Машинное обучение

Пропущенные значения. Часть 1

Материалы > Анализ и обработка данных

В реальных данных встречаются не только ошибки, но и **пропущенные значения** (missing values). При этом не все алгоритмы машинного обучения умеют работать с данными, в которых есть пропуски.

Вначале немного теории.

Типы пропусков

В 1976 году математик Дональд Рубин (Donald B. Rubin) предложил следующую классификацию пропущенных значений.

Полностью случайные пропуски

Полностью случайные пропуски (missing completely at random, MCAR) предполагают, что вероятность появления пропуска никак не связана с данными. Такие пропуски возникают, например, если измерительный прибор неисправен и случайным образом не записал часть наблюдений, или если один из образцов крови, изучаемых в лаборатории, оказался поврежден и по этой причине его характеристики выпали из исследования.

Интересно посмотреть на эту классификацию с точки зрения условной вероятности. Введем обозначения.

- **П** — пропуски в данных (missing data)
- **Н** — наблюдаемые значения, то есть те данные, которые мы собрали (observed data)
- **О** — отсутствующие значения, или те данные, которые собрать не удалось (unobserved data)

Тогда полностью случайные пропуски можно выразить следующим образом.

$P(П \mid Н, О) = \text{константа}$

> Какими бы ни были наблюдаемые и отсутствующие значения, вероятность пропусков всегда одинакова, так как они, эти пропуски, полностью случайны.

Эту же идею можно выразить и так.

P (П | Н, О) = P (П)

Считается, что в реальности наблюдать полностью случайные пропущенные значения очень сложно. Какие-либо закономерности (то есть связи с наблюдаемыми или отсутствующими значениями) все равно существуют. Это приводит нас ко второй категории пропусков.

Случайные пропуски

Случайные пропуски (missing at random, MAR) — вероятность появления пропуска зависит от некоторой *известной нам* переменной. Например, отсутствие ответа на определенный вопрос анкеты может зависеть от возраста респондента. Молодые охотнее отвечают на вопрос, люди более пожилого возраста скорее избегают ответа.

> Если *мы знаем* об этой особенности, то можем, правильно собирая и корректируя данные, добиться большей объективности.

Вероятность появления таких пропусков с учетом наблюдаемых и отсутствующих значений можно представить как *функцию от наблюдаемых значений*.

P (П | Н, О) = f(Н)

В нашем примере, такой функцией является функция возраста респондентов, f(возраст).

Неслучайные пропуски

Неслучайные пропуски (missing not at random, MNAR) — вероятность появления пропуска зависит, в том числе, от фактора, о котором мы *ничего не знаем*. Например, у весов может быть верхний предел измерения и любой образец выше этого предела автоматически не записывается. В опросах общественного мнения MNAR возникает, когда люди с более активной жизненной позицией (переменная, которую мы не измеряем) чаще дают ответы на вопросы интервьюера.

В таком случае условная вероятность пропусков зависит от функции, которая может учитывать как наблюдаемые, так и, что более важно, отсутствующие значения.

P (П | Н, О) = f(Н, О)

Проблема опять же в том, что мы не знаем, что это за функция, а значит не знаем как именно появились пропущенные значения. Теперь перейдем к практике.

> Откроем блокнот к этому занятию⧉

Выявление пропусков

Вначале подготовим необходимые данные. Сегодня мы снова будем работать с датасетом

«Титаник».

```
# импортируем датасет Титаник
titanic = pd.read_csv('/content/train.csv')
```

Базовые методы

Метод .info()

Рассмотрим базовые методы обнаружения пропусков. В первую очередь, можно использовать **метод .info()**. Этот метод соотносит максимальное количество записей в датафрейме с количеством записей в каждом столбце.

```
# применим этот метод к нашему датасету
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
#   Column       Non-Null Count  Dtype
---  ------       --------------  -----
0   PassengerId  891 non-null    int64
1   Survived     891 non-null    int64
2   Pclass       891 non-null    int64
3   Name         891 non-null    object
4   Sex          891 non-null    object
5   Age          714 non-null    float64
6   SibSp        891 non-null    int64
7   Parch        891 non-null    int64
8   Ticket       891 non-null    object
9   Fare         891 non-null    float64
10  Cabin        204 non-null    object
11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Как мы видим, всего в датасете может быть до 891 записи. При этом в столбцах Age, Cabin и Embarked записей меньше, а значит есть пропуски.

Также обратите внимание на одну особенность Питона. Столбец Age логично преобразовать в тип int, однако из-за того, что в нем есть пропущенные значения, сделать этого не получится. Для количественных данных с пропусками доступен только тип float.

```
# попробуем преобразовать Age в int
titanic.Age.astype('int')
```

```
---------------------------------------------------------------------------
IntCastingNaNError                        Traceback (most recent call last)
<ipython-input-33-9ac81bbfa473> in <module>()
      1 # попробуем преобразовать Age в int
----> 2 titanic.Age.astype('int')

                                 7 frames

/usr/local/lib/python3.7/dist-packages/pandas/core/dtypes/cast.py in astype_float_to_int_nansafe(values, dtype, copy)
```

```
   1212     if not np.isfinite(values).all():
   1213         raise IntCastingNaNError(
-> 1214             "Cannot convert non-finite values (NA or inf) to integer"
   1215         )
   1216     return values.astype(dtype, copy=copy)

IntCastingNaNError: Cannot convert non-finite values (NA or inf) to integer
```

SEARCH STACK OVERFLOW

Конечно, если столбцов много, результат метода **.info()** становится трудно воспринимать.

Методы .isna() и .sum()

Можно последовательно использовать **методы .isna()** и **.sum()**.

```
1  # .isna() выдает True или 1, если есть пропуск,
2  # .sum() суммирует единицы по столбцам
3  titanic.isna().sum()
```

```
1   PassengerId      0
2   Survived         0
3   Pclass           0
4   Name             0
5   Sex              0
6   Age            177
7   SibSp            0
8   Parch            0
9   Ticket           0
10  Fare             0
11  Cabin          687
12  Embarked         2
13  dtype: int64
```

Процент пропущенных значений

Также не сложно посчитать процент пропущенных значений.

```
1# для этого разделим сумму пропусков в каждом столбце на количество наблюдений,
2# округлим результат и умножим его на 100
3(titanic.isna().sum() / len(titanic)).round(4) * 100
```

```
1   PassengerId     0.00
2   Survived        0.00
3   Pclass          0.00
4   Name            0.00
5   Sex             0.00
6   Age            19.87
7   SibSp           0.00
8   Parch           0.00
9   Ticket          0.00
10  Fare            0.00
11  Cabin          77.10
12  Embarked        0.22
```

```
13          dtype: float64
```

Теперь нам гораздо проще оценить картину в целом.

Библиотека missingno

Библиотека missingno предоставляет удобные средства для визуальной оценки пропусков.

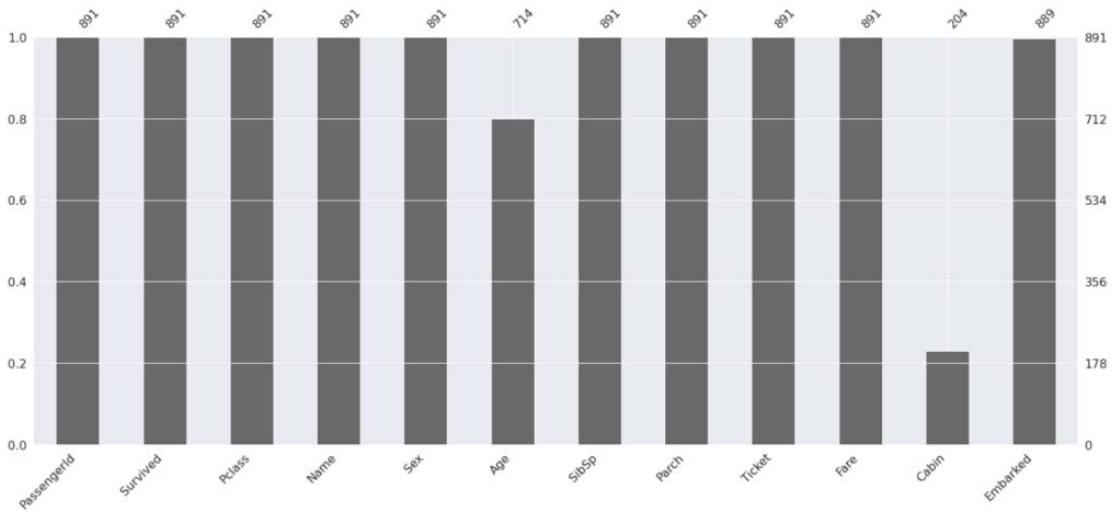
```
1 # импортируем библиотеку missingno с псевдонимом msno
2 import missingno as msno
```

Кроме того, для повышения качества визуализации сделаем стиль графиков seaborn основным.

```
1           sns.set()
```

В первую очередь на пропуски можно посмотреть с помощью **столбчатой диаграммы** (функция **msno.bar()**).

```
1       msno.bar(titanic);
```

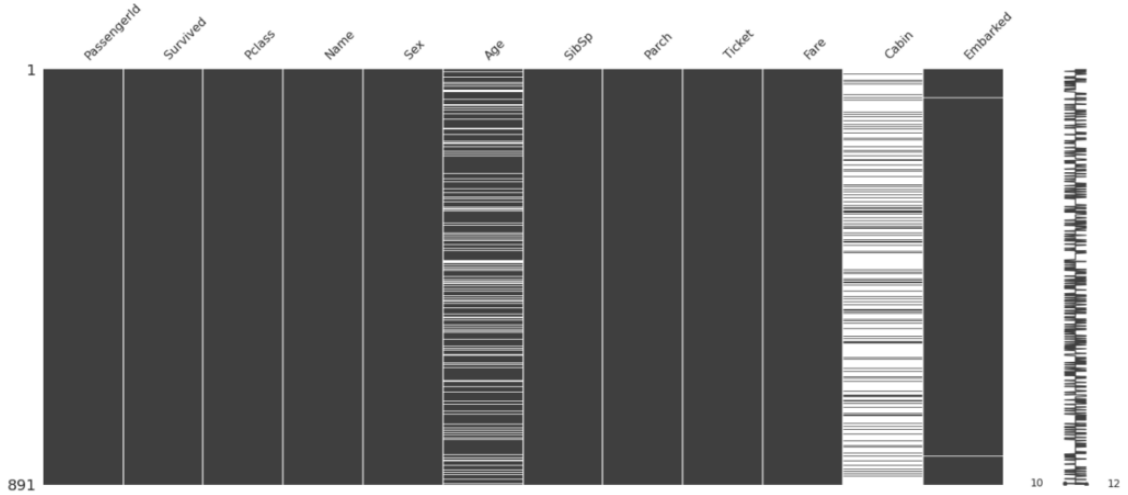


На этом графике мы четко видим процент (слева) и абсолютное количество (справа и сверху) заполненных значений.

> При этом столбчатая диаграмма не дает информации о том, где именно больше всего пропущенных значений. Другими словами, есть ли в пропусках какая-то закономерность или нет.

Для этого подойдет **матрица** пропущенных значений (**функция msno.matrix()**).

```
1 msno.matrix(titanic);
```



Распределение пропущенных значений в датасете «Титаник» выглядит случайным, закономерностью были бы пропуски, например, только в первой половине наблюдений.

При этом обратите внимание, мы говорим про случайность внутри столбцов с пропусками. О том, зависят ли пропуски от значений других столбцов, мы поговорим ниже.

Матрица корреляции пропущенных значений

Еще один интересный инструмент — **матрица корреляции пропущенных значений** (nullity correlation matrix).

> По сути она показывает, насколько сильно присутствие или отсутствие значений одного признака влияет на присутствие значений другого.

Если мы знаем, в каких столбцах есть пропуски, то можем просто последовательно применить к ним **методы .isnull()** и **.corr()**.

```
1 titanic[['Age', 'Cabin', 'Embarked']].isnull().corr()
```

	Age	Cabin	Embarked
Age	1.000000	0.144111	-0.023616
Cabin	0.144111	1.000000	-0.087042
Embarked	-0.023616	-0.087042	1.000000

В тех случаях, когда мы не знаем, в каких столбцах есть пропущенные значения, то можем использовать код ниже (взят из документации к библиотеке).

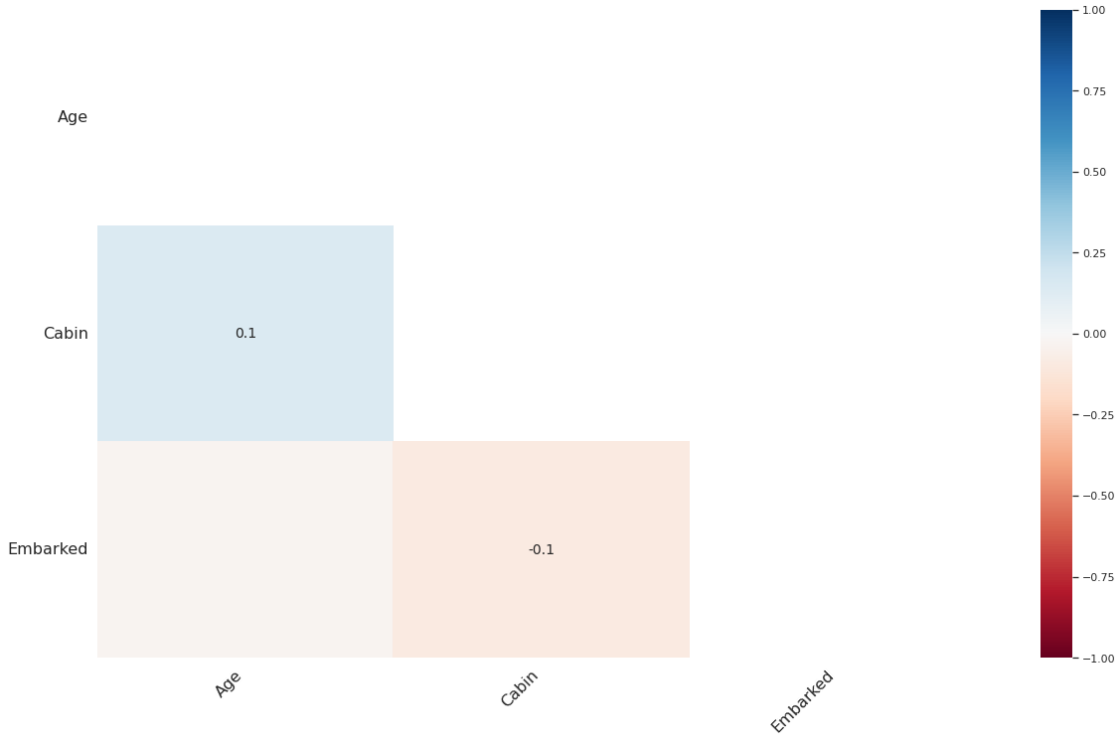
```
1 df = titanic.iloc[:, [i for i, n in enumerate(np.var(titanic.isnull(), axis = 'rows')) if n > 0]]
2 df.isnull().corr()
```

	Age	Cabin	Embarked
Age	1.000000	0.144111	-0.023616
Cabin	0.144111	1.000000	-0.087042
Embarked	-0.023616	-0.087042	1.000000

Значения корреляции могут быть от -1 (если значения одного признака присутствуют, значения другого — отсутствуют) до 1 (если присутствуют значения одного признака, то присутствуют значения и другого). Более подробно про корреляцию мы поговорим при изучении взаимосвязи переменных.

Визуально, корреляцию пропущенных значений можно представить с помощью **тепловой карты** (heatmap). Для этого есть функция **msno.heatmap()**.

```
1  msno.heatmap(titanic);
```



Мы видим, что корреляция пропусков близка к нулю для всех признаков. Другими словами, пропуски одного признака не влияют на пропуски другого.

Теперь рассмотрим стратегии работы с пропусками. По большому счету их две: удаление и заполнение. У обоих подходов есть свои достоинства и недостатки.

Удаление пропусков

Во многих случаях **удаление пропусков** (missing values deletion) может оказаться неплохим решением, потому что в этом случае мы не «портим» данные.

Удаление пропущенных значений хорошо работает (позволяет качественно обучить алгоритм), если мы считаем, что пропуски носят полностью случайный характер (MCAR).

Единственным ограничением в этом случае будет достаточность данных для обучения после удаления пропусков.

Удаление строк

Удаление строк (deleting rows или listwise deletion, также называется **анализом полных наблюдений**, complete case analysis), в которых есть пропуски — наиболее очевидный подход к работе с пропущенными значениями. Рассмотрим этот способ на практике.

В датасете «Титаник» только два пропущенных значения в столбце Embarked. Удалим соответствующие строки.

```
1# удаление строк обозначим через axis = 'index'
2# subset = ['Embarked'] говорит о том, что мы ищем пропуски только в столбце Embarked
3titanic.dropna(axis = 'index', subset = ['Embarked'], inplace = True)
```

```
1 # убедимся, что в Embarked действительно не осталось пропусков
2 titanic.Embarked.isna().sum()
```

```
1                                        '0'
```

Удаление строк не стоит применять, если пропущенные значения зависят от какого-либо неизвестного нам фактора (MNAR). Например, если на вопрос анкеты не склонны отвечать менее активные граждане, удаление строк с пропусками оставит в данных только определенную группу населения (появится bias, искажение) и алгоритм не будет репрезентативен.

Кроме того, если в одном из столбцов большой процент пропусков, построчное удаление просто оставит нас без данных. В датасете «Титаник» это относится к столбцу Cabin. В этом случае, если мы выбираем стратегию удаления данных, разумнее удалить сам столбец.

Удаление столбцов

Удаление столбцов (column deletion) несложно выполнить с помощью **метода .drop()**. Например, удалим столбец Cabin, в котором более 77 процентов пропусков.

```
1 # передадим в параметр columns тот столбец, который хотим удалить
2 titanic.drop(columns = ['Cabin'], inplace = True)
```

```
1  # убедимся, что такого столбца больше нет
2  titanic.columns
```

```
1 Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
```

```
2       'Parch', 'Ticket', 'Fare', 'Embarked'],
3      dtype='object')
```

Попарное удаление пропусков

Попарное удаление пропусков (pairwise deletion или, как еще говорят, **анализ доступных данных**, available case analysis) проще понять, если представить, что мы не удаляем пропуски, а игнорируем их или используем только доступные значения.

Расчет метрик. Принципа игнорирования пропусков придерживаются очень многие функции и методы в Питоне. Например, используем методы **.groupby()** и **.count()** для того, чтобы посчитать количество мужчин и женщин на борту и выведем данные по каждому из оставшихся признаков.

```
1  sex_g = titanic.groupby('Sex').count()
2  sex_g
```

Sex	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Embarked
female	312	312	312	312	259	312	312	312	312	312
male	577	577	577	577	453	577	577	577	577	577

Как вы видите, если верить столбцу Age, пассажиров на борту меньше, чем если руководствоваться данными, например, столбца PassengerId.

```
1 # сравним количество пассажиров в столбце Age и столбце PassengerId
2 sex_g['PassengerId'].sum(), sex_g['Age'].sum()
```

```
1              (889, 712)
```

Это значит, что метод **.count()** игнорировал пропуски. То же самое касается, например, метода **.mean()** или метода **.corr()**.

```
1 # метод .mean() игнорирует пропуски и не выдает ошибки
2 titanic['Age'].mean()
```

```
1       29.64209269662921
```

```
1  # то же можно сказать про метод .corr()
2  titanic[['Age', 'Fare']].corr()
```

	Age	Fare

Age	1.000000	0.096067
Fare	0.096067	1.000000

Построение модели. *Преимуществом* при построении модели будет то, что мы по максимуму используем имеющиеся данные. Например, у нас есть два признака, и в первом есть пропуск у четвертого наблюдения (с индексом «три»), а во втором — у пятого.

id	Признак 1	Признак 2	Целевая переменная
0	53	12	0
1	61	28	1
2	48	20	0
3	~~NaN~~	24	~~0~~
4	57	~~NaN~~	~~1~~
5	60	31	1
	Модель А	Модель В	

Если мы построим первую модель (А) на основе признака 1 (и соответственно удалим только четвертое наблюдение), а вторую (В) — на основе признака 2 (удалив пятое), то избежим необходимости каждый раз удалять два наблюдения и терять информацию.

Недостаток заключается в том, что эти модели по сути построены на разных данных (в реальности мы конечно удалим больше одного наблюдения в каждом случае), а значит сравнение моделей будет некорректным.

Заполнение пропусков

Как уже было сказано выше, удаление пропусков не всегда возможно. В этом случае прибегают к **заполнению пропусков** (missing values imputation). Подготовим данные.

```
# еще раз загрузим датасет "Титаник", в котором снова будут пропущенные значения
titanic = pd.read_csv('/content/train.csv')

# возьмем лишь некоторые из столбцов
titanic = titanic[['Pclass', 'Sex', 'SibSp', 'Parch', 'Fare', 'Age', 'Embarked']]

# закодируем столбец Sex с помощью числовых значений
map_dict = {'male' : 0, 'female' : 1}
titanic['Sex'] = titanic['Sex'].map(map_dict)

# посмотрим на результат
titanic.head()
```

	Pclass	Sex	SibSp	Parch	Fare	Age	Embarked
0	3	0	1	0	7.2500	22.0	S

1	1	1	1	0	71.2833	38.0	C
2	3	1	0	0	7.9250	26.0	S
3	1	1	1	0	53.1000	35.0	S
4	3	0	0	0	8.0500	35.0	S

Одномерные методы

Одномерные методы (single Imputation) — это заполнение с использованием данных одного столбца. Другими словами, чтобы заполнить пропуски мы берем данные того же признака.

Заполнение константой

Количественные данные. Самый простой способ работы с пропусками в количественных данных — заполнить пропуски константой. Например, нулем (подходит для алгоритмов, чувствительных к масштабу признаков).

Воспользуемся **методом .fillna()**.

```
# вначале сделаем копию датасета
fillna_const = titanic.copy()

# заполним пропуски в столбце Age нулями, передав методу .fillna() словарь,
# где ключами будут названия столбцов, а значениями - константы для заполнения пропусков
fillna_const.fillna({'Age' : 0}, inplace = True)
```

Заполнение константой позволяет не сокращать размер выборки, однако может внести системную ошибку в данные. Сравним медианный возраст до и после заполнения пропусков нулями.

```
titanic.Age.median(), fillna_const.Age.median()
```

```
(28.0, 24.0)
```

При использовании алгоримов, основанных на деревьях решений, пропуски можно заполнить значением, не встречающимся в выборке. Если все значения признака положительные, то пропуски заполняются, например, -1.

Категориальные данные. Для категориальных признаков в некоторых случаях можно провести дополнительное исследование. В частности, в датасете «Титаник» есть два пассажира с неизвестным портом посадки.

```
# найдем пассажиров с неизвестным портом посадки
# для этого создадим маску по столбцу Embarked и применим ее к исходным данным
missing_embarked = pd.read_csv('/content/train.csv')
missing_embarked[missing_embarked.Embarked.isnull()]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
61	62	1	1	Icard, Miss. Amelie	female	38.0	0	0	113572	80.0	B28	NaN
829	830	1	1	Stone, Mrs. George Nelson (Martha Evelyn)	female	62.0	0	0	113572	80.0	B28	NaN

При этом в Интернете можно найти информацию о том, что обе пассажирки (Mrs Stone и ее служанка Amelie Icard) зашли на борт в порту Southampton (S).

Mrs Stone boarded the *Titanic* in Southampton on 10 April 1912 and was travelling in first class with her maid Amelie Icard. She occupied cabin B-28.

Для заполнения строковым значением также подойдет метод **.fillna()**.

```
# метод .fillna() можно применить к одному столбцу
# два пропущенных значения в столбце Embarked заполним буквой S (Southampton)
fillna_const.Embarked.fillna('S', inplace = True)
```

Конечно, такая информация о пропущенных значениях бывает доступна далеко не всегда.

```
# убедимся, что в столбцах Age и Embarked не осталось пропущенных значений
fillna_const[['Age', 'Embarked']].isna().sum()
```

```
Age         0
Embarked    0
dtype: int64
```

Вместо метода **.fillna()** можно использовать инструмент библиотеки sklearn, который называется **SimpleImputer**. Создадим объект этого класса и обучим модель.

```
# сделаем копию датасета
const_imputer = titanic.copy()

# импортируем класс SimpleImputer из модуля impute библиотеки sklearn
from sklearn.impute import SimpleImputer

# создадим объект этого класса, указав,
# что мы будем заполнять константой strategy = 'constant', а именно нулем fill_value = 0
imp_const = SimpleImputer(strategy = 'constant', fill_value = 0)

# и обучим модель на столбце Age
# мы используем двойные скобки, потому что метод .fit() на вход принимает двумерный массив
imp_const.fit(const_imputer[['Age']])
```

```
1 SimpleImputer(fill_value=0, strategy='constant')
```

Теперь применим эту модель для заполнения пропусков.

```
1# также используем двойные скобки с методом .transform()
2const_imputer['Age'] = imp_const.transform(const_imputer[['Age']])
3
4# убедимся, что пропусков не осталось и посчитаем количество нулевых значений
5const_imputer.Age.isna().sum(), (const_imputer['Age'] == 0).sum()
```

```
1 (0, 177)
```

В конце раздела мы проведем сравнение эффективности различных методов заполнения пропусков и столбец Embarked нам уже не понадобится.

```
1 # удалим его
2 const_imputer.drop(columns = ['Embarked'], inplace = True)
3
4 # и посмотрим на размер получившегося датафрейма
5 const_imputer.shape
```

```
1 (891, 7)
```

```
1 # посмотрим на результат
2 const_imputer.head(3)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	3	0	1	0	7.2500	22.0
1	1	1	1	0	71.2833	38.0
2	3	1	0	0	7.9250	26.0

Заполнение средним арифметическим или медианой

Количественные данные можно заполнить средним арифметическим или медианой (statistical imputation). Вначале воспользуемся методом **.fillna()**.

```
1 # сделаем копию датафрейма
2 fillna_median = titanic.copy()
3
4 # заполним пропуски в столбце Age медианным значением возраста,
5 # можно заполнить и средним арифметическим через метод .mean()
6 fillna_median.Age.fillna(fillna_median.Age.median(), inplace = True)
7
8 # убедимся, что пропусков не осталось
9 fillna_median.Age.isna().sum()
```

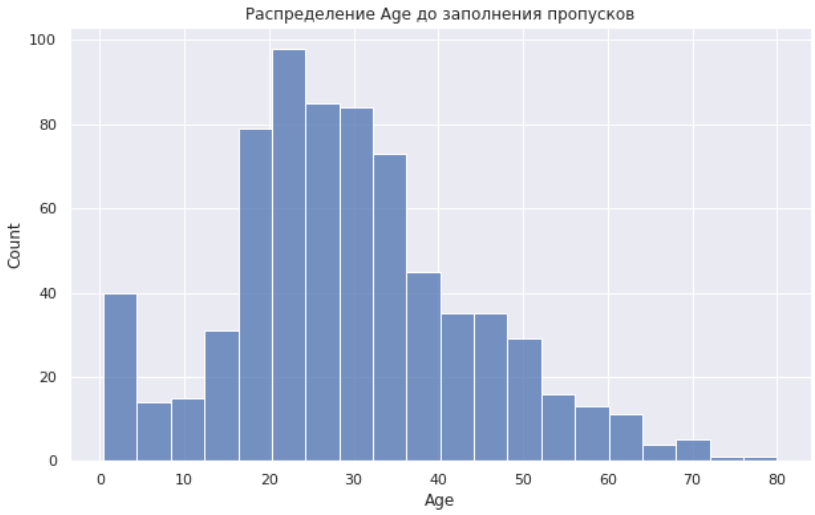
```
1                                   '0'
```

У такого простого и понятного подхода тем не менее есть ряд *недостатков*:

- во-первых, когда в данных появляется большое количество одинаковых близких к среднему значений, мы снижаем ценную вариативность в данных;
- кроме того, такое заполнение пропусков может быть некорректно; ниже мы рассмотрим пример данных кредитного скоринга, где, если заполнить пропуски в столбце «Стаж» средним значением или медианой, молодой сотрудник может получить больший стаж, чем у него есть на самом деле, а сотрудник в возрасте, меньший.

Еще раз обратимся к столбцу Age в датасете «Титаник» и рассмотрим распределение возраста до и после заполнения медианой.

```
1 # изменим размер последующих графиков
2 sns.set(rc = {'figure.figsize' : (10, 6)})
3
4 # скопируем датафрейм
5 median_imputer = titanic.copy()
6
7 # посмотрим на распределение возраста до заполнения пропусков
8 sns.histplot(median_imputer['Age'], bins = 20)
9 plt.title('Распределение Age до заполнения пропусков');
```



Посмотрим на среднее арифметическое и медиану.

```
1 median_imputer['Age'].mean().round(1), median_imputer['Age'].median()
```

```
1              (29.7, 28.0)
```

Используем класс **SimpleImputer** библиотеки sklearn для заполнения пропусков этим медианным значением.

```
# создадим объект класса SimpleImputer с параметром strategy = 'median'
# (для заполнения средним арифметическим используйте strategy = 'mean')
imp_median = SimpleImputer(strategy = 'median')

# применим метод .fit_transform() для одновременного обучения модели и заполнения пропусков
median_imputer['Age'] = imp_median.fit_transform(median_imputer[['Age']])

# убедимся, что пропущенных значений не осталось
median_imputer.Age.isna().sum()
```

```
1                                    '0'
```

Посмотрим на распределение возраста и его медианное значение после заполнения пропусков.

```
# посмотрим на распределение после заполнения пропусков
sns.histplot(median_imputer['Age'], bins = 20)
plt.title('Распределение Age после заполнения медианой');
```


```
# посмотрим на метрики после заполнения медианой
median_imputer['Age'].mean().round(1), median_imputer['Age'].median()
```

```
1            (29.4, 28.0)
```

```
# столбец Embarked нам опять же не понадобится
median_imputer.drop(columns = ['Embarked'], inplace = True)

# посмотрим на размеры получившегося датафрейма
median_imputer.shape
```

```
1                (891, 6)
```

Как мы видим, распределение претерпело существенные изменения. В частности, у нас появилось очень много медианных значений, которые доминируют в распределении возраста.

Заполнение внутригрупповым значением

Справиться с этой проблемой можно, в частности, через более сложный способ заполнения пропусков количественного признака — вначале разбить пассажиров на категории (bins), например, по полу или классу каюты, вычислить медианное значение для каждой категории и только потом заполнять им пропущенные значения.

```
1  # скопируем датафрейм
2  median_imputer_bins = titanic.copy()
```

Выберем столбец Age. Заполним пропуски в столбце Age, выполнив группировку по Sex и Pclass и применив функцию **median** через метод **.transform()**.

```
1 median_imputer_bins['Age'].fillna(median_imputer_bins\
2                                   .groupby(['Sex','Pclass'])['Age']\
3                                   .transform('median'), inplace = True)
```

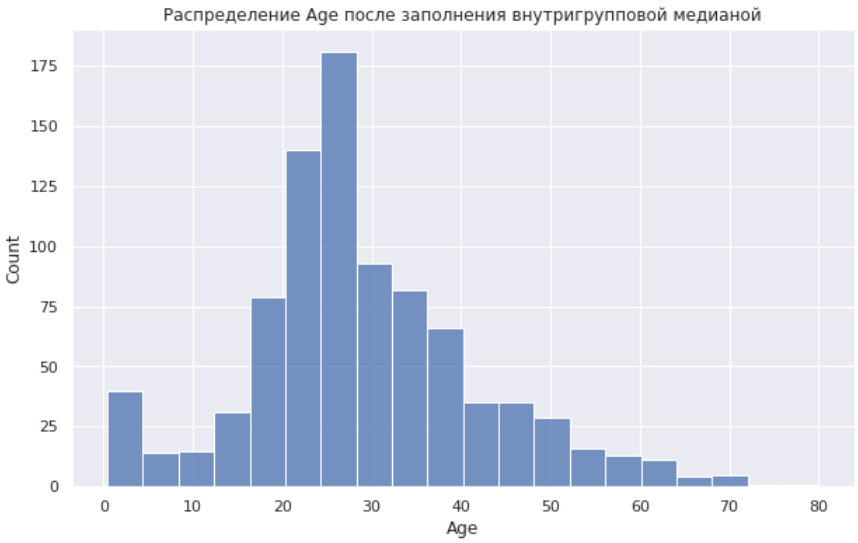
Убедимся, что в столбце Age не осталось пропусков.

```
1  # проверим пропуски в столбце Age
2  median_imputer_bins.Age.isna().sum()
```

```
1                                                    '0'
```

Посмотрим на распределение.

```
1 sns.histplot(median_imputer_bins['Age'], bins = 20)
2 plt.title('Распределение Age после заполнения внутригрупповой медианой');
```



Мы видим, что медианное значение доминирует гораздо меньше.

```
# столбец Embarked нам не понадобится
median_imputer_bins.drop(columns = ['Embarked'], inplace = True)

# посмотрим на размеры получившегося датафрейма
median_imputer_bins.shape
```

```
(891, 6)
```

Рассмотрим другие методы.

Заполнение наиболее частотным значением

Для заполнения пропусков в категориальных данных подойдет **метод заполнения наиболее часто встречающимся значением** (модой). Если пропусков немного, этот метод вполне обоснован. При большом количестве пропусков, можно попробовать создать на их основе новую категорию.

Подготовим данные и посмотрим на распределение категорий в столбце Embarked.

```
# скопируем датафрейм
titanic_mode = titanic.copy()

# посмотрим на распределение пассажиров по порту посадки до заполнения пропусков
titanic_mode.groupby('Embarked')['Survived'].count()
```

```
Embarked
C    168
Q     77
S    644
Name: Survived, dtype: int64
```

Модой будет порт Southampton (что одновременно является верным для заполнения пропусков значением, однако, опять же, в большинстве случаев мы не можем этого знать наверняка).

Воспользуемся классом **SimpleImputer** для заполнения пропусков.

```
# создадим объект класса SimpleImputer с параметром strategy = 'most_frequent'
imp_most_freq = SimpleImputer(strategy = 'most_frequent')

# применим метод .fit_transform() к столбцу Embarked
titanic_mode['Embarked'] = imp_most_freq.fit_transform(titanic_mode[['Embarked']]).ravel()

# убедимся, что пропусков не осталось
titanic_mode.Embarked.isna().sum()
```

```
1                                    '0'
```

Примечание. При работе со строками метод **.fit_transform()** возвращает двумерный массив Numpy. Для того чтобы записать его обратно в столбец Embarked, необходимо применить метод **.ravel()**.

Проверим результат.

```
1 # количество пассажиров в категории S должно увеличиться на два
2 titanic_mode.groupby('Embarked')['Survived'].count()
```

```
1   Embarked
2   C    168
3   Q     77
4   S    646
5   Name: PassengerId, dtype: int64
```

Примечание. Приведем еще два способа найти моду.

```
1   titanic.Embarked.value_counts().index[0]
```

```
1                                    'S'
```

```
1     imp_most_freq.statistics_
```

```
1     array(['S'], dtype=object)
```

В случае если у нас есть существенное количество пропусков в категориальной переменной, мы можем задуматься над созданием отдельной категории для пропущенных значений.

Очевидно, каким бы одномерным методом мы ни воспользовались, мы всегда ограничены данными одного признака.

```
1 # для работы с последующими методами столбец Embarked нам уже не нужен
2 titanic.drop(columns = ['Embarked'], inplace = True)
```