Машинное обучение

Дополнительные материалы

Материалы > Анализ и обработка данных

Тест Литтла для выявления MCAR

Для того чтобы количественно определить полностью случайным ли образом сформированы пропущенные значения (MCAR), существует тест Литтла (Little's test). Этот критерий был предложен в 1988 году Родериком Литтлом в качестве единого критерия оценки случайности пропущенных значений в многомерных *количественных* (!) данных.

> **Нулевая гипотеза** этого критерия утверждает, что пропуски полностью случайны (MCAR). **Альтернативная гипотеза** говорит о том, что пропуски зависят от наблюдаемых значений (MAR).

Создадим новый блокнот на R в Google Colab и воспользуемся соответствующей функцией одной из библиотек этого языка.

Откроем блокнот с кодом на R

Датасет airquality

Вначале попрактикуемся на встроенном в R датасете airquality.

```
# импортируем библиотеку datasets
library(datasets)
```

Посмотрим на первые строки датафрейма с помощью **функции head()**.

```
head(airquality)
```

A data.frame: 6 × 6

	Ozone	Solar.R	Wind	Temp	Month	Day
	<int>	<int>	<dbl>	<int>	<int>	<int>
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5

6	28	NA	14.9	66	5	6

Оценим размерность.

```
# выведем общее количество строк и столбцов
dim(airquality)
```

```
153 · 6
```

Посмотрим на общие статистические показатели с помощью **функции summary()**.

```
summary(airquality)
```

```
     Ozone           Solar.R          Wind             Temp      
 Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00  
 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00  
 Median : 31.50   Median :205.0   Median : 9.700   Median :79.00  
 Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88  
 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00  
 Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00  
 NA's   :37       NA's   :7                                       
     Month            Day      
 Min.   :5.000   Min.   : 1.0  
 1st Qu.:6.000   1st Qu.: 8.0  
 Median :7.000   Median :16.0  
 Mean   :6.993   Mean   :15.8  
 3rd Qu.:8.000   3rd Qu.:23.0  
 Max.   :9.000   Max.   :31.0  
```

Теперь установим и импортируем библиотеку **naniar**, которая и содержит необходимый нам тест Литтла.

```
# установим библиотеку
install.packages('naniar')
```

```
# импортируем ее
library(naniar)
```

Посмотрим на абсолютное количество и процент пропусков в каждом из столбцов.

```
miss_var_summary(airquality)
```

A tibble: 6 × 3

variable	n_miss	pct_miss
<chr>	<int>	<dbl>
Ozone	37	24.183007
Solar.R	7	4.575163
Wind	0	0.000000
Temp	0	0.000000
Month	0	0.000000
Day	0	0.000000

Перейдем к статистическому тесту.

```
# для теста Литтла используем функцию mcar_test()
mcar_test(airquality)
```

A tibble: 1 × 4

statistic	df	p.value	missing.patterns
<dbl>	<dbl>	<dbl>	<int>
35.10613	14	0.001417781	4

Вероятность (p-value) очень мала, ниже порогового значения, например, в пять или даже один процент, и мы можем отвергнуть нулевую гипотезу о полностью случайных пропусках.

Кроме этого, **функция mcar_test()** сообщает о четырех выявленных в пропущенных данных паттернах.

Вернемся к датасету, с которым работали до сих пор.

Датасет «Титаник»

Подгрузим файл train.csv в сессионное хранилище блокнота на R.

train.csv **Скачать**

Импортируем его с помощью **функции read.csv()**.

```
titanic = read.csv('/content/train.csv', na.strings = c("NA", "NaN", ""))
head(titanic)
```

A data.frame: 6 × 12

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
	<int>	<int>	<int>	<chr>	<chr>	<dbl>	<int>	<int>	<chr>	<dbl>	<chr>	<chr>
1	1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.2500	NA	S
2	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.9250	NA	S
4	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1000	C123	S
5	5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.0500	NA	S
6	6	0	3	Moran, Mr. James	male	NA	0	0	330877	8.4583	NA	Q

Обратите внимание, в параметр na.strings мы передали вектор с возможными вариантами записи пропущенных значений. Без этого параметра часть пропусков не была бы учтена.

Посмотрим на количество и процент пропусков в каждом столбце.

```
miss_var_summary(titanic)
```

A tibble: 12 × 3

variable	n_miss	pct_miss
<chr>	<int>	<dbl>
Cabin	687	77.1043771
Age	177	19.8653199
Embarked	2	0.2244669
PassengerId	0	0.0000000
Survived	0	0.0000000

Survived	0	0.0000000
Pclass	0	0.0000000
Name	0	0.0000000
Sex	0	0.0000000
SibSp	0	0.0000000
Parch	0	0.0000000
Ticket	0	0.0000000
Fare	0	0.0000000

Проведем тест Литтла.

```
1 mcar_test(titanic)
```

A tibble: 1 × 4

statistic	df	p.value	missing.patterns
<dbl>	<dbl>	<dbl>	<int>
598.5422	43	0	5

В данном случае вероятность (p-value) наблюдать такие пропуски при условии, что нулевая гипотеза верна, вообще равна нулю, и у нас появляется ещё больше оснований отвергнуть предположение о полностью случайных пропусках.

Функция сообщает, что в данных выявлено пять паттернов и собственно именно этими зависимостями между пропусками и наблюдаемыми данными мы и пользовались в многомерных способах заполнения пропусков.

Временная сложность алгоритма

Откроем блокнот с кодом на Питоне

Сравнение алгоритмов

Важность сравнения эффективности алгоритмов очевидна. Менее очевидным является способ их сравнения.

С одном стороны, мы могли бы замерять *фактическое время* работы алгоритма, однако такое измерение очень сильно зависело бы, в частности, от характеристик конкретного компьютера и в конечном счете не имело бы смысла.

Более удобно измерять работу алгоритма в *количестве операций*, необходимых для выполнения задачи.

Линейный и бинарный поиск

Вернёмся к рассмотрению алгоритмов линейного и бинарного поиска. Вначале приведем соответствующие функции.

```
def linear(arr, x):

  # объявим счетчик количества операций
  counter = 0

  for i in range(len(arr)):

    # с каждой итерацией будем увеличивать счетчик на единицу
    counter += 1

    if arr[i] == x:
      return i, counter
```

```
def binary(arr, x):

  # объявим счетчик количества операций
  counter = 0

  low, high = 0, len(arr) - 1

  while low <= high:

    # увеличиваем счетчик с каждой итерацией цикла
    counter += 1

    mid = low + (high - low) // 2

    if arr[mid] == x:
        return mid, counter

    elif arr[mid] < x:
        low = mid + 1

    else:
        high = mid - 1

  return -1
```

Эти алгоритмы аналогичны рассмотренным ранее, за исключением того, что в данном случае мы также будем считать количество операций. Под операцией в контексте поиска мы будем понимать сравнение искомого числа с очередным элементом массива.

Пусть даны два отсортированных массива из восьми и шестнадцати чисел.

```
arr8 = np.array([3, 4, 7, 11, 13, 21, 23, 28])
arr16 = np.array([3, 4, 7, 11, 13, 21, 23, 28, 29, 30, 31, 33, 36, 37, 39, 42])

len(arr8), len(arr16)
```

```
(8, 16)
```

Найдем в этих массивах индекс чисел 28 и 42 соответственно. Алгоритм линейного поиска ожидаемо справится за восемь и шестнадцать операций.

```
# первым результатом функции будет индекс искомого числа,
# вторым - количество операций сравнения
linear(arr8, 28), linear(arr16, 42)
```

```
((7, 8), (15, 16))
```

Заметим, что это **худший случай** из возможных (worst case scenario), искомые числа

оказались на самом неудачном месте.

Алгоритму же бинарного поиска потребуется всего лишь четыре и пять операций соответственно.

```
binary(arr8, 28), binary(arr16, 42)
```

```
((7, 4), (15, 5))
```

Из приведенного выше кода следует, что алгоритмы не только различаются по количеству затрачиваемых операций, но и рост количества вычислений с ростом объема данных происходит не одинаково.

> Количество операций (k) линейного поиска растет пропорционально количеству данных (n), т.е. $k = n$, это *линейная зависимость*, в бинарном же поиске *зависимость логарифмическая* $k = \log(n)$.

Сравним рост количества операций по мере роста объема данных.

```
# посчитаем количество операций для входных массивов разной длины
# создадим списки, куда будем записывать количество затраченных итераций
ops_linear, ops_binary = [], []

# будет 100 входных массивов длиной от 1 до 100 элементов
input_arr = np.arange(1, 101)

# на каждой итерации будем работать с массивом определенной длины
for i in input_arr:

  # внутри функций поиска создадим массив из текущего количества элементов
  # и попросим найти последний элемент i - 1
  _, l = linear(np.arange(i), i - 1)
  _, b = binary(np.arange(i), i - 1)

  # запишем количество затраченных операций в соответствующий список
  ops_linear.append(l)
  ops_binary.append(b)
```

Выведем результат на графике.

```
plt.plot(input_arr, ops_linear, label = 'Линейный поиск')
plt.plot(input_arr, ops_binary, label = 'Бинарный поиск')

plt.title('Зависимость количества операций поиска от длины массива')
plt.xlabel('Длина входного массива')
plt.ylabel('Количество операций в худшем случае')

plt.legend();
```

Вполне понятно, почему в алгоритме линейного поиска количество операций растет пропорционально объему данных. Выясним, почему количество операций бинарного поиска имеет логарифмическую зависимость.

Логарифмическая зависимость бинарного поиска

На каждом этапе мы делим данные пополам и выполняем операцию сравнения. Например, массив из 16-ти чисел мы разделим четыре раза.

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \text{т.е.} \frac{n}{2^k}$$

В нашем случае,

$$\frac{16}{2^k}$$

Выполнять такое деление мы будем до тех пор, пока количество чисел не будет равно одному (т.е. мы не найдем нужное число).

$$\frac{n}{2^k} = 1, 2^k = n$$

Тогда k (количество операций) будет равно

$$k = \log_2(n)$$

В нашем случае,

$$4 = \log_2(16)$$

Нотация «О» большого

Такая оценка называется **временной сложностью** (time complexity) алгоритма и обычно выражается в **О-символике** или **нотации «О» большого** (big-O notation). Для приведенных выше алгоритмов запись будет выглядеть так:

- линейный поиск: $O(n)$;
- бинарный поиск: $O(\log(n))$.

Повторюсь, что учитывается количество операций в худшем случае. При желании, можно сравнить и наиболее благоприятный сценарий (best case scenario). Тогда,

- для алгоритма линейного поиска его сложность в лучшем случае (если искомое значение стоит на первом месте) составит $O(1)$, то есть потребуется одна операция;

- при бинарном поиске, если искомое число находится, что наиболее удачно, в середине отсортированного массива, его сложность также составит $O(1)$.

Основание логарифма обычно не приводится поскольку

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Асимптотическое время

Нотация «О» большого отражает так называемое асимптотическое время работы алгоритма.

Асимптотический анализ (asymptotic analysis) изучает поведение функции при стремлении аргумента к определенному значению. В нашем случае мы смотрим на количество операций (поведение функции) при значительном увеличении объема данных (аргумент).

> Скорость изменения функции ещё называют *порядком* изменения, от англ. order или нем. Ordnung, отсюда заглавная буква «О» в нотации.

Таким образом говорят, что в худшем случае линейный поиск соответствует линейному времени, а бинарный — логарифмическому.

Существует также константное время (когда сложность алгоритма не зависит от объема данных, $O(1)$, квадратичное время $O(n^2)$ или, например, факториальное $O(n!)$. В последнем случае, количество операций растет наиболее стремительно.

Про константы

Внимательный читатель вероятно обратил внимание, что формула временной сложности бинарного поиска и фактическое количество операций приведенных алгоритмов не совпадают.

Мы взяли массивы из 8-ми и 16-ти чисел и по формуле, должны были уложиться в $\log_2(8) = 3$ и $\log_2(16) = 4$ операции. Фактически же мы затратили четыре и пять соответственно.

Другими словами сложность алгоритма бинарного поиска в худшем случае составляет $O(\log(n) + 1)$, однако так как нас интересует порядок (примерное понимание) роста функции при росте аргумента, а не точное значение, константы принято опускать.

Аналогично $O(2\log(n))$ сводится к $O(\log(n))$.

Временная сложность в ML

Как применять вычислительную сложность алгоритма на практике? Рассмотрим два примера.

Поиск ближайших соседей по k-мерному дереву существенно быстрее, чем простой

перебор всех векторов сравнения и расчет расстояния:

- во втором случае (brute force) временная сложность равна $O(n, m)$, где n — количество векторов запроса, а m — количество векторов сравнения;
- в первом (kdtree) — $O(n \log(m))$, правда без учета операций для создания k-мерного дерева.

Кроме того, если мы знаем из-за какого компонента данных количество необходимых операций растет наиболее быстро, именно с этого компонента мы и начнем оптимизацию модели.

Например, из документации мы знаем, что **сложность алгоритма IterativeImputer** составляет

$$O(knp^3 \min(n, p))$$

где k — максимальное число итераций, n — количество наблюдений, а p — количество признаков. Таким образом, очевидно, что для снижения количества необходимых операций, в первую очередь нужно работать с признаками.

Еще одно сравнение методов заполнения пропусков

Способ сравнения

Выше мы сравнили способы заполнения пропусков, вначале использовав одномерный метод или многомерную модель, а затем применив алгоритм логистической регрессии в задаче классификации.

Такой подход был выбран, чтобы показать, как работают методы заполнения пропусков в связке с реальной задачей машинного обучения. Для чистого же сравнения этих методов более правильным будет

- взять полный датасет (без пропущенных значений);
- случайным образом изъять часть значений;
- заполнить пропуски различными методами;
- сравнить результат с изъятыми данными.

Создание данных с пропусками

Возьмем уже знакомый нам по вводному курсу датасет, позволяющий классифицировать опухоли на доброкачественные и злокачественные.

```
1 # импортируем данные опухолей из модуля datasets библиотеки sklearn
2 from sklearn.datasets import load_breast_cancer
3
4 # выведем признаки и целевую переменную и поместим их в X_full и _ соответственно
5 X_full, _ = load_breast_cancer(return_X_y = True, as_frame = True)
```

```
6
7 # масштабируем данные
8 X_full = pd.DataFrame(StandardScaler().fit_transform(X_full), columns = X_full.columns)
```

Теперь напишем функцию, которая будет случайным образом добавлять пропуски в выбранные нами признаки.

```
1  # нам понадобится модуль random
2  import random
3
4  # на вход функция будет получать полный датафрейм, номера столбцов признаков,
5  # долю пропусков в каждом из столбцов и точку отсчета
6  def add_nan(x_full, features, nan_share = 0.2, random_state = None):
7
8    random.seed(random_state)
9
10   # сделаем копию датафрейма
11   x_nan = x_full.copy()
12
13   # вначале запишем количество наблюдений и количество признаков
14   n_samples, n_features = x_full.shape
15
16   # посчитаем количество признаков в абсолютном выражении
17   how_many = int(nan_share * n_samples)
18
19   # в цикле пройдемся по номерам столбцов
20   for f in range(n_features):
21     # если столбец был указан в параметре features,
22     if f in features:
23       # случайным образом отберем необходимое количество индексов наблюдений (how_many
24       # из перечня, длиной с индекс (range(n_samples))
25       mask = random.sample(range(n_samples), how_many)
26       # заменим соответствующие значения столбца пропусками
27       x_nan.iloc[mask, f] = np.nan
28
29   # выведем датафрейм с пропусками
30   return X_nan
```

Обратите внимание на один нюанс. **Функция random.sample()**, что удобно, выбирает элемент *без возвращения*, то есть один раз выбрав наблюдение с индексом «один», второй раз она это наблюдение не выберет.

```
1 # выведем пять чисел от 0 до 9
2 random.seed(42)
3 # с функцией random.sample() повторов не будет
4 random.sample(range(10), 5)
```

```
1 [1, 0, 4, 9, 6]
```

Мы могли бы использовать функции **np.random.randint()** или **random.choice()**, однако в этом случае из-за повторов процент пропусков был бы чуть ниже желаемого, например не 20, а 18.

```
1 np.random.seed(42)
2 # выберем случайным образом пять чисел от 0 до 9
3 np.random.randint(0, 10, 5)
```

```
1 array([6, 3, 7, 4, 6])
```

```
1 random.seed(42)
2 # выберем пять случайных чисел от 0 до 9
3 [random.choice(range(10)) for _ in range(5)]
```

```
1 [1, 0, 4, 3, 3]
```

Применим объявленную функцию к датафрейму и создадим 20 процентов пропусков в первом

столбце.

```
X_nan = add_nan(X_full,
                features = [0],
                nan_share = 0.2,
                random_state = 42)
```

Проверим результат.

```
(X_nan.isna().sum() / len(X_nan)).round(2)
```

```
mean radius                0.2
mean texture               0.0
mean perimeter             0.0
mean area                  0.0
mean smoothness            0.0
mean compactness           0.0
mean concavity             0.0
mean concave points        0.0
mean symmetry              0.0
mean fractal dimension     0.0
radius error               0.0
texture error              0.0
perimeter error            0.0
area error                 0.0
smoothness error           0.0
compactness error          0.0
concavity error            0.0
concave points error       0.0
symmetry error             0.0
fractal dimension error    0.0
worst radius               0.0
worst texture              0.0
worst perimeter            0.0
worst area                 0.0
worst smoothness           0.0
worst compactness          0.0
worst concavity            0.0
worst concave points       0.0
worst symmetry             0.0
worst fractal dimension    0.0
dtype: float64
```

Перейдем к заполнению пропусков.

Заполнение пропусков

Заполнение константой

```
# скопируем датасет
fill_const = X_nan.copy()
# заполним пропуски нулем
fill_const.fillna(0, inplace = True)
# убедимся, что пропусков не осталось
fill_const.isnull().sum().sum()
```

```
'0'
```

Заполнение медианой

```
# скопируем датасет
fill_median = X_nan.copy()
# заполним пропуски медианой
# по умолчанию, и .fillna(), и .median() работают со столбцами
fill_median.fillna(fill_median.median(), inplace = True)
# убедимся, что пропусков не осталось
fill_const.isnull().sum().sum()
```

```
'0'
```

Так как все признаки датасета количественные, не будем применять метод заполнения внутригрупповой медианой (для этого нам потребовалась бы хотя бы одна категориальная переменная).

Заполнение линейной регрессией

Напишем функцию для заполнения пропусков линейной регрессией.

```
# передадим функции датафрейм, а также название столбца с пропусками
def linreg_imputer(df, col):

  # обучающей выборкой будут строки без пропусков
  train = df.dropna().copy()
  # тестовой (или вернее выборкой для заполнения пропусков)
  # будут те строки, в которых пропуски есть
  test = df[df[col].isnull()].copy()

  # выясним индекс столбца с пропусками
  col_index = df.columns.get_loc(col)

  # разделим "целевую переменную" и "признаки"
  # обучающей выборки
  y_train = train[col]
  X_train = train.drop(col, axis = 1)

  # из "тестовой" выборки удалим столбец с пропусками
  test = test.drop(col, axis = 1)

  # обучим модель линейной регрессии
  model = LinearRegression()
  model.fit(X_train, y_train)

  # сделаем прогноз пропусков
  y_pred = model.predict(test)
  # вставим пропуски (value) на изначальное место (loc) столбца с пропусками (column)
  test.insert(loc = col_index, column = col, value = y_pred)

  # соединим датасеты и обновим индекс
  df = pd.concat([train, test])
  df.sort_index(inplace = True)

  return df
```

Заполним пропуски.

```
fill_linreg = X_nan.copy()
fill_linreg = linreg_imputer(X_nan, 'mean radius')
fill_linreg.isnull().sum().sum()
```

```
'0'
```

Заполнение с помощью MICE

```
fill_mice = X_nan.copy()
mice_imputer = IterativeImputer(initial_strategy = 'mean', # вначале заполним пропуски
                                estimator = LinearRegression(), # в качестве модели исп
                                random_state = 42 # добавим точку отсчета
                                )

# используем метод .fit_transform() для заполнения пропусков
fill_mice = pd.DataFrame(mice_imputer.fit_transform(fill_mice), columns = fill_mice.col
fill_linreg.isnull().sum().sum()
```

```
'0'
```

Заполнение с помощью KNNImputer

```
fill_knn = X_nan.copy()

# используем те же параметры, что и раньше: пять "соседей" с одинаковыми весами
knn_imputer = KNNImputer(n_neighbors = 5, weights = 'uniform')

fill_knn = pd.DataFrame(knn_imputer.fit_transform(fill_knn), columns = fill_knn.columns
fill_knn.isnull().sum().sum()
```

```
'0'
```

Оценка результата

Так как мы вычисляем количественные отклонения (разницу) прогнозного набора данных от полного, возведем отклонения в квадрат и суммируем вначале по столбцам (в нашем случае такой столбец один), а затем найдем общий квадрат всех отклонений датасета. Это и будет нашей метрикой.

Примечание. От RMSE метрика отличается только тем, что мы не делим на количество наблюдений (оно одинаковое в каждом датасете) и не извлекаем квадратный корень. Такое упрощение сделано для того, чтобы получившиеся в этом конкретном случае показатели было удобнее сравнивать.

```
# напишем функцию, которая считает сумму квадратов отклонений
# заполненного значения от исходного
def nan_mse(X_full, X_nan):
  return ((X_full - X_nan)**2).sum().sum().round(2)
```

```
# создадим списки с датасетами и названиями методов
imputer = [fill_const, fill_median, fill_linreg, fill_mice, fill_knn]
name = ['constant', 'median', 'linreg', 'MICE', 'KNNImputer']
```

```
# в цикле оценим качество каждого из методов и выведем результат
for i, n in zip(imputer, name):
  score = nan_mse(X_full, i)
  print(n + ': ' + str(score))
```

```
constant: 122.7
median: 137.04
linreg: 0.03
MICE: 0.03
KNNImputer: 9.77
```

Используя более объективный метод сравнения, мы видим ещё бóльшую разницу результатов

одномерного и многомерного способов заполнения данных.

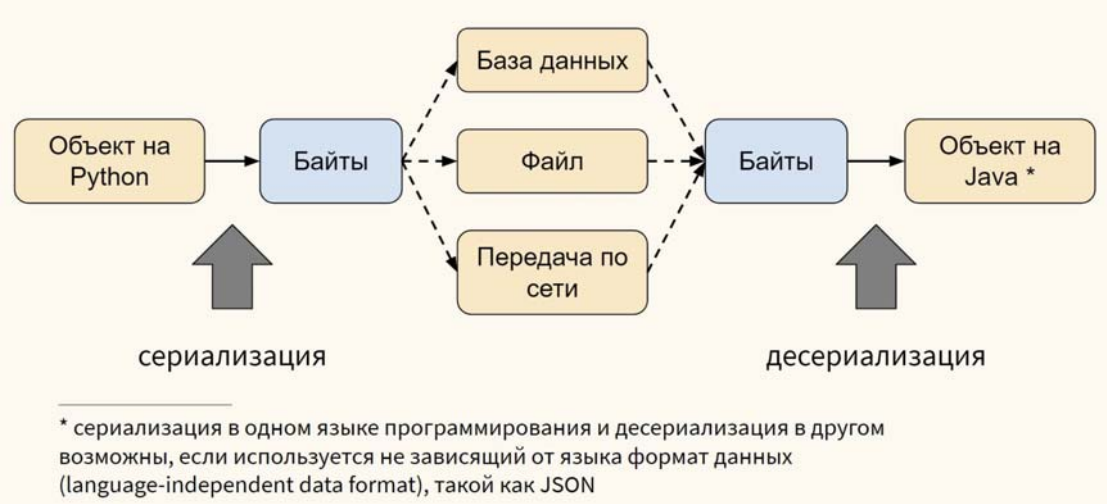
Обратите внимание, результат линейной регрессии совпал с результатом алгоритма MICE (как в случае с датасетом «Титаник», так и в текущем сравнении методов). Это логично, MICE, как и линейная регрессия использовали один и тот же алгоритм (класс LinearRegression) и одни и те же признаки без пропусков (и MICE не пришлось заполнять их средним значением).

Сериализация и десериализация данных

Некоторые библиотеки создают файлы модели, в частности, в форматах JSON и Pickle. Зачем это нужно?

> Благодаря этому у нас появляется возможность сохранить обученную модель на диске или передать ее по сети, а затем использовать для прогнозирования на новых данных *без необходимости повторного обучения*.

Процесс преобразования объекта (в частности, модели) в формат, пригодный для хранения и передачи данных, называется **сериализацией** (serialization). Процесс восстановления из этого формата называется соответственно **десериализацией** (deserialization).



Вначале рассмотрим в целом процесс серилизации в JSON и Pickle, а затем перейдем к работе с моделями.

Формат JSON

Формат JSON расшифровывается как JavaScript Object Notation, но, несмотря на наличие слова JavaScript в своем названии, *не зависит от языка программирования*. Кроме того, он позволяет сериализовать данные в *человекочитаемый формат*.

Этот формат очень часто используется для передачи данных между клиентом и сервером. Например, запросим информацию о случайно выбранном банковском учреждении.

```
# импортируем модуль json,
import json
```

```python
# функцию urlopen() из модуля для работы с URL-адресами,
from urllib.request import urlopen
# а также функцию pprint() одноименной библиотеки
from pprint import pprint

url = 'https://random-data-api.com/api/v2/banks'

# получаем ответ (response) в формате JSON
with urlopen(url) as response:
  # считываем его и закрываем объект response
  data = response.read()

# данные пришли в виде последовательности байтов
print(type(data))
print()
# выполняем десериализацию
output = json.loads(data)
pprint(output)
print()
# и смотрим на получившийся формат
print(type(output))
```

```
<class 'bytes'>

{'account_number': '6695451080',
 'bank_name': 'UBS CLEARING AND EXECUTION SERVICES LIMITED',
 'iban': 'GB57WYQG90913422454081',
 'id': 4063,
 'routing_number': '014681402',
 'swift_bic': 'BCYPGB2LXXX',
 'uid': 'e1ef18ac-e286-471b-858f-f1a7cb3d85e6'}

<class 'dict'>
```

По сети мы получили объект типа bytes, то есть последовательность байтов, и десериализовали его с помощью **метода .loads()** в формат питоновского словаря.

Убедиться в человекочитаемости этого формата можно просто перейдя по ссылке в браузере: https://random-data-api.com/api/v2/banks.

Вложенный словарь и список словарей

JSON-объект можно создать из вложенных питоновских словарей или списка словарей.

```python
# создадим вложенные словари
sales = {
    'PC' : {
        'Lenovo' : 3,
        'Apple'  : 2
        },
    'Phone' : {
        'Apple': 2,
        'Samsung': 5
        }
}

# и список из словарей
students = [
    {
        'id': 1,
        'name': 'Alex',
        'math': 5,
```

```
        'computer science': 4
    },
    {
        'id': 2,
        'name': 'Mike',
        'math': 4,
        'computer science': 5
     }
]
```

Методы .dumps() и .loads()

Вначале применим **метод .dumps()** для создания строкового JSON-объекта.

```
# преобразуем вложенный словарь в JSON
# дополнительно укажем отступ (indent)
json_sales = json.dumps(sales, indent = 4)

print(json_sales)
print(type(json_sales))
```

```
{
    "PC": {
        "Lenovo": 3,
        "Apple": 2
    },
    "Phone": {
        "Apple": 2,
        "Samsung": 5
    }
}
<class 'str'>
```

Обратите внимание, что хотя объект похож на питоновский словарь, тем не менее это строка. Восстановим словарь с помощью **метода .loads()**.

```
sales = json.loads(json_sales)
print(sales)
print(type(sales))
```

```
{'PC': {'Lenovo': 3, 'Apple': 2}, 'Phone': {'Apple': 2, 'Samsung': 5}}
<class 'dict'>
```

Методы .dump() и .load()

Метод .dump() создает последовательность байтов и используется для записи JSON-объекта в файл. Этот метод принимает два основных параметра: источник данных и файл, в который следует записать JSON-объект. Передадим ему этот файл с помощью конструкции **with open()**.

```
# создадим файл students.json и откроем его для записи
with open('/content/students.json', 'w') as wf:
  # поместим туда students, преобразовав в JSON
  json.dump(students, wf, indent = 4)
```

Восстановим список словарей из файла.

```
# прочитаем файл из сессионного хранилища
with open('/content/students.json', "rb") as rf:
```

```
  # и преобразуем обратно в список из словарей
  students_out = json.load(rf)
```

Посмотрим на результат.

```
students_out
```

```
[{'id': 1, 'name': 'Alex', 'math': 5, 'computer science': 4},
 {'id': 2, 'name': 'Mike', 'math': 4, 'computer science': 5}]
```

Обратите внимание, результат десериализации — это новый объект.

```
print(students == students_out)
print(students is students_out)
```

```
True
False
```

JSON и Pandas

Библиотека Pandas позволяет сохранить датафрейм в файл формата JSON, а также импортировать такой файл и соответственно восстановить датафрейм.

```
# импортируем датасет и преобразуем в датафрейм
from sklearn.datasets import load_breast_cancer
cancer, _ = load_breast_cancer(return_X_y = True, as_frame = True)

# создадим JSON-файл, поместим его в сессионное хранилище
cancer.to_json('/content/cancer.json')

# и сразу импортируем его и создадим датафрейм
pd.read_json('/content/cancer.json').head(3)
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness
0	17.99	10.38	122.8	1001.0	0.11840
1	20.57	17.77	132.9	1326.0	0.08474
2	19.69	21.25	130.0	1203.0	0.10960

Скриншот демонстрирует только часть датафрейма

Pickle

Если JSON-объект можно создать на Питоне, а восстановить на Java, то объект Pickle сериализуется и десериализуется только с помощью Питона. За это отвечает одноименная библиотека.

```
import pickle
```

Методы .dumps() и .loads()

Методы .dumps() и **.loads()** преобразуют объект в байты и восстанавливают исходный тип данных соответственно.

```
sales_pickle = pickle.dumps(sales)

print(sales_pickle)
print(type(sales_pickle))
```

```
b'\x80\x03}q\x00(X\x02\x00\x00\x00PCq\x01}q\x02(X\x06\x00\x00\x00Lenovoq\x03K\x03X\x05\
<class 'bytes'>
```

```
sales_out = pickle.loads(sales_pickle)

print(sales_out)
print(type(sales_out))
```

```
{'PC': {'Lenovo': 3, 'Apple': 2}, 'Phone': {'Apple': 2, 'Samsung': 5}}
<class 'dict'>
```

Методы .dump() и .load()

Методы .dump() и **.load()** сериализуют объект в файл и соответственно десериализуют объект из файла.

```
# создадим файл students.p
# и откроем его для записи в бинарном формате (wb)
with open('/content/students.p', 'wb') as wf:
  # поместим туда объект pickle
  pickle.dump(students, wf)
```

```
# достанем этот файл из сессионного хранилища
# и откроем для чтения в бинарном формате (rb)
with open('/content/students.p', 'rb') as rf:
  students_out = pickle.load(rf)
```

```
# выведем результат
students_out
```

```
[{'id': 1, 'name': 'Alex', 'math': 5, 'computer science': 4},
 {'id': 2, 'name': 'Mike', 'math': 4, 'computer science': 5}]
```

При создании файла в формате pickle можно использовать расширения .p, .pkl или .pickle.

Собственные объекты

В отличие JSON, который позволяет сериализовать только ограниченный набор объектов, Pickle может сериализовать любой питоновский объект, например собственную функцию или класс.

Начнем с **функций**.

```
# создадим функцию, которая будет выводить надпись "Some function!"
def foo():
  print('Some function!')

# преобразуем эту функцию в объект Pickle
foo_pickle = pickle.dumps(foo)

# десериализуем и
foo_out = pickle.loads(foo_pickle)

# вызовем ее
foo_out()
```

```
Some function!
```

То же самое можно сделать с **классом**.

```
# создадим класс и объект этого класса
class CatClass:

  def __init__(self, color):
    self.color = color
    self.type_ = 'cat'

Matroskin = CatClass('gray')
```

```
# сериализуем класс в объект Pickle и поместим в файл
with open('cat_instance.pkl', 'wb') as wf:
  pickle.dump(Matroskin, wf)
```

```
# достанем из файла и десериализуем
with open('cat_instance.pkl', 'rb') as rf:
  Matroskin_out = pickle.load(rf)
```

```
# выведем атрибуты созданного нами объекта класса
Matroskin_out.color, Matroskin_out.type_
```

```
('gray', 'cat')
```

Обратите внимание, что после десериализации мы смогли восстановить не только общую структуру класса, но и те данные (атрибут color), которые поместили в создаваемый объект.

Когда мы будем сохранять таким образом модели ML, вместо цвета шерсти животного мы будет хранить в Pickle, например, веса, полученные в результате обучения.

Сохраняемость модели ML

Чаще всего для этого используется формат Pickle, потому что он способен сериализовать и десериализовать сложные объекты, которыми являются модели.

> В машинном обучении говорят, что процесс сериализации и десериализации выполняется ради обеспечения **сохраняемости модели** (model persistance).

При этом, при использовании модуля Pickle, на этапах сериализации и десериализации важно работать с одинаковыми версиями Питона и используемых библиотек (например, sklearn). В противном случае результат может быть непредсказуемым. Это заставило некоторых специалистов предложить серилизацию модели ML с помощью JSON.

Рассмотрим сериализацию модели на практике. Вначале обучим модель логистической регрессии.

```
# импортируем датасет о раке груди
X, y = load_breast_cancer(return_X_y = True, as_frame = True)

# импортируем класс для масштабирования данных,
from sklearn.preprocessing import MinMaxScaler
# функцию для разделения выборки на обучающую и тестовую части,
from sklearn.model_selection import train_test_split
# класс логистической регрессии
from sklearn.linear_model import LogisticRegression

```

```
# разделим выборку
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.30,
                                                    random_state = 42)

# создадим объект класса MinMaxScaler
scaler = MinMaxScaler()

# масштабируем обучающую выборку
X_train_scaled = scaler.fit_transform(X_train)

# обучим модель на масштабированных train данных
model = LogisticRegression(random_state = 42).fit(X_train_scaled, y_train)

# используем минимальное и максимальное значения
# обучающей выборки для масштабирования тестовых данных
X_test_scaled = scaler.transform(X_test)

# сделаем прогноз
y_pred = model.predict(X_test_scaled)
```

Оценим результат.

```
# импортируем функцию для создания матрицы ошибок
from sklearn.metrics import confusion_matrix

# передадим матрице тестовые и прогнозные значения
# поменяем порядок так, чтобы злокачественные опухоли были положительным классом
model_matrix = confusion_matrix(y_test, y_pred, labels = [1,0])

# для удобства создадим датафрейм
model_matrix_df = pd.DataFrame(model_matrix)
model_matrix_df
```

	0	1
0	107	1
1	5	58

```
# рассчитаем accuracy
accuracy_score(y_test, y_pred).round(2)
```

```
0.96
```

Теперь выполним сериализацию и десериализацию модели.

```
# сериализуем и
with open('model.pickle', 'wb') as wf:
  pickle.dump(model, wf)
```

```
# десериализуем модель
with open('model.pickle', 'rb') as rf:
  model_out = pickle.load(rf)
```

Передадим десериализованной модели те же данные и убедимся, что она выдаст результат идентичный изначальной модели.

```
# напомню, десериализованная модель - это другой объект
y_pred_out = model_out.predict(X_test_scaled)

model_matrix = confusion_matrix(y_test, y_pred_out, labels = [1,0])
model_matrix_df = pd.DataFrame(model_matrix)
model_matrix_df
```

	0	1
0	107	1
1	5	58

```
accuracy_score(y_test, y_pred).round(2)
```

```
0.96
```

Подведем итог

В дополнительных материалах мы применили тест Литтла для выявления полностью случайных пропусков, познакомились с понятием временной сложности алгоритма, а также сериализацией и десериализацией модели ML.