**Содержание**

Практическое занятие. Пропущенные значения. Часть 1. (2 часа)………..2

Практическое занятие. Пропущенные значения. Часть 2. (2 часа)………..21

Практическое занятие. Пропущенные значения. Часть 3. (2 часа)………..28

Практическое занятие. Пропущенные значения. Часть 4. (2 часа)………..44

Ответы на вопросы……………………………………………………………61

**Пропущенные значения. Часть 1. (2 часа)**

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.

**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 | Н, О) = 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 связь 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.

Р (П | Н, О) = (Н)

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

Р (П | Н, О) = (Н, O)

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.

**Выявление пропусков**

В первую очередь подготовим необходимые данные. Сегодня мы снова будем работать с датасетом «Титаник».

# импортируем датасет Титаник

titanic = pd.read\_csv('/content/train.csv')

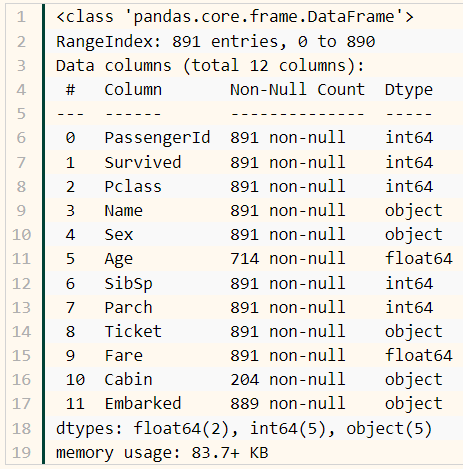
**Базовые методы**

**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

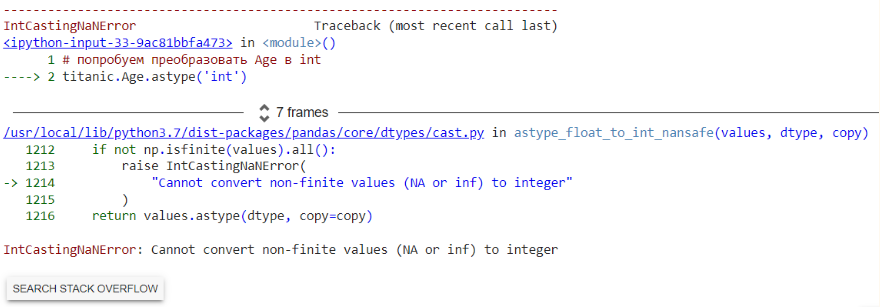
titanic.info()



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')

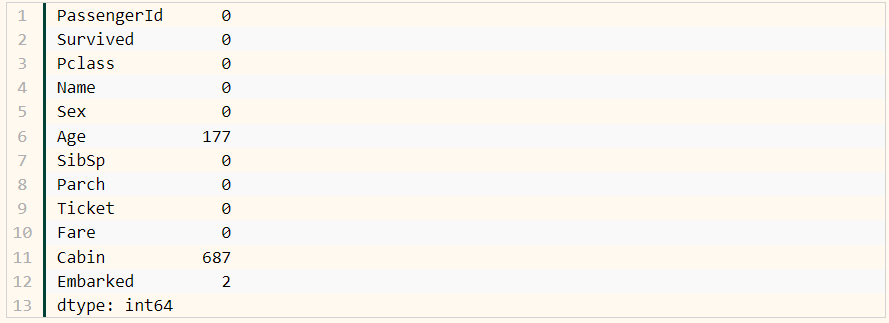


**Методы .isna() и .sum()**

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

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

titanic.isna().sum()



**Процент пропущенных значений**

Также не сложно посчитать процент пропущенных значений.

# для этого разделим сумму пропусков в каждом столбце на количество наблюдений,

# округлим результат и умножим его на 100

(titanic.isna().sum() / len(titanic)).round(4) \* 100

Теперь нам гораздо проще оценить «масштаб бедствия».

**Библиотека missingno**

Библиотека missingno предоставляет удобные средства для визуальной оценки пропусков.

# импортируем библиотеку missingno с псевдонимом msno

import missingno as msno

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

sns.set()

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

msno.bar(titanic)

***(предоставить график в отчете)***

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()).

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.

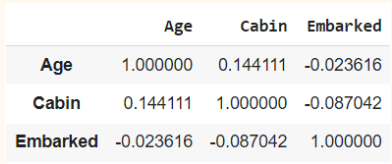
**Матрица корреляции пропущенных значений**

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().

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



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

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

df.isnull().corr()

***(Сравнить результаты)***

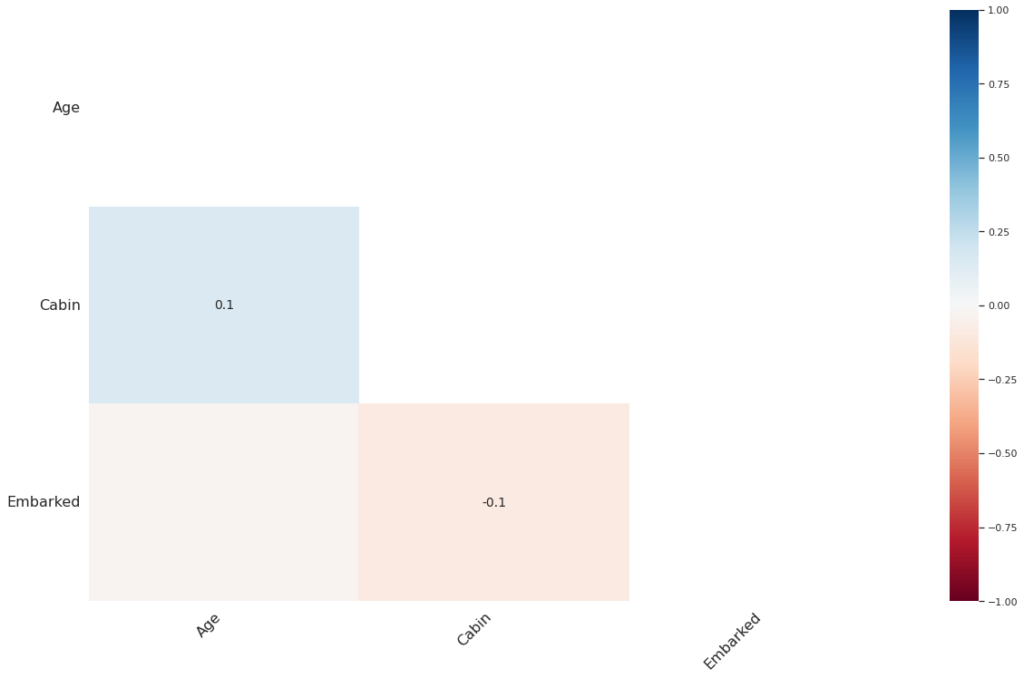
Значения корреляции могут быть от −1 (если значения одного признака присутствуют, значения другого – отсутствуют) до 1 (если присутствуют значения одного признака, то присутствуют значения и другого). Более подробно про корреляцию мы поговорим при изучении взаимосвязи переменных.

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

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.

# удаление строк обозначим через axis = 'index'

# subset = ['Embarked'] говорит о том, что мы ищем пропуски только в столбце Embarked

titanic.dropna(axis = 'index', subset = ['Embarked'], inplace = True)

# убедимся, что в Embarked действительно не осталось пропусков

titanic.Embarked.isna().sum()

***(проверить результат)***

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.

# передадим в параметр columns тот столбец, который хотим удалить

titanic.drop(columns = ['Cabin'], inplace = True)

# убедимся, что такого столбца больше нет

titanic.columns

***(сохранить список столбцов)***

**Попарное удаление пропусков**

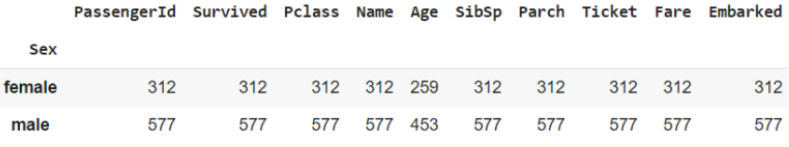
Попарное удаление пропусков (pairwise deletion или, как еще говорят, анализ доступных данных, available case analysis) проще понять, если представить, что мы не удаляем пропуски, а игнорируем их или используем только доступные значения.

**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.

sex\_g = titanic.groupby('Sex').count()

sex\_g

Результат:



# сравним количество пассажиров в столбце Age и столбце PassengerId

sex\_g['PassengerId'].sum(), sex\_g['Age'].sum()

**(889, 712)**

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

# метод .mean() игнорирует пропуски и не выдает ошибки

titanic['Age'].mean()

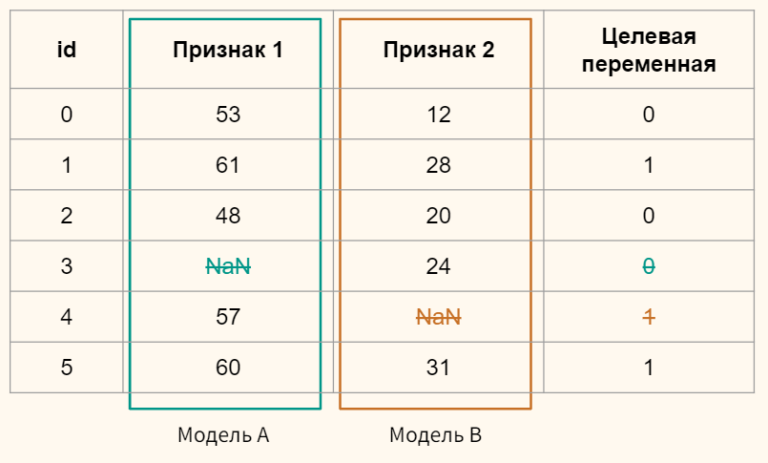
29.64209269662921

# то же можно сказать про метод .corr()

titanic[['Age', 'Fare']].corr()



**Построение модели.**



**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()

**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)

Сравним медианный возраст до и после заполнения пропусков нулями.

titanic.Age.median(), fillna\_const.Age.median()

Результат:

**(28.0, 24.0)**

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

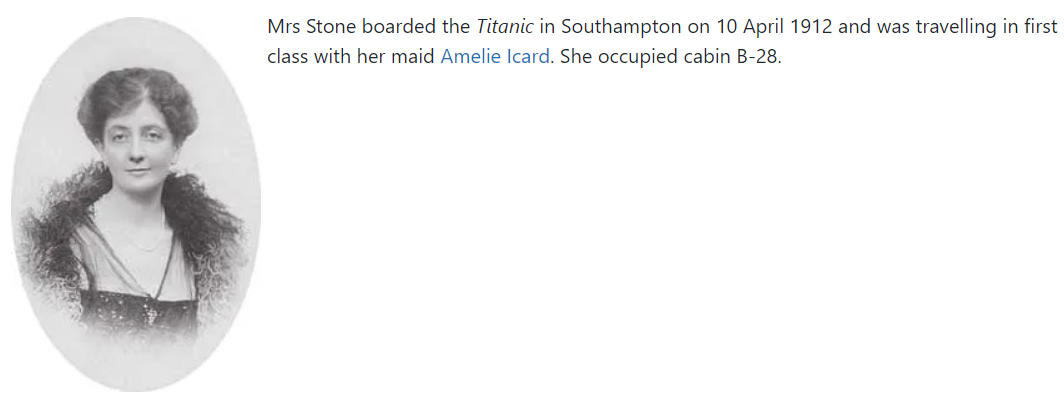
# найдем пассажиров с неизвестным портом посадки

# 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()]





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()

***(проверить результат)***

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']])

**(результат?)**

Теперь применим эту модель для заполнения пропусков.

# также используем двойные скобки с методом .transform()

const\_imputer['Age'] = imp\_const.transform(const\_imputer[['Age']])

# убедимся, что пропусков не осталось и посчитаем количество нулевых значений

const\_imputer.Age.isna().sum(), (const\_imputer['Age'] == 0).sum()

**(0, 177)**

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

# удалим его

const\_imputer.drop(columns = ['Embarked'], inplace = True)

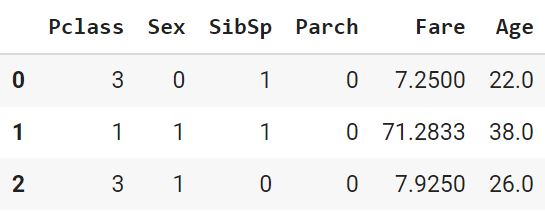
# и посмотрим на размер получившегося датафрейма

const\_imputer.shape

**(891, 7)**

# посмотрим на результат

const\_imputer.head(3)



**Заполнение средним арифметическим или медианой**

Количественные данные можно заполнить средним арифметическим или медианой (Statistical Imputation). Вначале воспользуемся методом .fillna().

# сделаем копию датафрейма

fillna\_median = titanic.copy()

# заполним пропуски в столбце Age медианным значением возраста,

# можно заполнить и средним арифметическим через метод .mean()

fillna\_median.Age.fillna(fillna\_median.Age.median(), inplace = True)

# убедимся, что пропусков не осталось

fillna\_median.Age.isna().sum()

**'0'**

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

***(какие?)***

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

# изменим размер последующих графиков

sns.set(rc = {'figure.figsize' : (10, 6)})

# скопируем датафрейм

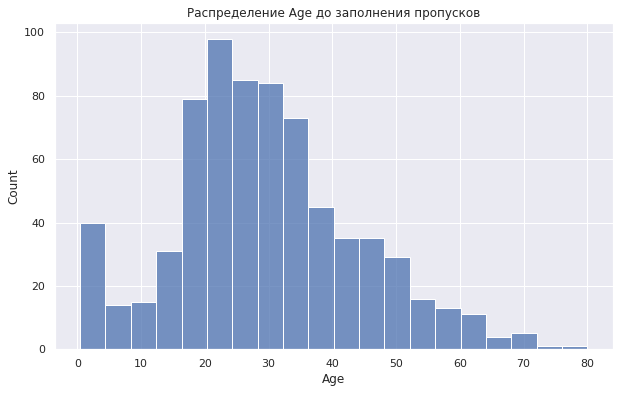
median\_imputer = titanic.copy()

# посмотрим на распределение возраста до заполнения пропусков

sns.histplot(median\_imputer['Age'], bins = 20)

plt.title('Распределение Age до заполнения пропусков');

Результат:



Посмотрим на среднее арифметическое и медиану.

median\_imputer['Age'].mean().round(1), median\_imputer['Age'].median()

(29.7, 28.0)

**Часть 2. (2 часа)**

Продолжаем рассмотрение методов предварительной обработки информации.

Предварительно загрузим необходимые библиотеки и датасет.

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import scipy

import missingno as msno

titanic = pd.read\_csv('train.csv')

Установим стиль отображения графики.

sns.set()

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

Если до сих пор этого не сделали, загружаем класс SimpleImputer.

from sklearn.impute import SimpleImputer

# 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()

**'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');

***(построить график, сравнить результаты)***

# посмотрим на метрики после заполнения медианой

median\_imputer['Age'].mean().round(1), median\_imputer['Age'].median()

**(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

**(результат?)**

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.

Выполним группировку с помощью метода .groupby() и найдем медианный возраст каждой группы.

# скопируем датафрейм

median\_imputer\_bins = titanic.copy()

# сгруппируем пассажиров по полу и классу каюты

Age\_bins = median\_imputer\_bins.groupby(['Sex', 'Pclass'])

# найдем медианный возраст с учетом получившихся групп

Age\_bins.Age.median()

Результат:

**Sex Pclass**

**0 1 40.0**

**2 30.0**

**3 25.0**

**1 1 35.0**

**2 28.0**

**3 21.5**

**Name: Age, dtype: float64**

Применим lambda-функцию к объекту SeriesGroupBy и заменим пропуски соответствующим медианным значением.

# объект SeriesGroupBy находится в переменной Age\_bins.Age,

# применим к нему lambda-функцию через метод .apply()

median\_imputer\_bins.Age = Age\_bins.Age.apply(lambda x: x.fillna(x.median()))

Убедимся, что в столбце Age не осталось пропусков.

# проверим пропуски в столбце Age

median\_imputer\_bins.Age.isna().sum()

**'0'**

Посмотрим на распределение.

sns.histplot(median\_imputer\_bins['Age'], bins = 20)

plt.title('Распределение Age после заполнения внутригрупповой медианой');



# 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()

**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']])

# убедимся, что пропусков не осталось

titanic\_mode.Embarked.isna().sum()

**'0'**

Проверим результат.

# количество пассажиров в категории S должно увеличиться на два

titanic\_mode.groupby('Embarked')['Survived'].count()

Embarked

C 168

Q 77

S 646

Name: PassengerId, dtype: int64

Примечание. Приведу еще один простой способ найти моду. Его можно использовать совместно с методом .fillna() для заполнения пропусков.

titanic.Embarked.value\_counts().index[0]

'S'

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.

# для работы с последующими методами столбец Embarked нам уже не нужен

titanic.drop(columns = ['Embarked'], inplace = True)

**Часть 3. (2 часа)**

**ПРО МНОГОМЕРНЫЕ МЕТОДЫ**

Более продвинутый подход — многомерные методы (Multivariate Imputation), заполнение пропусков одной переменной на основе данных других признаков. Другими словами, мы строим модель машинного обучения для заполнения пропусков.



Такой моделью может быть линейная регрессия для количественных признаков или логистическая — для категориальных. Рассмотрим пример линейной регрессии и сразу два подхода к ее реализации, детерминированный и стохастический.

**Линейная регрессия**

**Детерминированный подход**

Детерминированный подход (deterministic approach) предполагает, что мы заполняем пропуски строго теми значениями, которые будут предсказаны линейной регрессией.

**Подготовка данных**

Теперь давайте подготовим данные.

# сделаем копию датасета

lr = titanic.copy()

# импортируем класс StandardScaler модуля Preprocessing библиотеки sklearn

from sklearn.preprocessing import StandardScaler

# создаем объект этого класса

scaler = StandardScaler()

# применяем метод .fit\_transform() и сразу помещаем результат в датафрейм

lr = pd.DataFrame(scaler.fit\_transform(lr), columns = lr.columns)

# посмотрим на результат

lr.head(3)



В тестовую выборку мы поместим те наблюдения, в которых в столбце Age есть пропуски.

# создадим маску из пустых значений в столбце Age с помощью метода .isnull()

test = lr[lr['Age'].isnull()].copy()

test.head(3)



# посмотрим на количество таких строк

test.shape

(**проверить результат**)

В обучающей выборке напротив окажутся те строки, где в Age пропусков нет.

# используем метод .dropna(), чтобы избавиться от пропусков

train = lr.dropna().copy()

# оценим количество строк без пропусков

train.shape

результат: **(714, 7)**

Вместе обучающая и тестовая выборки должны дать \_\_\_\_ наблюдение.

|  |  |
| --- | --- |
|  | len(train) + len(test) |

Из датафрейма train выделим столбец Age. Это будет наша целевая переменная.

# целевая переменная может быть в формате Series

y\_train = train['Age']

# также не забудем удалить столбец Age из датафрейма признаков

X\_train = train.drop('Age', axis = 1)

# в test столбец Age не нужен в принципе

X\_test = test.drop('Age', axis = 1)

Оценим результат.

# на этих признаках мы будем учить нашу модель

X\_train.head(3)



# это будет нашей целевой переменной

y\_train.head(3)

**0 -0.530377**

**1 0.571831**

**2 -0.254825**

**Name: Age, dtype: float64**

# на этих данных мы будем строить прогноз (заполнять пропуски)

X\_test.head(3)



Мы готовы к обучению модели и заполнению пропусков.

**Обучение модели и заполнение пропусков**

Обучать модель линейной регрессии и строить прогноз мы уже умеем.

# импортируем класс LinearRegression

from sklearn.linear\_model import LinearRegression

# создадим объект этого класса

lr\_model = LinearRegression()

# обучим модель

lr\_model.fit(X\_train, y\_train)

# применим обученную модель к данным, в которых были пропуски в столбце Age

y\_pred = lr\_model.predict(X\_test)

# посмотрим на первые три прогнозных значения

y\_pred[:3]



Пропущенные значения заполнены. Остается обратно «собрать» датафрейм.

# присоединим прогнозные значения возраста к датафрейму test

test['Age'] = y\_pred

test.head(3)



Теперь у нас есть два датафрейма train и test со столбцом Age с заполненными пропусками.

# еще раз взглянем на датафрейм train

train.head(3)



Соединим их методом «один на другой» с помощью функции pd.concat().

lr = pd.concat([train, test])

lr.head(7)



Как вы видите, по сравнению с изначальным датафреймом порядок строк нарушился. После четвертого индекса сразу идет шестой, а строка с пятым индексом оказалась где-то в середине датафрейма.

# восстановим изначальный порядок строк, отсортировав их по индексу

lr.sort\_index(inplace = True)

lr.head(7)

Остается вернуть исходный масштаб.

# вернем исходный масштаб с помощью метода .inverse\_transform()

lr = pd.DataFrame(scaler.inverse\_transform(lr), columns = lr.columns)

# округлим столбец Age и выведем результат

lr.Age = lr.Age.round(1)

lr.head(7)



(**Оценить результат**)

В данном случае, если метод **.fit\_transform()** вычитает из каждого значения среднее и делит на СКО,

то метод **.inverse\_transform()** в обратном порядке умножает каждое число на СКО и прибавляет среднее арифметическое.

Проверим на наши данные. Подставим в формулу выше отмасштабированное значение возраста первого наблюдения (индекс 0).

(-0.530377 \* titanic.Age.std() + titanic.Age.mean()).round()

**22.0**

Убедимся в отсутствии пропусков и посмотрим на размеры получившегося датафрейма.

lr.Age.isna().sum(), lr.shape

**(0, (891, 7))**

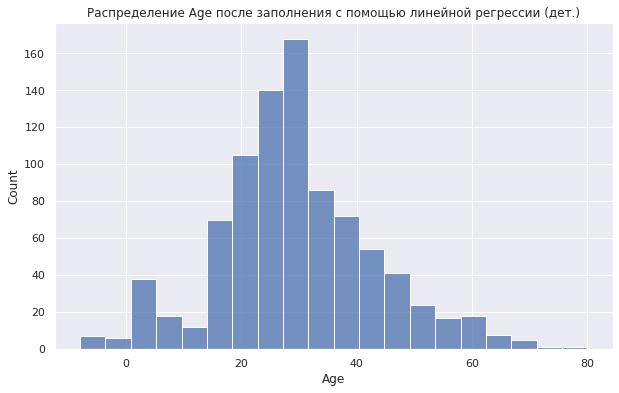
**Оценка результата**

Вначале построим гистограмму.

# посмотрим на распределение возраста после заполнения пропусков

sns.histplot(lr['Age'], bins = 20)

plt.title('Распределение Age после заполнения с помощью линейной регрессии (дет.)')



Распределение чуть больше похоже на нормальное, чем при заполнении медианой. Впрочем возникла одна проблема. Линейная регрессия предсказала нам отрицательные значения возраста, который разумеется должен быть только положительным. Как поступить? Просто удалить строки с неположительными значениями нам бы не хотелось, чтобы не терять с таким трудом восстановленные данные.

Воспользуемся методом .clip(), который установит минимальную границу значений столбца.

# установим минимальное значение на уровне 0,5 (полгода)

lr.Age.clip(lower = 0.5, inplace = True)

lr.Age.mean().round(1), lr.Age.median()

**Особенность детерминированного подхода**

В детерминированном подходе сохраняется та же особенность, которую мы наблюдали при заполнении пропусков медианой, а именно доминирование одного значения (в случае медианы) или узкого диапазона (в случае линейной регрессии).

Для того чтобы лучше это увидеть, во-первых, пометим изначальные (назовем их actual) и заполненные (imputed) значения столбца Age.

# сделаем копию датафрейма, которую используем для визуализации

lr\_viz = lr.copy()

# создадим столбец Age\_type, в который запишем actual, если индекс наблюдения есть в train,

# и imputed, если нет (т.е. он есть в test)

lr\_viz['Age\_type'] = np.where(lr.index.isin(train.index), 'actual', 'imputed')

# вновь "обрежем" нулевые значения

lr\_viz.Age.clip(lower = 0.5, inplace = True)

# посмотрим на результат

lr\_viz.head(7)



Создадим точечную диаграмму, где по оси x будет индекс датафрейма, по оси y — возраст, а цветом мы обозначим изначальное это значение, или заполненное.

sns.scatterplot(data = lr\_viz, x = lr\_viz.index, y = 'Age', hue = 'Age\_type')

plt.title('Распределение изначальных и заполненных значений (лин. регрессия, дет. подход)');

Результат:



На графике видно, что заполненные значения гораздо ближе к среднему значению (а зачастую просто равны ему), чем исходные данные. Аналогичную картину мы увидим, если рассчитаем соответствующие СКО.

lr\_viz[lr\_viz['Age\_type'] == 'actual'].Age.std(), lr\_viz[lr\_viz['Age\_type'] == 'imputed'].Age.std()

**Стохастический подход**

При применении стохастического подхода (Stochastic Regression Imputation) мы будем использовать гауссовский шум (Gaussian noise), то есть такой шум (элемент случайности), который следует нормальному распределению. Объявим соответствующую функцию.

# объявим функцию для создания гауссовского шума

# на входе эта функция будет принимать некоторый массив значений x,

# среднее значение mu, СКО std и точку отсчета для воспроизводимости результата

def gaussian\_noise(x, mu = 0, std = 1, random\_state = 42):

# вначале создадим объект, который позволит получать воспроизводимые результаты

rs = np.random.RandomState(random\_state)

# применим метод .normal() к этому объекту для создания гауссовского шума

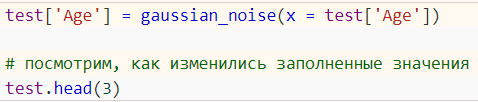
noise = rs.normal(mu, std, size = x.shape)

# добавим шум к исходному массиву

return x + noise

Среднее значение шума, равное нулю, мы взяли потому, что не хотим искажать колебания в ту или иную сторону. СКО равное, по умолчанию, единице указано исходя из того, что в линейной регрессии мы стремимся к работе со стандартизированными данными, отклонение которых как раз равно этой величине.

Заменим заполненные значения теми же значениями, но с добавлением шума.





Теперь соединим два датасета исходных и заполненных значений и оценим результат.

# соединим датасеты и обновим индекс

lr\_stochastic = pd.concat([train, test])

lr\_stochastic.sort\_index(inplace = True)

# вернем исходный масштаб с помощью метода .inverse\_transform()

lr\_stochastic = pd.DataFrame(scaler.inverse\_transform(lr\_stochastic), columns = lr\_stochastic.columns)

# округлим столбец Age и выведем результат

lr\_stochastic.Age = lr\_stochastic.Age.round(1)

lr\_stochastic.head(7)

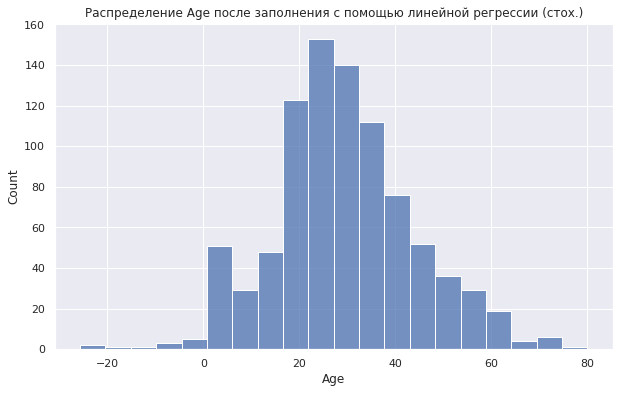


# посмотрим на распределение возраста

# после заполнения пропусков с помощью стохастического подхода

sns.histplot(lr\_stochastic['Age'], bins = 20)

plt.title('Распределение Age после заполнения с помощью линейной регрессии (стох.)');



Как мы видим, распределение еще больше похоже на нормальное.

# обрежем нулевые и отрицательные значения

lr\_stochastic.Age.clip(lower = 0.5, inplace = True)

Оценим среднее арифметическое и медиану.

lr\_stochastic.Age.mean().round(1), lr\_stochastic.Age.median()

**(29.3, 28.0)**

Медиана при стохастическом подходе вернулась к значению изначального распределения. Теперь с помощью точечной диаграммы оценим, как изменился разброс заполненных значений.

# сделаем копию датафрейма, которую используем для визуализации

lr\_st\_viz = lr\_stochastic.copy()

# создадим столбец Age\_type, в который запишем actual, если индекс наблюдения есть в train,

# и imputed, если нет (т.е. он есть в test)

lr\_st\_viz['Age\_type'] = np.where(lr\_stochastic.index.isin(train.index), 'actual', 'imputed')

# вновь "обрежем" нулевые значения

lr\_st\_viz.Age.clip(lower = 0.5, inplace = True)

# создадим график, где по оси x будет индекс датафрейма,

# по оси y - возраст, а цветом мы обозначим изначальное это значение, или заполненное

sns.scatterplot(data = lr\_st\_viz, x = lr\_st\_viz.index, y = 'Age', hue = 'Age\_type')

plt.title('Распределение изначальных и заполненных значений (лин. регрессия')



Как мы видим разброс заполненных значений существенно приблизился к разбросу изначальных данных. Сравним СКО.

lr\_st\_viz[lr\_st\_viz['Age\_type'] == 'actual'].Age.std().round(2), \

lr\_st\_viz[lr\_st\_viz['Age\_type'] == 'imputed'].Age.std().round(2)

Результат:

Забегая вперед скажем, что хотя у нас были основания для разработки стохастического подхода, детерминированный подход будет существенно более точно предсказывать пропуски, поскольку случайные колебания могут как улучшить, так и ухудшить качество заполненных пропусков.

Также заметим, что модель логистической регрессии для заполнения пропусков в категориальных данных строится аналогичным образом.

**Практическое занятие. Пропущенные значения. Часть 4. (2 часа)**

**MICE / IterativeImputer**

Описанный выше алгоритм регрессии используется в алгоритме MICE или IterativeImputer. MICE расшифровывается как Multiple Imputation by Chained Equations, многомерный способ заполнения пропущенных данных с помощью цепных уравнений.

**Принцип алгоритма MICE**

Рассмотрим MICE на несложном примере. Предположим, что у нас есть данные о заемщиках, а именно их возраст, стаж и уровень заработной платы, а также факт возвращения или невозвращения кредита. В этих данных есть пропущенные значения.



Кроме того, для целей оценки качества результата предположим, что нам известны истинные значения пропусков.



Теперь перейдем непосредственно к алгоритму MICE. Это итерационный алгоритм, другими словами, мы будем раз за разом повторять определенный набор действий до тех пор, пока не придем к нужному результату.

**Шаг 1.** Заполняем данные с помощью среднего арифметического (целевую переменную мы отбросим). Это наша отправная точка.



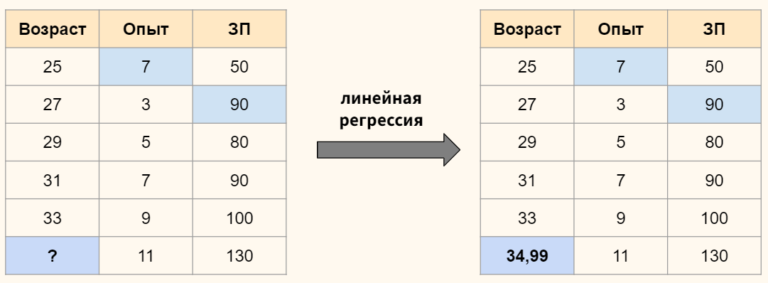
Эти значения очевидно далеко не оптимальны и еще раз демонстрируют ограниченность одномерных методов. В 25 лет сложно иметь семилетний стаж, три года стажа вряд ли обеспечат уровень заработной платы в 90 тысяч рублей (исходя из имеющихся данных), а человеку с 11 годами опыта скорее всего будет больше 29 лет.

Попробуем улучшить этот результат.

**Шаг 2.** Уберем заполненное только что значение возраста. Остальные заполненные значения трогать не будем.

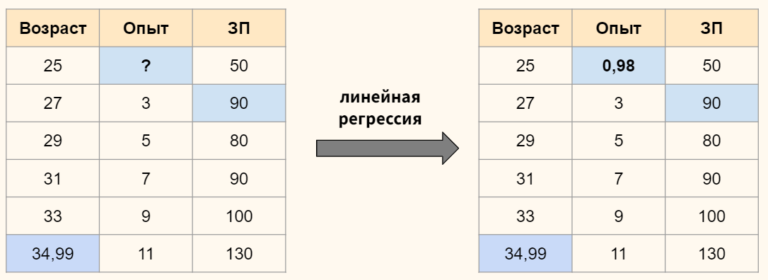


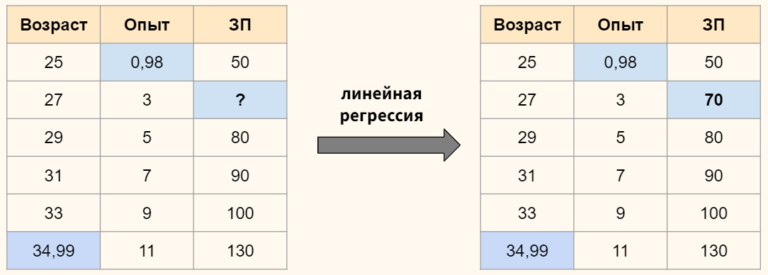
**Шаг 3.** Заполним пропуск с помощью линейной регрессии так, как мы это делали выше.



В данном случае «Опыт» и «ЗП» (кроме последней строки) будут признаками обучающей выборки (X\_train), возраст (кроме последней строки) — целевой переменной обучающей выборки (y\_train), «Опыт» и «ЗП» последней строки — признаками тестовой выборки (X\_test), а возраст последней строки — прогнозируемым значением (y\_pred).

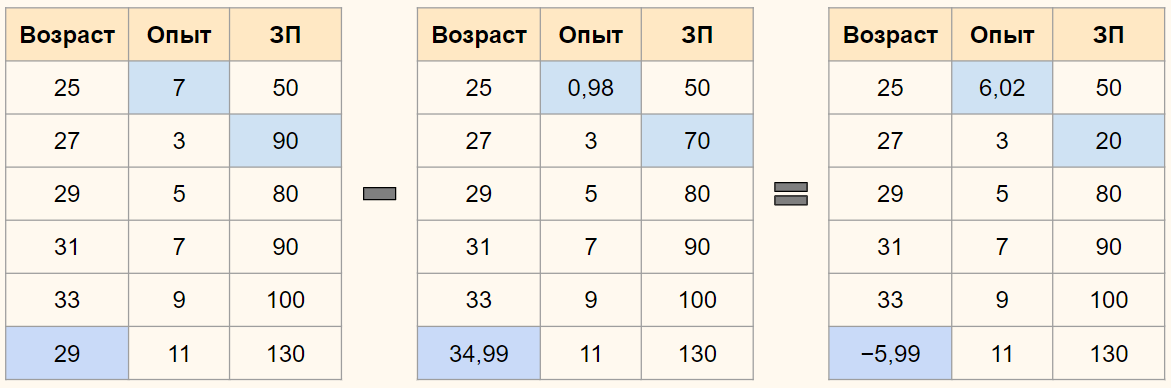
**Шаги 4 и 5.** Сделаем то же самое для двух других пропущенных значений.





Итак, мы завершили первую итерацию (цикл) работы алгоритма.

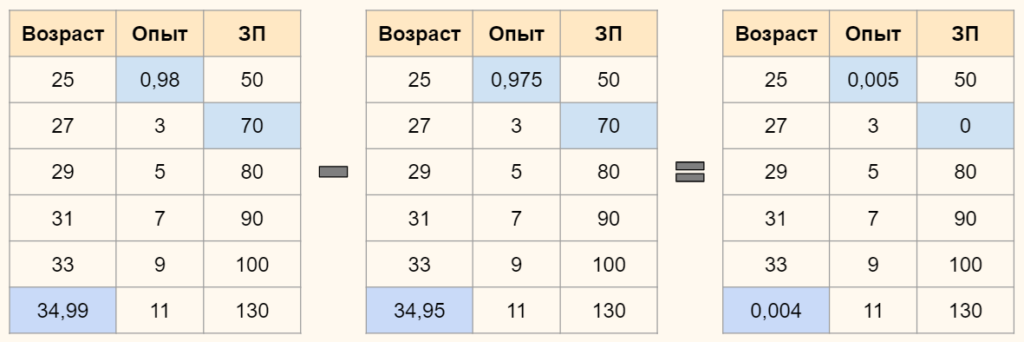
**Шаг 6.** Теперь давайте вычтем заполненные после первого цикла значения из исходного датасета, в котором пропуски представляют собой среднее арифметическое по столбцам.



Получившиеся разницы – (-5,99; 6,02; 20) – это критерий качества работы алгоритма.

*\*\*\* Наша задача повторять шаги с 3 по 5 (каждый раз используя новые заполненные значения в качестве отправной точки) до тех пор, пока разницы между двумя последними пропущенными значениями не будут близки к нулю. \*\*\**

Например, уже на второй итерации мы получим следующий результат.



Перейдем к практике.

**Реализация на Питоне через класс IterativeImputer**

Изначально алгоритм MICE был создан на языке R, но сегодня доступен в качестве экспериментального класса IterativeImputer в библиотеке sklearn.

# сделаем копию датасета для работы с методом MICE

mice = titanic.copy()

# предварительно нам нужно "включить" класс IterativeImputer,

from sklearn.experimental import enable\_iterative\_imputer

# затем импортировать его

from sklearn.impute import IterativeImputer

Теперь импортируем классы моделей, которые мы можем использовать внутри алгоритма MICE.

# в примере ограничимся использованием линейной регрессии

from sklearn.linear\_model import LinearRegression

from sklearn.linear\_model import BayesianRidge

from sklearn.ensemble import RandomForestRegressor

Так как в конечном счете мы снова имеем дело с линейной регрессией, будет разумно стандартизировать данные.

# создадим объект класса StandardScaler

scaler = StandardScaler()

# стандартизируем данные и сразу поместим их в датафрейм

mice = pd.DataFrame(scaler.fit\_transform(mice), columns = mice.columns)

Создадим объект класса IterativeImputer, заполним пропуски и укажем необходимые параметры.

mice\_imputer = IterativeImputer(initial\_strategy = 'mean', # вначале заполним пропуски средним значением

estimator = LinearRegression(), # в качестве модели используем линейную регрессию

random\_state = 42 # добавим точку отсчета

)

Используем метод .fit\_transform() для заполнения пропусков в датасете mice.

mice = mice\_imputer.fit\_transform(mice)

# вернем данные к исходному масштабу и округлим столбец Age

mice = pd.DataFrame(scaler.inverse\_transform(mice), columns = titanic.columns)

mice.Age = mice.Age.round(1)

mice.head(7)

Результатом будет что-то похожее на:



# убедимся, что пропусков не осталось

mice.Age.isna().sum(), mice.shape

Результат:

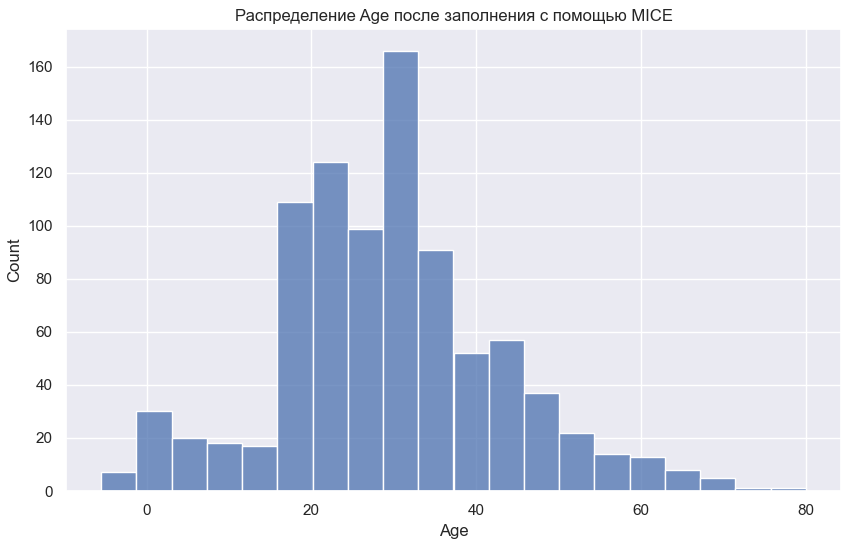
**(0, (891, 7))**

Оценим качество получившегося распределения.

# посмотрим на гистограмму возраста после заполнения пропусков

sns.histplot(mice['Age'], bins = 20)

plt.title('Распределение Age после заполнения с помощью MICE');



Так как мы заполняли пропуски линейной регрессией, у нас снова появились отрицательные значения.

# обрежем нулевые и отрицательные значения

mice.Age.clip(lower = 0.5, inplace = True)

Количественно оценим получившееся распределение.

# оценим среднее арифметическое и медиану

mice.Age.mean().round(1), mice.Age.median()

Результат:

**(29.3, 28.3)**

# сравним СКО исходного датасета и данных после алгоритма MICE

titanic.Age.std().round(2), mice.Age.std().round(2)

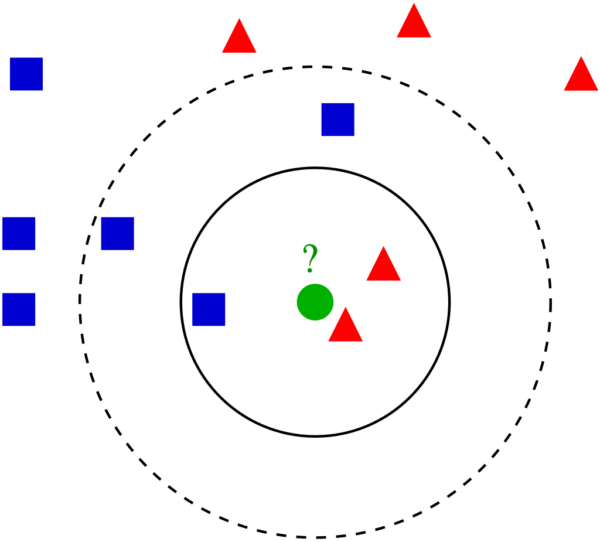
Результат:

**(14.53, 13.54)**

Рассмотрим принципиально другой алгоритм для заполнения пропусков, а именно метод ближайших соседей.

**Метод k-ближайших соседей**

(k-nearest neighbors algorithm, k-NN). рассмотрим работу этого алгоритма на примере задачи классификации.



На рисунке мы рассчитали расстояние от зеленой точки (наблюдение или вектор, класс которого мы хотим предсказать) до ближайших соседей.

Если мы возьмем k равное трем, то в число соседей войдут точки внутри меньшей окружности (сплошная линия). Если пяти – внутри большей окружности (прерывистая линия). При k = 3 большая часть соседей – красные треугольники. Именно к этой категории мы и отнесем зеленую точку. Если взять k = 5, зеленая точка будет классифицирована как синий квадрат.

Для количественной целевой переменной мы можем найти, например, среднее арифметическое k-ближайших соседей.

В задаче заполнения пропусков мы сначала найдем соседей наблюдения с пропущенным значением (например, других пассажиров «Титаника»), а затем заполним пропуск (в частности, возраст) средним арифметическим значений этого столбца наблюдений соседей (т.е. средним возрастом других пассажиров).

Также не будем забывать, что так как речь идет об алгоритме, учитывающем расстояние, нам обязательно нужно масштабировать данные.

Перейдем к практике.

**Sklearn KNNImputer**

Вначале рассмотрим реализацию этого алгоритма в библиотеке Sklearn. Скопируем и масштабируем данные.

# сделаем копию датафрейма

knn = titanic.copy()

# создадим объект класса StandardScaler

scaler = StandardScaler()

# масштабируем данные и сразу преобразуем их обратно в датафрейм

knn = pd.DataFrame(scaler.fit\_transform(knn), columns = knn.columns)

Теперь воспользуемся классом KNNImputer для заполнения пропусков.

# импортируем класс KNNImputer

from sklearn.impute import KNNImputer

# создадим объект этого класса с параметрами:

# пять соседей и однаковым весом каждого из них

knn\_imputer = KNNImputer(n\_neighbors = 5, weights = 'uniform')

# заполним пропуски в столбце Age

knn = pd.DataFrame(knn\_imputer.fit\_transform(knn), columns = knn.columns)

# проверим отсутствие пропусков и размеры получившегося датафрейма

knn.Age.isna().sum(), knn.shape

Результат:

**(0, (891, 7))**

Вернем исходный масштаб данных.

knn = pd.DataFrame(scaler.inverse\_transform(knn), columns = knn.columns)

# округлим значение возраста

knn.Age = knn.Age.round(1)

# посмотрим на результат

knn.head(7)

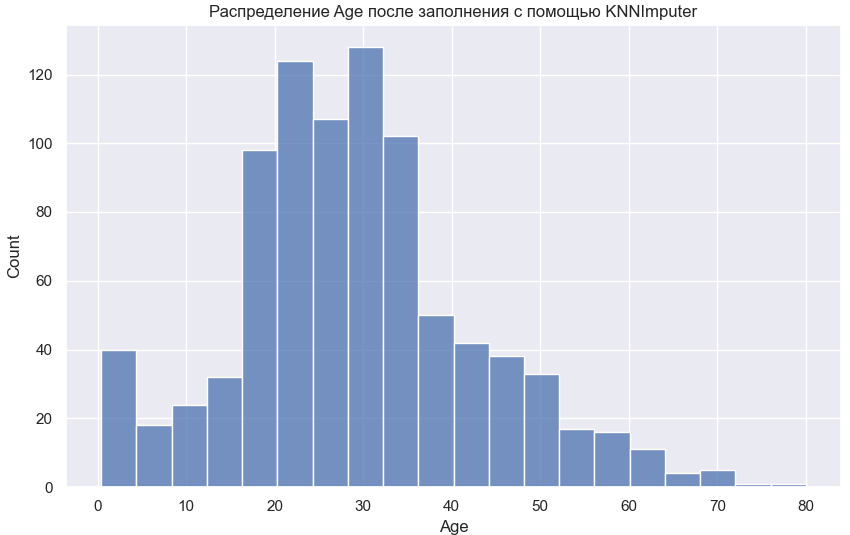


Осталось взглянуть на получившееся распределение.

# посмотрим на распределение возраста после заполнения пропусков

sns.histplot(knn['Age'], bins = 20)

plt.title('Распределение Age после заполнения с помощью KNNImputer');



Распределение близко к нормальному.

**Метод impyute fast\_knn**

**Особенности метода ближайших соседей.**

Метод ближайших соседей прост и в то же время эффективен.

*При этом в базовом варианте его реализации у него есть один недостаток – долгое время работы или, как правильнее сказать, высокая временная сложность (time complexity) алгоритма.*

Давайте рассмотрим механику этого метода чуть подробнее. В первую очередь, для лучшего понимания, введем несколько неформальных терминов:

* назовем вектором запроса (query vector) то новое наблюдение, для которого мы хотим найти ближайшие к нему вектора (зеленая точка на изображении выше)
* вектором сравнения (reference vector) будет то наблюдение, которое уже содержит разметку (класс или числовое значение) или в котором отсутствует пропуск (все остальные точки)

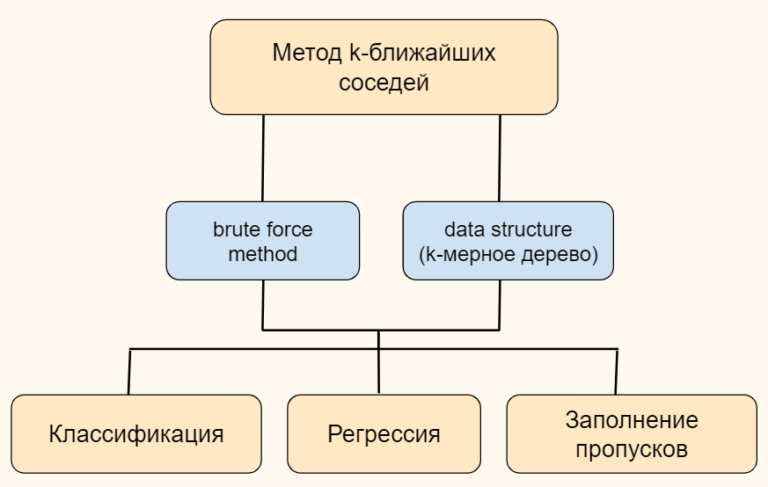
Алгоритм k-ближайших соседей потребует двух циклов.

1. В первом цикле мы будем поочередно брать по одному вектору запроса
2. Во втором вложенном в него цикле мы будем для каждого вектора запроса находить расстояние до всех векторов сравнения.
3. Наконец, найдя и отсортировав вектора сравнения по расстоянию, выберем для каждого вектора запроса k-ближайших.
4. Далее приступим к решению задачи классификации, регрессии (хотя термин регрессия здесь не вполне корректно применять) или заполнению пропуска.

При большом количестве наблюдений (как векторов запроса, так и векторов сравнения) такая работа может занять очень много времени. Такой вариант реализации алгоритма, его еще называют ***методом перебора*** или методом опробования (brute force method), является возможным, но не единственным решением поставленной задачи.

Другим возможным решением является преобразование векторов сравнения в такую ***структуру данных***, по которой поиск соседей будет происходить быстрее (то есть мы оптимизируем второй, вложенный цикл, с первым мы сделать ничего не можем, нам в любом случае нужно перебрать все вектора запроса).

Такой структурой данных может быть, в частности, **k-мерное дерево** или **k-d-дерево** (k-diminsional tree, k-d tree).



Во многом сказанное выше перекликается с различиями между алгоритмами линейного и бинарного поиска, которые мы рассмотрели ранее.

В дополнительных материалах к сегодняшнему занятию мы отдельно поговорим про временную сложность алгоритмов. Реализацию же различных вариантов метода k-ближайших соседей и создания k-мерных деревьев мы отложим до курса традиционных алгоритмов ML.

А сейчас давайте обратимся к библиотеке impyute и классу fast\_knn, который как раз реализует быстрый поиск ближайших соседей по k-мерному дереву (отсюда и название этого класса).

**Функция fast\_knn()**

Вначале подготовим данные.

# сделаем копию датафрейма и масштабируем данные

impyute = titanic.copy()

scaler = StandardScaler()

impyute = scaler.fit\_transform(impyute)

# на выходе в переменной impyute содержится массив Numpy

type(impyute)

Вывод:

**numpy.ndarray**

Установим библиотеку impyute.

Для Питона:

!pip install impyute

Для Анаконды:

conda install impyute

Теперь воспользуемся фунцией fast\_knn() для заполнения пропусков.

# импортируем функцию fast\_knn

from impyute.imputation.cs import fast\_knn

# передаем этой функции массив Numpy и количество соседей

# (если данные содержатся в формате датафрейма, нужно использовать атрибут .values)

impyute = fast\_knn(impyute, k = 5)

# возвращаем исходный масштаб и преобразуем в датафрейм

impyute = pd.DataFrame(scaler.inverse\_transform(impyute), columns = titanic.columns)

# округляем столбец возраста

impyute.Age = impyute.Age.round(1)

# проверим отсутствие пропусков и размеры получившегося датафрейма

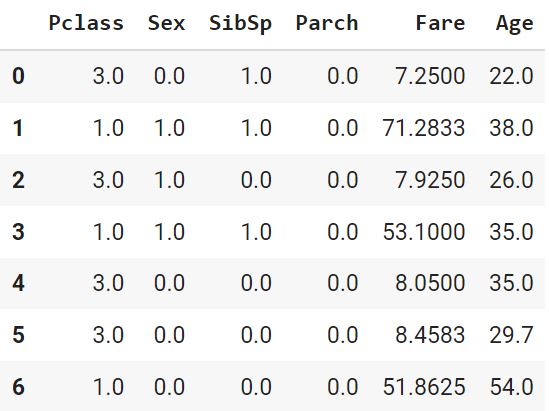
impyute.Age.isna().sum(), impyute.shape

Получим:

**(0, (891, 7))**

# оценим результат

impyute.head(7)



**Сравнение методов**

Пришло время оценить качество алгоритмов заполнения пропусков. Для сравнения рассмотренных выше методов сделаем следующее:

Возьмем получившиеся в результате применения каждого из методов одинаковые датасеты (различаются только значения, которыми были заполнены пропуски)

Используем модель логистической регрессии для построения прогноза выживания пассажиров. Тот метод заполнения пропусков, с которым модель логистической регрессии покажется наилучший результат и будет считаться победителем

Вначале создадим два списка:

* в первый поместим все датасеты с заполненным столбцом Age
* во второй, соответствующие названия методов

datasets = [const\_imputer, median\_imputer, lr, lr\_stochastic, mice, knn]

methods = ['constant', 'median', 'linear regression', 'stochastic linear regression', 'MICE','KNNImputer']

Возьмем целевую переменную из исходного файла, так как мы не использовали ее при заполнении пропусков.

y = pd.read\_csv('train.csv')['Survived']

Импортируем класс LogisticRegression и функцию accuracy\_score() для оценки качества модели.

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

Теперь в цикле обучим модель на каждом из датасетов, сделаем прогноз, оценим и выведем результат.

# в цикле пройдемся по датасетам с заполненными пропусками

# и списком названий соответствующих методов

for X, method in zip(datasets, methods):

# масштабируем признаки

X = StandardScaler().fit\_transform(X)

# для каждого датасета построим и обучим модель логистической регрессии

model = LogisticRegression()

model.fit(X, y)

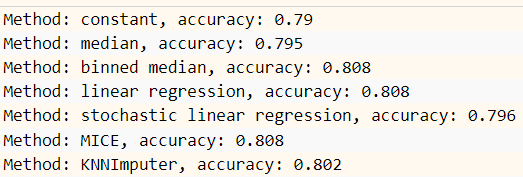
# сделаем прогноз

y\_pred = model.predict(X)

# выведем название использованного метода и достигнутую точность

print(f'Method: {method}, accuracy: {np.round(accuracy\_score(y, y\_pred), 3)}')

Результат:



Мы видим, что, используя более сложные методы заполнения пропусков, мы в среднем добились более высокой точности финальной модели.

Отдельно **обратите внимание** на хорошие результаты заполнения пропусков внутригрупповой медианой (binned median) и наоборот невысокую точность алгоритма стохастической линейной регрессии. Во втором случае, снижение точности объясняется как раз тем, что мы чаще не угадывали куда должна двигаться вариативность, нежели оказывались правы.

**Уточнение.** Хотя в классификации выше метод заполнения внутригрупповым показателем отнесен к одномерным методам, де факто это метод многомерный, поскольку среднее или медиана столбца с пропусками рассчитываются на основе группировки по другим столбцам датасета.

**Дополнение.** Еще одним способом работы с пропусками является создание переменной-индикатора (indicator method), которая принимает значение 1, если пропуск присутствует, и 0, если отсутствует.

На отдельной странице приведены дополнительные материалы к этому занятию.

**Подведем итог**

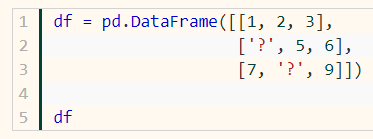
Сегодня мы рассмотрели теоретические основы, а также практические способы заполнения пропусков в перекрестных данных.

В частности мы узнали, что пропуски можно заполнить, используя одномерные или многомерные методы. Сравнение этих методов показало, что многомерные методы в среднем показывают более высокие результаты.

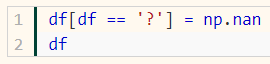
**Ответы на вопросы**

**Вопрос.** Что делать, если пропуски заполнены каким-либо символом, а не NaN? Например, знаком вопроса.

**Ответ.** Вначале нужно превратить этот или эти символы в NaN, а дальше работать как со стандартными пропусками.









**Вопрос.** Чем метод .isnull() отличается от метода .isna()?

**Ответ.** Это одно и то же.

**Вопрос.** Некоторые авторы указывают, что пропуски типа MNAR зависят только от отсутствующих значений. Другими словами,

**Ответ.** Да, действительно, встречаются разные определения. Здесь важно, что пропуски типа MNAR в любом случае в какой-то степени зависят именно от отсутствующих значений, о которых мы по определению ничего не знаем, а значит не знаем как предсказывать такие пропуски.