

# 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

<b>1 Вопросы</b>	<b>10</b>
<b>2 Основные термины</b>	<b>11</b>
<b>3 Оценка потребления ресурсов Python-приложением</b>	<b>12</b>
<b>4 Теория алгоритмов и структуры данных</b>	<b>12</b>
<b>5 NP-полнота</b>	<b>26</b>
<b>6 Настройка непрерывной интеграции (CI) на GitHub Actions</b>	<b>29</b>
<b>7 Настройка непрерывной доставки (CD) с помощью Codefresh</b>	<b>33</b>
<b>8 Разработка собственных Python-пакетов для PyPI</b>	<b>33</b>
<b>9 Формы записи дисперсии</b>	<b>51</b>
<b>10 Обработка исключений при чтении данных в Pandas</b>	<b>51</b>
<b>11 Выход из приложения при перехвате исключения ветками try-except</b>	<b>52</b>
<b>12 Логгирование в Python</b>	<b>52</b>
<b>13 Бинарный поиск</b>	<b>52</b>
<b>14 Временная сложность алгоритмов и нотация О-большое</b>	<b>53</b>
<b>15 Управление памятью в Python</b>	<b>54</b>
<b>16 REST API</b>	<b>57</b>
<b>17 Приемы работы с библиотекой argparse</b>	<b>58</b>
<b>18 Приемы работы с MLflow</b>	<b>58</b>
<b>19 WSGI- и ASGI-серверы</b>	<b>81</b>

<b>20 NGINX</b>	<b>82</b>
<b>21 Приемы разработки web-приложений с помощью FastAPI</b>	<b>82</b>
<b>22 Приемы разработки приложений с графическим интерфейсом пользователя с использованием библиотеки DearPyGui</b>	<b>99</b>
<b>23 Использование Google Drive как хранилище артифактов ML-пайплайнов</b>	<b>99</b>
<b>24 Конфигурационные файлы как интерфейс доступа к Python-сценарию</b>	<b>99</b>
<b>25 Упаковка ML-пайплайна в docker-образ</b>	<b>100</b>
<b>26 Приемы работы с dataclass</b>	<b>102</b>
<b>27 Тестирование в Python</b>	<b>104</b>
<b>28 Автоматическое тестирование в Python</b>	<b>117</b>
<b>29 Инструменты автоматического форматирования, инспектирования и анализа кода</b>	<b>117</b>
<b>30 Тонкости импортирования модулей и пакетов в Python</b>	<b>120</b>
<b>31 Логистическая функция потерь</b>	<b>122</b>
<b>32 Автоматический анализ кода с платформой DeepSource</b>	<b>123</b>
<b>33 Python и LATEX</b>	<b>125</b>
<b>34 Адаптивный бустинг</b>	<b>126</b>
<b>35 Градиентный бустинг</b>	<b>131</b>
<b>36 Экстремальный градиентный бустинг с XGBoost</b>	<b>136</b>
<b>37 Потоки и процессы. Глобальная блокировка интерпретатора</b>	<b>144</b>
<b>38 Форматирование строк в языке Python</b>	<b>145</b>
<b>39 SSH-клиент в браузере</b>	<b>145</b>
<b>40 Большие данные в Hadoop</b>	<b>146</b>
<b>41 Теорема Байеса</b>	<b>146</b>
<b>42 Глубокое обучение</b>	<b>148</b>
<b>43 Хэшируемые пользовательские классы в языке Python</b>	<b>151</b>
<b>44 Как интерпретировать связь между именем функции и объектом функции в Python</b>	<b>152</b>

45 Использование @contextmanager	153
46 Перегрузка операторов в языке Python	156
47 Области видимости в языке Python	160
48 Декораторы в Python	162
49 Методы в Python	174
50 Замыкания/фабричные функции в Python	175
51 Значения по умолчанию изменяемого типа данных в Python	177
52 Генераторы, сопрограммы в Python	177
53 Библиотека functools	178
54 Калибровка классификаторов	179
55 Приемы работы с менеджером пакетов conda	180
56 Инструмент автоматического построения дерева проекта под задачи машинного обучения	183
57 Управление локальными переменными окружения проекта	183
58 Приемы работы с модулем subprocess	184
59 Решающие деревья и сопряженные вопросы	185
60 Анализ временных рядов	185
61 Методологии проектирования реляционного хранилища данных	198
62 Кодирование признаков	203
63 Машинное обучение с AutoML	206
64 Хранилища данных. DWH	206
65 Приемы работы с ETL-инструментом Apache NiFi	208
66 Приемы работы с библиотекой Vowpal Wabbit	208
67 Приемы работы с Microsoft Machine Learning for Apache Spark	209
68 Приемы работы с библиотекой BeautifulSoup	210
69 Приемы работы с библиотекой pandas	211
70 Приемы работы с библиотекой Plotly	215

71 Максимальный информационный коэффициент	216
72 Подбор гиперпараметров	216
73 Отбор признаков. Очень простые приемы	221
74 Важность признаков	222
75 Интерпретация моделей и оценка важности признаков с библиотекой SHAP	225
76 Перестановочная важность признаков в библиотеке eli5	229
77 Приемы работы с библиотекой River	230
78 Регулярные выражения в Python	230
79 Неравенство Маркова	231
80 Асинхронное программирование в Python	231
81 Приемы работы с DVC	233
82 Работа с базами данных в Python	233
83 Особенности использования менеджера пакетов pip	241
84 Приемы работы с flake8	242
85 Особенности работы с форматером black	242
86 Разработка интерактивных карт с помощью библиотеки Folium	244
Список иллюстраций	246
Список литературы	246

## Содержание

<b>1 Вопросы</b>	<b>10</b>
1.1 Как строится прогноз в классическом и экстремальной градиентном бустинге? . . . . .	10
1.2 Как строится поверхность разбиения дерева принятия решения и вычисляется прогноз на дереве . . . . .	10
<b>2 Основные термины</b>	<b>11</b>
<b>3 Оценка потребления ресурсов Python-приложением</b>	<b>12</b>
<b>4 Теория алгоритмов и структуры данных</b>	<b>12</b>
4.1 Хеш-таблицы . . . . .	13
4.2 Фильтры Блума . . . . .	16
4.3 Кучи . . . . .	18

4.4 Стеки и очереди . . . . .	21
4.5 Деревья поиска . . . . .	21
4.6 Массивы и связанные списки . . . . .	25
4.7 В-деревья . . . . .	26
<b>5 НР-полнота</b>	<b>26</b>
5.1 Краткая сводка . . . . .	26
5.2 Интуитивное определение SAT, NP и P . . . . .	28
<b>6 Настройка непрерывной интеграции (CI) на GitHub Actions</b>	<b>29</b>
6.1 Порядок работы с pull-request . . . . .	29
6.2 Конфигурационные файлы для непрерывной интеграции . . . . .	30
<b>7 Настройка непрерывной доставки (CD) с помощью Codefresh</b>	<b>33</b>
<b>8 Разработка собственных Python-пакетов для PyPI</b>	<b>33</b>
8.1 Семантическое управление версиями . . . . .	33
8.2 Краткая дорожная карта разработки собственного пакета . . . . .	33
8.3 Инструменты и приемы разработки пакетов . . . . .	35
8.4 Примеры файлов <code>setup.py</code> . . . . .	38
8.5 Точки входа в <code>setup.py</code> файлах . . . . .	43
<b>9 Формы записи дисперсии</b>	<b>51</b>
<b>10 Обработка исключений при чтении данных в Pandas</b>	<b>51</b>
<b>11 Выход из приложения при перехвате исключения ветками <code>try-except</code></b>	<b>52</b>
<b>12 Логгирование в Python</b>	<b>52</b>
<b>13 Бинарный поиск</b>	<b>52</b>
<b>14 Временная сложность алгоритмов и нотация O-большое</b>	<b>53</b>
14.1 Простыми словами . . . . .	53
14.2 Формальное определение $O$ -большое . . . . .	53
14.3 Обозначение $\Omega$ -большое и $\Theta$ -большое . . . . .	54
14.4 Обозначение $o$ -малое . . . . .	54
<b>15 Управление памятью в Python</b>	<b>54</b>
<b>16 REST API</b>	<b>57</b>
<b>17 Приемы работы с библиотекой argparse</b>	<b>58</b>
<b>18 Приемы работы с MLflow</b>	<b>58</b>
18.1 Общий сценарий использования MLflow на примере библиотеки H2O . . . . .	58
18.2 Общие сведения . . . . .	60
18.3 Обучение модели линейной регрессии с двумя гиперпараметрами . . . . .	63
18.4 Подбор гиперпараметров с использованием различных стратегий . . . . .	66

18.5 Конвейер с использованием MLflow . . . . .	70
<b>19 WSGI- и ASGI-серверы</b>	<b>81</b>
<b>20 NGINX</b>	<b>82</b>
<b>21 Приемы разработки web-приложений с помощью FastAPI</b>	<b>82</b>
21.1 Разворачивание FastAPI-приложений на платформе Deta . . . . .	97
21.2 Разворачивание FastAPI-приложения в ручную . . . . .	98
<b>22 Приемы разработки приложений с графическим интерфейсом пользователя с использованием библиотеки DearPyGui</b>	<b>99</b>
<b>23 Использование Google Drive как хранилище артифактов ML-пайплайнов</b>	<b>99</b>
<b>24 Конфигурационные файлы как интерфейс доступа к Python-сценарию</b>	<b>99</b>
<b>25 Упаковка ML-пайплайна в docker-образ</b>	<b>100</b>
<b>26 Приемы работы с dataclass</b>	<b>102</b>
<b>27 Тестирование в Python</b>	<b>104</b>
27.1 Что можно тестировать в задачах анализа данных . . . . .	104
27.2 Пример организации директории под тесты . . . . .	105
27.3 Пропуск тестов . . . . .	106
27.4 Запуск определенных тестов . . . . .	106
27.5 Параллельный запуск тестов . . . . .	107
27.6 Создание объектов, используемых в тестах, с помощью фикстур . . . . .	108
27.7 Параметрические фикстуры и тестовые функции . . . . .	110
27.8 Управляемые тесты с объектами-пустышками . . . . .	111
27.9 Выявление непротестированного кода с помощью coverage . . . . .	114
27.10 Виртуальные окружения . . . . .	116
<b>28 Автоматическое тестирование в Python</b>	<b>117</b>
<b>29 Инструменты автоматического форматирования, инспектирования и анализа кода</b>	<b>117</b>
<b>30 Тонкости импортирования модулей и пакетов в Python</b>	<b>120</b>
<b>31 Логистическая функция потерь</b>	<b>122</b>
<b>32 Автоматический анализ кода с платформой DeepSource</b>	<b>123</b>
32.1 Связка DeepSource и Travis CI . . . . .	125
<b>33 Python и LATEX</b>	<b>125</b>

<b>34 Адаптивный бустинг</b>	<b>126</b>
34.1 Адаптивный бустинг широкими мазками . . . . .	126
34.2 Обучение на взвешенной выборке . . . . .	127
34.3 Основные концепции адаптивного бустинга. Разбор видео-лекции Josh Starmer . . . . .	128
34.4 Пример работы алгоритма адаптивного бустинга . . . . .	129
<b>35 Градиентный бустинг</b>	<b>131</b>
35.1 Особенности реализации в пакете XGBoost . . . . .	131
35.1.1 Установка пакета xgboost на Windows . . . . .	133
35.1.2 Простой пример работы с xgboost и shap . . . . .	133
35.2 Особенности реализации в пакете LightGBM . . . . .	136
35.3 Особенности реализации в пакете CatBoost . . . . .	136
<b>36 Экстремальный градиентный бустинг с XGBoost</b>	<b>136</b>
36.1 Регрессия . . . . .	136
36.2 Классификация . . . . .	144
<b>37 Потоки и процессы. Глобальная блокировка интерпретатора</b>	<b>144</b>
<b>38 Форматирование строк в языке Python</b>	<b>145</b>
<b>39 SSH-клиент в браузере</b>	<b>145</b>
<b>40 Большие данные в Hadoop</b>	<b>146</b>
<b>41 Теорема Байеса</b>	<b>146</b>
41.1 Регистрация пользовательских функций выхода из приложения . . . . .	147
<b>42 Глубокое обучение</b>	<b>148</b>
42.1 Функции активации . . . . .	148
42.2 Стохастический градиентный спуск . . . . .	150
<b>43 Хэшируемые пользовательские классы в языке Python</b>	<b>151</b>
<b>44 Как интерпретировать связь между именем функции и объектом функции в Python</b>	<b>152</b>
<b>45 Использование @contextmanager</b>	<b>153</b>
<b>46 Перегрузка операторов в языке Python</b>	<b>156</b>
46.1 Перегрузка оператора сложения . . . . .	157
46.2 Перегрузка оператора умножения на скаляр . . . . .	158
46.3 Операторы сравнения . . . . .	159
<b>47 Области видимости в языке Python</b>	<b>160</b>
<b>48 Декораторы в Python</b>	<b>162</b>
48.1 Реализация простого декоратора . . . . .	162
48.2 Кэширование с помощью functools.lru_cache . . . . .	165

48.3 Одиночная диспетчеризация и обобщенные функции . . . . .	165
48.4 Композиции декораторов . . . . .	166
48.5 Параметризованные декораторы . . . . .	167
48.6 Цепочка параметрических декораторов . . . . .	170
48.7 Обобщение по механизму работы декораторов . . . . .	172
48.8 Написание декораторов класса . . . . .	172
<b>49 Методы в Python</b>	<b>174</b>
49.1 Статические методы . . . . .	174
49.2 Классовые методы . . . . .	174
<b>50 Замыкания/фабричные функции в Python</b>	<b>175</b>
50.1 Области видимости и значения по умолчанию применительно к переменным цикла	175
<b>51 Значения по умолчанию изменяемого типа данных в Python</b>	<b>177</b>
<b>52 Генераторы, сопрограммы в Python</b>	<b>177</b>
<b>53 Библиотека functools</b>	<b>178</b>
53.1 Каррированные функции с помощью <code>functools.partial</code> . . . . .	178
<b>54 Калибровка классификаторов</b>	<b>179</b>
54.1 Непараметрический метод гистограммной калибровки (Histogram Binning) . . . . .	179
54.2 Непараметрический метод изотонической регрессии (Isotonic Regression) . . . . .	179
54.3 Параметрическая калибровка Платта (Platt calibration) . . . . .	179
54.4 Логистическая регрессия в пространстве логитов . . . . .	180
54.5 Деревья калибровки . . . . .	180
54.6 Температурное шкалирование (Temperature Scaling) . . . . .	180
<b>55 Приемы работы с менеджером пакетов conda</b>	<b>180</b>
55.1 Создание виртуального окружения . . . . .	180
55.2 Активация/деактивация виртуального окружения . . . . .	182
55.3 Обновление виртуального окружения . . . . .	182
55.4 Вывод информации о виртуальном окружении . . . . .	182
55.5 Удаление виртуального окружения . . . . .	183
55.6 Экспорт виртуального окружения в <code>environment.yml</code> . . . . .	183
<b>56 Инструмент автоматического построения дерева проекта под задачи машинного обучения</b>	<b>183</b>
<b>57 Управление локальными переменными окружения проекта</b>	<b>183</b>
<b>58 Приемы работы с модулем subprocess</b>	<b>184</b>
<b>59 Решающие деревья и сопряженные вопросы</b>	<b>185</b>
59.1 Коэффициент Джини . . . . .	185
59.2 Случайный лес . . . . .	185

<b>60 Анализ временных рядов</b>	<b>185</b>
60.1 Признаки на временных рядах . . . . .	185
60.2 Кросс-валидация на временных рядах . . . . .	186
60.3 Синус-косинусное кодирование во временных рядах . . . . .	187
60.4 Прогнозирование временных рядов. Метод имитированных исторических прогнозов	187
60.5 Обнаружение аномалий во временных рядах . . . . .	188
60.6 Приемы работы с библиотекой Prophet . . . . .	192
60.7 Преобразование нестационарного временного ряда в стационарный . . . . .	195
60.8 Стабилизация дисперсии . . . . .	196
<b>61 Методологии проектирования реляционного хранилища данных</b>	<b>198</b>
61.1 Основная терминология . . . . .	198
61.2 Базовые сведения о концепциях . . . . .	200
<b>62 Кодирование признаков</b>	<b>203</b>
<b>63 Машинное обучение с AutoML</b>	<b>206</b>
<b>64 Хранилища данных. DWH</b>	<b>206</b>
<b>65 Приемы работы с ETL-инструментом Apache NiFi</b>	<b>208</b>
<b>66 Приемы работы с библиотекой Vowpal Wabbit</b>	<b>208</b>
<b>67 Приемы работы с Microsoft Machine Learning for Apache Spark</b>	<b>209</b>
<b>68 Приемы работы с библиотекой BeautifulSoup</b>	<b>210</b>
68.1 Пример использования BeautifulSoup для скрапинга сайта . . . . .	210
<b>69 Приемы работы с библиотекой pandas</b>	<b>211</b>
69.1 Определить число уникальных значений в каждом категориальном признаке . . . . .	211
69.2 Срезы в мультииндексах . . . . .	212
69.3 Число уникальных значений категориальных признаков в объекте DataFrame . . . . .	212
69.4 Прочитать файл, распарсить временную метку, назначить временную метку индексом	212
69.5 Число пропущенных значений в объекте DataFrame . . . . .	212
69.6 Управление стилями объекта DataFrame . . . . .	212
69.7 Заполнить пропущенные значения средними по группе . . . . .	214
<b>70 Приемы работы с библиотекой Plotly</b>	<b>215</b>
<b>71 Максимальный информационный коэффициент</b>	<b>216</b>
<b>72 Подбор гиперпараметров</b>	<b>216</b>
72.1 Приемы работы с библиотекой hyperopt . . . . .	216
<b>73 Отбор признаков. Очень простые приемы</b>	<b>221</b>
<b>74 Важность признаков</b>	<b>222</b>

<b>75 Интерпретация моделей и оценка важности признаков с библиотекой SHAP</b>	<b>225</b>
75.1 Общие сведения о значениях Шепли . . . . .	225
75.2 Пример построения локальной и глобальной интерпретаций . . . . .	225
75.2.1 Локальная интерпретация отдельной точки данных обучающего набора . . .	226
75.2.2 Локальная интерпретация отдельной точки данных тестового набора . . .	226
75.2.3 Глобальная интерпретация модели на тестовом наборе данных . . . . .	228
<b>76 Перестановочная важность признаков в библиотеке eli5</b>	<b>229</b>
<b>77 Приемы работы с библиотекой River</b>	<b>230</b>
<b>78 Регулярные выражения в Python</b>	<b>230</b>
<b>79 Неравенство Маркова</b>	<b>231</b>
<b>80 Асинхронное программирование в Python</b>	<b>231</b>
80.1 Библиотека aiomisc . . . . .	231
<b>81 Приемы работы с DVC</b>	<b>233</b>
<b>82 Работа с базами данных в Python</b>	<b>233</b>
<b>83 Особенности использования менеджера пакетов pip</b>	<b>241</b>
<b>84 Приемы работы с flake8</b>	<b>242</b>
<b>85 Особенности работы с форматером black</b>	<b>242</b>
<b>86 Разработка интерактивных карт с помощью библиотеки Folium</b>	<b>244</b>
<b>Список иллюстраций</b>	<b>246</b>
<b>Список литературы</b>	<b>246</b>

## 1. Вопросы

- 1.1. Как строится прогноз в классическом и экстремальной градиентном бустинге?**
- 1.2. Как строится поверхность разбиения дерева принятия решения и вычисляется прогноз на дереве**

Для каждого узла дерева принятия решения строится разбиение. ВАЖНО: разбиение строится по какому-то одному признаку. Для построения разбиения нужно из всех доступных признаков матрицы признаков, выбрать тот признак и с тем пороговым значением, которые обеспечивают наибольшую эффективность разбиения родительского узла. Например, в задачах классификации в качестве показателя эффективности разбиения можно использовать информационный прирост, вычисленный либо на базе загрязненности Джини, либо на базе энтропии Шеннона. Чем больше информационный прирост, тем «чище» дочерние узлы. Или другими словами, чем больше информационный прирост, тем больше экземпляры, попавшие в рассматриваемый узел, похожи

друг на друга в смысле значений целевой переменной. В случае задач регрессии, экземпляры обучающего набора данных группируются по показателю разброса вокруг среднего. Чем больше значения вещественной целевой переменной похожи друг на друга (то есть имеют плюс/минус один порядок), тем меньше дисперсия вокруг среднего и тем чище дочерний узел.

Рекурсивное разбиение данных повторяется до тех пор, пока все точки данных в каждой области разбиения (каждом листе дерева решений) не будут принадлежать одному и тому же значению целевой переменной (классу или количественному значению).

Выход по листу в задаче регрессии вычисляется как среднее арифметическое значений вещественной целевой переменной, ассоциированных с экземплярами обучающего набора данных, попавшими в рассматриваемый лист. В задаче классификации выход по листу строится как среднее арифметическое вероятностей принадлежности экземпляра к определенному классу.

Прогноз для новой точки данных получают следующим образом: сначала выясняют, в какой области разбиения пространства признаков находится данная точка, а затем назначают класс, к которому относится большинство точек в этой области. Область может быть найдена с помощью обхода дерева, начиная с корневого узла, и путем перемещения влево или вправо, в зависимости от того, выполняется ли очередной тест.

В задаче регрессии используется точно такой же подход. Для получения прогноза мы обходим дерево на основе тестов в каждом узле и находим лист, в который попадает новая точка данных. Выходом для этой точки данных будет значение целевой переменной, усредненное по всем *обучающим* точкам в этом листе.

Или другими словами, после прохода по дереву решений и выяснению листа, к которому принадлежит новая точка данных (экземпляр *тестового* набора данных), этой точке данных будет назначен выход листа, вычисленный как среднее арифметическое значений вещественной целевой переменной, ассоциированных с экземплярами *обучающего* набора данных, попавшими в рассматриваемый лист.

**ВАЖНО:** разбиение строится по значениям *одного* выбранного признака, а «мера похожести» экземпляров обучающего набора данных, попавших в узел, и выход по листу строится на основании значений целевой переменной, ассоциированных с экземплярами обучающего набора данных, попавшими в рассматриваемый узел или лист. В результате (для задачи регрессии) поверхность принятия решения представляет собой ступенчатую поверхность, имеющую константное значение в пределах отдельно взятого листа.

## 2. 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  $\alpha$ -квантиль однозначно задается уравнением

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

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

$$\mathbb{P}\left(x_{\frac{1-\alpha}{2}} \leq X \leq x_{\frac{1+\alpha}{2}}\right) = \alpha.$$

*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 в приложениях, обрабатывающих информацию в «реальном времени» (IoT-приложения, чаты и пр.)

### 3. Оценка потребления ресурсов Python-приложением

Оценить потребление ресурсов Python-приложением можно с помощью модуля `memory_profiler`<sup>1</sup>

```
from memory_profiler import memory_usage
print(memory_usage())
```

### 4. 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.

Обозначение *O*-большое распределяет алгоритмы по группам согласно их асимптотическим временам работы в худшем случае, таким как линейно-временные ( $O(n)$ ) или логарифмически-временные ( $O(n \log n)$ ) алгоритмы и операции [13, 246].

<sup>1</sup>Ставится как обычно с помощью pip: `pip install memory_profiler`

Смысл *структуре данных* заключается в организации данных таким образом, чтобы к ним можно было получить доступ быстро и с пользой. Например, *очередь* последовательно организует данные так, что удаление объекта в ее начале или добавление объекта в ее конце занимает *постоянное время*. *Стек* в качестве структуры данных позволяет удалять объект или добавлять объект в его начало за *постоянное время*.

## 4.1. Хеш-таблицы

Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя поиск элемента в хеш-таблице может в *наихудшем* случае потребовать столько же времени, сколько на поиск в связанным списке, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях *среднее* время поиска элемента в хеш-таблице составляет  $O(1)$  [16, 285].

Хеш-таблицы, подобно кучам и деревьям поиска, поддерживают эволюционирующее множество объектов, ассоциированных с ключами. В отличие от кучи и деревьев поиска, они не поддерживают никакой информации об упорядочивании вообще. Смысл существования хеш-таблицы состоит в том, чтобы *облегчить сверхбыстрый поиск* (по ключу), который также в этом контексте называется просмотром (lookup). Хеш-таблица может сказать, что в ней есть, а чего нет, и может сделать это очень и очень быстро (намного быстрее, чем куча или дерево поиска).

Концептуально хеш-таблицу можно рассматривать как *массив* [13]. Единственно, чем хороши массивы, – это немедленным случайному к ним доступом. В массиве, чтобы, например, узнать что находится в позиции или изменить значение в этой позиции нужно просто обратиться к этой позиции за постоянное время. *Массивы* прекрасно подходят для *чтения элементов* в произвольных позициях, потому что обращение к любому элементу в массиве происходит мгновенно. *Связанные списки* отлично подходят в тех ситуациях, когда данные должны читаться *последовательно*: сначала читается один элемент, по адресу переходим к следующему элементу и т.д. Хеш-таблицы сочетают в себе лучшие возможности массивов и связанных списков.

**Когда использовать хеш-таблицу:** если приложению требуется *быстрый поиск* с динамически изменяющимся множеством объектов, то хеш-таблица обычно является предпочтительным вариантом.

Быстродействие хеш-таблиц

- *поиск*:

- в среднем операция выполняется за постоянное время  $O(1)$ ,
  - в худшем случае – за время  $O(n)$ ,

- *вставка*:

- в среднем операция выполняется за постоянное время  $O(1)$ ,
  - в худшем случае – за время  $O(n)$ ,

- *удаление*

- в среднем операция выполняется за постоянное время  $O(1)$ ,
  - в худшем случае – за время  $O(n)$ ,

Для достижения постоянно-временного *поиска* в хеш-таблице со сплением списки корзин должны оставаться *короткими*, в идеале – с длиной не более малой константы [13, 212].

**ВАЖНО:** любые операции в хеш-таблицах *в среднем* выполняются за постоянное<sup>2</sup> время  $O(1)$ , а в худшем – за *линейное* время  $O(n)$  (это очень медленно).

Для разрешения коллизий в хеш-таблицах, как правило, используется:

- либо стратегия с использованием *связанных списков* (список может быть как односторонним, так и двунаправленным),
- либо стратегия с использованием *метода открытой адресации* (линейное исследование, квадратичное исследование, двойное хеширование).

**ВАЖНО:** для предотвращения коллизий необходимы:

- низкий коэффициент загрузки  $\alpha = n/b$ , где  $n$  – число элементов коллекции, а  $b$  – число ячеек хеш-таблицы,
- хорошая хеш-функция.

В хеш-таблице с *открытой адресацией* время выполнения операции чтения или вставки масштабируется вместе с числом проб, необходимых для отыскания пустой ячейки (либо искомого объекта). Когда коэффициент загрузки хеш-таблицы равен  $\alpha$ , то это означает, что  $\alpha$ -доля ячеек хеш-таблицы заполнена, а оставшаяся часть  $(1 - \alpha)$  является пустой.

В поиске на основе хеша  $n$  элементов коллекции  $C$  сначала загружаются в хеш-таблицу  $H$  с  $b$  ячейками (еще их называют «корзинами»), структурированными в виде массива. Этот шаг предварительной обработки имеет производительность  $O(n)$ , но улучшает производительность будущих поисков с использованием концепции хеш-функции.

*Хеш-функция* представляет собой детерминированную функцию, которая отображает каждый элемент  $C_i$  коллекции на целочисленное значение  $h_i$  (индекс ячейки в хеш-таблице). На минуту предположим, что  $0 \leq h_i < b$ . При загрузке элементов в хеш-таблицу элемент  $C_i$  вставляется в ячейку  $H[h_i]$ . После того как все элементы будут вставлены, поиск элемента  $t$  становится поиском  $t$  в  $H[hash(t)]$ .

Хеш-функция гарантирует только, что если два элемента,  $C_i$  и  $C_j$ , равны, то  $hash(C_i) = hash(C_j)$ . Может случиться так, что несколько элементов в коллекции  $C$  имеют одинаковые значения хеша; такая ситуация называется *коллизией*, и хеш-таблице необходима стратегия для урегулирования подобных ситуаций.

**ВАЖНО:** размер хеш-таблицы следует увеличить, если *коэффициент загрузки*  $\alpha$  начинает превышать 0.70 (это хорошее эмпирическое правило); тогда, при наличии правильно выбранной хеш-функции и непатологических данных, все наиболее распространенные стратегии разрешения коллизий, как правило, приводят к постоянно-временным хеш-табличным операциям [13, 225].

Наиболее распространенным решением является хранение в каждой ячейке *связанного списка* (несмотря на то, что *многие* из этих связанных списков будут содержать *только один элемент*); при этом в хеш-таблице могут храниться все элементы, вызывающие коллизии.

Поиск в связанных списках должен выполняться линейно, но он будет *быстрым*, потому что каждый из них может хранить максимум несколько элементов. При поиске на основе хеша количество сравнений строк *связано с длиной связанных списков*, а не с размером коллекции.

Целочисленный индекс ячейки в хеш-таблице для элемента  $s$  вычисляется как

$$hash(s) = |hashFun(s)| \mod b,$$

где  $hashFun(s)$  – хеш-функция от элемента,  $b$  – число ячеек в хеш-таблице.

---

<sup>2</sup>Постоянное время не означает, что операции выполняются мгновенно; просто время остается постоянным независимо от размера хеш-таблицы

Первичная память хеш-таблицы  $H$  должна быть достаточно большой, а *связанные списки элементов*, хранящиеся в ячейках, должны быть как можно меньше.

В качестве размера хеш-таблицы  $H$  обычно выбирается простое число, чтобы гарантировать, что использование оператора вычисления остатка от деления эффективно распределяет вычисляемые номера ячеек. На практике хорошим выбором является  $2^k - 1$ , несмотря на то что это значение не всегда простое.

**ВАЖНО:** вставка элемента в связанный список выполняется за постоянное время, а удаление элемента пропорционально длине списка. Если хеш-функция равномерно распределяет элементы, отдельные списки оказываются относительно короткими.

После того как хеш-функция возвращает *индекс ячейки* в хеш-таблице, мы смотрим, пуста ли соответствующая ячейка таблицы. Если она пуста, возвращаем значение `False`, указывающее, что искомого элемента нет в коллекции. В противном случае просматриваем связанный список для этой ячейки, чтобы определить наличие или отсутствие в нем искомого элемента.

**Среднее** время, необходимое для *поиска* элемента, является константой, или  $O(1)$ . Поиск состоит из одного просмотра  $H$  с последующим линейным поиском в коротком списке коллизий.

Определим для хеш-таблицы *коэффициент загрузки*  $\alpha$  как среднее количество элементов в связанным списке для некоторой ячейки  $H[h]$ . Точнее,  $\alpha = n/b$ , где  $n$  – количество элементов отображаемой коллекции, а  $b$  – количество ячеек в хеш-таблице.

**ВАЖНО:** По мере увеличения числа ячеек  $b$  в хеш-таблице *максимальная* длина связанных списков *уменьшается*, а количество ячеек, содержащих единственный элемент увеличивается.

**ВАЖНО:** По мере уменьшения коэффициента загрузки  $\alpha = n/b$  средняя длина каждого списка элементов также уменьшается, что приводит к улучшению производительности. То есть при фиксированном количестве элементов коллекции чем больше ячеек в хеш-таблице, тем лучше.

Так как хеш-таблица обычно может расти до размера, достаточно большого, чтобы все *связанные списки элементов были небольшими*, производительность поиска считается равной  $O(1)$ .

Один популярный вариант поиска на основе хеша модифицирует обработку коллизий путем наложения ограничения, заключающегося в том, что *каждая ячейка содержит один элемент*.

Вместо создания *связанного списка* для хранения всех элементов, которые хешируются в одну и ту же ячейку в хеш-таблице, используется *метод открытой адресации*, который хранит элементы, вызвавшие коллизию, в *некоторых других пустых ячейках хеш-таблицы H* [14, 140].

При использовании открытой адресации хеш-таблица снижает накладные расходы на хранение элементов путем устранения всех связанных списков.

Чтобы *вставить* элемент с использованием открытой адресации, вычисляется индекс ячейки в хеш-таблице  $h_k = \text{hash}(e)$ , которая должна содержать элемент  $e$ . Если ячейка  $H[h_k]$  пуста, ей присваивается значение элемента  $H[h_k] = e$  так же, как и в стандартном алгоритме. В противном случае выполняется *исследование* (probe) ячеек  $H$  с использованием той или иной стратегии исследования и элемент  $e$  помещается в первую обнаруженную пустую ячейку.

Стратегии исследований:

- Линейное исследование; выполняется неоднократный поиск в ячейках  $h_k = (h_k + c \cdot i) \bmod b$ , где  $c$  – целочисленное смещение, а  $i$  – количество последовательных исследований  $H$  (часто  $c = 1$ ); при использовании этой стратегии в  $H$  могут появиться кластеры элементов,
- Квадратичное исследование; выполняется неоднократный поиск в ячейках  $h_k = (h_k + c_1 \cdot i + c_2 \cdot i^2) \bmod b$ , где  $c_1$  и  $c_2$  – константы; этот подход обычно позволяет избежать кластеризации; полезными для практического применения являются значения  $c_1 = c_2 = 1/2$ ,

- Двойное хеширование; сходно с линейным исследованием, но в отличие от него  $c$  не является константой, а определяется второй хеш-функцией.

**ВАЖНО:** Во всех случаях, если пустая ячейка после  $b$  проб не обнаружена, вставка считается неудачной.

В хеш-таблице с использованием открытой адресации *проблемой* является *удаление* элементов. Для поддержки удаления при использовании открытой адресации необходимо пометить ячейку как удаленную.

Открытая адресация уменьшает общее количество используемой памяти, но требует существенно большего количества проб в наихудшем случае. Второе следствие заключается в том, что линейное исследование приводит к большему количеству проб из-за кластеризации.

Вычисленный коэффициент загрузки для хеш-таблиц описывает ожидаемую производительность *поиска* и *вставки*. Если коэффициент загрузки слишком высок, количество проб при поиске элемента становится чрезмерным, будь то в связанном списке ячейки или в цепочке ячеек при использовании открытой адресации. Хеш-таблица может увеличить количество ячеек и перестроится с помощью процесса, известного как «перехеширование» – нечастой операции, которая уменьшает коэффициент загрузки, хотя и является дорогостоящей операцией, – со временем работы  $O(n)$ .

Типичный способ изменения размера – удвоить количество имеющихся ячеек и добавить еще одну (так как хеш-таблицы обычно содержат нечетное количество ячеек). Когда становится доступным большее количество ячеек, все существующие элементы в хеш-таблице перехешируются и вносятся в новую структуру. Эта дорогостоящая операция снижает общую стоимость будущих поисков, но должна выполняться как можно реже; в противном случае не удастся получить амортизационную производительность хеш-таблиц, равную  $O(1)$ .

Выбор стратегии разрешения коллизий зависит от многих обстоятельств. Например, сценарий разрешения коллизий на базе связанных списков занимает больше места, чем открытая адресация (для хранения указателей в связанных списках), поэтому открытая адресация может быть предпочтительнее, когда пространство является первоочередной проблемой. Реализация удалений выполняется сложнее с открытой адресацией, чем со сцеплением, поэтому сцепление может быть предпочтительнее в приложениях с большим числом удалений.

**ВАЖНО:** поскольку хеш-таблицы с открытой адресацией хранят не более одного объекта в расчете на позицию в массиве, у них никогда не может быть коэффициента загрузки больше 1. Как только коэффициент загрузки стал равен 1, вставить больше объектов уже не получится. А вот в хеш-таблицу со сцеплением может быть вставлено произвольное число объектов, хотя ее результативность снижается по мере вставки большего числа объектов. Например, если загрузка равна 100, то средняя длина списка корзины тоже равна 100 [4].

## 4.2. Фильтры Блума

Фильтр Блума предоставляет альтернативную структуру *битового массива*  $B$ , которая обеспечивает константную производительность при добавлении элементов из коллекции  $C$  в битовый массив  $B$  или проверку того, что элемент *не был добавлен* в битовый массив  $B$ .

Как ни удивительно, это поведение не зависит от количества элементов, уже добавленных в  $B$ . Однако имеется и ловушка: при проверке, находится ли некоторый элемент в  $B$ , фильтр Блума может возвратить ложно-положительный результат, даже если на самом деле элемент в

$C$  отсутствует. Фильтр Блума позволяет *точно* определить, что элемент не был добавлен в  $B$ , поэтому никогда не возвращает ложно-отрицательный результат.

На рис. 1 в битовый массив  $B$  вставлены два значения,  $u$  и  $v$ . Таблица в верхней части рисунка показывает позиции битов, вычисленные хеш-функциями ( $k = 3$ ). Как можно видеть, фильтр Блума в состоянии быстро определить, что третье значение  $w$  не было вставлено в битовый массив  $B$ , поскольку одно из его  $k$  вычисляемых значений битов равно нулю (в данном случае это бит с позицией 6). Однако для значения  $x$  фильтр возвращает ложно-положительный результат, поскольку, несмотря на то что это значение не было вставлено в  $B$ , все  $k$  вычисляемых битов равны единицы.

	$h_1$	$h_2$	$h_3$
$u$	1	3	8
$v$	2	4	8
$w$	1	2	6
$x$	1	2	3

Биты $B$	0	1	1	1	1	0	0	0	1	0	0
	0	1	2	3	4	5	6	7	8	9	10

Рис. 1. Пример работы фильтра Блума

Фильтр Блума обрабатывает значения во многом так же, как и поиск на основе хеша. Алгоритм начинает работу с массива из  $m$  битов, каждый из которых изначально равен нулю. Имеется  $k$  хеш-функций, вычисляющих при вставке значений (потенциально различные) позиции битов в этом массиве.

Фильтр Блума возвращает значение `false`, если может доказать, что целевой элемент  $t$  еще не был вставлен в битовый массив и, соответственно, отсутствует в коллекции  $C$ . Алгоритм может вернуть значение `true`, которое может быть ложно-положительным результатом, если искомый элемент  $t$  был вставлен в массив. То есть если хотя бы один из битов имеет значение 0, значит, искомое значение не было добавлено в массив, однако, если все  $k$  битов имеют значение 1, можно только утверждать, что искомое значение *могло* быть добавлено в массив.

**ВАЖНО:** Фильтр Блума демонстрирует эффективное использование памяти, но он полезен только тогда, когда допустимы ложно-положительные результаты.

Общее количество памяти, необходимое для работы фильтра Блума, фиксировано и составляет  $m$  битов, и оно не увеличивается независимо от количества хранящихся значений. Кроме того, алгоритм требует только фиксированного количества испытаний  $k$ , поэтому каждая вставка и поиск могут быть выполнены за время  $O(k)$ , которое рассматривается как константное [14, 149].

Основная сложность фильтра заключается в разработке эффективных хеш-функций, которые обеспечивают действительно равномерное распределение вычисляемых битов для вставляемых значений. Хотя размер битового массива является константой, для снижения количества ложно-положительных срабатываний он может быть достаточно большим. И, наконец, нет никакой возможности удаления элемента из фильтра, поскольку потенциально это может фатально нарушить обработку других значений.

Фильтр Блума имеет предсказуемую вероятность ложного-положительного срабатывания в предположении, что  $k$  хеш-функций дают равномерно распределенные случайные величины. Достаточно точной оценкой  $p_k$  является следующая [14, стр. 149]

$$p_k = \frac{1}{2} \left[ 1 - \left( 1 - \frac{1}{m} \right)^{kn} \right]^k,$$

где  $k$  – число хеш-функций;  $m$  – длина битового массива (размер фильтра Блума в битах);  $n$  – количество добавленных в фильтр значений.

Если требуется, чтобы ложно-положительные срабатывания не превышали некоторого небольшого значения, следует выбрать  $k$  и  $m$  после оценки числа  $n$  вставляемых элементов. В литературе предлагается обеспечить значение  $1 - (1 - 1/m)^{kn}$ , близкое к  $1/2$ . Например, чтобы гарантировать, что ложно-положительные срабатывания для списка из 211 422 будут для менее 10%, следует установить значение  $m$  равным по крайней мере 1 120 000 битам.

Фильтр Блума поддерживает операции Вставить и Просмотреть (Поиск) в *постоянной времени* и он предпочтительнее хеш-таблиц в приложениях, в которых пространство ценится не вес золота, а периодические ложные утверждения не являются проблемой.

### 4.3. Кучи

*Куча* – структура данных, которая отслеживает эволюционирующее множество объектов с *ключами* и может быстро идентифицировать объект с наименьшим ключом.

Самые главные вещи, которые следует помнить о любой структуре данных, – это операции, которые она поддерживает, и время, необходимое для каждой из них.

Две самые важные операции, поддерживаемые кучами, это операции

- Вставить,
- Извлечь минимум<sup>3</sup>

**ВАЖНО:** Алгоритм HeapSort использует кучу для сортировки массива длины  $n$  за время  $O(n \log n)$ .

**ВАЖНО:** Кучи могут быть визуализированы как полные бинарные деревья, но реализованы в виде массивов.

Было бы легко поддерживать только операцию Вставка, повторно присоединяя новые объекты в конец массива или связанного списка (за постоянное время). Проблема в том, что операция Извлечь минимум требует линейно-временного исчерпывающего поиска по всем объектам. Ясно также как поддерживать только операцию Извлечь минимум – отсортировать исходное множество  $n$  объектов по ключу заранее раз и навсегда (с использованием времени  $O(n \log n)$  на предобработку), а затем последовательно вызывать операцию Извлечь минимум выхватывания объектов по одному с начала отсортированного списка (каждая за постоянное время). Загвоздка здесь в том, что любая простая реализация операции Вставить требует линейного времени. Хитрость кроется в разработке структуры данных, которая позволяет обеим операциям выполняться очень быстро. Именно в этом заключается смысл существования кучи.

**Теорема** (Время выполнения основных операций кучи). В куче с  $n$  объектами операции Вставить и Извлечь минимум выполняются за время  $O(\log n)$ .

---

<sup>3</sup>Структуры данных, поддерживающие эти операции, также называют *очередями с приоритетом*

**ВАЖНО:** Ни один из вариантов кучи не поддерживает операции **Извлечь минимум** и **Извлечь максимум** одновременно за  $O(\log n)$  – нужно выбрать, какой вариант требуется<sup>4</sup>.

**Теорема** (Время выполнения дополнительных кучевых операций). В куче с  $n$  объектами операции **Найти минимум**, **Объединить** в кучу и **Удалить** выполняются за время  $O(1)$ ,  $O(n)$  и  $O(\log n)$  соответственно.

Обобщим и приведем итоговый перечень показателей для куч на рис. 2. В случае операции **Извлечь минимум** нам нужно найти объект с наименьшим ключом, удалить его из кучи, а затем вернуть – то есть требуется перестроить кучу, поэтому данная операция зависит от числа объектов в куче. В случае же операции **Найти минимум** требуется просто найти объект с наименьшим ключом и вернуть его, а значит кучу перестраивать не нужно, и потому операция выполняется за постоянное время.

Операция	Время выполнения
Вставить	$O(\log n)$
Извлечь минимум	$O(\log n)$
-----	-----
Найти минимум	$O(1)$
Объединить в кучу	$O(n)$
Удалить	$O(\log n)$

Рис. 2. Операции и время их выполнения, поддерживаемые кучами: где  $n$  – это число объектов, хранящихся в куче

**Когда использовать кучу** Если приложению требуются быстрые вычисления минимума (либо максимума) для динамически изменяющегося множества объектов, куча обычно является предпочтительной структурой данных.

**Кучи в виде дерева** Кучи можно рассматривать как корневое бинарное дерево. Если число хранимых объектов на один меньше степени 2 (например,  $15 = 2^4 - 1$ ), то каждый уровень является *полным*. Когда число объектов находится между двумя такими числами, то единственный неполный уровень является последним, который заполняется слева направо.

**ВАЖНО:** Бинарное дерево называют *полным*, если каждый его узел либо представляет собой лист, либо имеет строго два дочерних узла. Узлы только с одним дочерним узлом в полном бинарном дереве отсутствуют. Следовательно, порядок дочерних узлов сохраняет информацию о местоположении.

**Свойство кучи:** для каждого объекта  $x$  ключ объекта  $x$  меньше или равен ключам его потомков рис. 3.

Для каждой пары «родитель – потомок» ключ родителя не больше ключа потомка. По мере обхода кучи вверх, ключи будут только уменьшаться. При этом корневой ключ является настолько малым, насколько это может получиться. Это должно звучать обнадеживающе, учитывая, что *смысл существования кучи – быстрые вычисления минимума*.

<sup>4</sup>Если нужно иметь обе, то можно использовать одну кучу каждого типа, либо обновить представление данных до сбалансированного дерева поиска

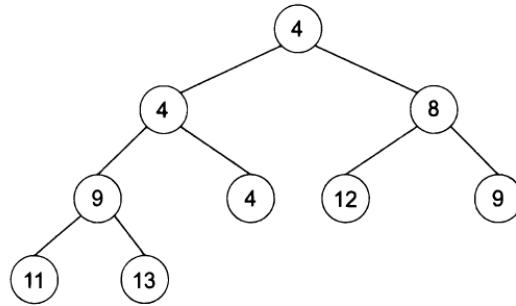


Рис. 3. Пример кучи в виде дерева с 9 объектами

**Кучи в виде массива** Мысленно мы визуализируем кучу как дерево, но в реализации мы используем массив с длиной, равной максимальному числу объектов, которые мы намерены хранить. Первый элемент массива соответствует корню дерева, последующие два элемента – следующему уровню дерева (в том же порядке), и так далее рис. 4.

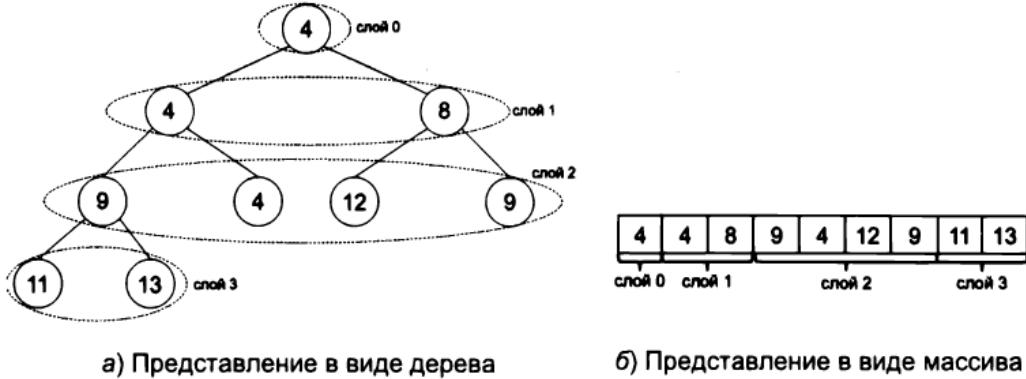


Рис. 4. Отображение древовидного представления кучи на его представление в виде массива

Для некорневого объекта (в позиции  $i \geq 2$ ) родителем  $i$  является объект в позиции  $\lfloor i/2 \rfloor$  (рис. 5).

Позиция родителя	$\lfloor i/2 \rfloor$ (при условии, что $i \geq 2$ )
Позиция левого потомка	$2i$ (при условии, что $2i \leq n$ )
Позиция правого потомка	$2i + 1$ (при условии, что $2i + 1 \leq n$ )

Рис. 5. взаимосвязь между позицией  $i \in \{1, 2, \dots, n\}$  объекта в куче и позициями его родителя, левого и правого потомка;  $n$  обозначает число объектов в куче

Такие простые формулы для перехода от потомка к его родителю и обратно существуют потому, что мы используем только *полные* бинарные деревья.

В общем случае операция Вставить присоединяет новый объект в конец кучи и многократно переставляет местами узлы нарушающие пары. В любой момент времени существует не более одной нарушающей пары «родитель – потомок» – пары, в которой новый объект является потомком.

Поскольку куча является полным бинарным деревом, она имеет  $\approx \log_2 n$  уровней, где  $n$  – число объектов в куче. Число перестановок местами не превышает число уровней, и на перестановку

требуется только постоянный объем работ. Таким образом, можно заключить, что в худшем случае время выполнения операции Вставить равно  $O(\log n)$ .

Так как операция Извлечь минимум предполагает удаление объекта с наименьшим ключом из кучи и возвращение этого объекта, то корень кучи гарантированно будет таким объектом. Задача заключается в восстановлении *полного бинарного дерева* и свойств кучи после удаления корня кучи.

В общем случае операция Извлечь минимум перемещает последний объект кучи в корневой узел (перезаписывая предыдущий корень) и повторно переставляя местами этот объект с его меньшим потомком. В любой момент времени существует не более двух нарушающих пар «родитель – потомок» – двух пар, в которых ранее последний объект является родителем.

Число перестановок не превышает число уровней, и на каждую перестановку требуется только постоянный объем работы. Поскольку существует  $\approx \log_2 n$  уровней, мы заключаем, что время выполнения операции Извлечь минимум в худшем случае равно  $O(\log n)$ , где  $n$  – это число объектов в куче.

#### 4.4. Стеки и очереди

Стек – последовательный контейнер, обеспечивающий вставку элемента в вершину стека и удаление элемента из вершины стека.

Очередь – последовательный контейнер, обеспечивающий добавление элементов в конец очереди и извлечение элементов с начала очереди.

Стеки и очереди представляют собой динамические множества, элементы из которых удаляются с помощью предварительно определенной операции Delete. Первым из *стека* (stack) удаляется элемент, который был помещен туда последним: в стеке реализуется стратегия «последним пришел – первым ушел»<sup>5</sup> (last-in, first-out – LIFO). Аналогично в *очереди* (queue) всегда удаляется элемент, который содержится в множестве дальше других: в очереди реализуется стратегия «первым пришел – первым ушел»<sup>6</sup> (first-in, first-out – FIFO) [16, 264].

Операция вставки Insert применительно к стекам часто называется записью в стек Push, а операция удаления Delete, которая вызывается без передачи аргумента, – снятием со стека Pop.

Применительно к очередям операция вставки называется Enqueue (поместить в очередь), а операция удаления – Dequeue (вывести из очереди).

#### 4.5. Деревья поиска

*Сбалансированное дерево поиска* (то есть дерево поиска, глубина которого всегда  $O(\log n)$ , где  $n$  – число объектов в дереве) расширяет функционал упорядоченного массива операциями вставки и удаления за логарифмическое время.

*Дерево поиска*, как и куча, представляет собой структуру данных для хранения эволюционирующего множества объектов, ассоциированных с ключами (и, возможно, с большим количеством других данных). Она поддерживает полное упорядочивание хранимых объектов и может поддерживать более богатое множество операций, несколько более медленного времени выполнения.

Использование упорядоченных массивов становится гораздо менее эффективным при частых изменениях базовой коллекции. Поиск на основе хеша может работать с динамическими коллекциями, но с целью эффективного использования ресурсов мы можем выбрать размер хеш-

<sup>5</sup>Стопка тарелок

<sup>6</sup>Обычная очередь в магазине

таблицы, который окажется слишком мал; зачастую у нас нет априорного знания о количестве элементов, которые будут храниться в таблице, так что *выбрать правильный размер хеш-таблицы оказывается очень трудной задачей*. Кроме того, хеш-таблицы не позволяют обходить все элементы в порядке сортировки [14, стр. 150].

Наиболее распространенным типом дерева поиска является *бинарное дерево поиска* (Binary Search Tree – BST) (рис. 6).

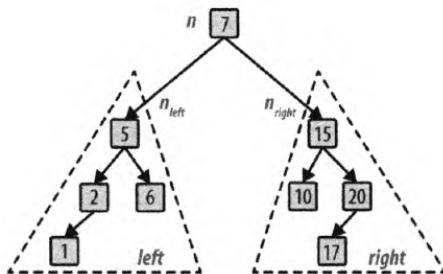


Рис. 6. Простое бинарное дерево поиска

Неплохая идея представлять дерево поиска как динамическую версию *отсортированного массива* – оно может делать то же, что и отсортированный массив, а также способно быстро производить операции вставки и удаления.

Поддерживаемые операции и их время выполнения в отсортированных массивах приведены на рис. 7. Смысл *дерева поиска* заключается в поддержке всех операций, поддерживаемых *отсортированным массивом*, а также *вставок* и *удалений*.

Операция	Время выполнения
Отыскать	$O(\log n)$
Минимум	$O(1)$
Максимум	$O(1)$
Предшественник	$O(\log n)$
Преемник	$O(\log n)$
Вывести в отсортированном порядке	$O(n)$
Выбрать	$O(1)$
Взять ранг	$O(\log n)$

Рис. 7. Отсортированные массивы:  $n$  – текущее число объектов в массиве

На рис. 8 приведен перечень показателей для деревьев поиска в сравнении с отсортированными массивами.

Важный нюанс: время выполнения на рис. 8 достигается только *сбалансированным* деревом поиска, то есть более сложной версией *стандартного бинарного дерева поиска*. Несбалансированное дерево поиска не гарантирует такое время выполнения.

*Сбалансированное дерево поиска* – дерево поиска, высота которого всегда равна  $O(\log n)$ .

**Когда использовать сбалансированное дерево поиска** Если приложение нуждается в поддержке упорядоченного представления (функционал отсортированного массива) динамически изменяющегося множества объектов (дополнительные операции удаления и вставки), то сба-

Операция	Отсортированный массив	Сбалансированное дерево поиска
Отыскать	$O(\log n)$	$O(\log n)$
Минимум	$O(1)$	$O(\log n)$
Максимум	$O(1)$	$O(\log n)$
Предшественник	$O(\log n)$	$O(\log n)$
Предыдущий	$O(\log n)$	$O(\log n)$
Вызвать в отсортированном порядке	$O(n)$	$O(n)$
Выбрать	$O(1)$	$O(\log n)$
Взять ранг	$O(\log n)$	$O(\log n)$
Вставить	$O(n)$	$O(\log n)$
Удалить	$O(n)$	$O(\log n)$

Рис. 8. Сбалансированные деревья поиска и отсортированные массивы:  $n$  – текущее число объектов в структуре данных

лансируемое дерево поиска (или структура на его основе) обычно является предпочтительной структурой данных.

Если обобщить бинарное дерево поиска имеет смысл использовать, когда [14, стр. 151]:

- требуется обход данных в возрастающем (или убывающем) порядке,
- размер множества данных неизвестен, а реализация должна быть в состоянии обработать любой возможный размер, помещающийся в памяти,
- множество данных – динамическое, с большим количеством вставок и удалений в процессе работы программы.

**ВАЖНО:** В худшем случае бинарное дерево поиска может выродиться и превратиться в связанный список. Для сохранения оптимальной производительности необходимо балансировать BST после каждого добавления (и удаления).

Если нужно поддерживать только *упорядоченное* представление, но *статической* совокупности данных (без вставок и удалений), то имеет смысл использовать вместо сбалансированного дерева поиска *отсортированный массив*. Если набор данных является динамическим, но вас интересуют только быстрые операции *взятия минимума* (либо *максимума*), то вместо сбалансированного дерева поиска используйте *кучи*.

AVL-деревья (деревья Адельсона-Вельского-Ландиса) – это самобалансирующиеся бинарные деревья поиска. Давайте определим понятие высоты AVL-узла. Высота листа равна 0, потому что он не имеет дочерних элементов. Высота узла, не являющегося листом, на 1 больше максимального значения высоты двух его дочерних узлов. Для обеспечения согласованности высоты несуществующего дочернего узла считается равной -1.

AVL-дерево гарантирует AVL-свойство для каждого узла, заключающееся в том, что разница высот для любого узла равна -1, 0 или 1. Разница высот определяется как  $height(left) - height(right)$ , т.е. высота левого поддерева минус высота правого поддерева. AVL-дерево должно обеспечивать выполнение этого свойства всякий раз при вставке значения в дерево или удалении из него.

**Детали реализации** (здесь описывается типичная реализация не обязательно сбалансированного бинарного дерева поиска) В бинарном дереве поиска каждый узел соответствует объекту (с ключом) и имеет три ассоциированных с ним указателя: указатель на родителя, указатель на

левого потомка и указатель на правого потомка. Любой из этих указателей может быть пустым, то есть иметь значение null, указывающее на отсутствие родителя или потомка.

*Свойство дерева поиска:*

- Для каждого объекта  $x$  объекты в левом поддереве объекта  $x$  имеют ключи меньше, чем у  $x$ ,
- Для каждого объекта  $x$  объекты в правом поддереве объекта  $x$  имеют ключи больше, чем у  $x$ .

Другими словами, значения ключа левого дочернего узла меньше значения ключа родительского узла, а значение ключа правого дочернего узла больше значения ключа родительского узла и, как следствие, ключи дочерних узлов упорядочены слева направо.

Свойство дерева поиска накладывает требование на каждый узел дерева поиска, а не только на корень (рис. 9).

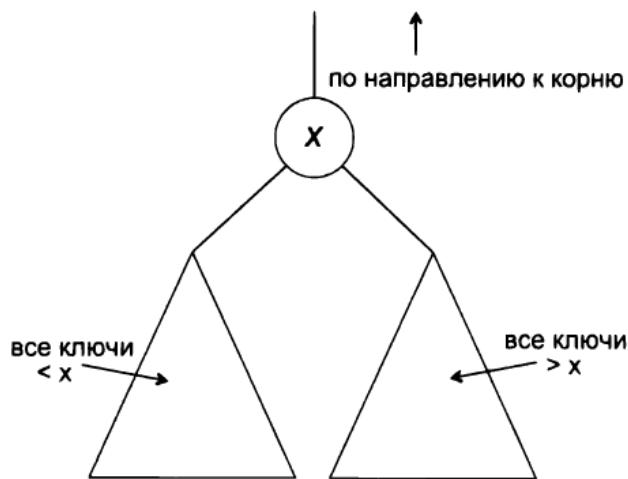


Рис. 9. Дерево поиска

Например, покажем дерево поиска, содержащее объекты с ключами  $\{1, 2, 3, 4, 5\}$ , и таблицу, перечисляющую места назначения трех указателей в каждом узле (рис. 10)

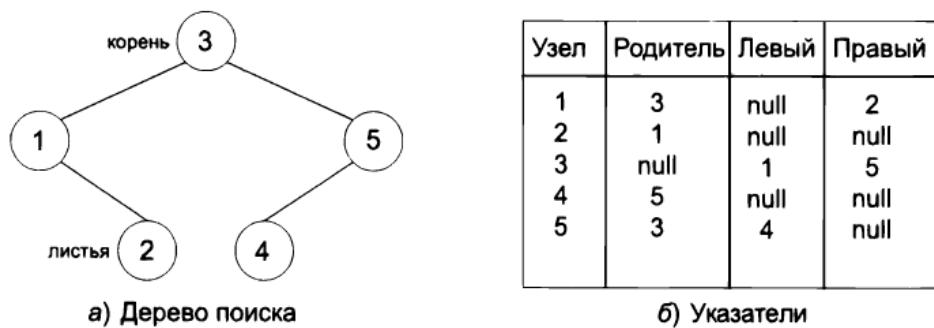


Рис. 10. Дерево поиска и соответствующие ему указатели на родителя и потомков

Бинарные деревья поиска и кучи различаются по некоторым направлениям. Кучи можно рассматривать как деревья, но реализованы они как массивы, без явных указателей между объектами. Дерево поиска явным образом хранит три указателя в расчете на объект и, следовательно, использует больше места (с постоянным множителем).

*Кучам* явные указатели не нужны, потому что они всегда соответствуют полным бинарным деревьям, в то время как бинарные деревья поиска могут иметь произвольную структуру.

Деревья поиска имеют иное предназначение, чем кучи. По этой причине свойство дерева поиска несопоставимо со свойством кучи. Кучи оптимизированы для вычислений *минимума*, а свойство кучи – то, что ключ потомка больше только ключа родителя, – упрощает поиск объекта с минимальным ключом (*это корень*). *Деревья поиска* оптимизированы, как ни странно, для *поиска*, и свойство дерева поиска определено соответствующим образом.

Например, если вы ищете объект с ключом 23 в дереве поиска и ключом корня является 17, то вы знаете, что объект может располагаться только в правом поддереве корня, и можете исключить объекты в левом поддереве из дальнейшего рассмотрения. Это должно напоминать о бинарном поиске, как и подобает структуре данных, смысл которой заключается в симулировании динамически изменяющегося отсортированного массива.

**Высота дерева поиска** Для заданного множества ключей существует множество разных деревьев поиска. Высота дерева определяется как длина самого длинного пути от корня до листа (ее еще называют глубиной). В общем случае *бинарное дерево поиска*, содержащее  $n$  объектов, может иметь какую угодно глубину от  $\approx \log_2 n$  (идеально сбалансированное бинарное дерево<sup>7</sup>) до  $n - 1$  (цепочка<sup>8</sup>).

**Реализация операции Отыскать со временем  $O(\text{высота})$**  Отыскать означает по ключу  $k$  вернуть указатель на объект в структуре данных с ключом  $k$  (либо сообщить, что такого объекта не существует). Свойство дерева поиска указывает, где именно искать объект с ключом  $k$ . Если  $k$  меньше (либо, соответственно, больше) ключа корня, то такой объект должен располагаться в левом поддереве (в правом поддереве соответственно) корня.

Время выполнения пропорционально количеству пройденных указателей, которое не превышает глубину дерева поиска.

**Реализация операций Минимум и Максимум за время  $O(\text{высота})$**  Ключи в левом поддереве корня могут быть только меньше ключа корня, и ключи в правом поддереве могут быть только больше. Если левое поддерево является пустым, то корень должен быть минимальным. В противном случае минимум левого поддерева также является минимумом всего дерева. Это наводит на мысль о том, чтобы следовать по указателю на левого потомка корня и повторять данный процесс.

Время работы пропорционально числу пройденных указателей, которое равно  $O(\text{высота})$ .

**Сбалансированные деревья поиска** Во всех наиболее распространенных реализациях сбалансированных деревьях поиска используются *повороты* – операции с постоянным временем, выполняющие небольшую локальную перебалансировку при сохранении свойства дерева поиска.

## 4.6. Массивы и связанные списки

Связанный список (linked list) – это структура данных, в которой объекты расположены в линейном порядке. Однако, в отличие от массива, в котором этот порядок определен индексами, порядок в связанном списке определяется указателями на каждый объект.

<sup>7</sup>Наилучший вариант

<sup>8</sup>Наихудший вариант

*Массивы* чрезвычайно полезны из-за того, что они поддерживают *произвольный доступ*, то есть другими словами *чтение элемента* в массиве выполняется за *постоянное время*  $O(1)$ .

Всего существует два вида доступа:

- *произвольный*,
- *последовательный*.

При последовательном доступе элементы читаются по одному, начиная с первого. *Связанные списки* поддерживают только *последовательный* доступ. Если требуется прочитать 10-ый элемент связанного списка, то придется прочитать первые 9 элементов и перейти по ссылкам к 10-ому элементу.

Вставка и удаление выполняются за постоянное время  $O(1)$  только в том случае, если доступ к элементу можно получить мгновенно. Для удаления элемента лучше подходит связанный список, потому что в нем достаточно изменить указатель в предыдущем элементе, а в массиве при удалении элемента все последующие элементы нужно будет сдвинуть вверх.

## 4.7. В-деревья

*В-деревья* представляют собой *сбалансированные деревья поиска*, созданные специально для эффективной работы с дисковой памятью (и другими типами вторичной памяти с непосредственным доступом). Многие СУБД используют для хранения информации именно В-деревья или их разновидности.

В-деревья отличаются от красно-черных деревьев тем, что узлы В-дерева могут иметь много дочерних узлов – от нескольких штук до тысяч, так что степень ветвления В-дерева может быть очень большой (хотя обычно она определяется характеристиками используемых дисков). В-деревья схожи с красно-черными деревьями в том, что все В-деревья с  $n$  узлами имеют высоту  $O(\lg n)$ , хотя само значение высоты В-дерева существенно меньше, чем у красно-черного дерева за счет более сильного ветвления. Таким образом, В-деревья также могут использоваться для реализации многих операций над динамическими множествами за время  $O(\lg n)$  [16, стр. 521].

В-деревья представляют собой естественное обобщение бинарных деревьев поиска.

## 5. NP-полнота

### 5.1. Краткая сводка

*Полиномиальные алгоритмы* – алгоритмы, время работы которых на входе длины  $n$  в最坏的情况下 равно  $O(n^k)$  для некоторой константы  $k$  (не зависящей от длины входа). Вообще говоря, о задачах, разрешимых с помощью алгоритмов с полиномиальным временем работы, возникает представление как о легко разрешимых или простых, а о задачах, время работы которых превосходит полиномиальное время, – как о трудно разрешимых или сложных [16, 1096].

*P-задачей* (полиномиальной задачей) называется задача, которую можно решить за полиномиальное (от длины входа) время. Точнее говоря, это задачи, которые можно решить за время  $O(n^k)$ , где  $k$  – некоторая константа, а  $n$  – размер входных данных задачи.

*NP-задачей* (недетерминированно полиномиальной задачей) называется задача, которую можно решить только на *недетерминированной машине*<sup>9</sup> Тьюринга за полиномиальное время. Други-

<sup>9</sup>В недетерминированных машинах следующее состояние не всегда однозначно определяется предыдущим. Работу такой машины можно представить как разветвляющийся на каждой неоднозначности процесс: задача считается решенной, если хотя бы одна ветвь процесса пришла к ответу

ми словами, класс NP состоит из задач, которые поддаются *проверке* в течение полиномиального времени. Имеется в виду, что если мы каким-то образом получаем «сертификат» решения, то в течение времени, полиномиальным образом зависящего от размера входных данных задачи, можно проверить корректность такого решения.

Задача называется *NP-полной*, если для нее не существует эффективных алгоритмов решения и к этой задаче можно свести любую другую задачу из класса NP за полиномиальное время. Таким образом, *NP-полные задачи образуют в некотором смысле подмножество «самых сложных» задач в классе NP*. И если для какой-то из них будет найден «быстрый» алгоритм решения, то и любая другая задача из класса NP может быть решена так же «быстро».

*Сводимость по Карпу.* Задача разрешения  $P_1$  полиномиально сводится к задаче разрешения  $P_2$ , если

- существует полиномиально вычислимая функция  $f : I_1 \rightarrow I_2$  (отображает входные данные  $I_1$  для  $P_1$  во входные данные  $I_2 \equiv f(I_1)$  для задачи  $P_2$ ),
- $\forall I_1$  совпадают ответы на вопросы « $P_1(I_1) ?$ » и « $P_2(f(I_1)) ?$ ».

**ВАЖНО:** NP-полные задачи это задачи, для которых не доказано ни то, что они не могут быть решены за полиномиальное время, ни то, что

**ВАЖНО:** Никакую NP-полную задачу нельзя решить никаким известным алгоритмом полиномиальной сложности.

На рис. 11 изображена диаграмма Венна взаимоотношений между классами  $P$ ,  $NP$ ,  $NP$ -полными и  $NP$ -трудными в случае если  $P \neq NP$ .

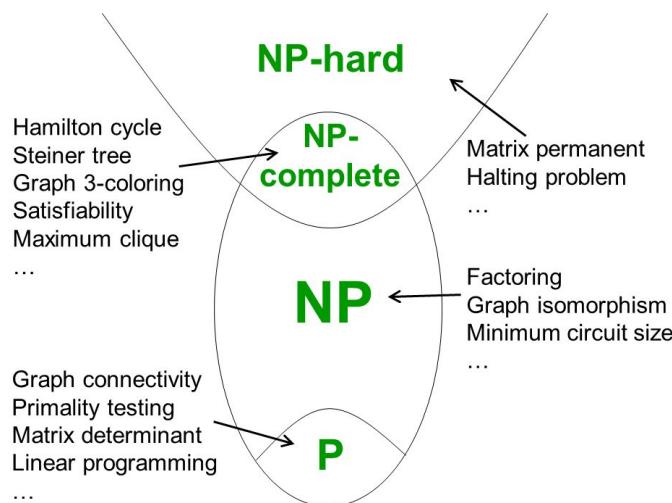


Рис. 11. Диаграмма Венна взаимоотношений между классами  $P$ ,  $NP$ ,  $NP$ -полными и  $NP$ -трудными

**ВАЖНО:** недетерминированность проще всего понять, рассматривая алгоритм, который производит вычисления до тех пор, пока не доходит до места, в котором должен быть сделан выбор из нескольких альтернатив. Детерминированный алгоритм исследовал бы сначала одну альтернативу, а потом вернулся бы для рассмотрения следующей альтернативы. *Недетерминированный* алгоритм может исследовать все альтернативы *одновременно*, по сути «копируя» самого себя для каждой альтернативы. Все копии работают независимо. Если копия обнаруживает, что данный путь безрезультатный, то она прекращает выполнение. Если копия находит требуемое решение, она объявляет об этом, и все копии прекращают работать. Детерминированная машина Тьюринга является частным случаем недетерминированной машины Тьюринга и не имеет копий.

Нахождение алгоритма, решающего какую-либо NP-полную задачу за полиномиальное время, позволит найти решение NP-задач за полиномиальное время, то есть позволит считать их P-задачами.

Трудоемкость более сложных задач растет экспоненциально с увеличением объема данных. Алгоритмы с экспоненциальным временем считаются неэффективными.

Если для какой-то NP-полной задачи найден полиномиальный алгоритм решения, то и любая другая задача из класса NP может быть решена за полиномиальное время.

Если установлено, что задача NP-полная, это служит достаточно надежным указанием на то, что она *трудноразрешимая* [16, 1088].

Для демонстрации сложности задач с помощью сведений нужно начать с одной *заведомо сложной задачи*. Самой сложной из всех *NP-полных задач* является логическая задача, называемая *задачей выполнимости (satisfiability) булевых формул* [15, 351].

Экземпляром задачи SAT является булева формула, состоящая только из имен переменных, скобок и операций И, ИЛИ и НЕ. Согласно теореме Кука, задача SAT для булевых формул, записанных в *конъюктивной нормальной форме*, является *NP-полней*. Требование к записи в конъюктивной нормальной форме существенно, так как, например, задача SAT для формул, представленных в дизъюктивной нормальной форме, тривиально решается за линейное время в зависимости от размера записи формулы.

Если известно эффективное решение задачи SAT, то можно эффективно решить любую задачу в классе NP. Иначе говоря, SAT – это задача-представитель класса, она является «самой сложнейшей» в своем классе и позволяет решить все другие задачи в NP.

**ВАЖНО:** вопрос о том, будет ли  $P = NP$  является открытой проблемой теории сложности. Широко распространено мнение, что  $P \neq NP$ , а следовательно  $P \subset NP$ .

## 5.2. Интуитивное определение SAT, NP и P

Формула  $F$  – это синтаксически правильный набор переменных и коннекторов, т.е. *следствие, И, ИЛИ* соединяют переменные или другие формулы, а *НЕ* стоит перед переменной или формулой. Пример  $F = (x \rightarrow (y \text{ OR } z)) \text{ AND } (x \rightarrow \text{NOT } x)$ .

Говорят, что формула  $F$  выполнима (SAT) тогда и только тогда (далее для краткости iff – if and only if), если ее переменным можно присвоить значения Истина/Ложь, таким образом, что  $F$  истина.

Любая пропозиционная формула  $F$  может быть приведена к виду CNF (Conjunctive Normal Form), т.е. быть представлена в виде

$$F' = c_1 \text{ AND } c_2 \text{ AND } \dots \text{ } c_n,$$

где  $c_i$  – это  $x \text{ OR } y \text{ OR } z$ , а  $x, y, z$  – это переменные или их отрицания.

Пример

$$F = (x \text{ OR } \text{NOT } y \text{ OR } \text{NOT } z) \text{ AND } (\text{NOT } x \text{ OR } \text{NOT } y \text{ OR } h) \text{ AND } (z \text{ OR } h).$$

Чтобы каждое предложение содержало не более трех переменных, потребуется ввести дополнительные переменные, но это исключительно технические детали.

В таком виде, когда формула имеет вид описанный выше задачи носит название 3-SAT, подчеркивая тот факт, что каждое предложение  $c_i$  содержит не более трех переменных или их отрицаний.

**класс P** (он же класс PTIME) – задачи разрешимые за *полиномиальное* время. Это значит, что число шагов алгоритма в данном классе растет не более, чем некоторый полином от входных данных.

Входным параметром является массив чисел для сортировки. Нас интересует рост времени в зависимости от роста длины массива, т.е. зависимость времени TIME (оно количество операций) от  $n$ .

В сортировке пузырьком количество операций (TIME) растет как квадрат от длины массива, т.е.  $TIME(n) = an^2 + bn + C$ . В данном случае *время работы алгоритма* растет не быстрее квадрата числа элементов и записывается это как  $TIME(n) \in O(n^2)$ .

**класс NP** обозначает класс недетерминированн полиномиальное время. Сама природа и внутренние механизмы NP следуют так называемому guess-and-check подходу (угадать и проверить). Пространство поиска решений экспоненциально (достаточно большое, чтобы исключить возможность перебора), а проверка решения является простой задачей. Можно рассматривать NP, как класс задач, в котором нужно найти решение (guess часть) среди большого количества вариантов, а потом проверить его корректность (check часть).

С точки зрения SAT, пространство решений это всевозможные наборы значений переменных, если у нас  $k$  различных переменных в формуле, то у нас  $2^k$  возможных «интерпретаций» формулы, т.е. пространство поиска  $I$  экспоненциально. Однако, если мы «угадали»  $I$ , то мы можем за полиномиальное время проверить истинность формулы.

## 6. Настройка непрерывной интеграции (CI) на GitHub Actions

### 6.1. 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 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>

```

## 6.2. 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 примера рассмотрим настройку непрерывной интеграции с помощью GitHub Actions.

Простой пример `ci.yaml`

```

.github/workflows/ci.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

```

## 7. Настройка непрерывной доставки (CD) с помощью Codefresh

## 8. Разработка собственных Python-пакетов для PyPI

### 8.1. 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`.

### 8.2. 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`

### 8.3. 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

- o <https://towardsdatascience.com/10-steps-to-set-up-your-python-project-for-success-14ff88b5c>
- o <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`.

## 8.4. 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` – односторонковое 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 подпакетов использовать функцию `find_packages()` (в данном случае функция найдет только пакет `example_pkg`),

- o **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
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{}\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 Orlow',
    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
...

```

## 8.5. 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 одноточечная 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

```
1 #!C:\Users\ADM\Anaconda3\envs\env_for_tests\python.exe
2 # EASY-INSTALL-ENTRY-SCRIPT: 'snake-package','console_scripts','snake'
3 import re
4 import sys
5
6 # for compatibility with easy_install; see #2198
7 __requires__ = 'snake-package'
8
9 try:
10     from importlib.metadata import distribution
11 except ImportError:
12     try:
13         from importlib_metadata import distribution
14     except ImportError:
15         from pkg_resources import load_entry_point
16
17
18 def importlib_load_entry_point(spec, group, name):
19     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, заменяет 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.
        ]
    }
)
```

```

        ],
        "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 связана 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 корне 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 функцию 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 организованы 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 `rucrond`, 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 rucrond с применением `pkg_resources` для поиска точек входа в программе под названием `pytimed.py`

pytimed.py

```
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() возвращает
                # 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
            ma 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=True` 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.

## 9. Формы записи дисперсии

Форма 1

$$D(X) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2 = \left( \frac{1}{n} \sum_{i=1}^n x_i^2 \right) - \mu^2, \quad \mu = \frac{1}{n} \sum_{i=1}^n x_i$$

Форма 2

$$D(X) = \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n \frac{1}{2} (x_i - x_j)^2 = \frac{1}{n^2} \sum_i \sum_{j>i} (x_i - x_j)^2.$$

## 10. 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 *обращивание 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() # напечатаем closed

```

## 11. 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() # завершим работу приложения и выбросим из сессии
    else:
        ...
    return data_filepath, output_fig_filepath
```

## 12. Логгирование в Python

Для того чтобы записи журнала уровня INFO не игнорировались логгером, следует установить базовый уровень логирования в INFO или DEBUG (уровень логирования 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 логирования
)
...
logging.info(...)
logging.error(...)
```

## 13. Бинарный поиск

В общем случае для списка из  $n$  элементов *бинарный поиск* (при бинарном поиске исключается половина элементов) выполняется за  $\log_2 n$  шагов, тогда как простой поиск (последовательным перебором) будет выполнен за  $n$  шагов. Бинарный поиск работает только в том случае, если список отсортирован!

Время выполнения бинарного поиска *логарифмическое*, то есть другими словами времененная сложность алгоритма «бинарный поиск»  $O(\log_2 n)$ , или так  $O(\log n)$  (в теории алгоритмов обычно под  $\log_2$  понимают  $\log$ ). У прямого поиска время выполнения линейное, т.е. временная сложность  $O(n)$ .

## 14. Временная сложность алгоритмов и нотация O-большое

### 14.1. Простыми словами

Специальная нотация «O-большое» описывает *скорость работы алгоритма*. Но нотация «O-большое» не сообщает время в секундах, а позволяет сравнить *количество операций*. Оно указывает насколько быстро возрастает время выполнения алгоритма. То есть  $O(n)$  – асимптотическое время работы алгоритма.

Можно сказать, что нотация «O-большое» – это просто форма записи асимптотического представления временной сложности алгоритма.

---

#### Замечание

Нотация «O-большое» описывает *худший* случай времени работы алгоритма и по сути сообщает насколько быстро возрастает время выполнения алгоритма с увеличением размера входных данных

---

Зачем при анализе времени исполнения алгоритма необходимо отбрасывать информацию, такую как постоянные коэффициенты и члены низших порядков? Дело в том, члены низших порядков по определению становятся все более и более неактуальными, поскольку вы сосредоточиваетесь на больших объемах входных данных, то есть входных данных, которые требуют алгоритмической изобретательности. Между тем постоянные коэффициенты обычно сильно зависят от деталей реализации. Если при анализе алгоритма мы не хотим привязываться к определенному языку программирования, архитектуре или компилятору, имеет смысл использовать формальный подход и не сосредоточиваться на постоянных коэффициентах.

Например, для сортировки слиянием верхний предел его *времени исполнения* равен  $6n \log_2 n + 6n$  примитивных операций, где  $n$  - длина входного массива.

Член низших порядков здесь  $6n$ . Поскольку  $n$  растет медленнее, чем  $n \log_2 n$ , он будет устранен в асимптотических обозначениях. Основной постоянные коэффициент 6 тоже будет устранен, в результате мы получаем намного более простое выражение  $n \log_2 n$ .

То есть *время работы* алгоритма «сортировка слиянием» составляет  $O(n \log n)$ .

Применение нотации  $O$ -большое позволяет ранжировать алгоритмы по группам согласно их асимптотически наихудшим *временам исполнения*, например: линейные  $O(n)$  алгоритмы,  $O(n \log n)$  алгоритмы, квадратичные  $O(n^2)$  алгоритмы,  $O(1)$  алгоритмы с *постоянным временем* и так далее.

Безусловно, здесь не утверждается, что постоянные коэффициенты никогда не имеют значения при разработке алгоритма. Скорее, верно то, что когда нужно провести сравнение между принципиально различными способами решения задачи, асимптотический анализ часто является правильным инструментом для понимания того, какой из них будет работать лучше, в особенностях на достаточно больших объемах входных данных.

### 14.2. Формальное определение $O$ -большое

Обозначение  $O$ -большое относится к функциям вида  $T(n)$ , определенным на положительных целых числах  $n = 1, 2, \dots$ . Для нас  $T(n)$  почти всегда будет обозначать *предел худшего времени исполнения алгоритма* как функцию длины  $n$  входных данных [12, 74].

Математическая версия  $O$ -большое:  $T(n) = O(f(n))$ , тогда и только тогда, когда существуют положительные константы  $c$  и  $n_0$ , такие что

$$O\text{-большое : } T(n) \leq c f(n), \quad c > 0$$

для всех  $n \geq n_0$ .

### 14.3. Обозначение $\Omega$ -большое и $\Theta$ -большое

Обозначение  $O$ -большое на сегодняшний день является наиболее важной и общепринятой категорией для обсуждения асимптотического времени работы алгоритма.

Если  $O$ -большое аналогично отношению «меньше или равно ( $\leq$ )», то  $\Omega$ -большое и  $\Theta$ -большое аналогичны, соответственно, отношениям «больше или равно ( $\geq$ )» и «равно (=)».

Математическая версия определения  $\Omega$ -большое:  $T(n) = \Omega(f(n))$ , тогда и только тогда, когда существуют положительные константы  $c$  и  $n_0$  такие, что

$$\Omega\text{-большое : } T(n) \geq c f(n), \quad c > 0$$

для всех  $n \geq n_0$ .

Обозначение  $\Theta$ -большое аналогично отношению «равно». Говоря, что  $T(n) = \Theta(f(n))$  мы, в сущности, подразумеваем, что  $T(n) = \Omega(f(n))$  и, одновременно,  $T(n) = O(f(n))$ . Эквивалентным образом  $T(n)$  в конечном счете зажата между двумя разными постоянными кратными  $f(n)$ .

Математическая версия определения  $\Theta$ -большое:  $T(n) = \Theta(f(n))$ , тогда и только тогда, когда существуют положительные константы  $c_1$ ,  $c_2$  и  $n_0$  такие, что

$$c_1 f(n) \leq T(n) \leq c_2 f(n), \quad c_1, c_2 > 0$$

для всех  $n \geq n_0$ .

### 14.4. Обозначение $o$ -малое

Если обозначение  $O$ -большое эквивалентно отношению «меньше или равно», то обозначение  $o$ -малое эквивалентно отношению «строго меньше». По аналогии, существует обозначение  $\omega$ -малое, которое соответствует «строго больше». Обозначения  $\theta$ -малое не существует.

## 15. Управление памятью в Python

Исходный код CPython <https://github.com/python/cpython>.

Python – интерпретируемый язык программирования. На первом этапе программа компилируется в байт-код (машинно-независимый код низкого уровня), а затем интерпретируется виртуальной машиной (например, Virtual CPython).

CPython не взаимодействует напрямую с регистрами и ячейками физической памяти – только с ее *виртуальным представлением*. В начале выполнения программы операционная система создает *новый процесс* и выделяет под него ресурсы.

Выделенную виртуальную память интерпретатор использует для:

1. собственной корректной работы,
2. стека вызываемых функций и их аргументов,

3. хранилища данных, представленного в виде кучи.

*ВременАя сложность алгоритма* определяется как функция от длины строки, представляющей входные данные, и равна времени работы алгоритма на данном входе. Временная сложность алгоритма обычно выражается с использованием нотации «*O* большое», которая учитывает только слагаемые самого высокого порядка, а также не учитывает константные множители, то есть коэффициенты. Если *временная сложность* выражена таким способом, то говорят об *асимптотическом описании временной сложности*, то есть при стремлении размера входа к бесконечности. Например, если существует число  $n_0$ , такое, что время работы алгоритма для всех входных длин  $n > n_0$  не превосходит  $5n^3 + 3n$ , то временную сложность данного алгоритма можно асимптотически оценить как  $O(n^3)$ .

В отличие от C/C++, мы не можем управлять состоянием кучи напрямую из Python. Функции низкоуровневой работы с памятью предоставляются Python/C API, но обычно интерпретатор просто обращается к хранилищу данных через *диспетчер памяти Python* (memory manager).

GIL – глобальная блокировка интерпретатора. GIL гарантирует, что в один и тот же момент времени байт-код выполняется только один потоком. Главное преимущество – безопасная работа с памятью, а основной недостаток в том, что многопоточное выполнение программ Python требует специфических решений.

Очевидно, программа не сама выполняет сохранение и освобождение памяти – ведь мы не пишем соответствующих инструкций. Интерпретатор лишь запрашивает диспетчер памяти сделать это. А диспетчер уже делегирует работу, связанную с хранением данных, *аллокаторам* – распределителям памяти.

Непосредственно с оперативной памятью взаимодействует *распределитель сырой памяти* (raw memory allocator). Поверх него работают аллокаторы, реализующие стратегии управления памятью, специфичные для отдельных типов объектов. Объектов разных типов – например, числа и строки – занимают разный объем, к ним применяются разные механизмы хранения и освобождения памяти. Аллокаторы стараются не занимать лишнюю память до тех пор, пока она не станет совершенно необходимой – этот момент определен стратегией распределения памяти CPython.

Python использует динамическую стратегию, то есть распределение памяти выполняется во время выполнения программы. Виртуальная память Python представляет иерархическую структуру, оптимизированную под объекты Python размером менее 256 Кб:

- *арена* – фрагмент памяти, расположенный в пределах непрерывного блока оперативной памяти объемом 256 Кб. Объекты размером более 256 Кб направляются в стандартный аллокатор C,
- *пул* – блок памяти внутри арены, занимающий 4 Кб, что соответствует одной странице виртуальной памяти. То есть одна арена включает до  $256/4 = 64$  пулов,
- *блок* – элемент пула размером от 16 до 512 байт. В пределах пула все блоки имеют одинаковый размер. Размер блока определяется тем, сколько байт требуется для представления конкретного объекта. Размеры блоков кратны 16 байт. То есть существует всего  $512/16 = 32$  классов блоков. То есть в одном пуле, в зависимости от класса, может находиться от 8 до 256 блоков.

*Блок* содержит не более одного объекта Python и находится в одном из трех состояний:

- *untouched* – блок еще не использовался для хранения данных,
- *free* – блок использовался механизмом памяти, но больше не содержит используемых программой данных,

- `allocated` – блок хранит данные, необходимые для выполнения программы.

В пределах пула блоки `free` организованы в односвязанный список с указателям `freeblock`. Если аллокатору для выделения памяти не хватит блоков списка `freeblock`, он задействует блоки `untouched`. Освобождение памяти означает всего лишь то, что аллокатор меняет статус блока с `allocated` на `free` и начинает отслеживать блок в списке `freeblock`.

*Пул* может находиться в одном из трех состояний: `used` (занят), `full` (заполнен), `empty` (пуст). Пустые пулы отличаются от занятых отсутствием блоков `allocated` и тем, что для них пока не определен `size class`. Пулы `full` полностью заполнены блоками `allocated` и недоступны для записи. Стоит освободиться любому из блоков заполненного пула – и он помечается как `used`. Пулы одного типа и одного размера блоков организованы в *двусвязные списки* (дву направленные связные списки). Это позволяет алгоритму легко находить доступное пространство для блока заданного размера. Алгоритм проверяет список `usedpools` и размещает блок в доступном пуле. Если в `usedpools` нет ни одного подходящего пула для запроса, алгоритм использует пул из списка `freepools`, который отслеживает пулы в состоянии `empty`.

Связный список – базовая динамическая структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки (*связки*) на следующий и/или предыдущий узел списка. Порядок обхода списка всегда явно задается его внутренними связями.

*Односвязный список* (однонаправленный связный список) – это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на `NULL`. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

В случае *двусвязного списка* (дву направленного связного списка) ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещаться в обе стороны. В этом списке проще производить удаление и перестановку элементов, так как легко доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

*Арены* содержат пулы любых видов и организованы в двусвязный список `use_arenas`. Список отсортирован по количеству доступных пустых пулов. Чем меньше в арене таких пулов, тем она ближе к началу списка. Для размещения новых данных выбирается область, наиболее заполненная данными.

Информацию о текущем распределении памяти в аренах, пулах и блоках можно посмотреть, запустив функцию `sys._debugmallocstats()`.

Чтобы не произошло утечки памяти, диспетчер памяти должен отследить, что вся выделенная память освободится после завершения работы программы. То есть при завершении программы CPython дает задание *очистить все арены*.

Именно количество используемых *арен* определяет *объем оперативной памяти*, занимаемой программой на Python – если в арене все пулы в состоянии `empty`, CPython делает запрос на освобождение этого участка виртуальной памяти. Но, чтобы пулы стали `empty`, все их блоки

должны быть `free` или `untouched`. Получается, нужно понять, как CPython освобождает память.

Для освобождения памяти используются два механизма: *счетчик ссылок* и *сборщик мусора*. Все в Python являются объектами, а прородителям всех типов объектов в реализации CPython является PyObject <https://docs.python.org/3/c-api/structures.html#c.PyObject>, то есть все типы объектов наследуют этот тип.

В PyObject определены счетчик ссылок и указатель на фактический тип объекта. Счетчик ссылок увеличивается на единицу, когда мы создаем что-то, что обращается к объекту, например, сохраняя объект в новой переменной. И наоборот, счетчик уменьшается на единицу, когда мы перестаем ссылаться на объект.

Счетчик ссылок любого объекта можно проверить с помощью `sys.getrefcount()`. Учтите, что передача объекта в `getrefcount()` увеличивает счетчик на единицу, так как сам вызов метода создает еще одну ссылку. Когда счетчик уменьшается до нуля, происходит вызов аллокатора для освобождения соответствующих блоков памяти.

Однако счетчик ссылок не способен отследить ситуации с циклическими ссылками. К примеру, возможна ситуация, когда два объекта ссылаются друг на друга, но оба уже не используются программой. Для борьбы с такими зависимостями используется сборщик мусора (*garbage collector*).

Если счетчик ссылок является свойством объекта, то сборщик мусора – механизм, который запускается на основе эвристик. Задача этих эвристик – снизить частоту и объем очищаемых данных. Основная стратегия заключается в разделении объектов на поколения: чем больше сборок мусора пережил объект, тем он значимее для выполнения работы программы. Сборщик мусора имеет интерфейс в виде модуля `gc`.

Сохранение и освобождение блоков памяти требует времени и вычислительных ресурсов. Чем меньше блоков задействовано, тем выше скорость работы программы.

Рекомендации по экономной работе с памятью:

- Обращайте внимание на работу с *неизменяемыми* объектами. К примеру, вместо использования оператора `+` для соединения строк используйте методы `.join()`, `.format()` или `f-строки`,
- Избегайте вложенных циклов. Создание сложных вложенных циклов приводит к генерации чрезмерно большого количества объектов, занимающих значительную часть виртуальной памяти. Большинство задач, решаемых с помощью вложенных циклов, разрешимы методами модуля `itertools`,
- Используйте кэширование. Если вы знаете, что функция или класс используют или генерируют набор однотипных объектов, применяйте кэширование. Часто для этого достаточно добавить всего лишь один декоратор из библиотеки `functools`,
- Профилируйте код. Если программа начинает «тормозить», то профилирование – самый быстрый способ найти корень всех зол.

## 16. REST API

REST (Represeatational State Transfer – передача состояния представления) – *архитектурный стиль взаимодействия компонентов распределенного приложения* в сети.

REST удобнее использовать для on-line приложений, когда ответ приложения ожидается в режиме близком к реальному времени.

Приложение считается RESTful, если

- Client-server,
- Stateless: сервисы не должны иметь состояния; другими словами, с точки зрения организации взаимодействия между сервисами результат запроса должен быть инвариантен по отношению к тому на какой сервис-реплику пришел этот запрос,
- Cache: запросы должны кешироваться,
- Uniform Interface: GET (получить, прочитать), POST (создать ресурс), PUT (обновить существующий ресурс), DELETE (удалить),
- Layered System,
- Code on demand (Optional).

## 17. 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
```

## 18. Приемы работы с MLflow

Ознакомиться с руководствами по использованию mlflow можно на ресурсе <https://mlflow.org/docs/latest/tracking.html>.

### 18.1. Общий сценарий использования MLflow на примере библиотеки H2O

Пусть директория проекта имеет вид

```
h2o/
|- MLproject
|- conda.yaml
|- random_forest.py
|- wine-quality.csv
```

Конфигурационный файл проекта имеет вид

```
MLproject
name: h2o-example

conda_env: conda.yaml # для запуска будет создано conda-окружение
```

```
entry_points:  
  main:  
    command: "python random_forest.py"
```

Файл зависимостей conda-окружения выглядит так

conda.yaml

```
name: h2o_example  
channels:  
  - defaults  
  - anaconda  
  - conda-forge  
dependencies:  
  - python=3.6  
  - numpy=1.14.2  
  - pandas  
  - pip  
  - pip:  
    - h2o  
  - mlflow>=1.0
```

Собственно сценарий

random\_forest.py

```
import h2o  
from h2o.estimators.random_forest import H2ORandomForestEstimator  
  
import mlflow  
import mlflow.h2o  
  
h2o.init()  
  
wine = h2o.import_file(path="wine-quality.csv")  
r = wine["quality"].runif()  
train = wine[r < 0.7]  
test = wine[0.3 <= r]  
  
def train_random_forest(ntrees):  
    # запуск будет инициализироваться для каждого значения ntrees  
    with mlflow.start_run():  
        rf = H2ORandomForestEstimator(ntrees=ntrees)  
        train_cols = [n for n in wine.col_names if n != "quality"]  
        rf.train(train_cols, "quality", training_frame=train, validation_frame=test)  
  
        mlflow.log_param("ntrees", ntrees)  
  
        mlflow.log_metric("rmse", rf.rmse())  
        mlflow.log_metric("r2", rf.r2())  
        mlflow.log_metric("mae", rf.mae())  
  
        mlflow.h2o.log_model(rf, "model")  
  
if __name__ == "__main__":  
    for ntrees in [10, 20, 50, 100, 200]:  
        train_random_forest(ntrees)
```

Запустить проект можно с помощью конструкции

```
mlflow run . # имя точки входа не указано, значит будет запущена точка входа по умолчанию, т.е.  
точка входа с именем main
```

После выполнения этой команды MLflow обратиться к файлу `MLproject` и на основании информации, приведенной в файле `conda.yaml` соберет conda-окружение. Затем будет найдена точка с именем `main`, которая требует запустить сценарий `random_forest.py`. В результате MLflow выполнит 5 запусков с различными значениями количества деревьев в случайному лесе.

Сравнить значения метрик качества можно с помощью графического интерфейса

```
mlflow ui
```

ВАЖНО: здесь каждый вызов функции `train_random_forest()` будет приводить к инициализации mlflow-запуска.

## 18.2. Общие сведения

Пример python-сценария с mlflow-вставками

train.py

```
import os  
import warnings  
import sys  
  
import pandas as pd  
import numpy as np  
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score  
from sklearn.model_selection import train_test_split  
from sklearn.linear_model import ElasticNet  
from urllib.parse import urlparse  
import mlflow  
import mlflow.sklearn  
  
import logging  
  
logging.basicConfig(level=logging.WARN)  
logger = logging.getLogger(__name__)  
  
  
def eval_metrics(actual, pred):  
    rmse = np.sqrt(mean_squared_error(actual, pred))  
    mae = mean_absolute_error(actual, pred)  
    r2 = r2_score(actual, pred)  
    return rmse, mae, r2  
  
  
if __name__ == "__main__":  
    warnings.filterwarnings("ignore")  
    np.random.seed(40)  
  
    # Read the wine-quality csv file from the URL  
    csv_url = (  
        "http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.  
        csv"  
    )  
    try:  
        data = pd.read_csv(csv_url, sep=";")  
    except Exception as e:
```

```

logger.exception(
    "Unable to download training & test CSV, check your internet connection. Error: %s",
)
# Split the data into training and test sets. (0.75, 0.25) split.
train, test = train_test_split(data)

# The predicted column is "quality" which is a scalar from [3, 9]
train_x = train.drop(["quality"], axis=1)
test_x = test.drop(["quality"], axis=1)
train_y = train[["quality"]]
test_y = test[["quality"]]

alpha = float(sys.argv[1]) if len(sys.argv) > 1 else 0.5
l1_ratio = float(sys.argv[2]) if len(sys.argv) > 2 else 0.5

with mlflow.start_run():
    lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
    lr.fit(train_x, train_y)

    predicted_qualities = lr.predict(test_x)

    (rmse, mae, r2) = eval_metrics(test_y, predicted_qualities)

    print("Elasticnet model (alpha=%f, l1_ratio=%f):" % (alpha, l1_ratio))
    print("  RMSE: %s" % rmse)
    print("  MAE: %s" % mae)
    print("  R2: %s" % r2)

    mlflow.log_param("alpha", alpha)
    mlflow.log_param("l1_ratio", l1_ratio)
    mlflow.log_metric("rmse", rmse)
    mlflow.log_metric("r2", r2)
    mlflow.log_metric("mae", mae)

    tracking_url_type_store = urlparse(mlflow.get_tracking_uri()).scheme

# Model registry does not work with file store
if tracking_url_type_store != "file":
    # Register the model
    # There are other ways to use the Model Registry, which depends on the use case,
    # please refer to the doc for more information:
    # https://mlflow.org/docs/latest/model-registry.html#api-workflow
    mlflow.sklearn.log_model(lr, "model", registered_model_name="ElasticnetWineModel")
else:
    mlflow.sklearn.log_model(lr, "model")

```

Запустив этот сценарий несколько раз с различными параметрами

```

python train.py 0.4 0.9
python train.py 0.9 0.35
...

```

МОЖНО ПОЛУЧИТЬ СВОДКУ ПО КАЖДОМУ ЗАПУСКУ, КОТОРЫЙ ДОСТУПЕН С ПОМОЩЬЮ КОМАНДЫ

```
mlflow ui
```

Затем, выяснив с помощью команды `mlflow ui` идентификационный номер запуска, можно запустить ML-модель как сервис

```
# разумеется порт 5000 должен быть свободен; то есть если mlflow ui не остановлен к настоящему моменту, то его следует остановить
mlflow models serve -m runs:/b64437a.../model
```

То есть, другими словами команда `mlflow models serve` превращает python-приложение в web-сервис, который работает на локальном петлевом интерфейсе на порту 5000.

Теперь можно послать запрос на предсказание

```
curl -X POST http://127.0.0.1:5000/invocations \
-H "Content-Type: application/json" \
-d '[{
    "fixed acidity" : 3.42,
    "volatile acidity" : 1.66,
    ...
}]' # 5.82505...
```

Запуск проекта может выглядеть так

```
mlflow models serve \
-m /Users/mlflow/mlflow-prototype/mlruns/0/7c1...174/artifacts/model \
-p 1234
```

У MLflow tracking server есть два компонента:

- backend store: здесь хранится метаинформация об экспериментах и запусках (параметры, метрики, теги и пр.); MLflow поддерживает два типа бекенд-хранилищ: *файловое хранилище* (file store) и *хранилище с поддержкой баз данных* (database-backed store),
- artifact store: здесь хранятся большие файлы, например, файлы с моделями.

Чтобы задать тип хранилища, нужно использовать флаг `--backend-store-uri`:

- для файлового хранилища `.path_to_store` или `file:/path_to_store`,
- для хранилища с поддержкой баз данных

```
<dialect>+<driver>://<username>:<password>@<host>:<port>/<database>
```

MLflow поддерживает следующие диалекты: `mysql`, `mssql`, `sqlite` и `postgresql`. Драйвер можно не указывать (если драйвер не указан, то будет использован драйвер по умолчанию для соответствующего диалекта). Например,

```
mlflow server \
--default-artifact-root mlruns/0/818...92/artifacts \
--backend-store-uri sqlite:///mlflow.db --host 0.0.0.0
```

По умолчанию `--backend-store-uri` указывает на локальный каталог `./mlruns`.

Для того чтобы разбить параметры по экспериментам можно использовать метод `set_experiment`

```
import mlflow

mlflow.set_experiment("experiment-with-random-forest")
with mlflow.start_run():
    mlflow.log_param("a", 1)
    mlflow.log_param("b", 2)
```

### 18.3. Обучение модели линейной регрессии с двумя гиперпараметрами

Пример

examples/sklearn\_elasticnet\_wine/train.py

```
import os
import warnings
import sys

import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import ElasticNet
from urllib.parse import urlparse
import mlflow
import mlflow.sklearn

import logging

logging.basicConfig(level=logging.WARN)
logger = logging.getLogger(__name__)

def eval_metrics(actual, pred):
    rmse = np.sqrt(mean_squared_error(actual, pred))
    mae = mean_absolute_error(actual, pred)
    r2 = r2_score(actual, pred)
    return rmse, mae, r2

if __name__ == "__main__":
    warnings.filterwarnings("ignore")
    np.random.seed(40)

    # Read the wine-quality csv file from the URL
    csv_url = (
        "http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv"
    )
    try:
        data = pd.read_csv(csv_url, sep=";")
    except Exception as e:
        logger.exception(
            "Unable to download training & test CSV, check your internet connection. Error: %s",
            e
        )

    # Split the data into training and test sets. (0.75, 0.25) split.
    train, test = train_test_split(data)

    # The predicted column is "quality" which is a scalar from [3, 9]
    train_x = train.drop(["quality"], axis=1)
    test_x = test.drop(["quality"], axis=1)
    train_y = train[["quality"]]
    test_y = test[["quality"]]

    alpha = float(sys.argv[1]) if len(sys.argv) > 1 else 0.5
    l1_ratio = float(sys.argv[2]) if len(sys.argv) > 2 else 0.5
```

```

with mlflow.start_run():
    lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
    lr.fit(train_x, train_y)

    predicted_qualities = lr.predict(test_x)

    (rmse, mae, r2) = eval_metrics(test_y, predicted_qualities)

    print("Elasticnet model (alpha=%f, l1_ratio=%f):" % (alpha, l1_ratio))
    print("  RMSE: %s" % rmse)
    print("  MAE: %s" % mae)
    print("  R2: %s" % r2)

    mlflow.log_param("alpha", alpha)
    mlflow.log_param("l1_ratio", l1_ratio)
    mlflow.log_metric("rmse", rmse)
    mlflow.log_metric("r2", r2)
    mlflow.log_metric("mae", mae)

tracking_url_type_store = urlparse(mlflow.get_tracking_uri()).scheme

# Model registry does not work with file store
if tracking_url_type_store != "file":
    # Register the model
    # There are other ways to use the Model Registry, which depends on the use case,
    # please refer to the doc for more information:
    # https://mlflow.org/docs/latest/model-registry.html#api-workflow
    mlflow.sklearn.log_model(lr, "model", registered_model_name="ElasticnetWineModel")
else:
    mlflow.sklearn.log_model(lr, "model")

```

Запустим сценарий несколько раз с различными параметрами

```
python sklearn_elasticnet_wine/train.py <alpha> <l1_ratio>
```

Теперь с помощью `mlflow ui` на `http://localhost:5000` можно посмотреть статистику по эксперименту.

Если в корне проекта расположить приведенные ниже файлы, то проект можно будет запускать с различными параметрами командой `mlflow run`.

`sklearn_elasticnet_wine/MLproject`

```

name: tutorial

conda_env: conda.yaml

entry_points:
  main:
    parameters:
      alpha: {type: float, default: 0.5}
      l1_ratio: {type: float, default: 0.1}
    command: "python train.py {alpha} {l1_ratio}"

```

`sklearn_elasticnet_wine/conda.yaml`

```

name: tutorial
channels:
  - defaults
dependencies:
  - python=3.6

```

```
- pip
- pip:
  - scikit-learn==0.23.2
  - mlflow>=1.0
```

ВАЖНО: настраивать окружение можно не только с помощью блока `conda_env: conda.yaml` в файле `MLproject`, но и с помощью блока `docker_env`; в отличие от `conda_env` блок `docker_env` позволяет включать не только python-зависимости.

Теперь, если запустить этот код командой

```
mlflow run sklearn_elasticnet_wine -P alpha=0.42
```

то MLflow запустит код в новом виртуальном окружении `conda`, зависимости которого описываются в `conda.yaml`.

ВАЖНО: параметры сценарию можно передать либо позиционно при запуске `python`, либо с помощью именованных флагов при запуске `python`, либо с помощью флага `-P` при `mlflow run`, т.е.

```
# позиционная передача значений аргументам
python script_name.py 0.3 1.0 # сценарий должен быть адаптирован для такой передачи
python script_name.py --learning-rate=0.3 --colsample-bytree=1.0 # сценарий должен быть адаптирован для такой передачи (argparse, click etc.)
mlflow run . -P learning_rate=0.3 -P colsample_bytree=1.0
```

Если в корне репозитория лежит файл `MLproject`, то можно запустить код из репозитория, просто указав его URL

```
mlflow run https://github.com/mlflow/mlflow-example.git -P alpha=5.0
```

Кроме того этот сценарий можно запустить как локальный web-сервис

```
# запуск приложения как сервиса на порту 1234
mlflow models serve \
-m /Users/mlflow/mlflow-prototype/mlruns/0/7c1...74/artifacts/model \ # -m -- модель
-p 1234 # -p -- порт
```

После чего, как обычно, можно послать запрос на прогноз

```
curl -X POST \
-H "Content-Type:application/json; format=pandas-split" \
--data '{
  "columns": [
    "alcohol",
    "chlorides",
    "citric acid",
    "density",
    "fixed acidity",
    "free sulfur dioxide",
    "pH",
    "residual sugar",
    "sulphates",
    "total sulfur dioxide",
    "volatile acidity"
  ],
  "data": [
    [12.8, 0.029, 0.48, 0.98,
     6.2, 29, 3.33, 1.2, 0.39,
     75, 0.66]
  ]
}'
```

```
}', http://127.0.0.1:1234/invocations # [6.379428821398614]
```

## 18.4. Подбор гиперпараметров с использованием различных стратегий

Рассмотрим пример подбора гиперпараметров с использованием различных стратегий

train.py

```
import warnings
import math
import keras
import numpy as np
import pandas as pd
import click

from keras.callbacks import Callback
from keras.models import Sequential
from keras.layers import Dense, Lambda
from keras.optimizers import SGD
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split

import mlflow
import mlflow.keras

def eval_and_log_metrics(prefix, actual, pred, epoch):
    rmse = np.sqrt(mean_squared_error(actual, pred))
    mlflow.log_metric("{}_rmse".format(prefix), rmse, step=epoch)
    return rmse

def get_standardize_f(train):
    mu = np.mean(train, axis=0)
    std = np.std(train, axis=0)
    return lambda x: (x - mu) / std

class MLflowCheckpoint(Callback):
    """
    Example of Keras MLflow logger.
    Logs training metrics and final model with MLflow.
    We log metrics provided by Keras during training and keep track of the best model (best loss
    on validation dataset). Every improvement of the best model is also evaluated on the test set.
    At the end of the training, log the best model with MLflow.
    """
    def __init__(self, test_x, test_y, loss="rmse"):
        self._test_x = test_x
        self._test_y = test_y
        self.train_loss = "train_{}".format(loss)
        self.val_loss = "val_{}".format(loss)
        self.test_loss = "test_{}".format(loss)
        self._best_train_loss = math.inf
        self._best_val_loss = math.inf
        self._best_model = None
```

```

self._next_step = 0

def __enter__(self):
    return self

def __exit__(self, exc_type, exc_val, exc_tb):
    """
    Log the best model at the end of the training run.
    """
    if not self._best_model:
        raise Exception("Failed to build any model")
    mlflow.log_metric(self.train_loss, self._best_train_loss, step=self._next_step)
    mlflow.log_metric(self.val_loss, self._best_val_loss, step=self._next_step)
    mlflow.keras.log_model(self._best_model, "model")

def on_epoch_end(self, epoch, logs=None):
    """
    Log Keras metrics with MLflow. If model improved on the validation data, evaluate it on
    a test set and store it as the best model.
    """
    if not logs:
        return
    self._next_step = epoch + 1
    train_loss = logs["loss"]
    val_loss = logs["val_loss"]
    mlflow.log_metrics({self.train_loss: train_loss, self.val_loss: val_loss}, step=epoch)

    if val_loss < self._best_val_loss:
        # The result improved in the validation set.
        # Log the model with mlflow and also evaluate and log on test set.
        self._best_train_loss = train_loss
        self._best_val_loss = val_loss
        self._best_model = keras.models.clone_model(self.model)
        self._best_model.set_weights([x.copy() for x in self.model.get_weights()])
        preds = self._best_model.predict(self._test_x)
        eval_and_log_metrics("test", self._test_y, preds, epoch)

@click.command(
    help="Trains a Keras model on wine-quality dataset."
    "The input is expected in csv format."
    "The model and its metrics are logged with mlflow."
)
@click.option("--epochs", type=click.INT, default=100, help="Maximum number of epochs to
    evaluate.")
@click.option("--batch-size", type=click.INT, default=16, help="Batch size passed to the learning algo.")
@click.option("--learning-rate", type=click.FLOAT, default=1e-2, help="Learning rate.")
@click.option("--momentum", type=click.FLOAT, default=0.9, help="SGD momentum.")
@click.option("--seed", type=click.INT, default=97531, help="Seed for the random generator.")
@click.argument("training_data")
def run(training_data, epochs, batch_size, learning_rate, momentum, seed):
    warnings.filterwarnings("ignore")
    data = pd.read_csv(training_data, sep=";")
    # Split the data into training and test sets. (0.75, 0.25) split.
    train, test = train_test_split(data, random_state=seed)
    train, valid = train_test_split(train, random_state=seed)
    # The predicted column is "quality" which is a scalar from [3, 9]
    train_x = train.drop(["quality"], axis=1).as_matrix()

```

```

train_x = (train_x).astype("float32")
train_y = train[[ "quality"]].as_matrix().astype("float32")
valid_x = (valid.drop([ "quality"], axis=1).as_matrix()).astype("float32")

valid_y = valid[[ "quality"]].as_matrix().astype("float32")

test_x = (test.drop([ "quality"], axis=1).as_matrix()).astype("float32")
test_y = test[[ "quality"]].as_matrix().astype("float32")

with mlflow.start_run():
    if epochs == 0: # score null model
        eval_and_log_metrics(
            "train", train_y, np.ones(len(train_y)) * np.mean(train_y), epoch=-1
        )
        eval_and_log_metrics("val", valid_y, np.ones(len(valid_y)) * np.mean(valid_y), epoch=-1)
        eval_and_log_metrics("test", test_y, np.ones(len(test_y)) * np.mean(test_y), epoch=-1)
    else:
        with MLflowCheckpoint(test_x, test_y) as mlflow_logger:
            model = Sequential()
            model.add(Lambda(get_standardize_f(train_x)))
            model.add(
                Dense(
                    train_x.shape[1],
                    activation="relu",
                    kernel_initializer="normal",
                    input_shape=(train_x.shape[1]),
                )
            )
            model.add(Dense(16, activation="relu", kernel_initializer="normal"))
            model.add(Dense(16, activation="relu", kernel_initializer="normal"))
            model.add(Dense(1, kernel_initializer="normal", activation="linear"))
            model.compile(
                loss="mean_squared_error",
                optimizer=SGD(lr=learning_rate, momentum=momentum),
                metrics=[],
            )

            model.fit(
                train_x,
                train_y,
                batch_size=batch_size,
                epochs=epochs,
                verbose=1,
                validation_data=(valid_x, valid_y),
                callbacks=[mlflow_logger],
            )

if __name__ == "__main__":
    run()

```

Как обычно, конфигурация проекта описывается в файле `MLproject`, а конфигурация виртуальной среды – в `conda.yaml`.

### MLproject

```

name: HyperparameterSearch

conda_env: conda.yaml

```

```

entry_points:
  # train Keras DL model
  train: # точка входа
    parameters:
      training_data: {type: string, default: "http://archive.ics.uci.edu/ml/machine-learning-
databases/wine-quality/winequality-white.csv"}
      epochs: {type: int, default: 32}
      batch_size: {type: int, default: 16}
      learning_rate: {type: float, default: 1e-1}
      momentum: {type: float, default: .0}
      seed: {type: int, default: 97531}
    command: "python train.py {training_data}
      --batch-size {batch_size}
      --epochs {epochs}
      --learning-rate {learning_rate}
      --momentum {momentum}"

# Use random search to optimize hyperparams of the train entry_point.
random: # точка входа
  parameters:
    training_data: {type: string, default: "http://archive.ics.uci.edu/ml/machine-learning-
databases/wine-quality/winequality-white.csv"}
    max_runs: {type: int, default: 8}
    max_p: {type: int, default: 2}
    epochs: {type: int, default: 32}
    metric: {type: string, default: "rmse"}
    seed: {type: int, default: 97531}
  command: "python search_random.py {training_data}
    --max-runs {max_runs}
    --max-p {max_p}
    --epochs {epochs}
    --metric {metric}
    --seed {seed}"

# Use GPyOpt to optimize hyperparams of the train entry_point.
gpyopt: # точка входа
  parameters:
    training_data: {type: string, default: "http://archive.ics.uci.edu/ml/machine-learning-
databases/wine-quality/winequality-white.csv"}
    max_runs: {type: int, default: 8}
    batch_size: {type: int, default: 2}
    max_p: {type: int, default: 2}
    epochs: {type: int, default: 32}
    metric: {type: string, default: "rmse"}
    gpy_model: {type: string, default: "GP"}
    gpy_acquisition: {type: string, default: "EI"}
    initial_design: {type: string, default: "random"}
    seed: {type: int, default: 97531}

  command: "python search_gpyopt.py {training_data}
    --max-runs {max_runs}
    --batch-size {batch_size}
    --max-p {max_p}
    --epochs {epochs}
    --metric {metric}
    --gpy-model {gpy_model}
    --gpy-acquisition {gpy_acquisition}
    --initial-design {initial_design}
    --seed {seed}"

```

```

# Use Hyperopt to optimize hyperparams of the train entry_point.
hyperopt: # точка входа
parameters:
  training_data: {type: string, default: "http://archive.ics.uci.edu/ml/machine-learning-
databases/wine-quality/winequality-white.csv"}
  max_runs: {type: int, default: 12}
  epochs: {type: int, default: 32}
  metric: {type: string, default: "rmse"}
  algo: {type: string, default: "tpe.suggest"}
  seed: {type: int, default: 97531}
command: "python -O search_hyperopt.py {training_data}
  --max-runs {max_runs}
  --epochs {epochs}
  --metric {metric}
  --algo {algo}
  --seed {seed}"

main: # точка входа
parameters:
  training_data: {type: string, default: "http://archive.ics.uci.edu/ml/machine-learning-
databases/wine-quality/winequality-white.csv"}
command: "python search_random.py {training_data}"

```

При запуске проекта можно указать точку входа с помощью флага `-e`. Тогда будет вызвана соответствующая команда. Например, если проект запустить как

```
mlflow run -e gpyopt --experiment-id <hyperparam_experiment_id> examples/hyperparam
```

то будет вызвана соответствующая команда из файла `MLproject`

```
python search_gpyopt.py ... # аргументы здесь опущены
```

`conda.yaml`

```

name: hyperparam_example
channels:
  - defaults
  - anaconda
  - conda-forge
dependencies:
  - python=3.6
  - numpy=1.14.3
  - pandas=0.22.0
  - scikit-learn=0.19.1
  - matplotlib=2.2.2
  - keras==2.2.2
  - pip
  - pip:
    - mlflow>=1.0
    - GPy==1.9.2
    - GPyOpt==1.2.5
    - pyDOE==0.3.8
    - hyperopt==0.1

```

## 18.5. Конвейер с использованием MLflow

Рассмотрим пример построения конвейера с использованием MLflow. В корне директории проекта, как обычно, лежит файл `MLproject` следующего содержания

## MLproject

```
name: multistep_example

conda_env: conda.yaml

# точки входа описывают базовую форму вызова python-сценариев;
# нужные значения будут передаваться в main.py через mlflow.run(".", entrypoint, parameters)
entry_points:
    load_raw_data:
        command: "python load_raw_data.py"

    etl_data:
        parameters:
            ratings_csv: path
            max_row_limit: {type: int, default: 100000}
        command: "python etl_data.py --ratings-csv {ratings_csv} --max-row-limit {max_row_limit}"

    als:
        parameters:
            ratings_data: path
            max_iter: {type: int, default: 10}
            reg_param: {type: float, default: 0.1}
            rank: {type: int, default: 12}
        command: "python als.py --ratings-data {ratings_data} --max-iter {max_iter} --reg-param {reg_param} --rank {rank}" # имитируем запуск из командной строки

    train_keras:
        parameters:
            ratings_data: path
            als_model_uri: string
            hidden_units: {type: int, default: 20}
        command: "python train_keras.py --ratings-data {ratings_data} --als-model-uri {als_model_uri} --hidden-units {hidden_units}"

main:
    parameters:
        als_max_iter: {type: int, default: 10}
        keras_hidden_units: {type: int, default: 20}
        max_row_limit: {type: int, default: 100000}
    command: "python main.py --als-max-iter {als_max_iter} --keras-hidden-units {keras_hidden_units} --max-row-limit {max_row_limit}"
```

Главный файл `main.py`, запускает всю цепочку преобразований

### main.py

```
import click
import os

import mlflow
from mlflow.utils import mlflow_tags
from mlflow.entities import RunStatus
from mlflow.utils.logging_utils import eprint

from mlflow.tracking.fluent import _get_experiment_id

def _already_ran(entry_point_name, parameters, git_commit, experiment_id=None):
    """Best-effort detection of if a run with the given entrypoint name,
```

```

parameters, and experiment id already ran. The run must have completed
successfully and have at least the parameters provided.

"""

experiment_id = experiment_id if experiment_id is not None else _get_experiment_id()
client = mlflow.tracking.MlflowClient()
all_run_infos = reversed(client.list_run_infos(experiment_id))
for run_info in all_run_infos:
    full_run = client.get_run(run_info.run_id)
    tags = full_run.data.tags
    if tags.get(mlflow_tags.MLFLOW_PROJECT_ENTRY_POINT, None) != entry_point_name:
        continue
    match_failed = False
    for param_key, param_value in parameters.items():
        run_value = full_run.data.params.get(param_key)
        if run_value != param_value:
            match_failed = True
            break
    if match_failed:
        continue

    if run_info.to_proto().status != RunStatus.FINISHED:
        eprint(
            ("Run matched, but is not FINISHED, so skipping " "(run_id=%s, status=%s)")
            % (run_info.run_id, run_info.status)
        )
        continue

    previous_version = tags.get(mlflow_tags.MLFLOW_GIT_COMMIT, None)
    if git_commit != previous_version:
        eprint(
            (
                "Run matched, but has a different source version, so skipping "
                "(found=%s, expected=%s)"
            )
            % (previous_version, git_commit)
        )
        continue

    return client.get_run(run_info.run_id)
eprint("No matching run has been found.")
return None

# TODO(aaron): This is not great because it doesn't account for:
# - changes in code
# - changes in dependant steps
def _get_or_run(entrypoint, parameters, git_commit, use_cache=True):
    existing_run = _already_ran(entrypoint, parameters, git_commit)
    if use_cache and existing_run:
        print("Found existing run for entrypoint=%s and parameters=%s" % (entrypoint, parameters))
        return existing_run
    print("Launching new run for entrypoint=%s and parameters=%s" % (entrypoint, parameters))
    submitted_run = mlflow.run(".", entrypoint, parameters=parameters) # запускаем соответствующий
    # сценарий из MLproject
    return mlflow.tracking.MlflowClient().get_run(submitted_run.run_id)

@click.command() # важный момент; сообщает, что задекорированная функция является командой
@click.option("--als-max-iter", default=10, type=int)
@click.option("--keras-hidden-units", default=20, type=int)
@click.option("--max-row-limit", default=100000, type=int)

```

```

def workflow(als_max_iter, keras_hidden_units, max_row_limit):
    # Note: The entrypoint names are defined in MLproject. The artifact directories
    # are documented by each step's .py file.
    with mlflow.start_run() as active_run:
        os.environ["SPARK_CONF_DIR"] = os.path.abspath(".") # создаем переменную окружения; здесь
        Spark будет искать файл spark-defaults.conf
        git_commit = active_run.data.tags.get(mlflow_tags.MLFLOW_GIT_COMMIT)
        # запускаем сценарий с точкой входа load_raw_data из MLproject
        load_raw_data_run = _get_or_run("load_raw_data", {}, git_commit)
        ratings_csv_uri = os.path.join(load_raw_data_run.info.artifact_uri, "ratings-csv-dir")
        # запускаем сценарий с точкой входа etl_data из MLproject
        etl_data_run = _get_or_run(
            "etl_data", {"ratings_csv": ratings_csv_uri, "max_row_limit": max_row_limit}, git_commit
        )
        ratings_parquet_uri = os.path.join(etl_data_run.info.artifact_uri, "ratings-parquet-dir")

        # We specify a spark-defaults.conf to override the default driver memory. ALS requires
        # significant memory. The driver memory property cannot be set by the application itself.
        # # запускаем сценарий с точкой входа als из MLproject
        als_run = _get_or_run(
            "als", {"ratings_data": ratings_parquet_uri, "max_iter": str(als_max_iter)}, git_commit
        )
        als_model_uri = os.path.join(als_run.info.artifact_uri, "als-model")

        keras_params = {
            "ratings_data": ratings_parquet_uri,
            "als_model_uri": als_model_uri,
            "hidden_units": keras_hidden_units,
        }
        # запускаем сценарий с точкой входа train_keras из MLproject
        _get_or_run("train_keras", keras_params, git_commit, use_cache=False)

if __name__ == "__main__":
    workflow()

```

Здесь функция `workflow` может принимать аргументы командной строки. Значения аргументам из командной строки можно передать либо непосредственно из командной строки, либо при запуске сценария `main.py` из-под файла `MLproject`

```
python main.py --als-max-iter {als_max_iter} --keras-hidden-units {keras_hidden_units} --max-row
-limit {max_row_limit}
```

После запуска сценария `main.py` вызывается функция `workflow`, которая инициирует «главный запуск». Далее с помощью функции `_get_or_run()` на основании данных, приведенных в файле `MLproject`, вызываются соответствующие точки входа. Например, вызов `_get_or_run("load_raw_data", {}, git_commit)` запускает сценарий `load_raw_data.py` как

```
python load_raw_data.py
```

#### load\_raw\_data.py

```

import requests
import tempfile
import os
import zipfile
import pyspark
import mlflow
import click

```

```

@click.command(
    help="Downloads the MovieLens dataset and saves it as an mlflow artifact "
    " called 'ratings-csv-dir'."
) # сообщает, что эта функция может принимать аргументы командной строки
@click.option("--url", default="http://files.grouplens.org/datasets/movielens/ml-20m.zip") # arg
    умень со значением по умолчанию
def load_raw_data(url):
    with mlflow.start_run() as mrun:
        local_dir = tempfile.mkdtemp()
        local_filename = os.path.join(local_dir, "ml-20m.zip")
        print("Downloading %s to %s" % (url, local_filename))
        r = requests.get(url, stream=True) # <- NB
        with open(local_filename, "wb") as f:
            for chunk in r.iter_content(chunk_size=1024): # файл можно скачивать блоками
                if chunk: # filter out keep-alive new chunks
                    f.write(chunk)

        extracted_dir = os.path.join(local_dir, "ml-20m")
        print("Extracting %s into %s" % (local_filename, extracted_dir))
        with zipfile.ZipFile(local_filename, "r") as zip_ref:
            zip_ref.extractall(local_dir)

        ratings_file = os.path.join(extracted_dir, "ratings.csv")

        print("Uploading ratings: %s" % ratings_file)
        mlflow.log_artifact(ratings_file, "ratings-csv-dir")

if __name__ == "__main__":
    load_raw_data()

```

В сценарии `load_raw_data.py` функция `load_raw_data` может принимать аргументы командной строки. В данном случае у аргумента функции `url` есть значение по умолчанию. Функция `load_raw_data` инициирует свой собственный запуск, в котором создается временная директория для файла `ml-20m.zip`, который скачивается по заданному URL.

Далее zip-файл с помощью `zip_ref.extractall(local_dir)` распаковывается во временную директорию. Остается только залогировать этот файл с помощью MLflow

```

mlflow.log_artifact(
    ratings_file, # это путь до файла, который требуется залогировать
    "ratings-csv-dir" # это путь куда нужно поместить файл; путь отсчитывается от поддиректории
    artifacts
)

```

Здесь файл `ratings.csv`, расположенный во временной директории, с помощью `mlflow.log_artifacts` будет размещен в поддиректории `artifacts`, ассоциированной с соответствующим запуском, а именно

```

...
|- artifacts/
  |- ratings-csv-dir/
    |- ratings.csv # <- NB

```

Теперь переменная `load_raw_data_run` в `main.py` может получить ссылку на объект запуска. Извлекаем путь до поддиректории `artifacts` этого запуска с помощью

```
load_raw_data_run.info.artifact_uri
```

Затем конструируем путь до поддиректории ratings-csv-dir

```
ratings_csv_uri = os.path.join(load_raw_data_run.info.artifact_uri, "ratings-csv-dir")
```

и этот путь передаем в следующую точку вызова

```
etl_data_run = _get_or_run(  
    "etl_data",  
    {  
        "ratings_csv": ratings_csv_uri, # путь до директории, в которой лежит ratings.csv  
        "max_row_limit": max_row_limit  
    },  
    git_commit  
) # запускаем сценарий etl_data на основании MLproject
```

Точка входа etl\_data в MLproject ожидает два параметра ratings\_csv и max\_row\_limit. Значения этим параметрам мы и передаем через функцию \_get\_or\_run()

```
...  
mlflow.run(  
    ".",  
    entrypoint, # etl_data  
    parameters=parametrs # {"ratings_csv": ratings_csv_uri, "max_row_limit": max_row_limit}  
)
```

Теперь нужно смотреть как организован сценарий etl\_data.py

etl\_data.py

```
import tempfile  
import os  
import pyspark  
import mlflow  
import click  
  
@click.command( # задекорированная функция может принимать аргументы командной строки  
    help="Given a CSV file (see load_raw_data), transforms it into Parquet "  
    "in an mlflow artifact called 'ratings-parquet-dir'"  
)  
@click.option("--ratings-csv")  
@click.option(  
    "--max-row-limit", default=10000, help="Limit the data size to run comfortably on a laptop."  
)  
def etl_data(ratings_csv, max_row_limit):  
    with mlflow.start_run() as mrlrun:  
        tmpdir = tempfile.mkdtemp()  
        ratings_parquet_dir = os.path.join(tmpdir, "ratings-parquet")  
        spark = pyspark.sql.SparkSession.builder.getOrCreate()  
        print("Converting ratings CSV %s to Parquet %s" % (ratings_csv, ratings_parquet_dir))  
        ratings_df = (  
            spark.read.option("header", "true")  
            .option("inferSchema", "true")  
            .csv(ratings_csv) # читается csv-файл из директории, путь до которой мы передали в main.py  
            .drop("timestamp")  
        ) # удаляем неиспользуемый столбец  
        ratings_df.show()  
        if max_row_limit != -1:  
            ratings_df = ratings_df.limit(max_row_limit)
```

```

ratings_df.write.parquet(ratings_parquet_dir) # запись csv-файла в формате parquet
print("Uploading Parquet ratings: %s" % ratings_parquet_dir)

mlflow.log_artifacts(
    ratings_parquet_dir, # путь до директории, содержание которой нужно скопировать в заданную
    # директорию
    "ratings-parquet-dir" # путь до директории относительно artifacts/, куда нужно скопировать
    # данные
)

if __name__ == "__main__":
    etl_data()

```

Сценарий `etl_data.py` содержит функцию `etl_data`, которая может принимать аргументы командной строки. Функция инициирует свой собственный запуск, в котором создается временная директория, к которой на следующей строке добавляется имя директории `ratings-parquet`.

Далее инициируется сессия PySpark, читается csv-файл, отсекаются первые 10000 строк этого файла и результат записывается в формате parquet в поддиректорию `ratings-parquet-dir`, расположенную во временной директории.

В результате в поддиректории `ratings-parquet-dir` директории `artifacts` будет лежать файл с расширением `*.parquet`

```

...
|- artifacts/
  |- ratings-parquet-dir/
    |- part-00000-5ab4571f-b0f0-438a-b5a5-...c000.snappy.parquet

```

То есть, теперь в указанной директории лежит parquet-файл. Путь до директории `ratings-parquet-dir` в поддиректории, ассоциированной с определенным запуском, через конструкцию

```

ratings_parquet_uri = os.path.join(
    etl_data_run.info.artifact_uri,
    "ratings-parquet-dir"
)

```

можно передать в следующую точку входа, а именно

```

als_run = _get_or_run(
    "als",
    {
        "ratings_data" : ratings_parquet_uri, # путь до директории с parquet-файлом
        "max_iter" : str(als_max_iter)
    },
    git_commit
)

```

В файле `MLproject` вызов точки входа `als` выглядит так

#### фрагмент `MLproject`

```

...
als:
parameters:
    ratings_data: path
    max_iter: {type: int, default: 10}
    reg_param: {type: float, default: 0.1}
    rank: {type: int, default: 12}
    command: python als.py

```

```
--ratings-data {ratings_data}
--max-iter {max_iter}
--reg-param {reg_param}
--rank {rank}""
...
```

Посмотрим на организацию сценария als.py

als.py

```
import click

import mlflow
import mlflow.spark

import pyspark
from pyspark.ml import Pipeline
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator

@click.command()
@click.option("--ratings-data")
@click.option("--split-prop", default=0.8, type=float)
@click.option("--max-iter", default=10, type=int)
@click.option("--reg-param", default=0.1, type=float)
@click.option("--rank", default=12, type=int)
@click.option("--cold-start-strategy", default="drop")
def train_als(ratings_data, split_prop, max_iter, reg_param, rank, cold_start_strategy):
    seed = 42

    spark = pyspark.sql.SparkSession.builder.getOrCreate()

    ratings_df = spark.read.parquet(ratings_data)
    (training_df, test_df) = ratings_df.randomSplit([split_prop, 1 - split_prop], seed=seed)
    training_df.cache()
    test_df.cache()

    mlflow.log_metric("training_nrows", training_df.count())
    mlflow.log_metric("test_nrows", test_df.count())

    print("Training: {}, test: {}".format(training_df.count(), test_df.count()))

    als = ( # создаем экземпляр класса ML-модели
        ALS()
        .setUserCol("userId")
        .setItemCol("movieId")
        .setRatingCol("rating")
        .setPredictionCol("predictions")
        .setMaxIter(max_iter)
        .setSeed(seed)
        .setRegParam(reg_param)
        .setColdStartStrategy(cold_start_strategy)
        .setRank(rank)
    )
    # собираем конвейер
    als_model = Pipeline(stages=[als]).fit(training_df)

    reg_eval = RegressionEvaluator(predictionCol="predictions", labelCol="rating", metricName="mse")
    # делаем предсказания
```

```

predicted_test_df = als_model.transform(test_df)

test_mse = reg_eval.evaluate(predicted_test_df)
train_mse = reg_eval.evaluate(als_model.transform(training_df))

print("The model had a MSE on the test set of {}".format(test_mse))
print("The model had a MSE on the (train) set of {}".format(train_mse))
# логируем метрики
mlflow.log_metric("test_mse", test_mse)
mlflow.log_metric("train_mse", train_mse)
# логируем модель относительно поддиректории artifacts
mlflow.spark.log_model(als_model, "als-model")

if __name__ == "__main__":
    train_als()

```

ВАЖНО: декоратор `@click.command()` превращает задекорированную функцию в вызываемый сценарий. То есть, если из командной строки вызывать сценарий, который содержит функцию, задекорированную `@click.command()`, то эта функция будет вызвана при запуске сценария.

В этом сценарии мы снова запускаем spark-сессию, а затем читаем parquet-файл. Потом разбиваем этот файл на обучающий и тестовый поднабор и кешируем их

```

ratings_df = spark.read.parquet(ratings_data)
(training_df, test_df) = ratings_df.randomSplit([split_prop, 1 - split_prop], seed=seed)
training_df.cache()
test_df.cache()

```

Затем конфигурируем модель, обучаем ее, делаем прогноз и логируем метрики и модель. Обученная модель будет располагаться в поддиректории `als-model` поддиректории `artifacts`

```

./
|- mlruns/
  |- 0/ # id эксперимента
    |- b552...699/ # id запуска
      |- artifacts/
        |- als-model
          |- sparkml
            MLmodel
            conda.yaml

```

Снова конструируем путь до директории, в которой лежит обученная spark-модель

```

als_model_uri = os.path.join(
    als_run.info.artifact_uri, "als-model"
)

```

Теперь этот путь можно передать в следующую точку входа

```

keras_params = {
    "ratings_data": ratings_parquet_uri, # путь до директории с parquet-файлом
    "als_model_uri": als_model_uri,       # путь до директории с обученной моделью
    "hidden_units": keras_hidden_units,
}

```

Наконец, можно запустить последнюю точку входа

```

mlflow.run(
    ".",
    entrypoint, # train_keras

```

```
    parameters=keras_params  
)
```

которая запускает

```
python train_keras.py  
--ratings-data {ratings_data}  
--als-model-uri {als_model_uri}  
--hidden-units {hidden_units}
```

Рассмотрим сценарий `train_keras.py`

```
train_keras.py  
  
import click  
  
import mlflow  
import mlflow.keras  
import mlflow.spark  
  
from itertools import chain  
import pyspark  
from pyspark.sql.functions import *  
from pyspark.sql.types import *  
  
import tensorflow as tf  
import tensorflow.keras as keras  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense  
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping  
import numpy as np  
import pandas as pd  
  
  
@click.command()  
@click.option("--ratings-data", help="Path readable by Spark to the ratings Parquet file")  
@click.option("--als-model-uri", help="Path readable by load_model to ALS MLmodel")  
@click.option("--hidden-units", default=20, type=int)  
def train_keras(ratings_data, als_model_uri, hidden_units):  
    np.random.seed(0)  
    tf.set_random_seed(42) # For reproducibility  
    # иницируем сессию  
    spark = pyspark.sql.SparkSession.builder.getOrCreate()  
    # загружаем обученную модель  
    als_model = mlflow.spark.load_model(als_model_uri).stages[0]  
    # читаем parquet-файл  
    ratings_df = spark.read.parquet(ratings_data)  
    # разбиваем на обучение и тест  
    (training_df, test_df) = ratings_df.randomSplit([0.8, 0.2], seed=42)  
    training_df.cache()  
    test_df.cache()  
  
    mlflow.log_metric("training_nrows", training_df.count())  
    mlflow.log_metric("test_nrows", test_df.count())  
  
    print("Training: {}, test: {}".format(training_df.count(), test_df.count()))  
  
    user_factors = als_model.userFactors.selectExpr("id as userId", "features as uFeatures")  
    item_factors = als_model.itemFactors.selectExpr("id as movieId", "features as iFeatures")  
    joined_train_df = training_df.join(item_factors, on="movieId").join(user_factors, on="userId")  
    joined_test_df = test_df.join(item_factors, on="movieId").join(user_factors, on="userId")
```

```

# We'll combine the movies and ratings vectors into a single vector of length 24.
# We will then explode this features vector into a set of columns.
def concat_arrays(*args):
    return list(chain(*args))

concat_arrays_udf = udf(concat_arrays, ArrayType(FloatType()))

concat_train_df = joined_train_df.select(
    "userId",
    "movieId",
    concat_arrays_udf(col("iFeatures"), col("uFeatures")).alias("features"),
    col("rating").cast("float"),
)
concat_test_df = joined_test_df.select(
    "userId",
    "movieId",
    concat_arrays_udf(col("iFeatures"), col("uFeatures")).alias("features"),
    col("rating").cast("float"),
)

pandas_df = concat_train_df.toPandas()
pandas_test_df = concat_test_df.toPandas()

# This syntax will create a new DataFrame where elements of the 'features' vector
# are each in their own column. This is what we'll train our neural network on.
x_test = pd.DataFrame(pandas_test_df.features.values.tolist(), index=pandas_test_df.index)
x_train = pd.DataFrame(pandas_df.features.values.tolist(), index=pandas_df.index)

# Show matrix for example.
print("Training matrix:")
print(x_train)

# Create our Keras model with two fully connected hidden layers.
model = Sequential()
model.add(Dense(30, input_dim=24, activation="relu"))
model.add(Dense(hidden_units, activation="relu"))
model.add(Dense(1, activation="linear"))

model.compile(loss="mse", optimizer=keras.optimizers.Adam(lr=0.0001))

filepath = "/tmp/ALS_checkpoint_weights.hdf5"
early_stopping = EarlyStopping(monitor="val_loss", min_delta=0.0001, patience=2, mode="auto")

model.fit(
    x_train,
    pandas_df["rating"],
    validation_split=0.2,
    verbose=2,
    epochs=3,
    batch_size=128,
    shuffle=False,
    callbacks=[early_stopping],
)

train_mse = model.evaluate(x_train, pandas_df["rating"], verbose=2)
test_mse = model.evaluate(x_test, pandas_test_df["rating"], verbose=2)
mlflow.log_metric("test_mse", test_mse)
mlflow.log_metric("train_mse", train_mse)

```

```

print("The model had a MSE on the test set of {}".format(test_mse))
mlflow.keras.log_model(model, "keras-model") # логируем модель в поддиректории keras-model под
# директории artifacts

if __name__ == "__main__":
    train_keras()

```

ВАЖНО: если в файле MLproject какой-то параметр не имеет значения по умолчанию

#### MLproject

```

name: docker-example

docker_env:
    image: mlflow-docker-example

entry_points:
    main:
        parameters:
            alpha: float # нет значения по умолчанию
            l1_ratio: {type: float, default: 0.1}
        command: "python train.py --alpha {alpha} --l1-ratio {l1_ratio}"

```

то значение этому параметру можно передать при запуске проекта, например

```
mlflow run example/docker -P alpha=0.3
```

Здесь проект запускается без указания точки входа, что сообщает MLflow о требовании запустить точку входа с именем по умолчанию (т.е. main). MLflow ищет в файле MLproject точку входа с именем main и запускает соответствующую команду.

Разумеется можно комбинировать именованные флаги и флаги -P

```

mlflow run . -e hyperopt \
    --experiment-name RAPIDS-CLI \ # специальный флаг, не имеющий отношения к опциям данной точки
    # входа
    --conda-env=$PWD/envs/conda.yaml \
    -P fpath=airline_small.parquet

```

Флаг -P указывает, что в файле MLproject у точки входа с именем hyperopt есть параметр с именем fpath. Однако сценарий адаптирован под ситуацию, когда нужно принять значение флага --conda-env, даже если он не описан в MLproject.

## 19. WSGI- и ASGI-серверы

WSGI (Web Server Gateway Interface) – протокол взаимодействия между *Python-приложением* и *веб-сервером*. WSGI-серверы появились потому, что веб-серверы раньше не умели взаимодействовать с приложениями, написанными на Python.

ASGI (Asynchronous Server Gateway Interface) – клиент-серверный протокол взаимодействия *веб-сервера и приложения* (это развитие технологии WSGI). По сравнению с WSGI предоставляет стандарт как для *асинхронных*, так и для *синхронных* приложений, с реализацией обратной совместимости WSGI.

## 20. NGINX

nginx <https://nginx.org/ru/> – это *HTTP-сервер* и обратный прокси-сервер, почтовый прокси-сервер, а также TCP/UDP прокси-сервер общего назначения.

ЗАМЕЧАНИЕ: web-сервер – это абстракция, которая вообщем-то ничего конкретного не означает. Или чуть подробнее, web-сервер – это программный комплекс, необходимый для поддержания работы web-протоколов и непосредственно железо, на котором эти программы работают (физические сервера). http-сервер – сервер, работающий по протоколу/ам HTTP/HTTPS. Или чуть более развернутъ, http-сервер – это всего лишь одна программа, реализующая взаимодействие по протоколу HTTP. Однако, когда говорят web-сервер, то обычно имеют ввиду именно http-сервер.

Когда говорят о связке «NGINX-gunicorn-Python\_webapp», то в ней nginx принимает запросы от клиентов (web-браузеров) и перенаправляет их WSGI-серверу (gunicorn), который в свою очередь перенаправляет их в Python-приложение. Когда Python-приложение возвращает какой-то ответ, этот ответ направляется на WSGI-сервер, который затем перенаправляет ответ на http-сервер nginx рис. 12.

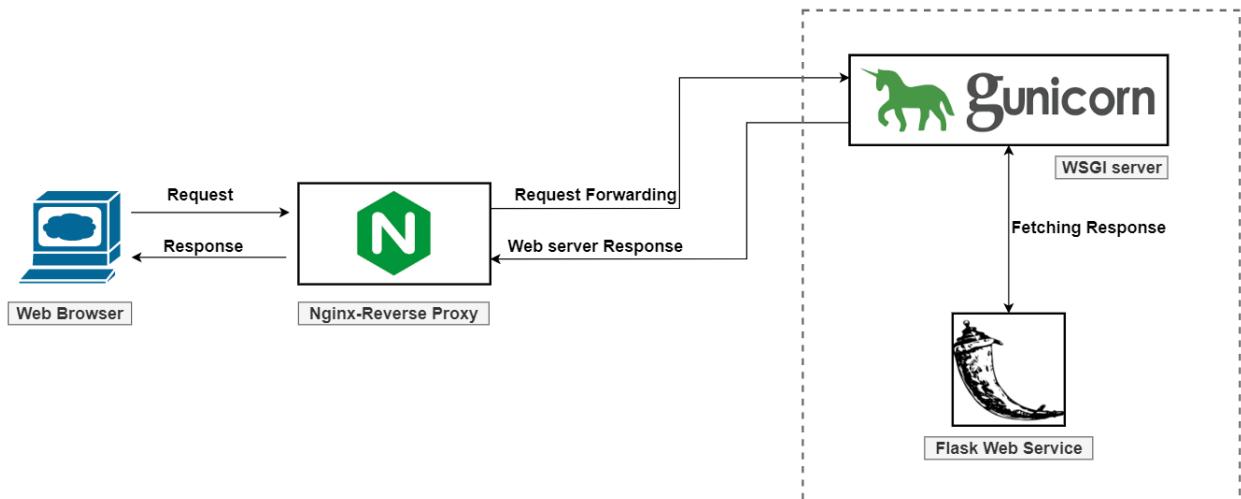


Рис. 12. Связка NGINX - gunicorn - Python-webapp

## 21. Приемы разработки web-приложений с помощью FastAPI

Много полезных материалов можно найти на официальном сайте FastAPI <https://fastapi.tiangolo.com/>.

Установка необходимых компонентов

```
pip install wheel -U  
pip install uvicorn fastapi pydantic
```

uvicorn <https://www.uvicorn.org/> – это ASGI веб-сервер<sup>10</sup>.

Последнюю часть URL, начинающуюся с первого символа /, называют endpoint или маршрутом (route). Например, в URL <https://example.com/items/foo> путем (конечной точкой или маршрутом) будет `/items/foo`.

Под операциями понимаются http-методы:

<sup>10</sup>Не путать с gunicorn (gunicorn – это WSGI-сервер, созданный для использования в UNIX-системах)

- POST: создание данных,
- GET: чтение данных,
- PUT: обновление данных,
- DELETE: удаление данных,
- OPTIONS,
- HEAD,
- PATCH,
- TRACE

Пример

main.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/") # HTTP-метод GET на пути /
async def root():
    return {"message": "Hello World"}
```

Здесь `@app.get("/")` говорит FastAPI, что задекорированная функция отвечает за обработку запросов, которые идут на «корень» web-сервиса, используя метод GET.

Функция `root` будет вызываться всякий раз, когда web-сервис получает запрос на «корень», то есть на `"/"` с использованием метода GET.

Аналогично можно использовать и другие операции: `@app.post()`, `@app.put()`, `@app.delete()` и т.д.

Запустить приложение из командной строки можно так

```
uvicorn main:app --reload # reload для перезапуска сервера после внесения изменений в код
```

Можно подставлять аргументы в маршрут. Например

```
from fastapi import FastAPI

app = FastAPI()

# http://127.0.0.1:8000/items/foo -> item_id = "foo"
@app.get("/items/{item_id}")
async def read_item(item_id): # item_id = "foo"
    return {"item_id": item_id}
```

Теперь, если запустить этот пример как

```
http://127.0.0.1:8000/items/foo
```

то значение `"foo"` из маршрута будет передано функции `read_item` в виде аргумента `item_id`.

Можно явно указывать тип параметра в функции

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

FastAPI действительно проверяет тип параметров. И если тип параметра не соответствует указанному в функции, то возникает ошибка.

Если нужно передать какой-то путь, то следует использовать такой синтаксис /files/file\_path:path

```
from fastapi import FastAPI

app = FastAPI()

# например, localhost:8000/files//leor.finkelberg/python -> file_path = "/leor.finkelberg/python"
@app.get("/files/{file_path:path}")
async def read_file(file_path: str):
    return {"file_path": file_path}
```

Еще можно использовать значения параметров, передаваемые в запросе, но не имеющие отношения к пути. Такие параметры передаются парами после знака вопроса и отделяются друг от друга знаком амперсанда, то есть

```
http://127.0.0.1:8000/10?key1=value1&key2=value2&...
```

```
from fastapi import FastAPI

app = FastAPI()

# http://127.0.0.1:8000/items/10?solver_type=iterative
@app.get("/items_id/{item_id}")
async def read_item(item_id: int, solver_type: str = "direct"):
    return {"item_id": item_id, "type": solver_type} # {"item_id": 10, "type": "iterative"}
```

В запросе можно передавать булевые значения

```
http://127.0.0.1:8000/items/100?q=test&short=True
# или так
http://127.0.0.1:8000/items/100?q=test&short=true
# или так
http://127.0.0.1:8000/items/100?q=test&short=1
# или так
http://127.0.0.1:8000/items/100?q=test&short=on
# или так
http://127.0.0.1:8000/items/100?q=test&short=yes
```

Маршрут может содержать несколько параметров, значения которых будут передаваться в функцию как аргументы

```
from typing import Optional, NoReturn
from fastapi import FastAPI

app = FastAPI()

# http://127.0.0.1:8000/users/1632/items/admin?q=test&short=true
@app.get("/users/{user_id}/items/{item_id}")
async def read_item(
    user_id: int,
    item_id: str,
    q: Optional[str] = None,
    short: bool = False
) -> NoReturn:
    item = {"user_id": user_id, "item_id": item_id}
```

```

if q:
    item.update({"q" : q})
if not short:
    item.update(
        {"description" : "This is an ..."}
    )

return item

```

С помощью специального класса `Query` можно добавить, например, ограничение по длине строки на значение параметра

```

from typing import Optional, Dict

from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
async def read_items(
    q: Optional[str] = Query(None, max_length=50) # если длина строки превысит 50 символов, то в
                                                # озникнет ошибка
) -> Dict:
    results = {"items" : [{"item_id" : "Foo"}, {"item_id" : "Bar"}]}
    if q:
        results.update({"q" : q})
    return results

```

ЗАМЕЧАНИЕ: конструкция `q: Optional[str] = Query(None)` эквивалентна конструкции `q: Optional[str] = None`, но первый вариант более явный. То есть с помощью `Query(None)` передается значение по умолчанию `None`, но без ограничений на строковую константу.

Разумеется можно добавить и ограничение на минимальную длину строки

```

@app.get("/items/")
async def read_items(
    q: Optional[str] = Query(
        None,
        min_length=3,
        max_length=50
    )
) -> Dict:
    ...

```

Есть возможность добавить регулярное выражение

```

@app.get("/items/")
async def read_items(
    q: Optional[str] = Query(
        None,
        min_length=3,
        max_length=50,
        regex="^.*python.*$"
    )
)

```

С помощью `Query` можно задавать значения по умолчанию и отличные от `None`

```

@app.get("/items/")
async def read_items(
    q: str = Query(

```

```
    "fixedquery",
    min_length=3
)
) -> Dict:
...

```

Чтобы сделать параметр обязательным, нужно просто не передавать ему значение по умолчанию и не использовать класс `Query`. Однако, даже когда используется класс `Query` параметр можно сделать обязательным, передав ему в качестве значения по умолчанию многоточие (...)

```
@app.get("/items/")
async def read_items(
    q: str = Query(
        ..., # многоточие сообщает FastAPI, что параметр 'q' обязательный
        min_length=3
)
) -> Dict:
...

```

При таком способе определения параметра со значением по умолчанию сохраняются все преимущества использования ограничений, накладываемых на параметр.

Для того чтобы сообщить FastAPI, что параметр может встречаться в URL несколько раз, можно использовать следующую конструкцию

```
# http://localhost:8000/items/?q=test&q=fortran
@app.get("/items/")
async def read_items(
    q: Optional[List[str]] = Query(None) # <- NB: обязательно нужен класс Query
) -> Dict:
    query_items = {"q": q}
    return query_items # {"q": ["test", "fortran"]}
```

Чтобы объявить параметр запроса с типом списка, как в примере выше, необходимо явно использовать класс `Query`. Иначе параметр запроса будет интерпретироваться как тело запроса.

Для параметра, который встречается в URL несколько раз, можно явно передать список его значений

```
# http://localhost:8000/items
@app.get("/items/")
async def read_items(
    q: List[str] = Query(
        ["foo", "bar"]
    )
) -> Dict:
    query_items = {"q": q}
    return query_items
```

ЗАМЕЧАНИЕ: проверить работоспособность функции `read_items` с методом GET (читать) можно несколькими способами:

- набрав в поисковой строке браузера, например, `http://localhost:8000/items/88`; тогда в ответ в браузере появится что-нибудь вроде `{"item" : 88}`,
- или с помощью утилиты `curl`

```
curl -X 'GET' \
    'http://localhost:8000/items/88' \
    -H 'accept: application/json'
```

Если требуется в URL указать параметр, имя которого считается недопустимым с точки зрения Python, то можно воспользоваться специальным параметром `alias` класса `Query`

```
# http://localhost:8000/items/?item-query=fortran
@app.get("/items/")
async def read_items(
    q: Optional[str] = Query(
        None,
        alias="item-query"
    ) # q <-> item-query
) -> Dict:
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results # {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}], "q": "fortran"}
```

Можно пометить параметр как «устаревший» с помощью параметра `deprecated` класса `Query`.

Ванильный Python ругается, если первым в функции указать параметр со значением по умолчанию (ведь сначала должны идти позиционные аргументы, а уже потом аргументы со значением по умолчанию), но для FastAPI это не имеет никакого значения.

Как и в ванильном Python при объявлении функции можно использовать `*`, для того чтобы сообщить FastAPI о том, что аргументам, стоящим справа от звездочки значения следует передавать явно (то есть по имени), а не позиционно

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/items/{item_id}")
async def read_items(
    *, # для FastAPI порядок позиционных и именновых аргументов не имеет значения
    item_id: int = Path(
        ..., # значения по умолчанию у item_id нет (обязательный параметр)
        title="The ID of the item to get"
    ),
    q: str
) -> Dict:
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

ВАЖНО: класс `Path` работает также как и класс `Query`, но только для переменных в контексте маршрута, например, `"/items/{item_id}"`. Класс `Query` применяется для аргументов, не относящихся к маршруту.

Ограничения можно накладывать разумеется не только на строковые константы, но и на вещественные значения аргументов функции

```
@app.get("/items/{item_id}")
async def read_items(
    *, # аргументам, расположенным правее *, значения следует передавать по имени
    item_id: int = Path(
        ..., # нет значения по умолчанию
        title="The ID ...",
        ge=1 # "Greater than or Equal", m.e. >= 1
    ),
    q: str,
```

```
) -> Dict:
    results = {"item_id" : item_id}
    if q:
        results.update({"q" : q})
    return results
```

Более сложный пример, обобщающий рассмотренные выше приемы оформления аргументов, относящихся и не относящихся к пути

```
# http://localhost:8000/items/1632?num_of_cores=12
@app.get("/items/{item_id}")
async def read_items(
    item_id : int = Path(
        ... # у item_id нет значения по умолчанию (обязательный параметр)
    ),
    *, # аргументам, стоящим справа от *, следует передавать значение по имени
    q: Optional[str] = Query( # необязательный параметр, принимающий объект опционального типа
        None,
        alias="item-query", # псевдоним
        title="Query string",
        description="Query string for the items to search in the database that...",
        min_length=3,
        max_length=50,
        regex="^.*python.*$",
        deprecated=True,
    ),
    solver_type: str = Query(
        "direct" # значение по умолчанию для переменной, не имеющей отношения к пути
    ),
    num_of_cores: int = Query(
        10, # значение по умолчанию для переменной, не имеющей отношения к пути
        ge=8, # >= 8
        le=15, # <= 15
    )
) -> Dict:
    results = {"items" : item_id, "solver_type" : solver_type, "num_of_cores" : num_of_cores}
    if q:
        results.update({"q" : q})

    return results # {"items":1632,"solver_type":"direct","num_of_cores":12}
```

Аналогично можно задать ограничения и для вещественных значений: для вещественных значений следует указывать строгие условия (>, <, но не <= и т.д.)

```
@app.get("/items/{item_id}")
async def read_items(
    *,
    item_id : int = Path(
        ... # нет значения по умолчанию (обязательный параметр)
        ge=0, # >= 0
        le=1000, # <= 1000
    )
    q: str,
    size: float = Query(
        ... # нет значения по умолчанию (обязательный параметр)
        gt=0, # > 0
        lt=10.5 # < 10.5
    )
)
```

Если нужно передать в запрос дополнительный параметр, не изменения структуры сложных типов других параметров запроса, то можно воспользоваться классом `Body`

```
from typing import Optional
from fastapi import Body, FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

class User(BaseModel):
    username: str
    full_name: Optional[str] = None

@app.put("/items/{item_id}")
async def update_item(
    item_id: int,
    item: Item,
    user: User,
    importance: int = Body(...),
):
    results = {
        "item_id": item_id, "item": item,
        "user": user, "importance": importance
    }
    return results
```

В этом случае FastAPI ожидает получить тело вида

```
{
    "item": {
        "name": "Foo",
        "description": "The pretender",
        "price": 42.0,
        "tax": 3.2
    },
    "user": {
        "username": "dave",
        "full_name": "Dave Grohl"
    },
    "importance": 5
}
```

Конечно, вместе с `Body` можно передавать и обычные параметры запроса

```
@app.put("/items/{item_id}")
async def update_item(
    *,
    item_id: int,
    item: Item,
    user: User,
    importance: int = Body(..., gt=0),
    q: Optional[str] = None
):
    ...
```

Если тело запроса должно быть вложенным, то это легко организовать с помощью параметра `embed`

```
@app.put("/items/{item_id}")
async def update_item(
    item_id: int,
    item: Item = Body(..., embed=True)
) -> Dict:
    results = {"item_id": item_id, "item": item}
    return results
```

В этом случае FastAPI будет ожидать тело вида

```
{
    "item": {
        "name": "Foo",
        "description": "The pretender",
        "price": 42.0,
        "tax": 3.2
    }
}
```

Вместо

```
{
    "name": "Foo",
    "description": "The pretender",
    "price": 42.0,
    "tax": 3.2
}
```

Так же как и для классов `Query`, `Path` и `Body` можно организовать дополнительную проверку внутри классов `pydantic` с помощью `Field`

```
from typing import Optional

from fastapi import Body, FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):
    name: str
    desctiption: Optional[str] = Field(
        None,
        title = "The desc...",
        max_length=300
    )
    price: float = Field(
        ...,
        gt=0,
        description="The ...",
    )
    tax: Optional[float] = None

@app.put("/items/{item_id}")
async def update_item(
    item_id: int,
    item: Item = Body(
        ...,
        embed=True
    )
) -> Dict:
    results = {"item_id": item_id, "item": item}
    return results
```

```

        )
) -> Dict:
    results = {"item_id" : item_id, "item" : item}
    return results

```

Field работает так же как Query, Path и Body и принимает те же параметры.

Можно объявить тип возвращаемого объекта в параметре response\_model

```

# НИ В КОЕМ СЛУЧАЕ НЕ ДЕЛАТЬ ТАК В ПРОДЕ!!!
class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: Optional[str] = None

# создать пользователя с помощью метода POST
@app.post("/user/", response_model=UserIn)
async def create_user(user: UserIn):
    return user

```

Здесь класс UserIn используется как для объявления входных данных, так и выходных данных. Теперь, когда создается пользователь с паролем, этот пароль возвращается в ответе.

Проверить работу функции create\_user с методом POST можно так

```

curl -X 'POST' \
'http://localhost:8000/user/' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "username": "string",
    "password": "string",
    "email": "user@example.com",
    "full_name": "string"
}'

```

Можно создать класс для описания выходной модели без поля для пароля

```

class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: Optional[str] = None

class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: Optional[str] = None

@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn):
    return user

```

ЗАМЕЧАНИЕ: в функциях с методом POST (создать), как в пример выше

```

@app.post("/items/", response_model=UserOut) # POST
async def create_user(user: UserIn):
    return user

```

соответствующий аргумент функции (в данном случае аргумент `user`) связывается с json-объектом, который передается через флаг `-d` в `curl`. То есть неявно переменная `user` как бы получает ссылку на json-объект. В функциях с методом GET (читать) значение переменной функции передается в маршруте, например

```
@app.get("/items/{item_id}")
async def read_items(item_id: int):
    ...
```

Для проверки работы функции с новой классом на выходных параметров воспользуемся таким запросом

```
curl -X 'POST' \
  'http://localhost:8000/user/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  # то, что передается через -d связывается с аргументом user функции create_user, объявлённой выше
  -d '{
    "username": "Leor",
    "password": "Evdimonia",
    "email": "leor.finkelberg@yandex.ru",
    "full_name": "Leor Finkelberg"
}'
```

Но ответ не будет содержать пароля

```
{ # без пароля!
  "username": "Leor",
  "email": "leor.finkelberg@yandex.ru",
  "full_name": "Leor Finkelberg"
}
```

Если ответы содержат много параметров со значением по умолчанию, а интерес представляют только те параметры, которым были переданы значения явно в текущей сессии, то можно использовать параметр `response_model_exclude_unset`

```
@app.get(
    "/items/{item_id}",
    response_model=Item,
    response_model_exclude_unset=True # <- NB
)
async def read_item(item_id: str):
    ...
```

Тогда ответ будет содержать только те параметры, которым были переданы значения в текущей сессии (параметры со значениями по умолчанию будут опущены).

Отфильтровать параметры на выходе можно с помощью параметров `response_model_exclude` и `response_model_include`

```
@app.get(
    "/items/{item_id}/name",
    response_model=Item,
    response_model_include={"name", "description"},
)
async def read_item_name(item_id: str):
    return items[item_id] # ответ будет включать только ключи name и description

@app.get(
```

```

"/items/{item_id}/public",
response_model=Item,
response_model_exclude={"tax"}
)
async def read_item_public_data(item_id: str):
    return items[item_id] # отсек будем включать все ключи кроме tax

```

Обновлять данные можно с помощью метода PUT

```

from typing import List, Optional

from fastapi import FastAPI
from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: Optional[str] = None
    description: Optional[str] = None
    price: Optional[float] = None
    tax: float = 10.5
    tags: List[str] = []

items = {
    "foo": {"name": "Foo", "price": 50.2},
    "bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},
    "baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},
}

@app.get("/items/{item_id}", response_model=Item) # GET
async def read_item(item_id: str):
    return items[item_id]

@app.put("/items/{item_id}", response_model=Item) # PUT
async def update_item(item_id: str, item: Item):
    update_item_encoded = jsonable_encoder(item)
    items[item_id] = update_item_encoded
    return update_item_encoded

```

Теперь чтобы обновить данные направим два запроса

```

$ curl -X 'PUT' \
'http://localhost:8000/items/2' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "name": "Ansys",
    "description": "CAE package...",
    "price": 250,
    "tax": 0.5,
    "tags": []
}'

$ curl -X 'PUT' \
'http://localhost:8000/items/42' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \

```

```

-d '{
    "name": "Nastran",
    "description": "CAE package...",
    "price": 480,
    "tax": 8.5,
    "tags": []
}',

```

А чтобы прочитать данные запросы для разных идентификационных номеров должны выглядеть так

```

$ curl -X 'GET' \
'http://localhost:8000/items/2' \
-H 'accept: application/json' # вернем json для Ansys
$ curl -X 'GET' \
'http://localhost:8000/items/42' \
-H 'accept: application/json' # вернем json для Nastran

```

В случае, когда данные требуется обновлять частично, удобно использовать `exclude_unset`

```

class Item(BaseModel):
    name: Optional[str] = None
    description: Optional[str] = None
    price: Optional[float] = None
    tax: float = 10.5
    tags: List[str] = []

@app.patch("/items/{item_id}", response_model=Item) # PATCH
async def update_item(item_id: str, item: Item):
    stored_item_data = items[item_id]
    stored_item_model = Item(**stored_item_data)
    update_data = item.dict(exclude_unset=True) # <- NB
    updated_item = stored_item_model.copy(update=update_data)
    items[item_id] = jsonable_encoder(updated_item)
    return updated_item

```

Теперь будут учитываться только те значения, которые были переданы явно (т.е. значения по умолчанию учитываться не будут).

Еще можно создавать так называемые подприложения. Пример

```

from fastapi import FastAPI

app = FastAPI()

@app.get("/app")
async def read_main():
    return {"message": "Hello World from main app"}

subapi = FastAPI()

@subapi.get("/sub")
async def read_sub():
    return {"message": "Hello World from sub API"}

```

Вызов может выглядеть так

```

http://localhost:8000/app # {"message": "Hello world from main app"}
http://localhost:8000/subapi/sub # {"message": "Hello world from sub API"}

```

Для упрощения процедуры передачи данных в приложение, удобно использовать web-сокеты. Пример

### main.py

```
from fastapi import FastAPI, WebSocket
from fastapi.responses import HTMLResponse

app = FastAPI()

html = """
<!DOCTYPE html>
<html>
<head>
    <title>Main form for ML-app</title>
</head>
<body>
    <h1>Enter your params here</h1>
    <form action="" onsubmit="sendMessage(event)">
        <input type="text" id="messageText" autocomplete="off"/>
        <button>Run</button>
    </form>
    <ul id='messages'>
    </ul>
    <script>
        var ws = new WebSocket("ws://localhost:8000/ws");
        ws.onmessage = function(event) {
            var messages = document.getElementById('messages')
            var message = document.createElement('li')
            var content = document.createTextNode(event.data)
            message.appendChild(content)
            messages.appendChild(message)
        };
        function sendMessage(event) {
            var input = document.getElementById("messageText")
            ws.send(input.value)
            input.value = ''
            event.preventDefault()
        }
    </script>
</body>
</html>
"""

@app.get("/")
async def get():
    return HTMLResponse(html)

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()

    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f "Message text was: {data}")


Запускаем как обычно
```

```
uvicorn main:app --reload
```

Если все прошло как полагается, то по маршруту `http://localhost:8000` должна быть доступна форма-интерфейс.

Если требуется, чтобы какие-то операции были выполнены перед запуском приложения, то это можно следующим образом

```
from fastapi import FastAPI

app = FastAPI()

items = {}

@app.on_event("startup")
async def startup_event():
    items["foo"] = {"name": "Fighters"}
    items["bar"] = {"name": "Tenders"}

@app.get("/items/{item_id}")
async def read_items(item_id: str):
    return items[item_id]
```

То есть в данном случае, перед запуском приложения будет создан словарь `items`.

Аналогично можно создать функцию, которая будет вызываться по завершении работы приложения

```
from fastapi import FastAPI

app = FastAPI()

@app.on_event("shutdown")
def shutdown_event():
    with open("log.txt", "a") as log:
        log.write("Application shutdown")

@app.get("/items/")
async def read_items():
    return [{"name": "Foo"}]
```

Иногда бывает нужно использовать в сессии пользовательские переменные окружения. Сделать это можно так

```
# ADMIN_EMAIL="leor.finkelberg@yandex.ru" APP_NAME="CAD.ai" uvicorn main:app
from fastapi import FastAPI
from pydantic import BaseSettings

class Settings(BaseSettings):
    app_name: str = "Awesome API"
    admin_email: str
    items_per_user: int = 50

settings = Settings()
app = FastAPI()

@app.get("/info")
async def info():
    return {
        "app_name": settings.app_name,
        "admin_email": settings.admin_email,
        "items_per_user": settings.items_per_user,
    } # {"app_name": "CAD.ai", "admin_email": "leor.finkelberg@yandex.ru", "items_per_user": 50}
```

Тогда запуск приложения будет выглядеть следующим образом

```
ADMIN_EMAIL="leor.finkelberg@yandex.ru" APP_NAME="CAD.ai" uvicorn main:app
```

Вот еще один любопытный способ использовать информацию, представленную в конфигурационном файле. Этот способ может быть полезен в ситуациях, когда только часть параметров изменяется в зависимости от окружения.

Создадим скрытый файл .env

```
.env  
# здесь используются только те параметры, которые изменяются в зависимости от окружения  
ADMIN_EMAIL="leor.finkelberg@yandex.ru"  
APP_NAME="CAD.ai"
```

Затем создадим модуль, который будет вычитывать конфигурацию

config.py

```
from pydantic import BaseSettings  
  
class Settings(BaseSettings):  
    # это базовые параметры  
    app_name: str = "Awesome API"  
    admin_email: str  
    items_per_user: int = 50  
  
    class Config:  
        env_file = ".env" # здесь используются только те параметры, которые изменяются в зависимости от окружения
```

А затем собственно модуль, который будет использовать данные из конфигурационного файла

main.py

```
from functools import lru_cache  
from fastapi import Depends, FastAPI  
from config import Settings  
  
app = FastAPI()  
  
@lru_cache()  
def get_settings():  
    return Settings()  
  
@app.get("/info")  
async def info(settings: Settings = Depends(get_settings)):  
    return {  
        "app_name": settings.app_name,  
        "admin_email": settings.admin_email,  
        "items_per_user": settings.items_per_user,  
    } # {"app_name": "CAD.ai", "admin_email": "leor.finkelberg@yandex.ru", "items_per_user": 50}
```

## 21.1. Развёртывание FastAPI-приложений на платформе Deta

Структура проекта для развертывания на платформе Deta <https://www.deta.sh/?ref=fastapi>

```
.  
`-- main.py  
`-- requirements.txt
```

Модуль `main.py` имеет следующее содержание

`main.py`

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

а файл с зависимостями такое содержание

`fastapi`

Устанавливать `uvicorn` для развертывания на Deta не нужно!!!

Теперь нужно создать бесплатный аккаунт на Deta. Устанавливаем утилиту командной строки `deta`

```
curl -fsSL https://get.deta.dev/cli.sh | sh
```

После чего нужно залогиниться

`deta login`

и, находясь в корне проекта, запустить процедуру развертывания

```
deta new
-----
{
  "name": "fastapideta",
  "runtime": "python3.7",
  "endpoint": "https://qltnci.deta.dev", # <---- NB
  "visor": "enabled",
  "http_auth": "enabled"
}
```

Для того чтобы убедиться в работоспособности приложения нужно вставить в браузер URL, созданный платформой Deta (в данном случае это `https://qltnci.deta.dev`).

Чтобы можно было делиться этой ссылкой следует выполнить «публичную аутентификацию»

`deta auth disable`

## 21.2. Развёртывание FastAPI-приложения в ручную

Для того чтобы развернуть FastAPI-приложение в ручную нужно просто установить ASGI-совместимый сервер

```
pip install uvicorn[standard]
```

И запустить приложение, как обычно, но без опции `--reload`

```
uvicorn main:app --host 0.0.0.0 --port 80
```

## 22. Приемы разработки приложений с графическим интерфейсом пользователя с использованием библиотеки DearPyGui

## 23. Использование Google Drive как хранилище артифактов ML-пайплайнов

Чтобы работать с файлом, размещенным на Google Drive, нужно просто выбрать интересующий файл, открыть публичный доступ к нему Открыть доступ → Разрешить доступ всем, у кого есть ссылка и скопировать ссылку

```
import pandas as pd
import requests

url = "https://drive.google.com/file/d/1-7TLGI-6zHxM3ASkCQj6yMLg6mEu5q9n/view?usp=sharing"
download_path = url.replace("view?usp=sharing", "uc?export=download&id={}>".format(url.split('/')[-2]))
# 'https://drive.google.com/uc?export=download&id=1-7TLGI-6zHxM3ASkCQj6yMLg6mEu5q9n'

df = pd.read_csv(download_path)
df.head()
```

## 24. 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)
```

## 25. Упаковка ML-пайплайна в docker-образ

В целом порядок работы выглядит следующим образом:

- пишем ML-пайплайн, который, скажем, обучает модель случайного леса для задачи регрессии,
- тестируем этот ML-пайплайн, смотрим на метрики и пр.,
- когда нам кажется, что пайплайн работает как полагается,
  - либо сохраняем обученную модель (модель обучается не в контейнере!) в формате .pkl в облачном хранилище (например, в S3),
  - либо (если по каким-то соображениям требуется, чтобы модель обучалась в контейнере) собираем проект с ML-пайплайном в пакет с помощью `python setup.py sdist bdist_wheel` и публикуем его на PyPI с помощью `twine upload dist/*` (в директории `dist` будет лежать и tar-архив, и whl-архив); затем этот пакет можно будет установить с помощью утилиты `pip` внутри docker-образа; к слову, также можно собрать этот пайплайн и без публикации на PyPI: т.е. можно собрать ML-пайплайн в пакет с помощью `python setup.py sdist`, а затем установить пакет инструкцией `pip install dist/package_name.tar.gz` внутри контейнера (но так лучше не делать!!!),
  - с помощью инструкции `pip install package_name` или в более простом, но менее правильном случае, с помощью инструкции `pip install dist/package_name.tar.gz` устанавливаем наш пакет в окружение docker-образа,
  - получаем доступ к пакету: получить доступ к нашему пакету можно следующим образом
  - в файле `setup.py`, который используется при сборке пакета можно создать точку входа со специальным ключевым словом `console_scripts`

```

      setup.py
1  from setuptools import find_packages, setup
2
3  setup(
4      name="ml_example",
5      packages=find_packages(),
6      version="0.1.0",
7      description="Example of ml project",
8      author="Your name (or your organization/company/team)",
9      entry_points={
10          "console_scripts": [ # <-- NB
11              "ml_example_train = ml_example.train_pipeline:train_pipeline_command"
12              # в Dockerfile должны быть строки
13              #     ENTRYPOINT ["ml_example_train"]
14              #     CMD ["confis/train_config.yaml"]
15          ]
16      },
17      install_requires=[
18          "click",
19          "Sphinx",
20          ...
21          "marshmallow-dataclass==8.3.0",
22          "pandas",
23      ],
24      license="MIT",
25  )

```

Конструкцию на строке 11 (с оговорками) допустимо интерпретировать так: при запуске docker-контейнера (на базе образа Python), у которого в `Dockerfile` должна быть

прописана инструкция ENTRYPPOINT вида `ENTRYPPOINT ["ml_exmaple_train"]` и инструкция CMD `["configs/train_config.yaml"]`, содержащая аргументы для ENTRYPPOINT, объект `ml_example_train` будет преобразован в утилиту, которая строку

`ml_example.train_pipeline:train_pipeline_command` превратит в  
`from ml_example.train_pipeline import train_pipeline_command` и запустит функцию `train_pipeline_command`, передав ей аргументы из CMD. В данном случае передача аргументов в функцию таким образом возможна, так как предполагается, что функция организована так

`train_pipeline.py`

```
...
@click.command(name="train_pipeline")
@click.argument("config_path") # <- принимает значение, переданное как аргумент командной строки или переданное через инструкцию CMD файла Dockerfile
def train_pipeline_command(config_path: str):
    params = read_training_pipeline_params(config_path)
    train_pipeline(params)
...
```

А сам Dockerfile может выглядеть так

```
FROM python:3.6 # <-- NB
RUN mkdir -p /build
...
ENTRYPOINT ["ml_example_train"]
CMD ["configs/train_config.yaml"]
```

- Подгружаем данные в docker-образ, например, из облачного хранилища,
- запускаем python-сценарий, который читает все, что нужно для его работы – предобученную модель, данные, конфигурационные файлы и пр. – ждем запроса от пользователя и возвращаем прогноз.

ВАЖНО: на этапе проверки работы приложения стартовый сценарий, должен располагаться в корне проекта и использовать абсолютные пути (от корня проекта) до интересующих модулей и пакетов. При сборке пакета для публикации на PyPI PE нужно изменять пути (они как раньше должны включать все узлы цепочки импорта от корня проекта). Например, если директория, на основании которой мы собираемся сделать пакет для публикации на PyPI, называется `ml_example` и лежит в корне проекта, то лежащие внутри нее модули, естественно, должны включать ее имя, например, `from ml_example.data import read_data`. После сборки проекта в пакет эти абсолютные пути по-прежнему будут иметь смысл, так как в этот поиск нужных модулей будет идти относительно имени опубликованного пакета.

## 26. Приемы работы с dataclass

Для того чтобы превратить обычный python-класс в data-класс, который служит в числе прочего для валидации схемы конфигурационного файла<sup>11</sup>, используемого в качестве интерфейса к python-приложению, нужно просто задекорировать соответствующий класс декоратором `dataclass`

<sup>11</sup>Или, проще говоря, для проверки типов значений, указанных в конфигурационном файле

ml\_example/entities/train\_pipelines\_params.py

```
import yaml
from dataclasses import dataclass, field
from marshmallow_dataclass import class_schema

# это модули, в которых содержится описание классов, которые
# будут использоваться как подтипы в главном дата-классе
from .feature_params import FeatureParams
from .split_params import SplittingParams
...

@dataclass()
class TrainingPipelineParams:
    # ожидается, что параметры с приведенными ниже именами,
    # встречаются в конфигурационном файле
    input_data_path: str
    output_model_path: str
    metric_path: str
    feature_params: FeatureParams      # класс из ml_example/entities/feature_params.py
    splitting_params: SplittingParams  # класс из ml_example/entities/split_params.py
    ...

TrainingPipelineParamsSchema = class_schema(TrainingPipelineParams)

def read_training_pipeline_params(
    path: str
) -> TrainingPipelineParams: # класс как тип
    with open(path, "r") as input_schema: # input_schema -- файловый объект
        schema = TrainingPipelineParamsSchema()
    return schema.load(yaml.safe_load(input_schema))
```

Типичный класс, который используется как подтип в главном дата-классе, имеет следующий вид

ml\_example/entities/feature\_params.py

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

@dataclass()
class FeatureParams:
    categorical_features: List[str],
    numerical_features: List[str],
    features_to_drop: List[str],
    target_col: Optional[str],
    use_log_trick: bool = field(default = True) # этого параметры нет в конфигурационном файле,
    но ни что не мешает его объявить в дата-классе
```

Этот подкласс отвечает следующему фрагменту конфигурационного файла

train\_config.yaml

```
...
feature_params:
    categorical_features:
        - "MSZoning"
        - "Neighborhood"
        ...
    numerical_features:
        - "OverallQual"
        - "MSSubClass"
```

```
...  
features_to_drop:  
    - "YrSold"  
target_col: "SalePrice"
```

## 27. 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 Hypotheses).

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 `AssertionError`.

### 27.1. 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 поломать модель, тесты должны это показать
- Тесты собственно данных: имена колонок, границы, дубли и пр.

## 27.2. 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__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` функция `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 функции 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 соответствующая функция
```

### 27.3. 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

```
import 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")
```

### 27.4. 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 аргумента, 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.

## 27.5. 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 --maxfail=3
# запустить сеанс отладки
pytest -v --lf -x --pdb
```

В сессии `pdb` можно использовать следующие команды<sup>12</sup>

- `r 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`.

## 27.6. 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.

<sup>12</sup>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-фиксюра-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 затраты 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.

## 27.7. 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`

## 27.8. 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`, 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): # заменяйте функцию os.unlink в нижеприведенной строке на fake_os_unlink
    os.unlink("foobar") # то есть будем 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("Something 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`, которая 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.

## 27.9. 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 строки кода, которые не были выполнены, а значит, не тестились. 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. 13)

```
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()
53 |

```

Рис. 13. Страница html-отчета о покрытии кода тестами

Из рис. 13 можно заключить, что:

- о функция `list_tasks` не тестируется для случая, когда задано имя владельца (эта строка отмечена красным),
- о 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 некоторые 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.

## 27.10. 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 = -rsX -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`, которая 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`. Страна `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 `rpypy`.

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 переопределели 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}
```

## 28. 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 мап,
- Подходит для тестирования кода, который работает с пользователем напрямую.

Можно тестировать модели.

## 29. 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 поддерживаются 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](https://black.readthedocs.io/en/stable/installation_and_usage.html).

Инструмент `black` поддерживается vim [https://black.readthedocs.io/en/stable/editor\\_integration.html#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 «`FileNotFoundException: [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/activate # для Linux/MacOs etc
$ env\Scripts\activate.bat # для Windows
```

## 30. 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 обратиться 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

---

## 31. 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

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

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)
```

## 32. Автоматический анализ кода с платформой DeepSource

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.flit]`),
- `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"
]
```

- o `transformers`: список трансформеров

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

### 32.1. 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
```

## 33. Python и L<sup>A</sup>T<sub>E</sub>X

Для компиляции L<sup>A</sup>T<sub>E</sub>X-документов прямо из-под Python можно использовать библиотеку `pylateX`<sup>13</sup> <https://jeltef.github.io/PyLaTeX/current/>.

---

<sup>13</sup>Устанавливается как обычно `pip install pylatex`

## 34. Адаптивный бустинг

### 34.1. Адаптивный бустинг широкими мазками

Бустинг является одной из самых мощных идей машинного обучения. Первоначально он был разработан для задач *классификации*, но, его можно распространить и на *регрессию*. Цель бустинга – создание процедуры, которая объединяет результаты многих слабых классификаторов для создания мощного комитета.

Проблема переобучения в случае адаптивного бустинга выражена не так явно, как, например, в случае простых деревьев принятия решений, так как для того чтобы модель могла переобучиться у нее должно быть достаточное число степеней свободы (т.е. модель должна быть достаточно сложной), а мета-классификатор в адаптивном бустинге строится, как правило, на очень простых моделях, которые не могут «запомнить» всех особенностей данных.

Слабым классификатором является тот, уровень ошибок которого ненамного лучше случайного угадывания. Цель бустинга состоит в том, чтобы последовательно применять алгоритм слабой классификации к многократно измененным версиям данных, тем самым создавая последовательность слабых классификаторов  $G_m(x)$ ,  $m = 1, \dots, M$ . Прогнозы, полученные от всех этих классификаторов, затем объединяются, и окончательное решение принимается по взвешенному большинству голосов

$$G(x) = \text{sign}\left(\sum_{m=1}^M \alpha_m G_m(x)\right)$$

Здесь коэффициенты  $\alpha_1, \alpha_2, \dots, \alpha_M$  рассчитываются с помощью алгоритма бустинга и взвешивают вклад каждого соответствующего классификатора  $G_m(x)$ . Их эффект состоит в том, чтобы придать больший вес более точным классификаторам, входящим в последовательность.

Модификации данных на каждом этапе бустинга состоят из применения весов  $w_1, w_2, \dots, w_N$  к каждому обучающему наблюдению  $(x_i, y_i)$ ,  $i = 1, 2, \dots, N$ . Первоначально все веса установлены равными  $w_i = 1/N$ , так что первый шаг обучает классификатор на данных обычным способом. Для каждой последующей итерации  $m = 2, 3, \dots, M$  веса наблюдений модифицируются индивидуально, и к взвешенным наблюдениям применяется алгоритм классификации. На этапе  $m$  те наблюдения, которые были неправильно классифицированы классификатором  $G_{m-1}(x)$ , созданном на предыдущем этапе, получают повышенные веса, тогда как веса тех наблюдений, которые были классифицированы правильно, уменьшаются.

Таким образом, по мере продолжения итераций наблюдения, которые *трудно* правильно классифицировать, получают все большее влияние. Следовательно, каждый последующий классификатор вынужден концентрироваться на тех *обучающих* наблюдениях, которые пропущены предыдущими классификаторами, входящими в последовательность.

Алгоритм адаптивного бустинга

1. Инициализируем веса экземпляров обучающего набора данных  $w_i = 1/N$ ,  $i = 1, 2, \dots, N$ ,
2. Для  $m = 1$  до  $M$ 
  - (a) Настраиваем классификатор  $G_m(x)$  на обучающих данных, используя веса  $w_i$ ,

(b) Вычисляем

$$err_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}$$

(c) Вычисляем вес классификатора-примитива

$$\alpha_m = \frac{1}{2} \log \frac{1 - err_m}{err_m},$$

(d) Обновляем

$$w_i \leftarrow w_i \cdot \exp\{\alpha_m \cdot I(y_i \neq G_m(x_i))\}, \quad i = 1, 2, \dots, N$$

3. Выводим результат

$$G(x) = \text{sign}\left[\sum_{m=1}^M \alpha_m G_m(x)\right].$$

Здесь  $i$  – индекс экземпляра обучающего набора данных;  $m$  – индекс классификатора.

**ВАЖНО:** одноуровневые деревья решений (пеньки решений) в случае *взвешенной выборки* строятся как обычно, с отбором признаков и пороговых значений на основании метрик качества разбиения родительского узла (например, на основании информационного прироста, вычисленного по загрязненности Джини или энтропии Шеннона), но вместо вычисления отношения числа экземпляров  $k$ -ого класса к общему числу экземпляров, попавших в рассматриваемый лист, оперировать приходится весами этих экземпляров.

### 34.2. Обучение на взвешенной выборке

Отличная лекция Разинкова [https://www.youtube.com/watch?v=jjwH1qn\\_C0k](https://www.youtube.com/watch?v=jjwH1qn_C0k).

Целевая функция для *регрессии* на взвешенной выборке вычисляется как

$$E(\theta) = \frac{1}{2} \sum_{i=1}^N w_i (t_i - y(\mathbf{x}_i, \theta))^2,$$

а для классификации – как

$$E(\theta) = - \sum_{i=1}^N w_i \sum_{k=1}^K t_i^{(k)} \ln y(\mathbf{x}_i, \theta).$$

Информационный прирост на деревьях решений по *взвешенной выборке* определяется как

$$I_w = H_w(S_i) - \sum_i \frac{\tilde{N}_{ij}}{\tilde{N}_i} H_w(S_{ij}), \quad H_w(S_i) = - \sum_k \frac{\tilde{N}_i^{(k)}}{\tilde{N}_i} \log \frac{\tilde{N}_i^{(k)}}{\tilde{N}_i},$$

здесь  $i$  – индекс родительского узла;  $j$  – индекс дочернего узла;  $k$  – индекс класса;  $\tilde{N}_i = \sum_{x_l \in S_i} w_l$  – это не количество экземпляров, попавших в рассматриваемый узел, а *сумма их весов*.

### 34.3. Основные концепции адаптивного бустинга. Разбор видео-лекции Josh Starmer

Адаптивный бустинг чувствителен к шуму и выбросам в данных. AdaBoost вызывает слабые классификаторы в цикле  $t = 1, \dots, T$ . После каждого вызова обновляется распределение весов  $D_t$ , которые отвечают важности каждого экземпляра обучающего набора данных. На каждой итерации веса каждого неверно классифицированного объекта, возрастают, таким образом новый комитет классификаторов «фокусирует» свое внимание на этих объектах.

В случайном лесе, как правило, мы строим деревья в полную глубину – некоторые деревья могут более глубокими чем прочие деревья в ансамбле, другие менее глубокие, но ни одно из этих деревьев не имеет явного ограничения на глубину.

В адаптивном бустинге, напротив, деревья, как правило, имеют один узел и два листа (такие деревья называют *пнями*). Пень это далеко не самый эффективный ученик.

В случайном лесе на задачах классификации все деревья имеют *равные* голоса. Напротив, в лесу пней, построенном с помощью адаптивного бустинга, голос некоторых пней может быть весомее голосов других пней.

В случайном лесе каждое дерево решения строится независимо от других. Напротив, в лесу пней, построенном с помощью адаптивного бустинга, порядок построения пней имеет значение. То есть ошибки первого пня влияют на то, как будет строится второй пень и т.д.

Случайный лес просто усредняет прогнозы деревьев решения (с равными весами) в ансамбле, а адаптивный бустинг вычисляет прогноз с помощью мета-классификатора, который представляет собой взвешенную сумму классификаторов-примитивов.

Три ключевые идеи, лежащие в основе адаптивного бустинга

- адаптивный бустинг состоит из «слабых учеников» (weak learners), которые почти всегда представляют собой пни,
- отдельные пни могут иметь более весомые голоса, чем другие пни ансамбля,
- каждый последующий пень учитывает ошибки предыдущих пней.

Построение леса пней начинают с того, что задают веса экземплярам обучающего набора данных. На первой итерации всем экземплярам обучающего набора назначается один и тот же вес, который вычисляется как  $1/n$ , где  $n$  – число экземпляров обучающего набора данных. То есть все образцы обучающего набора имеют одинаковую важность. Однако, после построения первого пня эти веса изменяются.

В качестве первого пня в лесу выбирается тот, которому отвечает меньшее значение индекса Джини, который вычисляется так (ЕСТЬ ОПАСЕНИЯ, ЧТО ЭТУ ФОРМУЛУ В ТАКОМ ВИДЕ ИСПОЛЬЗОВАТЬ НЕЛЬЗЯ)

$$IndexGini = \frac{N_{left}}{N} I_{left} + \frac{N_{right}}{N} I_{right}, \quad I = 1 - \sum_{i=1}^m p_i^2,$$

здесь  $p_i$  – доля экземпляров  $i$ -ого класса,  $m$  – число классов.

Обычно для того чтобы количественно оценить эффективность разбиения узла используют информационный прирост

$$IG(D_p) = I(D_p) - \sum_{k=1}^m \frac{N_k}{N} I(D_k),$$

где  $D_{p,k}$  – набор данных родительского узла и  $k$ -ого дочернего узла соответственно;  $N_p$  – общее количество экземпляров в родительском узле;  $N_k$  – число экземпляров, попавших в  $k$ -ый узел;  $m$  – число дочерних узлов (в случае бинарных деревьев  $m = 2$ );  $I$  – мера загрязненности.

Информационный прирост оценивает насколько стали чище дочерние узлы после разбиения. Чем информационный прирост больше, тем лучше.

В качестве  $I(D_{p,k})$  обычно используется либо загрязненности Джини, либо энтропия Шеннона. Таким образом, выходит, что формула для расчета индекса Джини вытекает из только что рассмотренной формулы при нулевом информационном приросте. **Этот результат следует использовать очень осторожно.**

Веса экземпляров, на которых модель адаптивного бустинга ошиблась, вычисляются следующим образом

$$NewSampleWeight = sampleWeight \cdot e^{amountOfSay}, \quad amountOfSay = \frac{1}{2} \log \frac{1 - TotalError}{TotalError},$$

где  $TotalError$  – сумма весов неправильно классифицированных экземпляров.

Например,  $NewSampleWeight = \frac{1}{8} e^{0.97} = 0.33$  (было  $1/8 = 0.125$ ).

Теперь нужно уменьшить веса правильно классифицированных экземпляров.

Веса экземпляров, которые модель адаптивного бустинга классифицировала правильно, вычисляются так

$$NewSampleWeight = sampleWeight \cdot e^{-amountOfSay}.$$

Например, для правильно классифицированных экземпляров новый вес будет

$$NewSampleWeight = \frac{1}{8} e^{-0.97} = 0.05 \text{ (было } 1/8 = 0.125\text{).}$$

**ВАЖНО:** в адаптивном бустинге большие веса назначаются тем экземплярам, на которых модель ошибалась и соответственно меньше веса – тем экземплярам, которые модель классифицировала верно.

Теперь нужно нормализовать новые веса таким образом, чтобы они складывались в единицу. То есть нужно разделить вес каждого образца на сумму весов всех образцов.

Сейчас мы можем использовать модифицированные веса, чтобы построить второй пень в ансамбле.

#### 34.4. Пример работы алгоритма адаптивного бустинга

Рассмотрим решение задачи бинарной классификации. Постановка:  $C_{-1}, C_1, t_i \in \{-1, 1\}, i = (1, \dots, N)$ . Слабые классификаторы  $y_j(\mathbf{x}) \rightarrow \{-1, 1\}, (j = 1, \dots, M)$ . Требуется построить классификатор вида

$$Y(\mathbf{x}) = \text{sign}\left(\sum_{j=1}^M \alpha_j y_j(\mathbf{x})\right).$$

Без весов  $\alpha_j$  прогноз по рассмотренной формуле будет эквивалентен обычному голосованию в случайному лесу: положительный знак суммы – положительный класс, отрицательный знак суммы – отрицательный класс, так как классификаторы-примитивы возвращают либо 1, либо -1.

Если вес  $j$ -ого классификатора отрицательный, то ответ этого классификатора следует инвертировать.

Обучение комитета классификаторов прекращается по достижении критерия останова (например, по достижении заданного числа классификаторов-примитивов в комитете, минимального прироста метрики качества на валидационном наборе и т.д.).

Веса элементов обучающей выборке на шаге  $j$ :  $w_i^{(j)} \geq 0$ ,  $\sum_{i=1}^N w_i^{(j)} = 1$ . На первой итерации для всех экземпляров обучающего набора веса определяются как  $w_i^{(1)} = 1/N$ ,  $i = 1, \dots, N$ .

**Обучение слабого классификатора** Минимизируемая целевая функция

$$J_j = \sum_{i=1}^N w_i^{(j)} I(y_j(\mathbf{x}_i) \neq t_i), \quad I(z) = \begin{cases} 1, & z = \text{True}, \\ 0, & z = \text{False} \end{cases}.$$

То есть  $J_j$  – это сумма весов экземпляров обучающего набора данных, на которых классификатор-примитив допустил ошибку. Другими словами, мы пытаемся минимизировать сумму весов на ошибочно классифицированных экземплярах обучающего набора.

Ошибка  $j$ -ого классификатора и есть целевая функция на  $j$ -ом шаге обучения

$$\varepsilon_j = \sum_{i=1}^N w_i^{(j)} I(y_j(\mathbf{x}_i) \neq t_i), \quad \varepsilon \in [0; 1].$$

Вес  $j$ -ого классификатора-примитива<sup>14</sup>

$$\alpha_j = \ln \frac{1 - \varepsilon_j}{\varepsilon_j}.$$

Положительные веса – если ошибка, меньше  $1/2$ , отрицательные веса – если больше  $1/2$  (выгоднее инвертировать знак классификатора), нулевые веса – если, ошибка равна  $1/2$ . То есть классификаторы, которые имеют высокую точность, будут иметь большой вес, а те классификаторы, которые имеют низкую точность буду иметь малый вес, и, наконец, те классификаторы, которые занимаются случайным угадыванием получают нулевой вес и никак не влияют на прогноз.

Обновление весов

$$w_i^{(j+1)} = \frac{w_i^{(j)} e^{\alpha_j I(y_j(\mathbf{x}_i) \neq t_i)}}{Z_j}, \quad i = 1, \dots, N,$$

$$Z_j = \sum_{i=1}^N w_i^{(j)} e^{\alpha_j I(y_j(\mathbf{x}_i) \neq t_i)}.$$

Здесь  $Z_j$  это просто сумма всех *новых* весов. Обновляем веса каждого экземпляра обучающего набора, а затем каждый новый вес делим на сумму новых весов. То есть в итоге веса экземпляров, на которых классификатор-примитив ошибается – *увеличиваются*, а на которых делает правильный прогноз – *уменьшаются*. Это делается для того, чтобы следующий классификатор-примитив обращал большее внимание на сложноклассифицированные экземпляры.

---

<sup>14</sup>Иногда в этой формуле встречается коэффициент пропорциональности  $1/2$  – это, так называемая, скорость обучения (в оригинальном адаптивном бустинге она равна 1)

**ВАЖНО:** если в данных есть ошибки, то адаптивный бустинг будет работать очень плохо, так как ему придется на каждой итерации обучения акцентировать внимание на ошибочных экземплярах.

Классификатор, который имеет ошибку  $1/2$ , имеет нулевой вес в комитете ( $\alpha_j = 0$ ). Это значит, что веса всех экземпляров останутся без изменений, не зависимо от того ошибся следующий классификатор-примитив или нет, т.е.

$$w_i^{(j+1)} = \frac{w_i^{(j)} \overbrace{e^{0 \cdot I(y_j(\mathbf{x}_i) \neq t_i)}}^1}{Z_j} = \frac{w_i^{(j)}}{Z_j}$$

А если веса экземпляров не изменились, то это значит, что и на следующий итерации классификатор останется тем же. То есть тоже будет заниматься случайным угадыванием и тоже не изменит веса экземпляров. Поэтому, когда ошибка становится равной  $1/2$ , обучение прекращают (такое обучение не сдвигается с места никогда).

Итоговый классификатор будет иметь вид

$$Y(\mathbf{x}) = \text{sign}\left(\sum_{j=1}^M \alpha_j y_j((\mathbf{x}))\right).$$

Если сумма имеет положительный знак, то прогнозируемый класс считается положительным (класс 1), а если сумма отрицательная, то класс считается отрицательным (минус 1).

**ВАЖНО:** адаптивный бустинг – это жадный алгоритм. Выбрав на какой-то итерации один раз наилучший классификатор-примитив (наилучший для той итерации), AdaBoost не меняет своего мнение относительно этого классификатора (не смотря на то, что он был наилучшим на той итерации, но может не оказаться наилучшим на других итерациях). То есть последовательность построения классификаторов-примитивов в адаптивном бустинге важна.

Адаптивный бустинг работает только на сбалансированной выборке. В случае обучения адаптивного бустинга на *несбалансированной* выборки можно начальные веса экземпляров задать так

$$w_i^{(1)} = \begin{cases} \frac{1}{2N_{-1}} & , N_{-1} + N_1 = 1. \\ \frac{1}{2N_1} & \end{cases}$$

То есть веса экземпляров миноритарного класса получают большие значения, а веса экземпляров мажоритарного класса – меньшие значения.

## 35. Градиентный бустинг

### 35.1. Особенности реализации в пакете XGBoost

Алгоритм экстремального градиентного бустинга XGBoost добавляет больше масштабируемых методов, которые задействуют многопоточность 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 градиентный 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,
- *внедерные 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*<sup>15</sup> пропущенные данные, используя внешкальные значения (в частности, *большое отрицательное число*, например, -999999), чтобы выработать надлежащее ветвление дерева в случае пропущенных значений.

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 (температура) 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,

<sup>15</sup>Импутация – процесс замещения пропущенных, некорректных или несостоятельных значений другими значениями

- `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
```

### 35.1.1. 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. 14), по которым будет проводиться поиск интересующего пакета (в данном случае пакета `xgboost`), а затем можно воспользоваться конструкцией

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

После, выбрав канал, можно приступать к установке пакета

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

### 35.1.2. 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

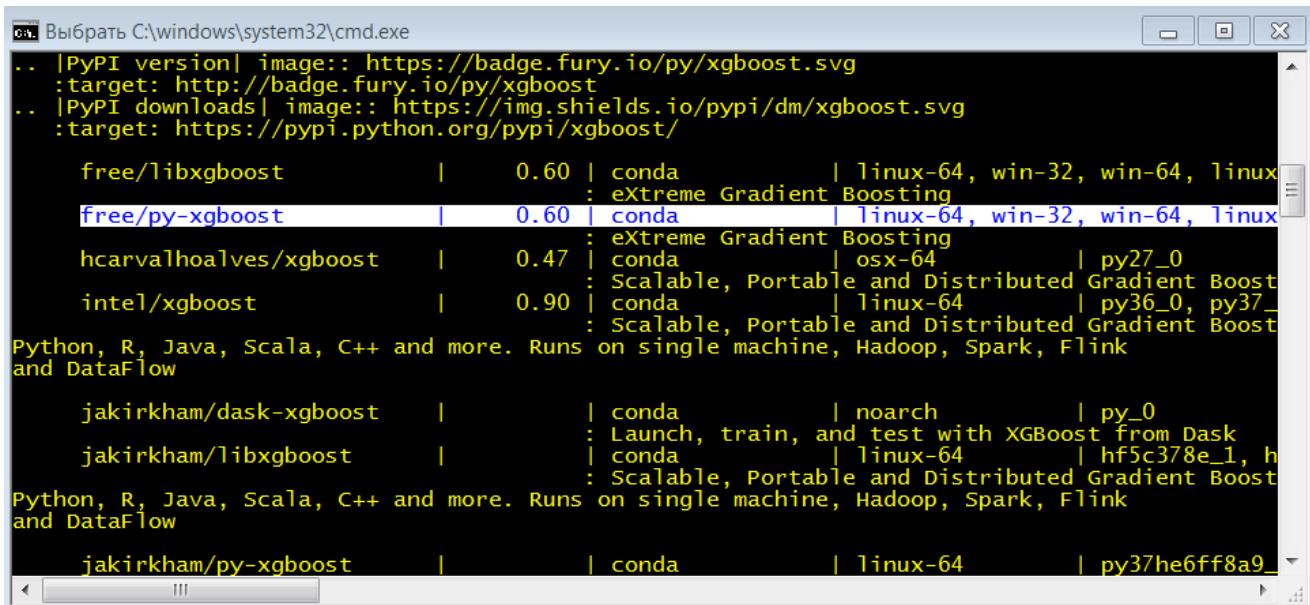


Рис. 14. Окно командной оболочки cmd.exe со списком доступных каналов, по которым будет проводиться поиск пакета xgboost

```

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')], # выводит результат на каждой 100-ой итерации бустинга
                      verbose_eval=100, early_stopping_rounds=20)

xgboost.plot_importance(model)

```

На рис. 15, рис. 16 и рис. 17 изображены графики важности признаков.

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,

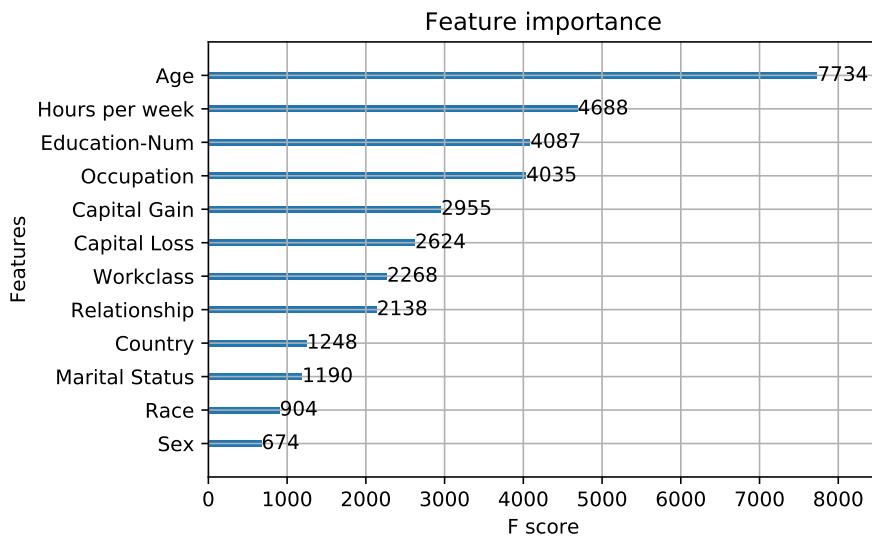


Рис. 15. График важности признаков `xgboost.plot_importance(model)`, построенный с помощью пакета `xgboost`

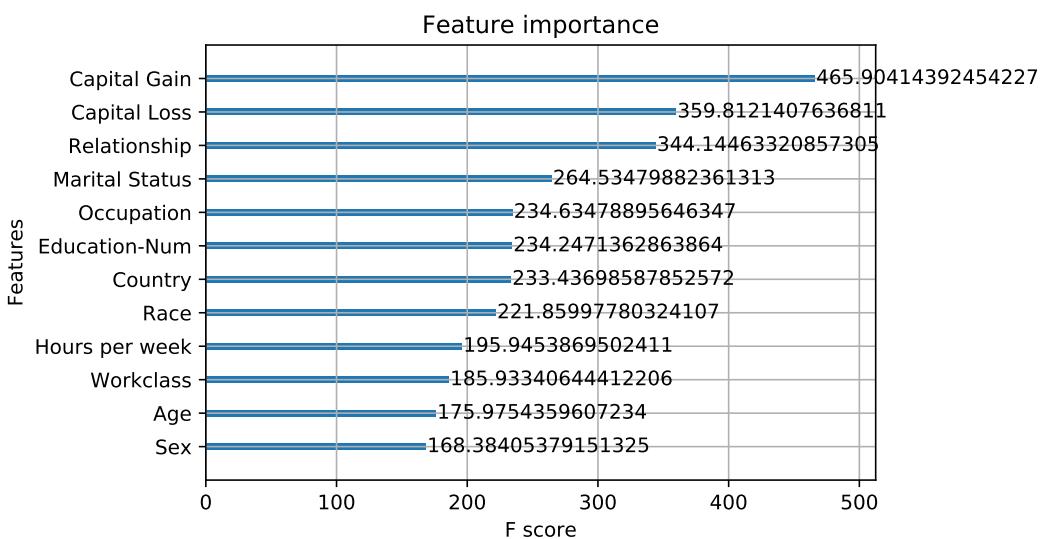


Рис. 16. График важности признаков `xgboost.plot_importance(model, importance_type='cover')`, построенный с помощью пакета `xgboost`

- **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)
```

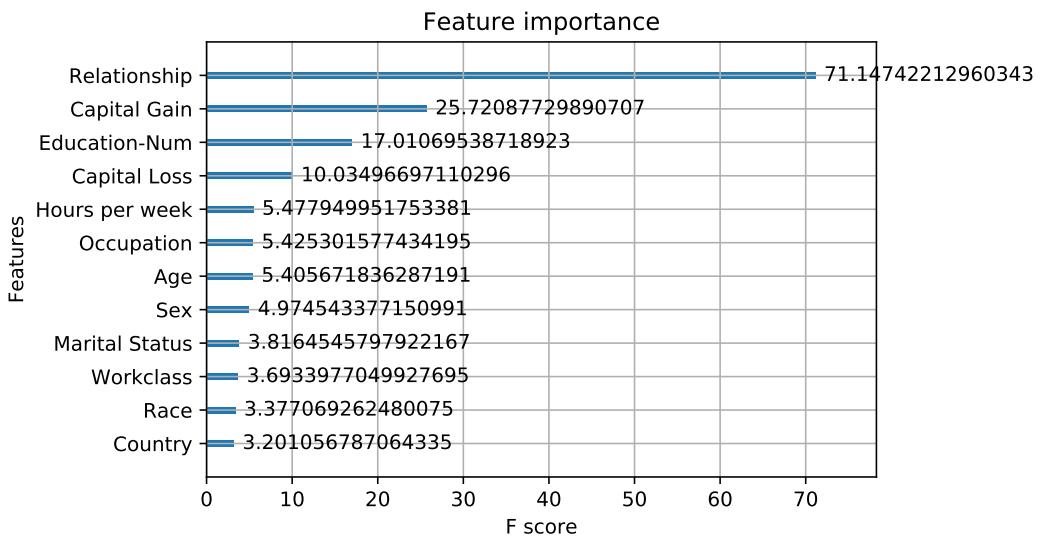


Рис. 17. График важности признаков `xgboost.plot_importance(model, importance_type='gain')`, построенный с помощью пакета `xgboost`

```

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_splits=10, 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)

```

### 35.2. Особенности реализации в пакете LightGBM

### 35.3. Особенности реализации в пакете CatBoost

## 36. Экстремальный градиентный бустинг с XGBoost

### 36.1. Регрессия

Для простоты рассмотрим призывное описание объекта с одним признаком Drug Dosage и целевой переменной Drug Effectiveness (см. рис. 18).

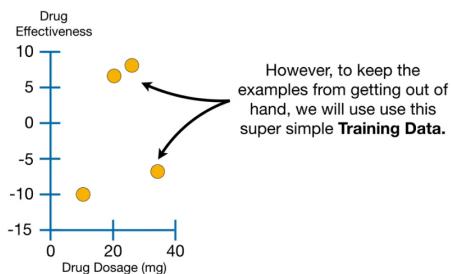


Рис. 18. Простой набор данных

На рис. 18 нижние две точки (большие отрицательные значения) отвечают низкой эффективности препарата, а верхние две точки, напротив, – высокой эффективности.

На самом первом этапе построения дерева принятия решения в XGBoost задается *базовое/начальное значение прогноза* (см. рис. 19), которое по умолчанию принимается равным 0.5 независимо от того, для какой задачи используется XGBoost – для регрессии или классификации.

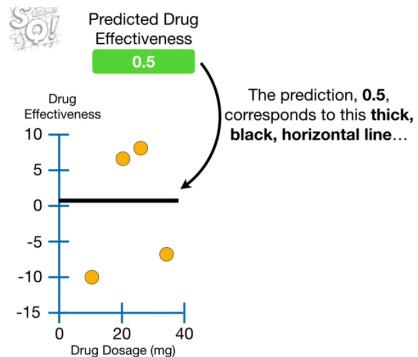


Рис. 19. Базовое/начальное значение прогноза

Далее вычисляются *остатки* (residuals), которые представляют собой разность, составленную из наблюдаемых значений и значения прогноза (рис. 20)

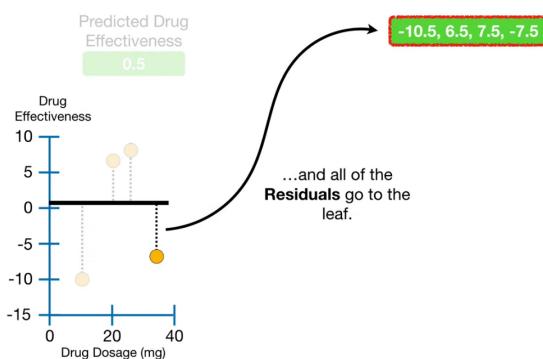


Рис. 20. Остатки, как разность наблюдаемых значений и значения прогноза

**ВАЖНО:** *Экстремальный градиентный бустинг* как и *классический градиентный бустинг* настраивается на *остатки*, но в отличие от классического градиентного бустинга, который использует обычные деревья регрессии, экстремальный градиентный бустинг использует специальные XGBoost-деревья.

Существует множество способов построить XGBoost-дерево. Ниже приводится один из наиболее общих подходов к построению XGBoost-деревьев для задач регрессии.

Каждое дерево начинается с одиночного листа (узла), в которое попадают все остатки, т.е. разность между наблюдаемыми значениями и значением начального прогноза (рис. 20).

Теперь можно вычислить *сходство* (similarity score) на остатках по формуле (см. рис. 21)

$$\text{similarity}_p = \frac{(\sum_k^n r_{k,p})^2}{n + \lambda},$$

где  $r_{k,p}$  – значения *остатка*, по которому вычисляется сходство, в  $p$ -ом родительском или до-чернем узле бинарного дерева,  $n$  – число точек, попавших в поднабор (например, если слева от порога оказалось 3 точки, то  $n = 3$ ),  $\lambda$  – параметр регуляризации (чем этот параметр больше, тем проще подрезать дерево, так как значения прироста получаются меньше).

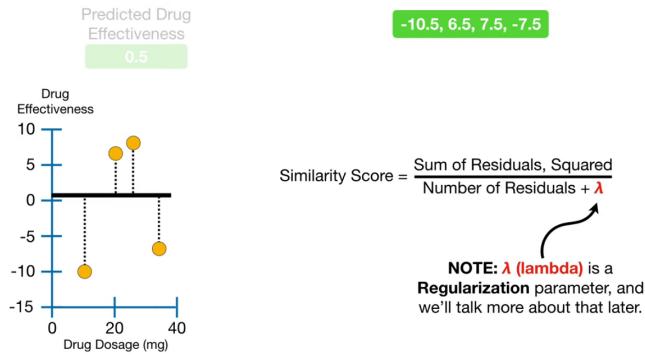


Рис. 21. Вычисление сходства

Например, сходство для родительского (в данном случае корневого) узла дерева при  $\lambda = 0$  составит (рис. 22)

$$\text{similarity} = \frac{(-10.5 + 6.5 + 7.5 + 7.5)^2}{4 + 0} = 4.$$

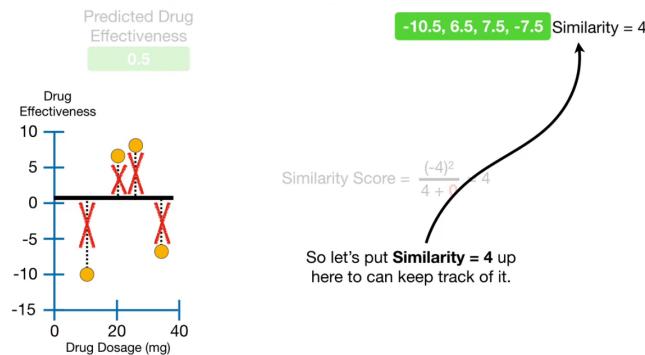


Рис. 22. Вычисление сходства для родительского узла дерева

Теперь надо ответить на вопрос о том сможем мы или нет повысить качество «кластеризации» таких остатков, если их разбить на две группы. Чтобы ответить на этот вопрос, начнем с самых низких значений диапазона изменения признака Drug Dosage. Для первой пары значений признака Drug Dosage вычислим среднее – Drug Dosage = 15 (рис. 23).

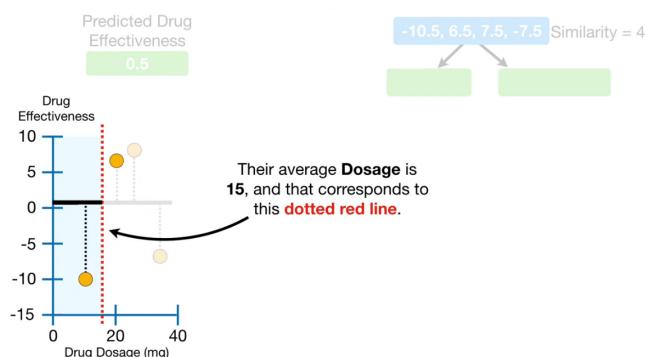


Рис. 23. Первый этап процедуры вычисления порога для признака Drug Dosage

Разобъем значения признака Drug Dosage по условию  $\text{Drug Dosage} < 15$ . В результате получится два листа – в левый попадет только одно значение признака Drug Dosage, а в правый лист – оставшиеся 3 (рис. 24). Можно вычислить сходство для дочерних узлов рис. 25.

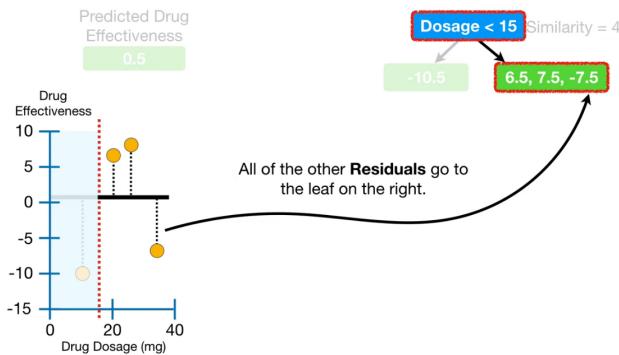


Рис. 24. Первое разбиение значений признака Drug Dosage

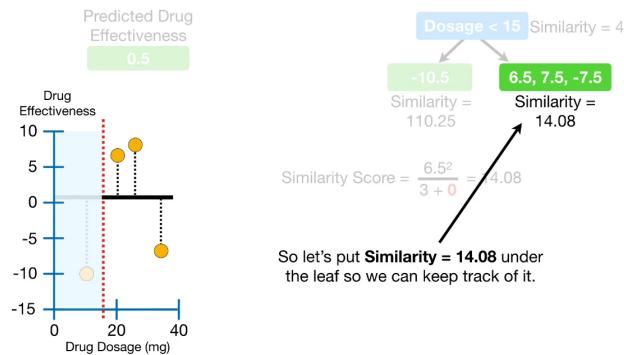


Рис. 25. Вычисление сходства для дочерних узлов

Когда узел состоит из сильно различающихся значений, то сходство для такого узла принимает небольшие значения, напротив, когда узел состоит из значений одного порядка (или только из одного значения), то сходство принимает большие значения.

**ВАЖНО:** мы ищем такое разбиение, при котором в дочерние узлы попадут экземпляры наиболее похожие друг на друга; например, в случае задачи регрессии мы ищем такое разбиение, которое поместит экземпляры с остатками одного порядка в один и тот же дочерний узел

Теперь нужно количественно оценить насколько дочерние узлы лучше группируют остатки, чем родительский узел (рис. 26).

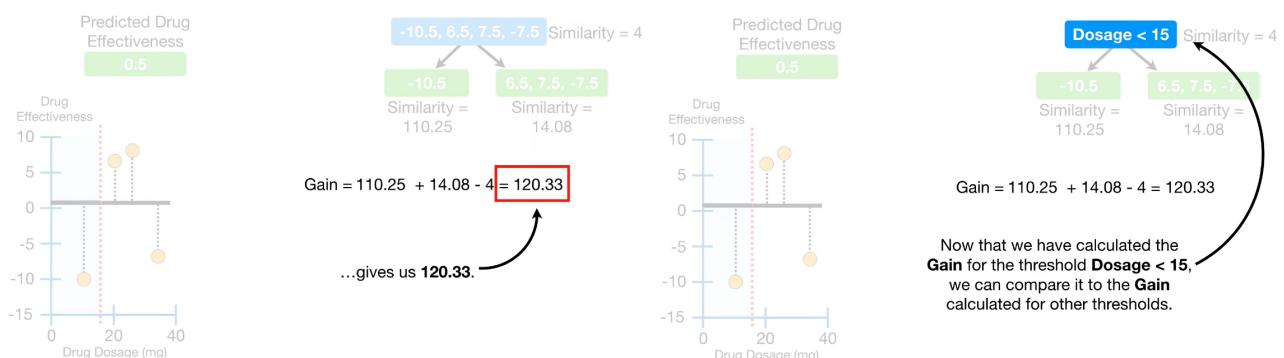


Рис. 26. Вычисление прироста

Сделать это можно, вычислив прирост (gain). Вообще для каждого *условия расщепления t* в дереве принятия решения (в случае задачи регрессии) для оценки качества группировки остатков

в дочерних узлах вычисляется *прирост*

$$\text{gain}_t = \text{similarity}_{left} + \text{similarity}_{right} - \text{similarity}_{root},$$

где  $\text{similarity}_p$  – сходство для родительского узла, левого или правого узлов.

Располагая значением прироста, мы можем сравнить различные стратегии разбиения (по сути мы сравниваем разбиения при различных значениях порога). Смещаем порог вправо, вычисляя среднее значение признака Drug Dosage для следующей пары точек (рис. 27).

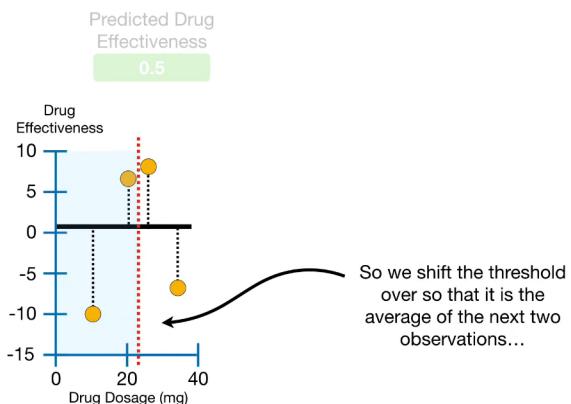


Рис. 27. Следующее разбиение признака Drug Dosage

Как в первый раз, строим дерево решения (рис. 28), но с новым значением порога (Drug Dosage = 22.5).

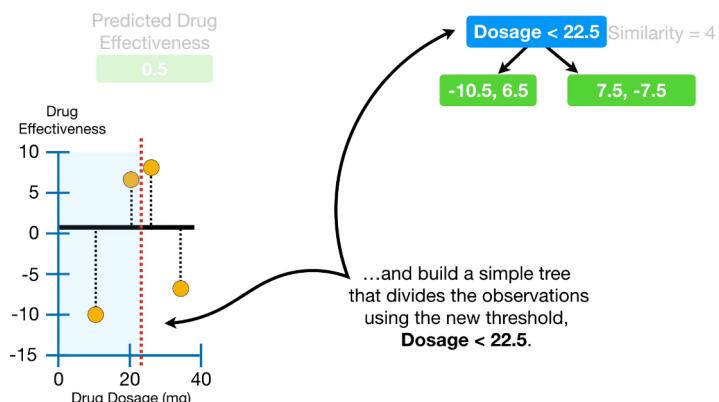


Рис. 28. Дерево решения для нового порога разбиения признака Drug Dosage

Снова вычисляем сходства для родительского и дочерних узлов, а затем прирост. Пусть на текущем этапе прирост составил 4, а не предыдущем шаге – 120.3. Это означает, что разбиение по условию  $\text{Drug Dosage} < 15$  ( $\text{gain} = 120.3$ ) лучше кластеризует остатки, чем условие  $\text{Drug Dosage} < 22.5$  ( $\text{gain} = 4$ ).

Снова смещаем порог вправо, снова вычисляем сходства для родительского и дочерних узлов при разбиении по условию  $\text{Drug Dosage} < 30$ , снова находим прирост. Пусть в данном случае прирост составил 56.33. Значит, что как и раньше наилучшим разбиением является разбиение по условию  $\text{Drug Dosage} < 15$  ( $\text{gain} = 120.3$ ).

Других пар точек у признака Drug Dosage нет, поэтому мы останавливаем процедуру выбора порога, а в качестве условия разбиения возьмем то, которому отвечает наибольшее значение *прироста* (рис. 29), т.е.  $\text{Drug Dosage} < 15$  ( $\text{gain} = 120.3$ ).

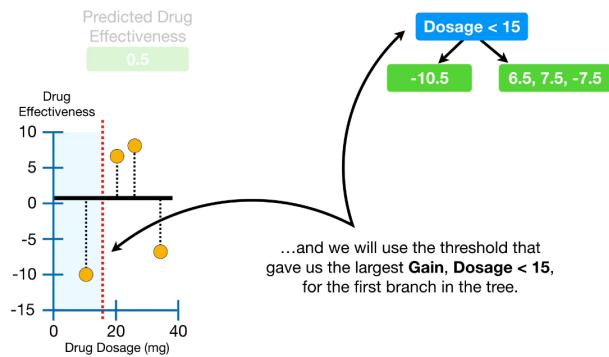


Рис. 29. Наилучшее разбиение признака Drug Dosage по значению прироста

В полученном разбиении в левом узле находится только один элемент и потому этот узел мы не можем разбивать дальше, но для правого узла можно поискать разбиение остатков (рис. 30).

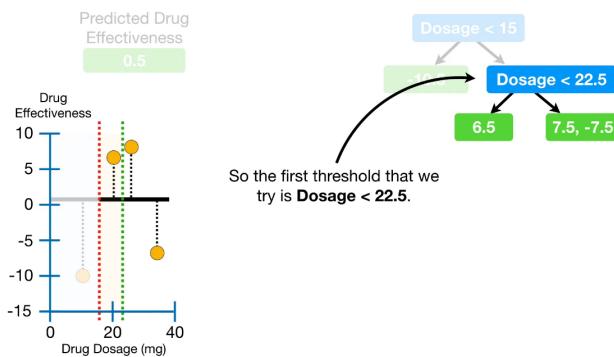


Рис. 30. Начало процедуры поиска следующего лучшего разбиения для признака Drug Dosage

Теперь, как раньше, вычисляем для разбиения по условию  $\text{Drug Dosage} < 22.5$  сходство для дочерних узлов (сходство для родительского узла мы выясняли выше, когда вычисляли сходства для разбиения по условию  $\text{Drug Dosage} < 15$  – сходство составило 14.08). И вычисляем прирост. Он составил в данном случае 28.17 (рис. 31)

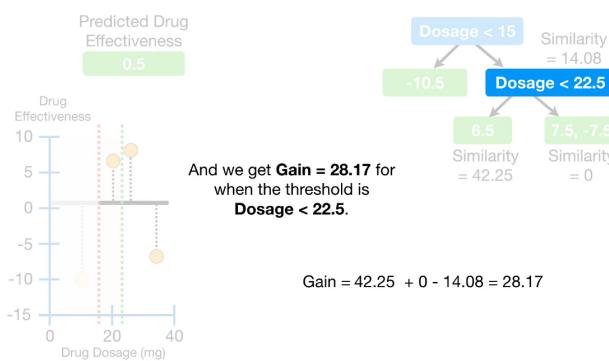


Рис. 31. Вычисление прироста для второго разбиения признака Drug Dosage

Смещаем порог вправо и повторяем процедуру с вычислением сходства и прироста. Пусть прирост для разбиения по условию  $\text{Drug Dosage} < 30$  составил 140.17. Будем использовать разбиение  $\text{Drug Dosage} < 30$  (рис. 32), как отвечающее наибольшему приросту ( $gain = 120.3$ ).

Для простоты ограничимся двумя уровнями дерева (по умолчанию дерево строится с 6 уровнями) и будем считать, что дерево решений построено.

Теперь стоит поговорить о *подрезке* (prune). В XGBoost деревья подрезаются на основании значений прироста.

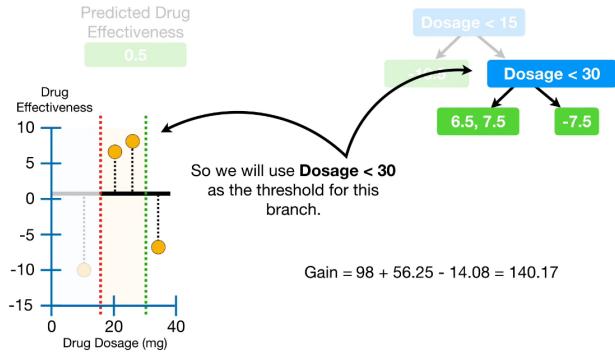


Рис. 32. Построение второго уровня дерева решений. Для простоты ограничиваемся двумя уровнями и не будем строить разбиение для левого узла

Для подрезания дерева принятия решения используется следующая стратегия. Задаются некоторым значением параметра сложности дерева  $\gamma$  (например,  $\gamma = 130$ ), а затем вычисляют  $gain - \gamma$ , начиная с самой нижней ветки дерева. Если разность получается отрицательной, то ветку удаляют и поднимаются на следующий уровень, снова вычисляя  $gain - \gamma$  и т.д. Если разность положительная, то ветку не удаляют. Теоретически так можно дойти и до корня дерева. Если для корня дерева разность получается отрицательной, то корень удаляется и у нас остается только базовое прогнозное значение, которое мы задавали в самом начале процедуры построения дерева (0.5) и которое является довольно экстремальной подрезкой.

**ВАЖНО:** нулевое значение параметра  $\gamma$  не отменяет подрезки дерева! Положительные значения  $\lambda$  снижают эффект переобучения, так как параметр  $\lambda$  – параметр регуляризации, а регуляризация нужно для упрощения модели.

Выходное значение  $m$ -ого листа (output value) рассчитывается как

$$(\text{output value})_m = \frac{\sum_k^n r_{k,m}}{n + \lambda}$$

Очевидно, что при  $\lambda = 0$  выходное значение (output value), представляет собой *обычное среднее* по остаткам (рис. 33).

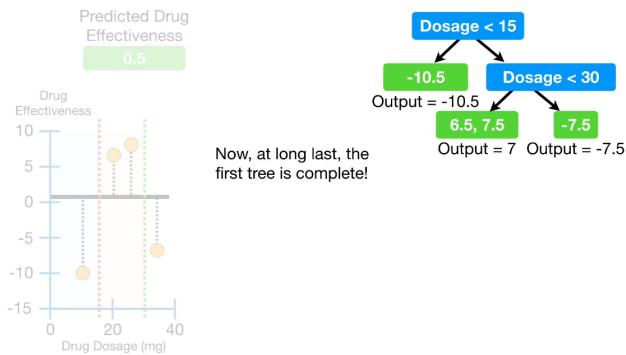


Рис. 33. Выходы по листьям дерева

Вычислить прогноз для конкретной точки на дереве экстремального градиентного бустинга (рис. 34), можно, как и в случае обычного градиентного бустинга, начав с базового прогноза (в данном случае 0.5) и добавив значение соответствующего листа (нужно пройти по всем условиям в дереве, используя указанные признаки и пороговые значения), умноженное на скорость обучения (learning rate)

```
new_residual = 0.5 + learning_rate * output_value_from_leaf
```

Например, для листа с выходом  $-10.5$  (значение признака Drug Dosage=10) прогноз будет  $0.5 + 0.3 \cdot (-10.5) = -2.65$ . Тогда новое значение остатка на этой точке данных будет меньше чем раньше (рис. 35). Аналогично можно вычислить прогноз (рис. 36) для другого значения признака Dosage = 20:  $0.5 + 0.3 \cdot 7 = 2.6$  (было 6.5).

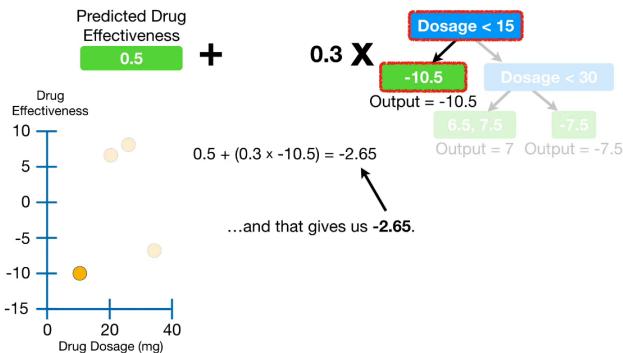


Рис. 34. Процедура построения прогноза в XGBoost для значения признака Dosage=10

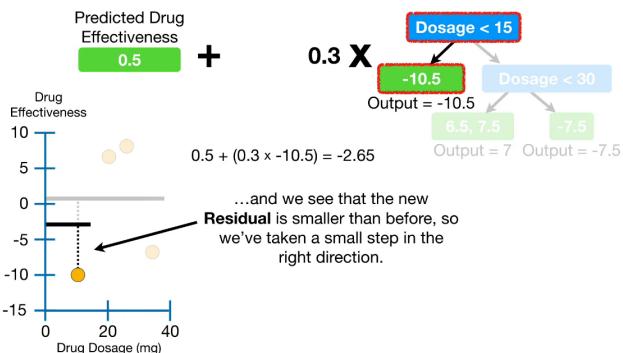


Рис. 35. Остаток на точке Dosage=10 стал меньше

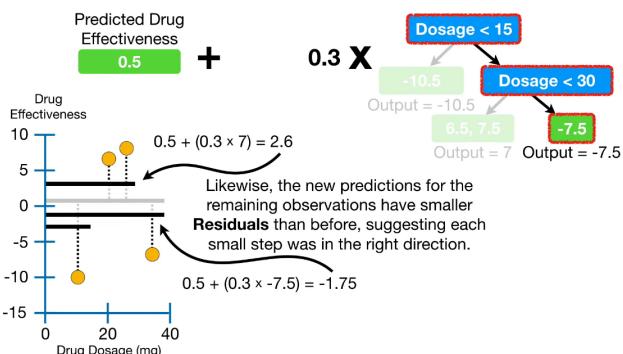


Рис. 36. Остаток на точке Dosage=20 стал меньше

Теперь можно построить другое дерево на *новых остатках* (residuals) и снова вычислить

```
# предсказание -- это по сути новые остатки
prediction (new_residual) = 0.5 + learning_rate * output_value_from_leaf
```

Например, значения целевой переменной, отвечающей значению признака Dosage=10, с учетом новых значений остатков будет

$$-2.65 + 0.3 \cdot (-10.5 + 2.65) = -5.005$$

Мы продолжаем строить деревья до тех пор пока остатки не станут меньше заданного порога или пока мы не достигнем максимального числа итераций.

Порядок построения ансамбля деревьев экстремального градиентного бустинга для задач регрессии:

- вычислить сходство (similarity score) для родительского, левого и правого узлов,
- вычислить прирост (gain) для соответствующего условия, ассоциированного с родительским узлом,
- вычислить разность между найденным значением прироста и заданным пользователем *параметром сложности дерева*  $gain - \gamma$ ; если результат положительный, то подрезание (prune) дерева не выполняется, в противном случае выполняется,
- вычислить выходные значения листьев,

## 36.2. Классификация

# 37. 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 виртуальной 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<sup>16</sup> (при вводе-выводе происходит освобождение глобальной блокировки интерпретатора, что способствует повышению степени распараллеливания), а не приложений, ограничиваемых пропускной способностью процессора<sup>17</sup>. В последнем случае для достижения более высокой степени распараллеливания необходим иметь возможность *параллельного выполнения* процессов несколькими ядрами или *процессорами* [11, стр. 229].

---

<sup>16</sup>Так называемые IO-bound задачи

<sup>17</sup>CPU-bound задачи

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

## 38. 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

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

```
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 находиться 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.
```

## 39. SSH-клиент в браузере

В работе клиенты используют протокол SSH (Secure Shell) – сетевой протокол, позволяющий осуществлять удаленное управление различными операционными системами. Поддержива-

ет 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](#). Поддерживает Windows, MacOS, Linux.

## 40. 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.

## 41. Теорема Байеса

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

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

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

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  $\vec{X} = (X_1, \dots, X_n)$  используют функцию  $n$  переменных, которая в точке  $(x_1, \dots, x_n) \in \mathbb{R}^n$ , 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  $\vec{X} = (X_1, \dots, X_n)$  определяется в виде

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

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

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

#### 41.1. 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)
```

## 42. Глубокое обучение

### 42.1. Функции активации

Без функции активации (например, такой как ReLU) полносвязанный слой `keras.layers.Dense` сможет обучаться только на *линейных* (аффинных) преобразованиях входных данных: пространство гипотез было бы совокупностью всех возможных линейных преобразований входных данных.

Такое пространство гипотез слишком ограничено, и наложение нескольких слоев представлений друг на друга не приносит 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<sup>18</sup>  $p$ ,  $D_{KL}(p||q)$  – дивергенция Кульбака-Лейблера<sup>19</sup> 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.

<sup>18</sup>Мера неопределенности некоторой системы

<sup>19</sup>Дивергенция Кульбака-Лейблера – неотрицательнозначный функционал, являющийся несимметричной мерой удаленности друг от друга двух вероятностных распределений, определенных на общем пространстве элементарных событий

Настраиваем модель оптимизатором rmsprop и функцией потерь binary\_crossentropy

```
model.compile(optimizer='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
# 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_train,
    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.

## 42.2. 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 несовпадения 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 полностью связанный 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.

## 43. Хэшируемые пользовательские классы в языке Python

Чтобы класс был хэшируемым<sup>20</sup>, 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]
            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)
```

<sup>20</sup>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 изменяться и он поддерживает сравнение с другими объектами

```

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<sup>21</sup> *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]

## 44. 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):

```

<sup>21</sup>Побитовые операторы рассматривают операнды как бинарные последовательности

```

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    def __exit__(self, exc_type, exc_value, traceback):
14        import sys
15        # переменная sys.stdout.write "через" атрибут экземпляра self.original_write
16        # ссылается на объект функции sys.stdout.write
17        sys.stdout.write = self.original_write
18
```

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 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. 37)

```

>>> 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 функции 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*!

## 45. Использование @contextmanager

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

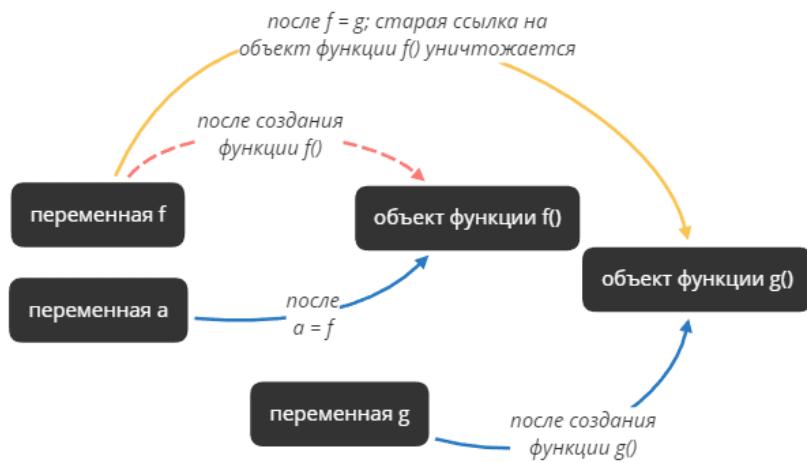


Рис. 37. Схема, описывающая связи между именами функций и их объектами

- 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

```

1 # mirror_gen.py
2 import contextlib
3
4 @contextlib.contextmanager # декорируем генераторную функцию
5 def looking_glass(): # генераторная функция
6     import sys
7     original_write = sys.stdout.write # (1)
8
9     def reverse_write(text): # замыкание
10        original_write(text[::-1]) # здесь original_write -- свободная переменная
11
12    sys.stdout.write = reverse_write # (2)
13    # все что выше 'yield' выполняется в начале блока with
14    yield 'jabberwocky'.upper() # (3)
15    # все что ниже 'yield' выполняется в конце блока with
16    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 будем 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` обертыывает функцию классом, который реализует методы `__enter__` и `__exit__`<sup>22</sup>.

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

1. Вызывает генераторную функцию `looking_glass()`<sup>23</sup> 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 `__exit__` выполняет следующие действия:

1. Сматривает, 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
```

<sup>22</sup>Этот класс называется `_GeneratorContextManager`

<sup>23</sup>При вызове генераторной функции возвращается объект-генератор

```
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

## 46. 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<sup>24</sup>,
- 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).

## 46.1. 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`<sup>25</sup> (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__`<sup>26</sup>, 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)

```

<sup>24</sup>При условии, что оператор в случае данных операндов имеет смысл

<sup>25</sup>`NotImplemented` – это значение-синглтон, которое должен возвращать специальный метод инфиксного оператора, чтобы сообщить интерпретатору, что не умеет обрабатывать данный операнд

<sup>26</sup>Иногда такие методы называют «инверсными» методами, но лучше их представлять как *правые* методы, так как они вызываются от имени правого операнда

```

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__`)

---

## 46.2. 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 operand имеет другой тип [4, стр. 425]

### 46.3. 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).

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 NotImplemented

```

и какой-то другой класс `Vector2d`

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	<code>a == b</code>	<code>a.__eq__(b)</code>	<code>b.__eq__(a)</code>	<code>return id(a) == id(b)</code>
	<code>a != b</code>	<code>a.__ne__(b)</code>	<code>b.__ne__(a)</code>	<code>return not (a == b)</code>
Порядок	<code>a &gt; b</code>	<code>a.__gt__(b)</code>	<code>a.__lt__(b)</code>	<code>raise TypeError</code>
	<code>a &lt; b</code>	<code>a.__lt__(b)</code>	<code>a.__gt__(b)</code>	<code>raise TypeError</code>
	<code>a &gt;= b</code>	<code>a.__ge__(b)</code>	<code>a.__le__(b)</code>	<code>raise TypeError</code>
	<code>a &lt;= b</code>	<code>a.__le__(b)</code>	<code>a.__ge__(b)</code>	<code>raise TypeError</code>

```
# в классе Vector2d
def __eq__(self, other):
    return 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)` видит, 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`

## 47. 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<sup>27</sup>* [1, стр. 477]:

- Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости – 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.

---

<sup>27</sup>Local, Enclosing, Global, Built-in

Если и в этих *пространствах имен* искомое имя не будет найдено, поиск будет продолжен в *глобальном пространстве имен*, а затем во *встроенным 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 имеет ссылок на какой-либо объект и потому возникает ошибка `UnboundLocalError`

---

## 48. 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»<sup>28</sup>, например, печатает строку в терминале. Задекорированные же функции выполняются строго в результате явного вызова [4, стр. 217].

### 48.1. Реализация простого декоратора

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

---

`clockdeco.py`, не очень удачный пример декоратора

---

<sup>28</sup>Если декоратор простой одноуровневый, то под верхним уровнем понимается его локальная область видимости, а если декоратор содержит замыкание, то – понимается область видимости объемлющей функции

```

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 загрузки 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():
23     print('test string from `simple_func_1`')
24
25 @simple_deco_2 # simple_func_2 = simple_deco_2(f=simple_func_2) -> simple_func_2
26 def simple_func_2():
27     print('test string from `simple_func_2`')
28
29 @clock # snooze = clock(func=snooze) -> clocked
30 def snooze(seconds):
31     time.sleep(seconds)
32
33 @clock
34 def factorial(n):
35     return 1 if n < 2 else n*factorial(n-1)
36
37
38 if __name__ == '__main__':
39     print('*'*10, 'Calling snooze(.123)')
40     print('snooze_result = {}'.format(snooze(.123)))
41     print('*'*10, 'Calling factorial(6)')

```

```

42 print('6! = ', factorial(6))
43 print(f'This is result from \'simple_func_1\': {simple_func_1()}' )
44 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

```

## 48.2. 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*.

---

## 48.3. 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 вызываться 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].

#### 48.4. 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`.

## 48.5. 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 представить, 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

@fabricdeco() # 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 переменной `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 область 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
```

## 48.6. 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 декорирует 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 строка `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`.

## 48.7. 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. 38).

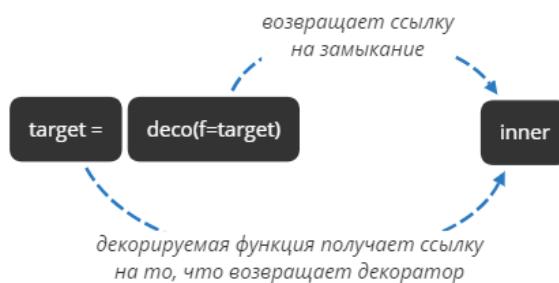


Рис. 38. К вопросу о механизме работы декоратора с вложенной функцией

## 48.8. 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_and_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.

## 49. Методы в Python

### 49.1. 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 ли метод статическим, – это издержки дизайна языка.

### 49.2. 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())

```

## 50. 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<sup>29</sup>: *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

```
>>> 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].

### 50.1. 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].

---

<sup>29</sup>Определение содержит авторские правки

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 обземлюющей 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

---

## 51. 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<sup>30</sup> (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*

---

## 52. 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)
```

<sup>30</sup>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

```
# 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
```

## 53. Библиотека functools

### 53.1. 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'])
# передаем 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 там в свою очередь get_live()
# разумеется можно вызвать метод set_live(False) и напрямую

```

## 54. 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.

### 54.1. 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 «равношириинном 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](#).

### 54.2. 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.

### 54.3. 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](#).

#### 54.4. Логистическая регрессия в пространстве логитов

#### 54.5. Деревья калибровки

Стандартный алгоритм 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.

#### 54.6. 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.

### 55. Приемы работы с менеджером пакетов conda

#### 55.1. 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`<sup>31</sup>

```
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
```

<sup>31</sup>В данном случае чтобы удалить виртуальную среду достаточно просто удалить директорию `envs`

## 55.2. 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}}
```

и теперь имя окружения будет (`envs`).

Деактивировать виртуальное окружение

```
conda deactivate
```

## 55.3. 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`<sup>32</sup>, 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.

## 55.4. 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
```

<sup>32</sup>Этот файл должен находиться в той же директории что и директория окружения `envs`

## 55.5. Удаление виртуального окружения

Удалить виртуальное окружение heroku\_env

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

## 55.6. Экспорт виртуального окружения в environment.yml

Экспортировать активное виртуальное окружение в yml-файл

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

# 56. 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 Mac OS 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.

## 57. Управление локальными переменными окружения проекта

Для того чтобы создать локальные переменные проекта<sup>33</sup> 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
```

<sup>33</sup> То есть переменные, привязанные к текущему проекту

```

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)))

```

## 58. Приемы работы с модулем subprocess

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

```

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}\n')

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.')

```

## 59. Решающие деревья и сопряженные вопросы

### 59.1. Коэффициент Джини

Коэффициент Джини<sup>34</sup> (Gini impurity) это просто вероятность неверной маркировки в узле случайно выбранного образца (для чистых листьев коэффициент Джини равен 0)

$$I_G(n) = 1 - \sum_{i=1}^J p_i^2, \quad (1)$$

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

$$I_G(n) = 1 - \left( \left( \frac{2}{6} \right)^2 + \left( \frac{4}{6} \right)^2 \right) = 0,444. \quad (2)$$

### 59.2. Случайный лес

Случайный лес – это модель, представляющая ансамбль решающих деревьев, дополненная двумя концепциями:

- концепцией бутстрапированных 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.

## 60. Анализ временных рядов

### 60.1. Признаки на временных рядах

Можно выделить следующие несколько групп признаков, которые можно вычислить на временных рядах:

- признаки на основе коэффициентов автокорреляции и частных коэффициентов автокорреляции,
- оптимальное значение параметра  $\lambda$  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,

<sup>34</sup>Еще говорят индекс Джини или загрязненность Джини

- дивергенция Кульбака-Лейблера в следующих друг за другом отрезках,
- спектральная энтропия ряда,
- дисперсия средних значений,
- минимальное число дифференцирований временного ряда, необходимое для достижения его стационарности.

## 60.2. Кросс-валидация на временных рядах

Оригинал статьи: [Time Series Nested Cross-Validation](#).

В случае обычной  $k$ -блочной перекрестной проверки мы имеющийся набор данных разбиваем на обучающий и тестовый. А затем обучающий разбиваем на подобучающий и валидационный. На подобучающем наборе данных мы обучаем модель, на валидационном настраиваем гиперпараметры. Затем модель обучается на полном обучающем наборе данных с использованием лучшей комбинации параметров. На тестовом наборе данных определяется обобщающая способность модели (рис. 39).

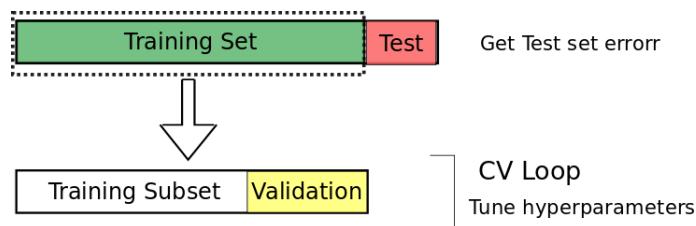


Рис. 39. Стандартная процедура кросс-валидации

На временных рядах традиционная перекрестная проверка не должна использоваться, так как возможна утечка данных из «будущего». Рекомендуется использовать *вложенную перекрестную проверку* (nested cross-validation). Вложенная перекрестная проверка включает внешний цикл для вычисления ошибок и внутренний цикл для подбора параметров.

Внутренний цикл работает как и раньше: обучающий набор данных разбивается на подобучающий набор и валидационный набор. Модель обучается на подобучающем наборе, а на валидационном наборе подбираются параметры. А во внешнем цикле исходный набор данных (временной ряд) разбивается на обучающий и тестовый наборы данных, и ошибка каждого разбиения усредняется (рис. 40).

Вложенная перекрестная проверка (чуть подробнее):

- Весь временной ряд разбиваем на несколько блоков,
- Первый блок разбивается на *обучающий* и *тестовый* набор данных,
- Обучающий набор данных первого сплита разбивается на *подобучающий* и *валидационный* набор данных,
- Для каждой комбинации гиперпараметров обучаем модель на *подобучающем* наборе данных, а затем вычисляем метрку качества на валидационном наборе данных (первого сплита),
- Выбираем такую комбинацию гиперпараметров, которая отвечает наибольшему значению метрики качества (или наименьшему значению ошибки), и обучаем модель еще раз с наилучшей комбинацией параметров, но уже на всем обучающем наборе данных первого сплита,
- Вычисляем метрику качества (или ошибку) на тестовом наборе данных первого сплита,
- Переходим к следующему блоку и повторяем всю цепочку.

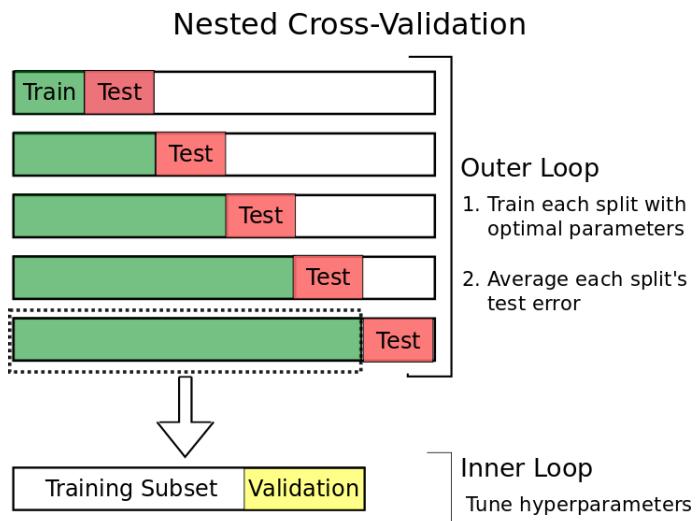


Рис. 40. Вложенная перекрестная проверка на временном ряде

- В итоге полученные ошибки тестового набора данных на каждом разбиении усредняем по этим разбиениям.

### 60.3. Синус-косинусное кодирование во временных рядах

Признаки типа «день недели», «время суток», «день месяца» и пр., строго говоря, имеют циклическую природу и игнорировать ее нельзя. Таким образом, обычно рекомендуется признаки, имеющие отношение к временной шкале кодировать с помощью синус-косинусного преобразования

$$\tilde{f}_i^{(s)} = \sin(f_i \cdot \frac{2\pi}{n_i}), \quad \tilde{f}_i^{(c)} = \cos(f_i \cdot \frac{2\pi}{n_i}),$$

где  $f_i$  –  $i$ -ый признак;  $\tilde{f}_i^{(s)}, \tilde{f}_i^{(c)}$  – новое закодированное значение  $i$ -ого признака для синуса и косинуса соответственно;  $n_i$  – число уникальных значений признака (например, для времени суток это 24, для дней недели это 7 и т.д.).

Пример

```
df["hr_sin"] = np.sin(df["hr"]*2*np.pi/24)
df["hr_cos"] = np.cos(df["hr"]*2*np.pi/24)

df["mnth_sin"] = np.sin((df["mnth"] - 1)*2*np.pi/12)
df["mnth_cos"] = np.cos((df["mnth"] - 1)*2*np.pi/12)
```

### 60.4. 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 (на-

пример, 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 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 расположеными 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 предсказанием 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  $k$ -й 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  $3H$ , однако этот параметр можно изменить.

Каждый раз после подгонки модели на обучающих данных из  $k$ -го 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*.

## 60.5. 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 экземпляров является коллективной аномалией; кроме того, если точечные или контекстуальные аномалии могут наблюдаться в любом наборе данных, то коллективные наблюдаются только в тех, где данные связаны 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 отклонений 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 распознавание. 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 распознавания 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 распознавания 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 противоположный. Алгоритмы, работающие в режиме распознавания 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<sup>35</sup> 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',
          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')
```

<sup>35</sup> В `sklearn` есть реализация одноклассового метода опорных векторов `OneClassSVM` (позволяет задать долю аномальных объектов в выборке с помощью параметра `nu`)

```
plt.legend()
```

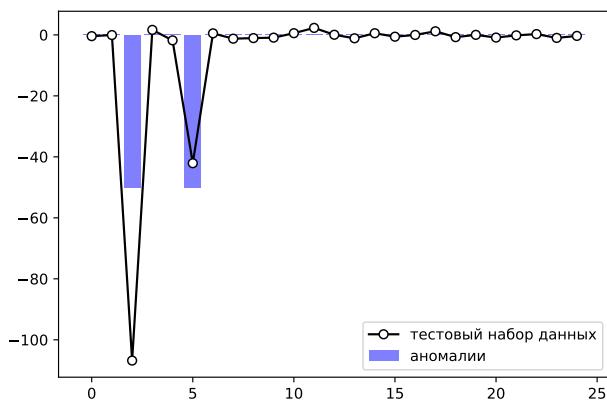


Рис. 41. Пример детектирования аномалий на тестовой наборе данных

**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 реализаций 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  $\theta$  и функцией плотности вероятности  $\mathbb{P}(x, \theta)$ , где  $x$  – наблюдение. Аномалия является обратной функцией распределения. Эти методы часто основываются на Гауссовой 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 исследуемых 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 основываются 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 удален 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.

## 60.6. 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*<sup>36</sup>);  $s(t)$  – сезонная компонента, отвечающая за периодические/квазипериодические изменения, связанные с *недельной* и *годовой сезонностью*<sup>37</sup>;  $h(t)$  – отвечает за аномальные дни (праздники, Black Fridays и т.д.);  $\varepsilon$  – 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.

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.

<sup>36</sup>Логистическая функция удобна для моделирования роста с насыщением, когда при увеличении показателя снижается темп его роста

<sup>37</sup>Моделируется с помощью рядов Фурье

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 точки 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 (например, 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

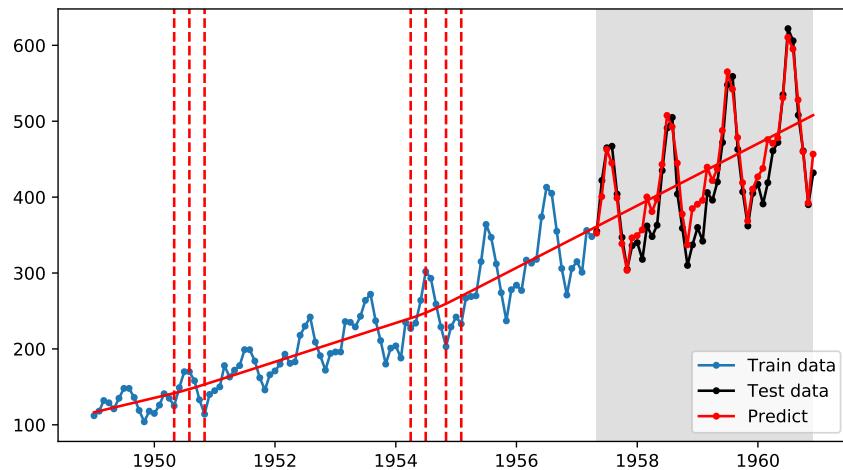


Рис. 42. Пример использования библиотеки `fbprophet`

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 пределах 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 выраженным 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 позволяют 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 МCMC. 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 a posteriori probability, MAP) тесно связана с *методом наибольшего правдоподобия* (ML), но дополнительно при оптимизации использует априорное распределение величины, которую оценивает.

Можно записать

$$\hat{\theta}_{\text{MAP}}(x) = \arg \max_{\theta} f(x|\theta)g(\theta),$$

где  $f(x|\theta)$  – функция правдоподобия,  $g(\theta)$  – априорная плотность распределения оцениваемого параметра  $\theta$ .

Пример. Предположим, что у нас есть последовательность  $(x_1, \dots, x_n)$  i.i.d (независимых и одинаково распределенных)  $N(\mu, \sigma_v^2)$  случайных величин и априорное распределение  $\mu$  задано  $N(0, \sigma_m^2)$ . Требуется найти MAP-оценку  $\mu$ .

Функция, которую нужно максимизировать задана

$$\pi(\mu)L(\mu) = \frac{1}{\sqrt{2\pi}\sigma_m} \exp\left(-\frac{1}{2}\left(\frac{\mu}{\sigma_m}\right)^2\right) \prod_{j=1}^n \frac{1}{\sqrt{2\pi}\sigma_v} \exp\left(-\frac{1}{2}\left(\frac{x_j - \mu}{\sigma_v}\right)^2\right).$$

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

$$\hat{\mu}_{\text{MAP}} = \frac{\sigma_m^2}{n\sigma_m^2 + \sigma_v^2} \sum_{j=1}^n x_j.$$

## 60.7. Преобразование нестационарного временного ряда в стационарный

Чтобы превратить нестационарный ряд в стационарный можно использовать следующие общие приемы:

- о 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,
- о 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()`.

## 60.8. 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](#)<sup>38</sup>, 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  $y = \{y_1, \dots, y_n\}$ ,  $y_i > 0$ ,  $i = (1, \dots, n)$ .

Пример использования преобразования Бокса-Кокса приведен на рис. 43. Такого рода преобразования полезны в ситуациях, связанных с проблемой *гетероскедастичности* (непостоянная дисперсия), или в ситуациях, где требуется *гауссовость* данных.

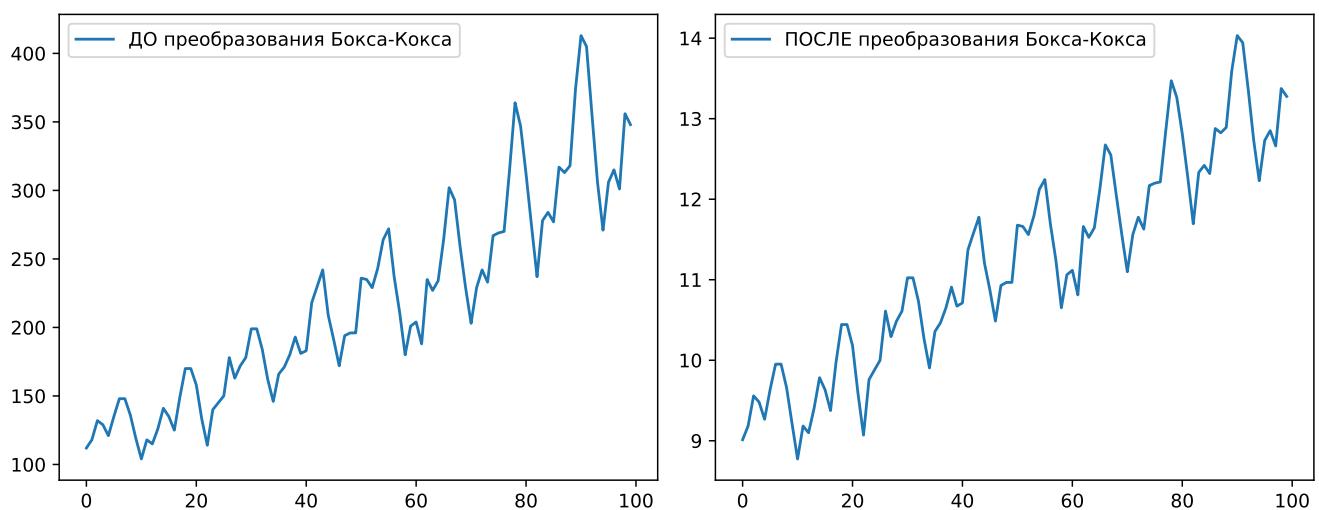


Рис. 43. Влияние преобразования Бокса-Кокса на временной ряд с изменяющейся во времени дисперсией

<sup>38</sup>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 асимметрию*

Параметр  $\lambda$  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.holtwinters import (
    #boxcox,
    inv_boxcox
)

# пользуемся готовым решением для обратного преобразования Бокса-Кокса
lmbda = 0.25
arr = np.array([3, 5, 10])
# можно задать значение ламбда 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
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  $\alpha$  так, чтобы выполнялось условие  $(y_i + \alpha) > 0, i = 1, \dots, n$

$$\hat{y}_i = \begin{cases} \log(y_i + \alpha), & \lambda = 0, \\ \frac{(y_i + \alpha)^\lambda - 1}{\lambda}, & \lambda \neq 0. \end{cases}$$

Также для того чтобы сделать данные «более гауссовскими» можно воспользоваться *преобразованием Йео-Джонсона* (Yeo-Johnson)

$$\hat{y}_i = \begin{cases} \frac{(y_i + 1)^\lambda - 1}{\lambda}, & \lambda \neq 0, y_i \geq 0, \\ \ln(y_i + 1), & \lambda = 0, y_i \geq 0, \\ -\frac{(-y_i + 1)^{2-\lambda} - 1}{2 - \lambda}, & \lambda \neq 2, y_i < 0, \\ -\ln(-y_i + 1), & \lambda = 2, y_i < 0. \end{cases}$$

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

---

## 61. Методологии проектирования реляционного хранилища данных

### 61.1. Основная терминология

Data Vault – гибридный подход, объединивший достоинства схемы «звезды» и третьей нормальной формы. Состоит из трех основных компонентов (рис. 44) – хабов (Hub), ссылок (Link) и спутников (Satellite).

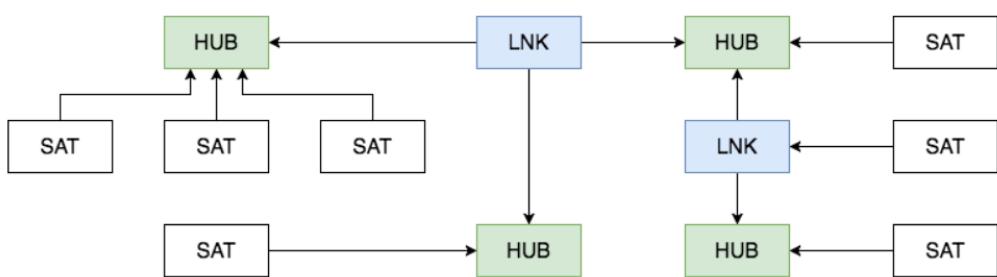


Рис. 44. Пример хранилища, построенного с помощью подхода Data Vault

*Хаб* – основное представление сущности (Клиент, Продукт, Заказ) с позиций бизнеса. Таблица-хаб содержит одно или несколько полей, отражающих сущность в понятиях бизнеса. В совокупности эти поля называются «бизнес-ключ». Идеальный кандидат на звание бизнес-ключа это ИНН организации или VIN номер автомобиля, а сгенерированный системой ID будет наихудшим вариантом. Бизнес-ключ всегда должен быть уникальным и неизменяемым. Хаб также содержит мета-поля load timestamp и record source, в которых хранятся время первоначальной загрузки сущности в хранилище и ее источник (название системы, базы или файла, откуда данные были загружены). В качестве первичного ключа Хаба рекомендуется использовать MD5 или SHA-1 хеш от бизнес-ключа.

*Ссылка* – связь типа «многие-ко-многим», связывающая несколько хабов. Она содержит те же метаданные, что и хаб. Ссылка может быть связана с другой ссылкой, но такой подход создает проблемы при загрузке, так что лучше выделить одну из ссылок в отдельный хаб.

Все описательные атрибуты Хаба или Ссылки (контекст) помещаются в таблицы-Спутники. Помимо контекста Спутник содержит стандартный набор метаданных (load timestamp и record source) и *один и только один* ключ «родителя». В Спутниках можно без проблем хранить историю изменения контекста, каждый раз добавляя новую запись при обновлении контекста в системе-источнике. Для упрощения процесса обновления большого спутника в таблицу можно добавить поле hash diff: MD5 или SHA-1 хеш от всех его описательных атрибутов. Для Хаба или Ссылки может быть сколь угодно Спутников, обычно контекст разбивается по частоте обновления. Контекст из разных систем-источников принято класть в отдельные Спутники.

Сначала данные из операционных систем поступают в staging area. Staging area используется как промежуточное звено в процессе загрузки данных. Одна из основных функций Staging зоны это снижение нагрузки на операционные базы при выполнении запросов. Таблицы здесь

полностью повторяют исходную структуру, но любые ограничения на вставку данных, вроде not null или проверки целостности внешних ключей, должны быть выключены с целью оставить возможность вставить даже поврежденные или неполные данные (особенно это актуально для excel-таблиц и прочих файлов). Дополнительно в stage таблицах содержатся хеши бизнес-ключей и информация о времени загрузки и источнике данных (рис. 45).

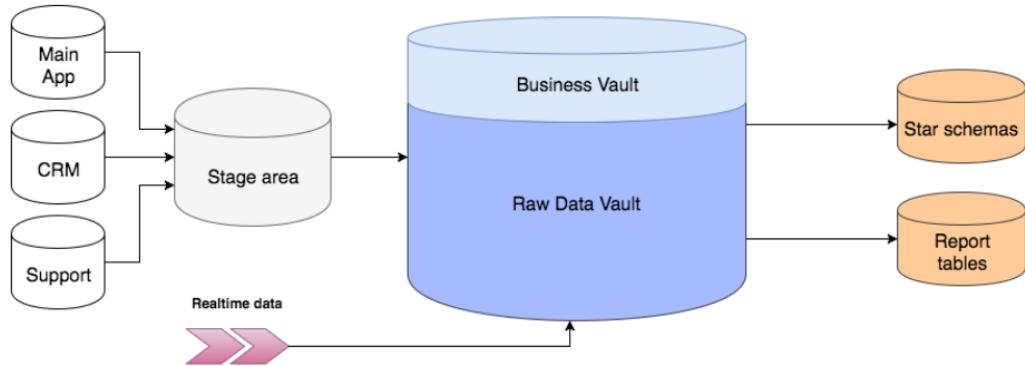


Рис. 45. Высокоуровненное представление хранилища данных

После этого данные разбиваются на Хабы, Ссылки и Сателлиты и загружаются в Raw Data Vault (в процессе загрузки они никак не агрегируются и не пересчитываются).

Business Vault — опциональная вспомогательная надстройка над Raw Data Vault. Строится по тем же принципам, но содержит переработанные данные: агрегированные результаты, сконвертированные валюты и прочее. Разделение чисто логическое, физически Business Vault находится в одной базе с Raw Data Vault и предназначен в основном для упрощения формирования витрин.

Когда нужные таблицы созданы и заполнены, наступает очередь витрин данных (Data Marts). Каждая витрина это отдельная база данных или схема, предназначенная для решения задач различных пользователей или отделов. В ней может быть специально собранная «звезда» или коллекция денормализованных таблиц. Если возможно, *таблицы* внутри *витрин* лучше делать *виртуальными*, то есть вычисляемыми «на лету». Для этого обычно используются SQL представления (SQL views).

Заполняется Data Vault следующим образом: сначала загружаются Хабы, потом Ссылки и затем Сателлиты; Хабы можно загружать параллельно, так же как и Сателлиты и Ссылки, если конечно не используется связь link-to-link.

#### Преимущества и недостатки

- (+) Гибкость и расширяемость. С Data Vault перестает быть проблемой как расширение структуры хранилища, так и добавление и сопоставление данных из новых источников. Максимально полное хранилище «сырых» данных и удобная структура их хранения позволяют нам сформировать витрину под любые требования бизнеса, а существующие решения на рынке СУБД хорошоправляются с огромными объемами информации и быстро выполняют даже очень сложные запросы, что дает возможность виртуализировать большинство витрин,
- (+) Agile-подход из коробки. Моделировать хранилище по методологии Data Vault довольно просто. Новые данные просто «подключаются» к существующей модели, не ломая и не модифицируя существующую структуру. При этом поставленная задача решается максимально изолировано,

- (-) Обилие соединений. За счет большого количества операций соединения запросы могут быть медленее, чем в традиционных хранилищах данных, где таблицы денормализованы,
- (-) Сложность. В описаной выше методологии есть множество важных деталей, разобраться в которых вряд ли получится за пару часов. К этому можно прибавить малое количество информации. Как следствие, при внедрении Data Vault возникают проблемы с обучением команды. Большой недостаток – это обязательное требование наличия витрин данных, так как сам по себе Data Vault плохо подходят для прямых запросов,
- (-) Избыточность.

## 61.2. Базовые сведения о концепциях

Многое в жизни проекта зависит от того, насколько хорошо продумана объектная модель и структура базы на старте.

Общепринятым подходом были и остаются различные варианты сочетания схемы «звезды» с третьей нормальной формой. Как правило, по принципу: исходные данные — 3NF, витрины — звезда. Этот подход, проверенный временем и подкрепленный большим количеством исследований — первое (а иногда и единственное), что приходит в голову опытному DWH-шнику при мысли о том, как должно выглядеть *аналитическое хранилище*.

С другой стороны данные могут расти как «вглубь», так и «вширь». И вот тут проявляется основной недостаток звезды — *ограниченная гибкость*.

Ниже рассматриваются две самые популярные для хранилищ данных методологии гибкого проектирования – Anchor model и Data Vault. За скобками остаются такие приемы, как, например, EAV<sup>39</sup> (Entity-attribute-value), 6NF и все относящееся к NoSQL решениям.

**Проблемы «классического» подхода и их решения в гибких методологиях** Во-первых, в основу такой модели закладывается четкое разделение данных на *измерения* и *факты*. И это логично. Ведь в подавляющем большинстве случаев анализ сводится именно к анализу определенных числовых показателей (фактов) в определенных разрезах (измерениях).

При этом связи между объектами закладываются в виде связей между таблицами по внешнему ключу. Это выглядит вполне логично, но сразу же приводит к первому ограничению гибкости – жесткому определению кардинальности связей.

Это значит, что на этапе проектирования таблиц нужно точно определить для каждой пары связанных объектов могут ли они относиться как многие-ко-многим, или только как 1-ко-многим, и «в какую сторону». От этого напрямую зависит в какой из таблиц будет первичный ключ, а в какой – внешний. Изменение этого отношения при получении новых требований с большой вероятностью приведет к переработке базы данных.

В Data Vault таблицы-связки называют Link, а в Якорной модели – Tie. В обеих архитектурах таблицы-связки могут связывать любое количество сущностей.

Эта на первый взгляд избыточность дает существенную гибкость при доработках (рис. 46).

**Хранение объектов и атрибутов в Data Vault и Anchor Model** Подход, предлагаемый авторами гибких архитектур, можно сформулировать так: необходимо отделить то, что изменяется, от того, что остается неизменным; то есть хранить ключи отдельно от атрибутов.

---

<sup>39</sup>Это модель данных для кодирования пространственно-эффективным способом сущностей, в которых число атрибутов (свойств, параметров) потенциально огромно, но число фактически применяемых к данной сущности относительно мало. Такие сущности соответствуют математическому понятию разреженной матрицы

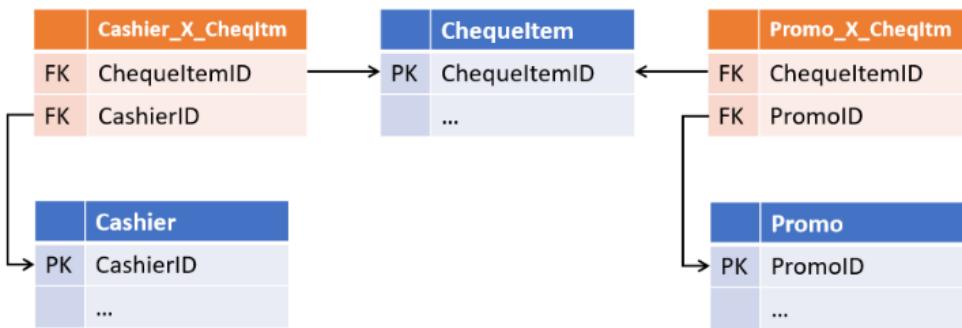


Рис. 46. Пример Структура толерантная к изменению кардинальности существующих связей

Точки зрения на то, что именно можно считать неизменным в Data Vault и Якорной модели расходятся. С точки зрения архитектуры Data Vault, неизменным можно считать весь набор ключей – натуральные (ИНН организации, код товара в системе-источнике и т.п.) и суррогатные<sup>40</sup>. При этом остальные атрибуты можно разделить на группы по источнику и/или частоте изменений и для каждой группы вести отдельную таблицу с независимым набором версий.

В Anchor Model неизменным считается только суррогатный ключ сущности. Все остальное (включая натуральные ключи) – просто частный случай его атрибутов. При этом все атрибуты по умолчанию независимы друг от друга, поэтому для каждого атрибута должна быть создана отдельная таблица.

В Data Vault таблицы, содержащие ключи сущностей, называются Хабами (Hub). Хабы всегда содержат фиксированный набор полей:

- Натуральные ключи сущности,
- Суррогатный ключ,
- Ссылку на источник,
- Время добавления записи.

Записи в Хабах никогда не изменяются и не имеют версий. Внешние хабы очень похожи на таблицы типа ID-мап, применяемые в некоторых системах для генерации суррогатов, однако в качестве суррогатов в Data Vault рекомендуется применять не целочисленный сиквенс, а хэш от набора бизнес-ключей. Такой подход упрощает загрузку связей и атрибутов из источников, но может вызвать другие проблемы (поэтому не является общепринятым).

Все остальные атрибуты сущностей хранятся в специальных таблицах, называемых Сателлитами. Один хаб может иметь несколько сателлитов, хранящих разные наборы атрибутов.

Распределение атрибутов по сателлитам происходит по принципу совместного изменения – в одном сателлите могут храниться не версионные атрибуты (например, дата рождения и СНИЛС для физ. лица), в другом – редко изменяющиеся версионные (например, фамилия и номер паспорта), в третьем – часто изменяющиеся (например, адрес доставки, категория, дата последнего заказа и т.п.).

Версионность при этом ведется на уровне отдельных сателлитов, а не сущности в целом, поэтому распределение атрибутов целесообразно проводить так, чтобы пересечение версий внутри одного сателлита было минимальным (что сокращает общее количество хранимых версий).

<sup>40</sup>Суррогатный ключ – это искусственно введенный в отношение атрибут, предназначенный исключительно для идентификации кортежей в отношении. Другими словами, суррогатный ключ не образовывается на основании каких-либо существующих атрибутах отношения, а генерируется искусственно

Также, для оптимизации процесса загрузки данных, в отдельные сателлиты часто выносятся атрибуты, получаемые из различных источников. Сателлиты связываются с хабом по внешнему ключу (что соответствует связи один-ко-многим). Это значит, что множественные значения атрибутов (например, несколько контактных номеров телефона у одного клиента) поддерживаются такой архитектурой по умолчанию.

В Якорной модели таблицы, хранящие ключи, называются Якорями (Anchor) и хранят они:

- только суррогатные ключи,
- ссылку на источник,
- время добавления записи.

Натуральные ключи с точки зрения Якорной модели считаются обычными атрибутами. Например, если данные об одной и той же сущности могут поступать из разных систем, в каждой из которых используется свой натуральный ключ.

В Data Vault типичный объект для хранения фактов - Связь (Link), в Сателлиты которой складываются вещественные показатели.

Такой подход выглядит интуитивно понятным. Он дает простой доступ к анализируемым показателям и в целом похож на традиционную таблицу фактов (только показатели хранятся не в самой таблице, а в «соседней»). Но есть и подводные камни: одна из типовых доработок модели — расширение ключа факта — вызывает необходимость добавления в Link нового внешнего ключа. А это в свою очередь «ломает» модульность и потенциально вызывает необходимость доработок других объектов.

В Якорной модели Связь не может иметь собственных атрибутов, поэтому такой подход не прокатит — абсолютно все атрибуты и показатели обязаны иметь привязку к одному конкретному якорю. Вывод из этого простой — для каждого факта тоже нужен свой якорь. Для части того, что мы привыкли воспринимать как факты, это может выглядеть естественно — например, факт покупки прекрасно сводится к объекту «заказ» или «чек», посещение сайта — к сессии и т.п. Но встречаются и факты, для которых найти естественный «объект-носитель» не так просто — например, остатки товаров на складах на начало каждого дня.

Соответственно, проблем с модульностью при расширении ключа факта в Якорной модели не возникает (достаточно просто добавить новую Связь к соответствующему Якорю), но проектирование модели для отображения фактов менее однозначно, могут появляться «искусственные» Якоря, отображающие объектную модель бизнеса не очевидно.

Получившаяся конструкция в обоих случаях содержит существенно больше таблиц, чем традиционное измерение. Но может занимать существенно меньше дискового пространства при том же наборе версионных атрибутов, что и традиционное измерение. Никакой магии тут, естественно, нет — всё дело в нормализации. Распределяя атрибуты по Сателлитам (в Data Vault) или отдельным таблицам (Anchor Model), мы уменьшаем (или совсем исключаем) дублирование значений одних атрибутов при изменении других.

Для Data Vault выигрыш будет зависеть от распределения атрибутов по Сателлитам, а для Якорной модели — практически прямо пропорционален среднему количеству версий на объект измерения.

Однако выигрыш по занимаемому месту — важное, но не главное преимущество отдельного хранения атрибутов. Вместе с отдельным хранением связей, такой подход делает хранилище модульной конструкцией. Это значит, что добавление как отдельных атрибутов, так и целых но-

вых предметных областей в такой модели выглядит как надстройка над существующим набором объектов без их изменения. И это именно то, что делает описанные методологии гибкими.

Декомпозиция данных, лежащая в основе модульности гибких архитектур, приводит к увеличению количества таблиц и, соответственно, накладных расходов на джойны при выборке. Для того, чтобы просто получить все атрибуты измерения, в классическом хранилище достаточно одного селекта, а гибкая архитектура потребует целого ряда джойнов. Причем если для отчетов все эти джойны можно написать заранее, то аналитики, привыкшие писать SQL руками, будут страдать вдвойне.

Многое зависит от движка. У многих современных платформ есть внутренние механизмы оптимизации джойнов. Например, MS SQL и Oracle умеют «пропускать» джойны на таблицы, если их данные не используются нигде, кроме других джойнов и не влияют на финальную выборку (table/join elimination), а MPP *Vertica* по опыту коллег из Авито, показала себя как прекрасный *двизжок* для *Якорной модели* с учетом некоторой ручной оптимизации плана запроса.

С другой стороны, хранить Якорную модель, например, на Click House, имеющем ограниченную поддержку join, пока выглядит не очень хорошей идеей.

Выводы:

- После некоторой начальной подготовки, связанной с развертыванием метаданных и написанием базовых алгоритмов ETL, быстро предоставить заказчику первый результат в виде парочки отчетов, содержащих данные всего нескольких объектов источников. Полностью продумывать (даже верхнеуровнево) всю объектную модель для этого не обязательно,
- Модель данных может начать работать (и приносить пользу) всего с 2-3 объектами, разрас-таясь со временем,
- Большинство доработок, включая расширение предметной области и добавление новых ис-точников не затрагивает существующий функционал и не вызывает опасность сломать что-то уже работающее,
- Благодаря декомпозиции на стандартные элементы, ETL-процессы в таких системах выгля-дят однотипно, их написание поддается алгоритмизации и, в конечном счете, автоматизации.

Ценой такой гибкости является производительность. Это не значит, что достигнуть приемлемой производительности на таких моделях невозможно. Чаще всего, просто может потребоваться больше усилий и внимания к деталям для достижения нужных метрик.

Сводная таблица с общими чертами и различиями рассмотренных подходов приведена на рис. 47.

## 62. 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/><sup>41</sup>

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

---

<sup>41</sup>Установить можно как обычно: pip install category-encoders

## Data Vault vs Anchor Model

Гибкая структура, позволяющая быстро добавлять в модель новые атрибуты и источники

Нормализация, больше таблиц, более тяжелые запросы по сравнению со звездой и ЗНФ.

Связи между объектами – не один из атрибутов объекта, а отдельный тип объектов

«Объектно-ориентированный» подход – концентрация на объектной структуре бизнеса, а не структуре данных

Сложная система метаданных, но более простые и типовые ETL

Data Vault	Anchor Model
<b>update</b> – не предпочтительно, но допустимо	<b>insert only!</b>
Некоторые изменения модели не вызывают изменения существующих объектов	Любая доработка – надстройка над существующей структурой таблиц
Связи могут иметь <b>свои атрибуты</b> и могут отражать <b>факты</b> .	Связи отражают <b>только взаимоотношения</b> объектов и не могут иметь своих атрибутов
<b>Объединение</b> атрибутов по частоте обновления и источникам	Максимальная <b>декомпозиция</b>

Рис. 47. Сводка по гибким методологиям

- 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 мультиколлинеарность, что может приводить к появлению неустойчивых оценок [10, 153]),
- M-вероятностная оценка правдоподобия (другие варианты: аддитивное сглаживание; кодирование сглаженным средним): можно использовать класс `category_encoders.m_estimate.MEstimateEncoder`. Каждая категория категории

ального признака кодируются по следующей формуле

$$e_k = \frac{s_k^t + m \frac{s^t}{n}}{c_k + m}, \quad (k = 1, \dots, K),$$

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](https://contrib.scikit-learn.org/category_encoders/woe.html)): чаще всего применяется в кредитном scoringе.
- *CatBoost-кодировщик*: можно использовать готовый класс CatBoostEncoder [https://contrib.scikit-learn.org/category\\_encoders/catboost.html](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.

## 63. 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>.

## 64. Хранилища данных. DWH

Хранилище данных (Data WareHouse, DWH) – предметно-ориентированная информационная база данных, специально разработанная и предназначена для подготовки отчетов и бизнес-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 сконструировать 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 базе 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 (ПоТ), данные с датчиков можно собирать в «озеро данных»<sup>42</sup> 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.

---

<sup>42</sup>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

## 65. 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>.

## 66. Приемы работы с библиотекой Vowpal Wabbit

Vowpal Wabbit – библиотека с открытым исходным кодом, ориентированная на крупно-масштабные онлайн<sup>43</sup>-задачи машинного обучения, в основе которых лежат так называемые алгоритмы, работающие во внешней памяти (их еще называют *внеклерными* (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).

Внеклерные 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.

<sup>43</sup> В компьютерных науках онлайнное машинное обучение (online machine learning [https://en.wikipedia.org/wiki/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. Онлайнное 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
```

## 67. 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()

```

## 68. Приемы работы с библиотекой BeautifulSoup

### 68.1. 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=eudimonia')
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)
# выведем
# [(0, 'Мясников', 'Александр', 'Алексеевич'),
# (1, 'Медовщук', 'Ирина', 'Сергеевна'),
# (2, 'Матыцын', 'Александр', 'Петрович'),
# (3, 'Смирнова', 'Оксана', 'Константиновна'),
# ...]

```

## 69. Приемы работы с библиотекой pandas

### 69.1. 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()
```

## 69.2. Срезы в мультииндексах

Для того чтобы выбрать определенные элементы индекса определенного уровня, нужно воспользоваться конструкцией

```
df = ...
df.xs("XOne", level="Platform")
```

## 69.3. 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]
```

## 69.4. 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
```

## 69.5. Число пропущенных значений в объекте DataFrame

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

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

## 69.6. 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 выделяют 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 минимальных 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. 48.

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('abbbabbaab'),

```

	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. 48. Отформатированный вывод DataFrame

```
'col2' : list('cdcccddcsd'),
'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. 49. 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. 49. Результат применения функции color\_code\_freq\_cat

## 69.7. 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()
)

```

## 70. 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')
```

## 71. 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 *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`).

## 72. Подбор гиперпараметров

### 72.1. Приемы работы с библиотекой `hyperopt`

На сегодняшний день существует несколько популярных подходов к решению задачи подбора гиперпараметров:

- Прямой поиск по сетке (например, `GridSearchCV` в `sklearn`),
- Случайный поиск (например, `RandomizedSearchCV` в `sklearn`),
- Байесовская оптимизация.

В случае байесовской оптимизации значения гиперпараметров в текущей итерации выбираются с учетом результатов на предыдущем шаге. Основная идея алгоритма заключается в следующем – на каждой итерации подбора находится компромисс между исследованием регионов

с самой удачной из найденных комбинаций гиперпараметров и исследований регионов с большой неопределенностью (где могут находиться еще более удачные комбинации). Это позволяет во многих случаях найти лучшие значения параметров модели за меньшее количество времени, так как алгоритм старается сосредоточиться именно на самых перспективных регионах. Порядок вычислений следующий

1. Для целевой функции построить суррогатную функцию (такую функцию удобнее оптимизировать),
2. На этой суррогатной функции найти гиперпараметры с самым лучшим результатом,
3. Найти значения целевой функции для найденных гиперпараметров,
4. Обновить суррогатную функцию с учетом результатов п. 3,
5. Повторять п. 2 - 4 до достижения критерия останова.

В библиотеке `hyperopt` реализовано 3 алгоритма оптимизации:

- классический случайный поиск Random Search,
- метод байесовской оптимизации Tree of Parzen Estimators (TPE),
- метод имитации отжига Simulated Annealing.

Пример

```
# выбираем категориальные (тип object)
# и численные признаки (остальные типы)
num_columns = np.where(X.dtypes != 'object')[0]
cat_columns = np.where(X.dtypes == 'object')[0]

# пайпайн для категориальных признаков
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(missing_values='?', strategy='most_frequent')),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])

# пайпайн для численных признаков
num_pipe = Pipeline([('scaler', StandardScaler())])

# соединяем пайпайны вместе
transformer = ColumnTransformer(
    transformers=[
        ('cat', cat_pipe, cat_columns),
        ('num', num_pipe, num_columns)
    ], remainder='passthrough'
)

# итоговая модель
model = Pipeline([
    ('transformer', transformer),
    ('lr', LogisticRegression(
        random_state=RS, solver='liblinear')
)
])
```

Зададим пространство поиска. Hyperopt позволяет работать с параметрами разного типа – непрерывными, дискретными и категориальными

```
search_space = {
    'lr_penalty' : hp.choice(label='penalty',
        options=['l1', 'l2']),
    'lr_C' : hp.loguniform(label='C',
        low=-4*np.log(10),
```

```

    high=2*np.log(10))
}

```

Здесь параметр регуляризации выбирается из лог-равномерного распределения  $[-4 \ln 10, 2 \ln 10]$ , и может принимать значения  $[10^{-4}, 10^2]$ . Тип регуляризации равновероятно выбирается из  $[L_1, L_2]$ .

Укажем функцию, которую будем оптимизировать. Она принимает на вход гиперпараметры, модель и данные, после чего возвращает точность на кросс-валидации (заметим, что точность возвращается со знаком минус, т.к. мы минимизируем функцию)

```

def objective(params, pipeline, X_train, y_train, r_state=None):
    """
    Кросс-валидация с текущими гиперпараметрами

    :params: гиперпараметры
    :pipeline: модель
    :X_train: матрица признаков
    :y_train: вектор меток объектов
    :return: средняя точность на кросс-валидации
    """

    # задаём параметры модели
    pipeline.set_params(**params)

    # задаём параметры кросс-валидации (стратифицированная 3-фолдовая с перемешиванием)
    skf = StratifiedKFold(n_splits=3, shuffle=True, random_state=r_state)

    # проводим кросс-валидацию
    score = cross_val_score(
        estimator=pipeline,
        X=X_train,
        y=y_train,
        scoring='roc_auc',
        cv=skf,
        n_jobs=-1
    )

    # возвращаем результаты, которые записываются в Trials()
    return {'loss': -score.mean(), 'params': params, 'status': STATUS_OK}

```

Укажем объект для сохранения истории поиска (Trials). Подбор выполняется с помощью функции fmin. Укажем в качестве алгоритма поиска tpe.suggest – байесовскую оптимизацию (для случайногопоиска – tpe.rand.suggest)

```

# запускаем hyperopt
trials = Trials()
best = fmin(
    # функция для оптимизации
    fn=partial(objective, pipeline=model, X_train=X, y_train=y, r_state=RS),
    # пространство поиска гиперпараметров
    space=search_space,
    # алгоритм поиска
    algo=tpe.suggest,
    # число итераций
    # (можно ещё указать и время поиска)
    max_evals=40,
    # куда сохранять историю поиска
    trials=trials,
    # random state
    rstate=np.random.RandomState(RS),

```

```
# progressbar
show_progressbar=True
)
```

Результаты можно посмотреть в `trails.results`. С hyperopt можно одновременно исследовать несколько моделей. Пример для трех классификаторов – логистической регрессии, дерева решений и метода ближайших соседей – выглядит так

```
search_space = {
    'clf_type': hp.choice(
        label='clf_type',
        options=[

            { # логистическая регрессия
                'type': 'lr',
                'params':{
                    'penalty': hp.choice(
                        label='penalty',
                        options=['l1', 'l2']
                    ),
                    'C': hp.loguniform(
                        label='C',
                        low=-4*np.log(10),
                        high=2*np.log(10)
                    )
                }
            },
            { # дерево решений
                'type': 'dt',
                'params':{
                    'max_depth': scope.int(
                        hp.uniform(
                            label='max_depth',
                            low=4,
                            high=15
                        )
                    ),
                    'max_features': hp.uniform(
                        label='max_features',
                        low=0.25,
                        high=0.9
                    )
                }
            },
            { # k ближайших соседей
                'type': 'knn',
                'params':{
                    'n_neighbors': scope.int(
                        hp.uniform(
                            label='n_neighbors',
                            low=10,
                            high=31
                        )
                    ),
                    'p' : scope.int(
                        hp.uniform(
                            label='p',
                            low=1,
                            high=5
                        )
                    )
                }
            },
        ]
    )
}
```

```
    }
}
])}
```

Модифицируем целевую функцию

```
def build_model(parameters, feat_transformer, r_state):
    """
Формируем модель с указанными гиперпараметрами

:parameters: параметры
:feat_transformer: column transformer для модели
:r_state: random state
:return: модель
"""

if parameters['clf_type']['type'] == 'lr':
    classifier = LogisticRegression(
        random_state=r_state, solver='liblinear',
        **parameters['clf_type']['params']
    )

elif parameters['clf_type']['type'] == 'dt':
    classifier = DecisionTreeClassifier(
        random_state=r_state,
        **parameters['clf_type']['params']
    )

elif parameters['clf_type']['type'] == 'knn':
    classifier = KNeighborsClassifier(
        n_jobs=-1,
        **parameters['clf_type']['params']
    )
else:
    raise KeyError('Unknown classifier: {}'.format(parameters['clf_type']['type']))

# pipeline с трансформером для признаков и классификатором
model = Pipeline([
    ('transformer', feat_transformer),
    ('clf', classifier)
])
return model

# target function
def objective(parameters, feat_transformer, X_train, y_train, r_state):
    """
Кросс-валидация с текущими гиперпараметрами

:param params: гиперпараметры
:feat_transformer: column transformer для модели
:X_train: матрица признаков
:y_train: вектор меток объектов
:r_state: random state
:return: средняя точность на кросс-валидации
"""

model = build_model(parameters=parameters, feat_transformer=feat_transformer, r_state=r_state)
skf = StratifiedKFold(
    n_splits=3, shuffle=True, random_state=r_state)
score = cross_val_score(estimator=model, X=X_train, y=y_train,
cv=skf, scoring='roc_auc', n_jobs=-1)
```

```
return {'loss': -score.mean(), 'params': parameters['clf_type']['params'],
       'clf':parameters['clf_type']['type'], 'status': STATUS_OK}
```

Запускаем подбор

```
trials = Trials() # для записи истории поиска
best = fmin(
    fn=partial(
        objective,
        feat_transformer=transformer,
        X_train=X,
        y_train=y,
        r_state=RS
    ),
    space=search_space,
    algo=tpe.suggest, # байесовская оптимизация
    max_evals=40,
    trials=trials,
    rstate=np.random.RandomState(RS),
    show_progressbar=True
)
```

## 73. Отбор признаков. Очень простые приемы

Существует три основные стратегии отбора признаков:

- одномерные статистики,
- отбор на основе модели,
- итеративный отбор.

Встречает еще такая классификация

- фильтры (filter methods),
- встроенные методы (embedded methods),
- обертки (wrapper methods).

Методы *фильтрации* применяются *до* обучения модели и, как правило, имеют низкую стоимость вычислений. К ним можно отнести визуальный анализ (например, удаление признака, у которого только одно значение, или большинство значений пропущено), оценку признаков с помощью какого-нибудь статистического критерия (корреляция,  $\chi^2$  и пр.) и экспертную оценку (удаление признаков, которые не подходят по смыслу, или признаков с некорректными значениями).

Простейшим способом оценки пригодности признаков является разведочный анализ данных (например, с помощью библиотеки `pandas-profiling`).

Более сложные методы автоматического отбора признаков реализованы в `sklearn`. `VarianceThreshold` отбирает признаки, у которых дисперсия меньше заданного значения. `SelectKBest` и `SelectPercentile` оценивают взаимосвязь предикторов с целевой переменной, используя статистические тесты, позволяя отобрать соответственно заданное количество и долю лучших по заданному критерию признаков. В качестве статистических тестов используются F-тест и взаимная информация.

Методы, основанные на F-тестах, оценивают степень *линейной* зависимости между двумя случайными величинами. Методы на базе *взаимной информации* могут фиксировать *любой вид статистической зависимости*, но будучи непараметрическими, требуют большего объема набора данных.

F-тест реализован в sklearn как `f_regression` и `f_classif` соответственно для *регрессии* и *классификации*, а  $\chi^2$  используется для *классификации* и оценивает зависимость между признаками и классами целевой переменной (требует неотрицательных и правильно отмасштабированных признаков).

*Взаимная информация* (Mutual Information) является мерой взаимной зависимости двух дискретных случайных величин

$$MI(Y, X) = \sum_{y_j \in Y} \sum_{x_i \in X} p(x_i, y_j) \log \frac{p(x_i, y_j)}{p(x_i)p(y_j)}$$

Сводка:

- для задач регрессии: `f_regression`, `mutual_info_regression`,
- для задач классификации: `f_classif`, `chi2`, `mutual_info_classif`.

**ВАЖНО:** *взаимная информация* является мерой взаимной зависимости целевой переменной и рассматриваемого признака. Не ограничиваясь вещественно-значными случайными величинами и линейными зависимостями, как коэффициент корреляции Пирсона, взаимная информация определяет насколько совместное распределение пары  $(X, Y)$  отличается от произведения маргинальных распределений  $X$  и  $Y$ . Этот тип тестов считается самым удобным в использовании – хорошо работает «из коробки» и позволяет находить нелинейные зависимости.

Встроенные методы выполняют отбор признаков во время обучения модели, оптимизируя их набор для достижения лучших значений метрик качества. К этим методам можно отнести регуляризацию в линейных моделях (обычно L1) и расчёт важности признаков в алгоритмах с деревьями. Отметим, что для линейных моделей требуется масштабирование и нормализация данных.

Выходы:

- К достоинствам фильтров можно отнести низкую стоимость вычислений (линейно зависит от количества признаков) и интерпретируемость. К недостаткам – то, что они рассматривают *каждый признак изолировано*, поэтому не могут выявить более сложные зависимости в данных, например, зависимость от нескольких предикторов. Эти методы хорошо подойдут, если в данных большое количество признаков, но малое количество объектов (что встречается, например, в медицинских, или биологических исследованиях),
- Встроенные методы, в отличие от фильтров, требуют больших вычислительных ресурсов, а так же более точной настройки и подготовки данных. Однако, эти методы могут выявить уже более сложные зависимости. Для менее сменёной интерпретации коэффициентов признаков необходимо настроить регуляризацию модели. Важно помнить, что распределение коэффициентов у линейных моделей зависит от способа предобработки данных.

## 74. Важность признаков

Важность признаков – числовые оценки, насколько каждый признак важен для решения поставленной задачи. Влияет на полученный результат, обеспечивает качество решения задачи.

С помощью *одномерных статистик* мы определяем наличие статистически значимой взаимосвязи между каждым признаком и зависимой переменной. Затем отбираем признаки, сильнее всего связанные с зависимой переменной (имеющие достигаемый уровень значимости *p-value*, не превышающий заданного порогового значения). В случае *классификации* эта процедура известна

как *дисперсионный анализ* (ANOVA). Ключевым свойством этих тестов является то, что они являются *одномерными*, то есть они рассматривают каждую характеристику по отдельности (и не учитывают взаимодействие признаков). Следовательно признак будет исключен, если он становится информативным лишь в сочетании с другим признаком. Как правило, одномерные тесты очень быстро вычисляются и не требуют построения модели. С другой стороны, они являются полностью независимыми от модели, которой вы, возможно, захотите применить после отбора признаков.

**ВАЖНО:** L1-регуляризацию можно рассматривать как способ отбора признаков.

*Отбор признаков на основе модели* использует модель машинного обучения с учителем, чтобы вычислить важность каждого признака, и оставляет только самые важные из них. Модель машинного обучения с учителем, которая используется для отбора признаков, не должна использоваться для построения итоговой модели. Модель, применяющаяся для отбора признаков, требует вычисления определенного показателя важности для всех признаков, с тем чтобы характеристики можно было ранжировать по этой метрике.

В отличие от одномерного отбора *отбор на основе модели* рассматривает все признаки сразу и поэтому может обнаружить взаимодействие (если модель способна выявить их).

В одномерном отборе признаков мы не использовали модель, а в отборе признаков на основе модели мы построили одну модель, чтобы выбрать характеристики. В *итеративном отборе признаков* строится последовательность моделей с различным количеством признаков. Существует два основных метода. Первый метод начинается шага, когда в модель включена лишь одна константа (входных признаков нет) и затем добавляет признак за признаком до тех пор, пока не будет достигнут критерий остановки. Второй метод начинается с шага, когда все признаки включены в модель, и затем начинает удалять признак за признаком, пока не будет достигнут критерий остановки. Поскольку строится последовательность модели, эти методы с вычислительной точки зрения являются гораздо более затратными в отличие от ранее обсуждавшихся методов.

Рекомендации:

- Нельзя увлекаться выбрасыванием «неважных» признаков: оценка качества признаков не всегда адекватная; если много хороших признаков кррелированы друг с другом, то их важность будет низкой; не рекомендуется оценивать важность и решать одним и тем же алгоритмом,

*Информационный прирост* при построении расщепления в решающих деревьях определяется по формуле [10, 122]

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N_p} I(D_j),$$

где  $f$  – признак для выполнения разбиения,  $D_p$  и  $D_j$  - набор данных родительского и  $j$ -ого дочернего узла,  $I$  – мера загрязненности (либо загрязненность Джини, либо энтропия Шенонна),  $N_p$  – общее количество образцов в родительском узле,  $N_j$  – количество образцов в  $j$ -ом дочернем узле.

В случае бинарных деревьев принятия решения каждый родительский узел разделяется на два дочерних – левый  $D_{left}$  и правый  $D_{right}$

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}).$$

Любопытно, что *информационный прирост* может интерпретироваться как *важность по неоднородности*. Важность по неоднородности (impurity-based importance) еще называют Gini importance (в sklearn), IncNodePurity (в R) и MDI (mean decrease impurity), когда усредняют по деревьям.

### Энтропия

$$I_H(t) = - \sum_{i=1}^c p(i|t) \log_2 p(i|t),$$

где  $p(i|t)$  – доля образцов, которые принадлежать классу  $i$  для индивидуального узла  $t$ . Следовательно энтропия равна нулю, когда все образцы в узле принадлежать одному и тому же классу, и максимальна при равномерном распределении классов. Таким образом, можно говорить, что энтропия стремится довести до максимума полное количество информации в дереве.

### Заргязненность Джини

$$I_G(t) = \sum_{i=1}^c p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2.$$

#### *Важность по неоднородности:*

- (+) автоматически вычисляется при построении деревьев (реализован в sklearn),
- (-) годится только для деревьев и ансамблей на основании деревьев,
- (-) имеет смещённость в сторону признаков с большим числом значений, также в случае разномасштабности признаков.

*Перестановочная важность* (Permutation Feature Importance) – признак важный, если перестановка его значений снижает качество:

- (+) не требуется переобучать модель – смотрим как меняется качество на отложенной выборке,
- (+) перестановка не меняет распределение значений,
- (+) можно применять на любых алгоритмах,
- (+) самый надежный метод,
- (-) очень медленный.

#### Особенности перестановочной важности:

- если есть несколько сильно коррелированных признаков, то перестановка значений одного из них может слабо влиять на качество решения: вариант решения – кластеризация признаков (по функции схожести) и формирование признакового пространства из представителей кластеров,
- оценка зависит от перестановки: на практике делают несколько перестановок (такая реализация становится медленнее обычного варианта с одинарной перестановкой),
- метод универсальный, но зависит от конкретной модели и критерия качества.

## 75. Интерпретация моделей и оценка важности признаков с библиотекой SHAP

### 75.1. Общие сведения о значениях Шепли

В библиотеке SHAP <https://github.com/slundberg/shap> для оценки важности признаков используются *значения Шепли*<sup>44</sup> (Shapley value) [https://en.wikipedia.org/wiki/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*<sup>45</sup>  $i$ -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)

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} \underbrace{\left( v(S \cup \{i\}) - v(S) \right)}_{f_i\text{-contribution}},$$

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](#))

---

### 75.2. 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](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
```

<sup>44</sup>Термин пришел из теории кооперативных игр

<sup>45</sup>Еще эту оценку можно интерпретировать как *вклад*

```

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'] # нумеру-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

```

### 75.2.1. Локальная интерпретация отдельной точки данных обучающего набора

Теперь можно построить локальную интерпретацию для одной точки данных из обучающего набора (см. рис. 50)

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()`.

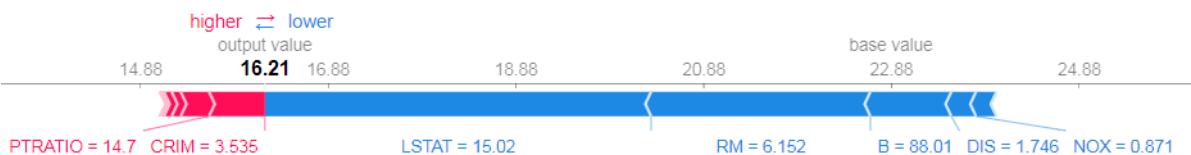


Рис. 50. Локальная интерпретация для одной точки данных обучающего набора

Еще можно построить график частичной зависимости (рис. 51)

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

### 75.2.2. 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)

```

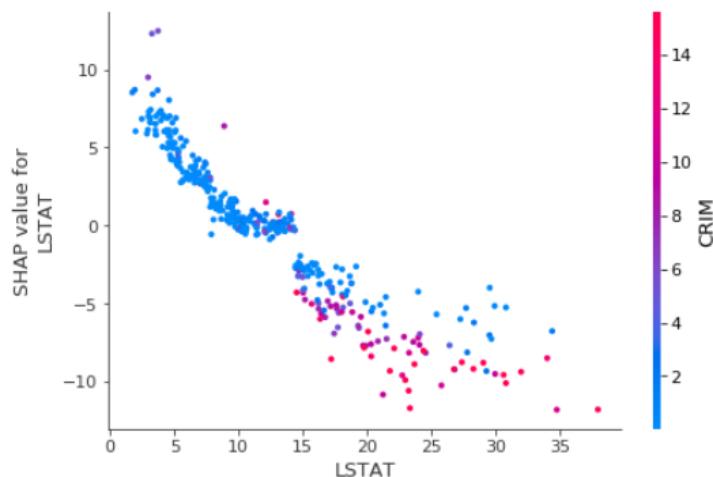


Рис. 51. График частичной зависимости признака LSTAT от значений Шепли с учетом влияния признака CRIM

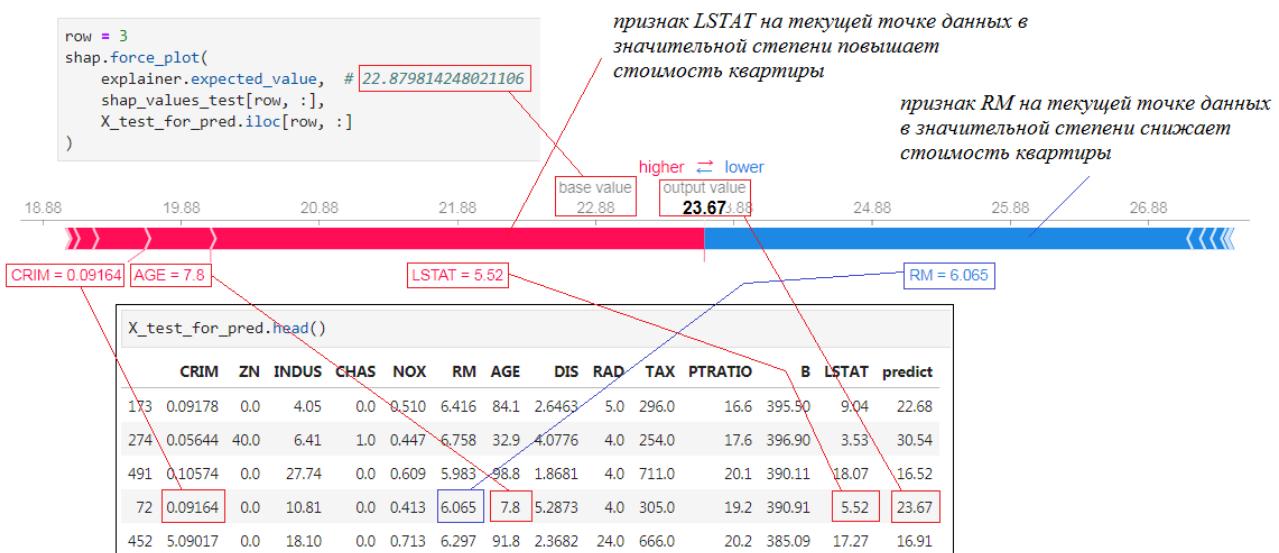


Рис. 52. Локальная интерпретация для одной точки данных тестового набора

```

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. 52).

Из рис. 52 видно, что признаки с различной «силой»<sup>46</sup>, которая определяется значениями Шепли, смещают предсказание модели на данной точке. Например, признак LSTAT (процент населения с низким социальным статусом) в значительной степени *повышает*<sup>47</sup> 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

<sup>46</sup>Ширина полосы

<sup>47</sup>Потому что значение этого признака невелико; чем меньше процент населения с низким социальным статусом проживает в округе, тем выше стоимость квартиры

```

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, :]
)

```

### 75.2.3. Глобальная интерпретация модели на тестовом наборе данных

Удобно работать с диаграммой рассеяния `shap.summary_plot` (рис. 53), 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)
```

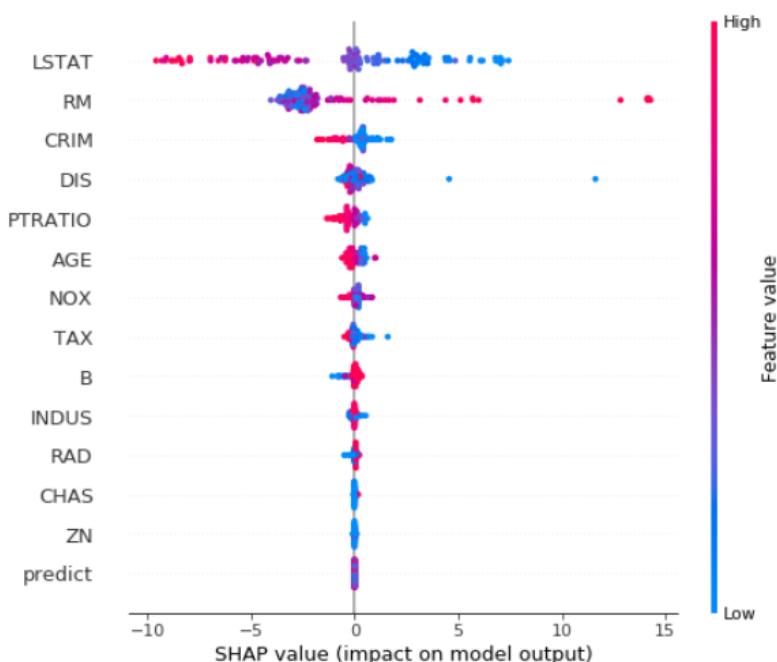


Рис. 53. Диаграмма рассеяния для точек тестового набора данных

Какие выводы можно сделать из рис. 53:

- 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. 54).

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

Какие выводы можно сделать из рис. 54:

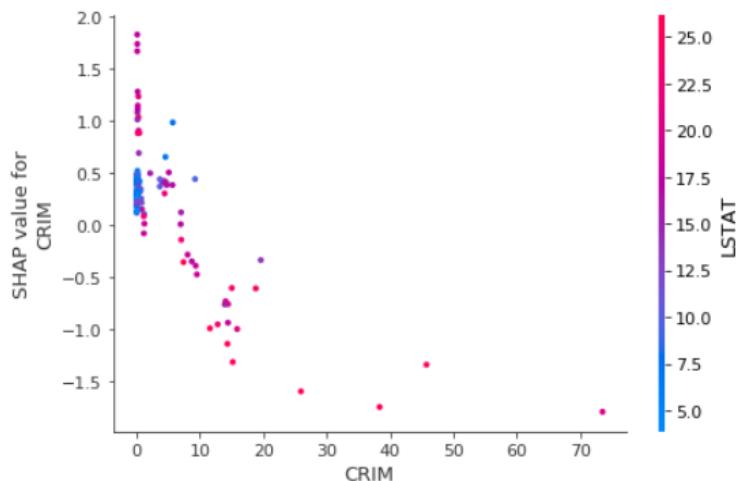


Рис. 54. График частичной зависимости признака CRIM от значений Шепли с учетом влияния LSTAT

- 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.

## 76. 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 перестановочные
# важности признаков

```

## 77. Приемы работы с библиотекой River

River <https://riverml.xyz/dev/> – библиотека динамического машинного обучения.

## 78. Регулярные выражения в Python

В языке Python есть несколько тонких особенностей, связанных с регулярными выражениями, а именно 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'

```

## 79. 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  $X : \Omega \rightarrow \mathbb{R}^+$  определена на вероятностном пространстве  $(\Omega, F, \mathbb{P})$ , и ее математическое ожидание  $\mathbb{E}X$  конечно. Тогда

$$\mathbf{P}(X \geq a) \leq \frac{\mathbb{E}X}{a}, \quad a > 0.$$

Пример. Пусть в среднем ученики опаздывают на 3 минуты (оценка математического ожидания), и нас интересует какова вероятность того, что ученик опаздывает на 15 и более минут. Чтобы получить грубую оценку сверху

$$\mathbf{P}(|X| \geq 15) \leq \frac{3}{15} = 0,2.$$

## 80. Асинхронное программирование в Python

### 80.1. 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)

# ИО 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 параллельно. '''
    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(mdt_hash.hexdigest().encode())

```

Декораторы `@threaded` делают из обычной функции *awaitable-объект*. Нельзя выполнить функцию, обернутую `@threaded` из функции, обернутой `@threaded`.

## 81. Приемы работы с DVC

С основами использования инструмента DVC можно познакомиться по статье <https://prolib.io/p/git-dlya-data-science-kontrol-versiy-modeley-i-datasetov-s-pomoshchyu-dvc-2020-12-02>.

## 82. 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('postgresql://postgres@localhost:5432/demo')
cur_pg = conn_pg.cursor()
# возвращаем название источника данных в формате строки
conn_pg.dsn # 'postgresql://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`<sup>48</sup> служит для синхронного выполнения сопрограмм.

По рекомендациям разработчиков `asyncio`

- следует использовать `asyncio.run()`, а цикл не нужен!,
- должна быть одна 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)'
```

<sup>48</sup>Запускает сопрограмму из асинхронного кода с явным переключением контекста

```

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 параллельно """
    # 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'))
    # 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 выполняются параллельно!!!
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`<sup>49</sup> <https://github.com/vertica/vertica-python>

```

import 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': 'vlcluster.example.com'
}

with vertica_python.conn(**conn_info) as conn:

```

<sup>49</sup> Устанавливается как обычно с помощью менеджера пакетов pip: `pip install vertica-python`

```
# 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')
    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`<sup>50</sup>

```
# 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<sup>51</sup> <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`<sup>52</sup>

<sup>50</sup> Устанавливается с помощью менеджера пакетов pip: `pip install clickhouse-driver`

<sup>51</sup> Существует соответствующая python-библиотека `neo4j` <https://neo4j.com/developer/python/>. Используется собственные язык запросов Cypher, но поддерживается и Gremlin

<sup>52</sup> Устанавливается как обычно с помощью менеджера пакетов pip: `pip install gremlinpython`

```
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

...  
...
```

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).

## 83. 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.

## 84. Приемы работы с flake8

flake8 – инструмент, позволяющий выявить стилистические ошибки в коде.

Установить 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
```

## 85. 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<sup>53</sup>

pyproject.toml

```
[tool.black]  
line-length = 80  
target-version = ['py37']  
include = '\.pyi?$'  
exclude = ''
```

<sup>53</sup>TOML – формат конфигурационных файлов

```
(  
  \/  
    \.eggs  
    | \.git  
    | \.hg  
    | \.mypycache  
    | \.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`<sup>54</sup> <https://pre-commit.com>. Нужно просто указать список хуков, а `pre-commit` будет запускать хук перед каждым коммитом.

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  
# вывождение  
[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.

<sup>54</sup>Установить можно как обычно `pip install pre-commit`

## 86. 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<sup>55</sup> <https://pypi.org/project/folium/> (предпочтнее отдаётся этой библиотеке).

Для развертывания приложения, созданного с помощью библиотеки `folium` на облачной платформе `Streamlit` требуется использовать библиотеку `streamlit_folium`<sup>56</sup> <https://github.com/randyzwitche/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 (ещё 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

```
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

1	Пример работы фильтра Блума . . . . .	17
2	Операции и время их выполнения, поддерживаемые кучами: где $n$ – это число объектов, хранящихся в куче . . . . .	19
3	Пример кучи в виде дерева с 9 объектами . . . . .	20

<sup>55</sup>Устанавливается как обычно `pip install folium`

<sup>56</sup>Установить можно так: `pip install streamlit-folium` или `conda install -c conda-forge streamlit-folium`

4	Отображение древовидного представления кучи на его представление в виде массива . . . . .	20
5	взаимосвязи между позицией $i \in \{1, 2, \dots, n\}$ объекта в куче и позициями его родителя, левого и правого потомка; $n$ обозначает число объектов в куче . . . . .	20
6	Простое бинарное дерево поиска . . . . .	22
7	Отсортированные массивы: $n$ – текущее число объектов в массиве . . . . .	22
8	<i>Сбалансированные</i> деревья поиска и отсортированные массивы: $n$ – текущее число объектов в структуре данных . . . . .	23
9	Дерево поиска . . . . .	24
10	Дерево поиска и соответствующие ему указатели на родителя и потомков . . . . .	24
11	Диаграмма Венна взаимоотношений между классами P, NP, NP-полными и NP-трудными . . . . .	27
12	Связка NGINX - gunicorn - Python-webapp . . . . .	82
13	Страница html-отчета о покрытии кода тестами . . . . .	115
14	Окно командной оболочки cmd.exe со списком доступных каналов, по которым будет проводиться поиск пакета xgboost . . . . .	134
15	График важности признаков xgboost.plot_importance(model), построенный с помощью пакета xgboost . . . . .	135
16	График важности признаков xgboost.plot_importance(model, importance_type='cover'), построенный с помощью пакета xgboost . . . . .	135
17	График важности признаков xgboost.plot_importance(model, importance_type='gain'), построенный с помощью пакета xgboost . . . . .	136
18	Простой набор данных . . . . .	136
19	Базовое/начальное значение прогноза . . . . .	137
20	Остатки, как разность наблюдаемых значений и значения прогноза . . . . .	137
21	Вычисление сходства . . . . .	138
22	Вычисление сходства для родительского узла дерева . . . . .	138
23	Первый этап процедуры вычисления порога для признака Drug Dosage . . . . .	138
24	Первое разбиение значений признака Drug Dosage . . . . .	139
25	Вычисление сходства для дочерних узлов . . . . .	139
26	Вычисление прироста . . . . .	139
27	Следующее разбиение признака Drug Dosage . . . . .	140
28	Дерево решения для нового порога разбиения признака Drug Dosage . . . . .	140
29	Наилучшее разбиение признака Drug Dosage по значению прироста . . . . .	141
30	Начало процедуры поиска следующего лучшего разбиения для признака Drug Dosage . . . . .	141
31	Вычисление прироста для второго разбиения признака Drug Dosage . . . . .	141
32	Построение второго уровня дерева решений. Для простоты ограничиваемся двумя уровнями и не будем строить разбиение для левого узла . . . . .	142
33	Выходы по листьям дерева . . . . .	142
34	Процедура построения прогноза в XGBoost для значения признака Dosage=10 . . . . .	143
35	Остаток на точке Dosage=10 стал меньше . . . . .	143
36	Остаток на точке Dosage=20 стал меньше . . . . .	143
37	Схема, описывающая связи между именами функций и их объектами . . . . .	154

38	К вопросу о механизме работы декоратора с вложенной функцией . . . . .	172
39	Стандартная процедура кросс-валидации . . . . .	186
40	Вложенная перекрестная проверка на временном ряде . . . . .	187
41	Пример детектирования аномалий на тестовой наборе данных . . . . .	191
42	Пример использования библиотеки <code>fbs prophet</code> . . . . .	194
43	Влияние преобразования Бокса-Кокса на временной ряд с изменяющейся во времени дисперсией . . . . .	196
44	Пример хранилища, построенного с помощью подхода <code>Data Vault</code> . . . . .	198
45	Высокоуровневое представление хранилища данных . . . . .	199
46	Пример Структура толерантная к изменению кардинальности существующих связей . . . . .	201
47	Сводка по гибким методологиям . . . . .	204
48	Отформатированный вывод <code>DataFrame</code> . . . . .	214
49	Результат применения функции <code>color_code_freq_cat</code> . . . . .	214
50	Локальная интерпретация для одной точки данных обучающего набора . . . . .	226
51	График частичной зависимости признака <code>LSTAT</code> от значений Шепли с учетом влияния признака <code>CRIM</code> . . . . .	227
52	Локальная интерпретация для одной точки данных тестового набора . . . . .	227
53	Диаграмма рассеяния для точек тестового набора данных . . . . .	228
54	График частичной зависимости признака <code>CRIM</code> от значений Шепли с учетом влияния <code>LSTAT</code> . . . . .	229

## 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. *Элbon K.* Машинное обучение с использованием Python. Сборник рецептов. – СПб.: БХВ-Петербург, 2019. – 384 с.
9. *Karay X., 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.
12. *Раффарден, Т.* Совершенный алгоритм. Основы. – СПб.: Питре, 2019, – 256 с.

13. Рафгарден, Т. Совершенный алгоритм. Графовые алгоритмы и структуры данных. – СПб.: Питре, 2019, – 256 с.
14. Хайнеман, Дж., Поллис, Г., Селков, С. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб.: ООО «Альфа-книга», 2017. – 432 с.
15. Скиена С. Алгоритмы. Руководство по разработке. – СПб.: БХВ-Петербург, 2011. – 720 с.
16. Кормен Т. Алгоритмы: построение и анализ. – М.: ООО «И.Д. Вильямс», 2013. – 1328 с.