.summary_plot` (рис. 12), 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)
```

Какие выводы можно сделать из рис. 12:

- 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. 13).

```
shap.dependence_plot('CRIM', shap_values_test[:, :-1], X_test_pred.iloc[:, :-1])
```

Какие выводы можно сделать из рис. 13:

- Чем выше уровень преступности в городе, тем в большей степени снижается стоимость квартиры,

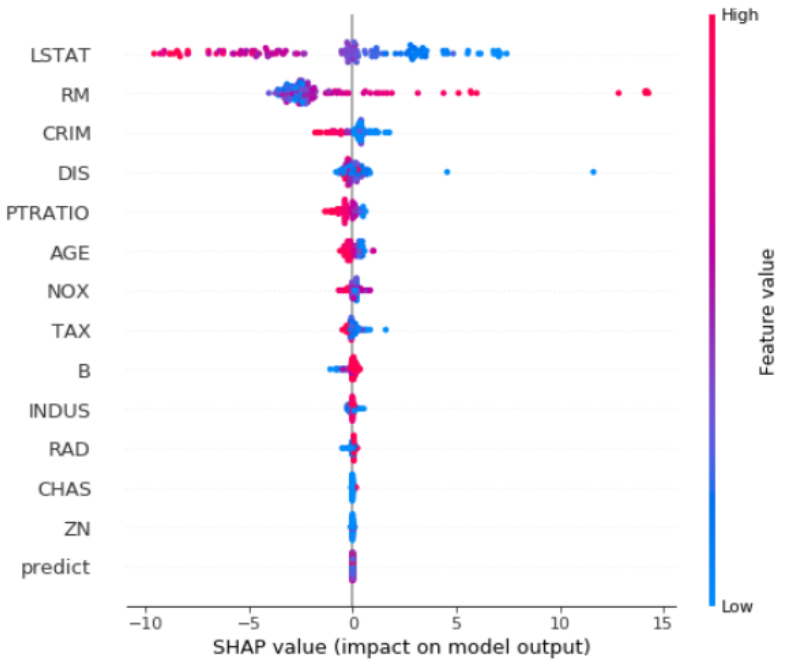

Рис. 12. Диаграмма рассеяния для точек тестового набора данных

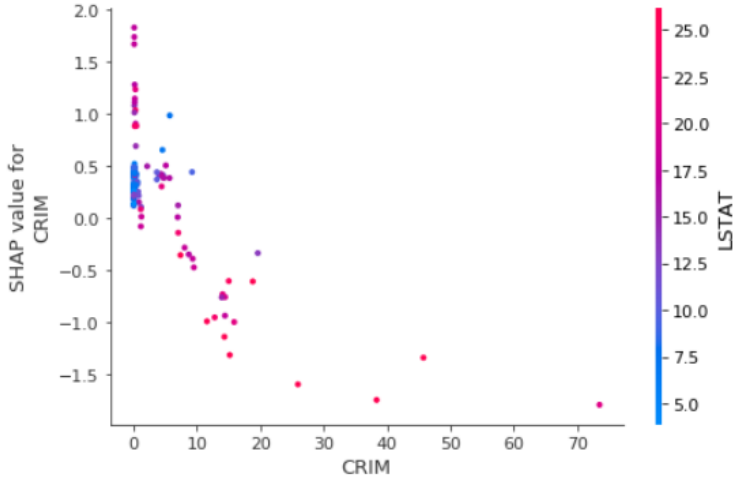

Рис. 13. График частичной зависимости признака `CRIM` от значений Шепли с учетом влияния `LSTAT`

- 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.

19. 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
```

20. 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'
```

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. *Сляткин 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.