Машинное обучение

Пропущенные значения. Часть 2

Материалы > Анализ и обработка данных

Перейдем к многомерным методам заполнения пропусков.

> Продолжим работу в том же блокноте

Про многомерные методы

Многомерные методы (multivariate imputation) предполагают заполнение пропусков одной переменной на основе данных других признаков. Другими словами, мы строим модель машинного обучения для заполнения пропусков.

id	Признак 1	Признак 2	Признак 3	Целевая переменная
0	53	12	0	1
1	~~NaN~~	28	1	0
2	48	20	0	0
3	~~NaN~~	24	0	1
4	57	35	1	0
5	60	31	1	1

Такой моделью может быть линейная регрессия для количественных признаков или логистическая — для категориальных.

> Обратите внимание, что хотя технически мы могли бы использовать исходную целевую переменную в качестве одного из признаков для заполнения пропусков, делать этого не

стоит, потому что в этом случае мы создадим между ними взаимосвязь, которой изначально могло не быть.

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)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	0.827377	-0.737695	0.432793	-0.473674	-0.502445	-0.530377
1	-1.566107	1.355574	0.432793	-0.473674	0.786845	0.571831
2	0.827377	1.355574	-0.474545	-0.473674	-0.488854	-0.254825

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)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
5	0.827377	-0.737695	-0.474545	-0.473674	-0.478116	NaN
17	-0.369365	-0.737695	-0.474545	-0.473674	-0.386671	NaN
19	0.827377	1.355574	-0.474545	-0.473674	-0.502949	NaN

```
1  # посмотрим на количество таких строк
2  test.shape
```

```
1                (177, 7)
```

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

```
1 # используем метод .dropna(), чтобы избавиться от пропусков
2 train = lr.dropna().copy()
3
4 # оценим количество строк без пропусков
5 train.shape
```

```
1                (714, 7)
```

Вместе обучающая и тестовая выборки должны дать 891 наблюдение.

```
1      len(train) + len(test)
```

```
1                          891
```

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

```
1 # целевая переменная может быть в формате Series
2 y_train = train['Age']
3
4 # также не забудем удалить столбец Age из датафрейма признаков
5 X_train = train.drop('Age', axis = 1)
6
7 # в test столбец Age не нужен в принципе
8 X_test = test.drop('Age', axis = 1)
```

Оценим результат.

```
1  # на этих признаках мы будем учить нашу модель
2  X_train.head(3)
```

	Pclass	Sex	SibSp	Parch	Fare
0	0.827377	-0.737695	0.432793	-0.473674	-0.502445
1	-1.566107	1.355574	0.432793	-0.473674	0.786845
2	0.827377	1.355574	-0.474545	-0.473674	-0.488854

```
1  # 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)
```

	Pclass	Sex	SibSp	Parch	Fare
5	0.827377	-0.737695	-0.474545	-0.473674	-0.478116
17	-0.369365	-0.737695	-0.474545	-0.473674	-0.386671
19	0.827377	1.355574	-0.474545	-0.473674	-0.502949

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]
```

```
array([-0.09740093,  0.37999257, -0.31925429])
```

Пропущенные значения заполнены. Остается «собрать» датафрейм обратно.

```
# присоединим прогнозные значения возраста к датафрейму test
test['Age'] = y_pred
test.head(3)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
5	0.827377	-0.737695	-0.474545	-0.473674	-0.478116	-0.097401
17	-0.369365	-0.737695	-0.474545	-0.473674	-0.386671	0.379993

19	0.827377	1.355574	-0.474545	-0.473674	-0.502949	-0.319254

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

```
1 # еще раз взглянем на датафрейм train
2 train.head(3)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	0.827377	-0.737695	0.432793	-0.473674	-0.502445	-0.530377
1	-1.566107	1.355574	0.432793	-0.473674	0.786845	0.571831
2	0.827377	1.355574	-0.474545	-0.473674	-0.488854	-0.254825

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

```
1 lr = pd.concat([train, test])
2 lr.head(7)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	0.827377	-0.737695	0.432793	-0.473674	-0.502445	-0.530377
1	-1.566107	1.355574	0.432793	-0.473674	0.786845	0.571831
2	0.827377	1.355574	-0.474545	-0.473674	-0.488854	-0.254825
3	-1.566107	1.355574	0.432793	-0.473674	0.420730	0.365167
4	0.827377	-0.737695	-0.474545	-0.473674	-0.486337	0.365167
6	-1.566107	-0.737695	-0.474545	-0.473674	0.395814	1.674039
7	0.827377	-0.737695	2.247470	0.767630	-0.224083	-1.908136

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.

```
1 # восстановим изначальный порядок строк, отсортировав их по индексу
2 lr.sort_index(inplace = True)
3 lr.head(7)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	0.827377	-0.737695	0.432793	-0.473674	-0.502445	-0.530377
1	-1.566107	1.355574	0.432793	-0.473674	0.786845	0.571831
2	0.827377	1.355574	-0.474545	-0.473674	-0.488854	-0.254825
3	-1.566107	1.355574	0.432793	-0.473674	0.420730	0.365167
4	0.827377	-0.737695	-0.474545	-0.473674	-0.486337	0.365167
5	0.827377	-0.737695	-0.474545	-0.473674	-0.478116	-0.097401
6	-1.566107	-0.737695	-0.474545	-0.473674	0.395814	1.674039

Остается вернуть исходный масштаб.

```
1 # вернем исходный масштаб с помощью метода .inverse_transform()
2 lr = pd.DataFrame(scaler.inverse_transform(lr), columns = lr.columns)
3
4 # округлим столбец Age и выведем результат
5 lr.Age = lr.Age.round(1)
6 lr.head(7)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	3.0	0.0	1.0	0.0	7.2500	22.0
1	1.0	1.0	1.0	0.0	71.2833	38.0
2	3.0	1.0	0.0	0.0	7.9250	26.0
3	1.0	1.0	1.0	0.0	53.1000	35.0
4	3.0	0.0	0.0	0.0	8.0500	35.0
5	3.0	0.0	0.0	0.0	8.4583	28.3
6	1.0	0.0	0.0	0.0	51.8625	54.0

В данном случае, если **метод .fit_transform()** класса StandardScaler вычитает из каждого значения среднее и делит на СКО,

$$z = \frac{x - \overline{x}}{\sigma}$$

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

$$x = z \cdot \sigma + \overline{x}$$

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

```
1 (-0.530377 * titanic.Age.std() + titanic.Age.mean()).round()
```

```
1    22.0
```

Убедимся в отсутствии пропусков и посмотрим на размеры получившегося датафрейма.

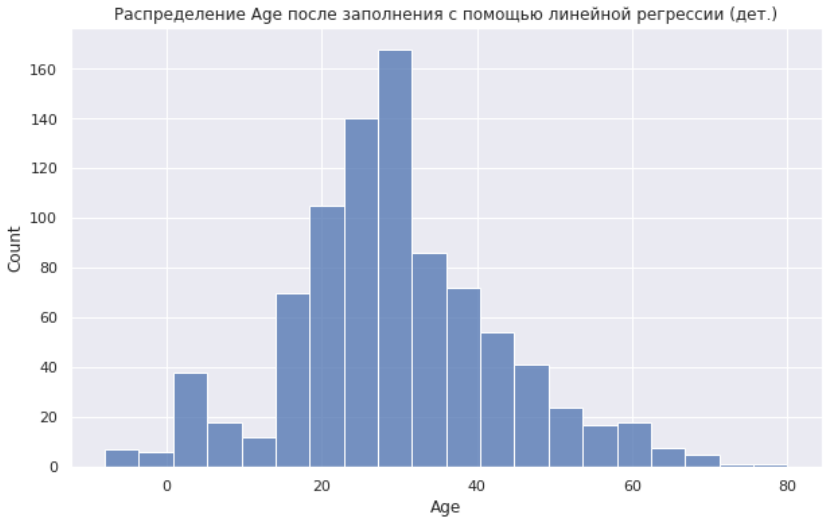
```
1 lr.Age.isna().sum(), lr.shape
```

```
1    (0, (891, 6))
```

Оценка результата

Вначале построим гистограмму.

```
1# посмотрим на распределение возраста после заполнения пропусков
2sns.histplot(lr['Age'], bins = 20)
3plt.title('Распределение Age после заполнения с помощью линейной регрессии (дет.)');
```



Как мы видим, 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.

```
1 # установим минимальное значение на уровне 0,5 (полгода)
2 lr.Age.clip(lower = 0.5, inplace = True)
```

Остается посмотреть на новые средние показатели.

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

```
1               (29.3, 28.3)
```

Среднее арифметическое и медиана практически не изменились.

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 значение 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)
```

	Pclass	Sex	SibSp	Parch	Fare	Age	Age_type
0	3.0	0.0	1.0	0.0	7.2500	22.0	actual
1	1.0	1.0	1.0	0.0	71.2833	38.0	actual
2	3.0	1.0	0.0	0.0	7.9250	26.0	actual
3	1.0	1.0	1.0	0.0	53.1000	35.0	actual
4	3.0	0.0	0.0	0.0	8.0500	35.0	actual
5	3.0	0.0	0.0	0.0	8.4583	28.3	imputed
6	1.0	0.0	0.0	0.0	51.8625	54.0	actual

Во-вторых, 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)');
```

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().round(2), \
lr_viz[lr_viz['Age_type'] == 'imputed'].Age.std().round(2)
```

```
(14.53, 8.33)
```

Можно предположить, что детерминированный подход переоценивает корреляцию между признаками, и, как следствие, преувеличивает точность прогноза пропущенных значений.

Попробуем преодолеть этот недостаток через внесение в заполняемые данные элемента случайности (дополнительных колебаний).

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.

```
test['Age'] = gaussian_noise(x = test['Age'])

# посмотрим, как изменились заполненные значения
test.head(3)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
5	0.827377	-0.737695	-0.474545	-0.473674	-0.478116	0.399313
17	-0.369365	-0.737695	-0.474545	-0.473674	-0.386671	0.241728

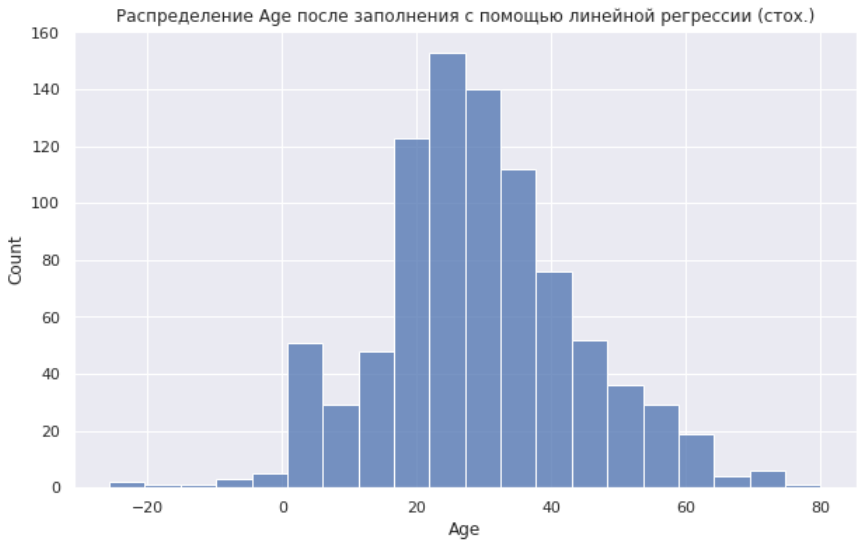
19	0.827377	1.355574	-0.474545	-0.473674	-0.502949	0.328434

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)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	3.0	0.0	1.0	0.0	7.2500	22.0
1	1.0	1.0	1.0	0.0	71.2833	38.0
2	3.0	1.0	0.0	0.0	7.9250	26.0
3	1.0	1.0	1.0	0.0	53.1000	35.0
4	3.0	0.0	0.0	0.0	8.0500	35.0
5	3.0	0.0	0.0	0.0	8.4583	35.5
6	1.0	0.0	0.0	0.0	51.8625	54.0

```
# 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
```

```
2 lr_stochastic.Age.clip(lower = 0.5, inplace = True)
```

Оценим среднее арифметическое и медиану.

```
1 lr_stochastic.Age.mean().round(1), lr_stochastic.Age.median()
```

```
1 (29.3, 28.0)
```

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

```
1 # сделаем копию датафрейма, которую используем для визуализации
2 lr_st_viz = lr_stochastic.copy()
3
4 # создадим столбец Age_type, в который запишем actual, если индекс наблюдения есть в train,
5 # и imputed, если нет (т.е. он есть в test)
6 lr_st_viz['Age_type'] = np.where(lr_stochastic.index.isin(train.index), 'actual', 'imputed')
7
8 # вновь "обрежем" нулевые значения
9 lr_st_viz.Age.clip(lower = 0.5, inplace = True)
10
11 # создадим график, где по оси x будет индекс датафрейма,
12 # по оси y - возраст, а цветом мы обозначим изначальное это значение, или заполненное
13 sns.scatterplot(data = lr_st_viz, x = lr_st_viz.index, y = 'Age', hue = 'Age_type')
14 plt.title('Распределение изначальных и заполненных значений (лин. регрессия, стох. подход)');
```

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

```
1 lr_st_viz[lr_st_viz['Age_type'] == 'actual'].Age.std().round(2), \
2 lr_st_viz[lr_st_viz['Age_type'] == 'imputed'].Age.std().round(2)
```

1 (14.53, 14.34)

Забегая вперед скажу, 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 замечу, что модель логистической регрессии для заполнения пропусков в категориальных данных строится аналогичным образом.

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.

Возраст	Опыт	ЗП	Кредит
25		50	1
27	3		1
29	5	80	0
31	7	90	0
33	9	100	1
	11	130	0

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

Возраст	Опыт	ЗП	Кредит
25	1	50	1
27	3	70	1
29	5	80	0
31	7	90	0
33	9	100	1
35	11	130	0

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.

Возраст	Опыт	ЗП
25	7	50
27	3	90
29	5	80
31	7	90
33	9	100
29	11	130

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.

Возраст	Опыт	ЗП
25	7	50
27	3	90
29	5	80
31	7	90
33	9	100
	11	130

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

Возраст	Опыт	ЗП
25	7	50
27	3	90
29	5	80
31	7	90
33	9	100
?	11	130

Возраст	Опыт	ЗП
25	7	50
27	3	90
29	5	80
31	7	90
33	9	100
34,99	11	130

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.

Возраст	Опыт	ЗП
25	**?**	50
27	3	90
29	5	80
31	7	90
33	9	100
34,99	11	130

линейная регрессия

Возраст	Опыт	ЗП
25	**0,98**	50
27	3	90
29	5	80
31	7	90
33	9	100
34,99	11	130

Возраст	Опыт	ЗП
25	0,98	50
27	3	**?**
29	5	80
31	7	90
33	9	100
34,99	11	130

линейная регрессия

Возраст	Опыт	ЗП
25	0,98	50
27	3	**70**
29	5	80
31	7	90
33	9	100
34,99	11	130

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.

Возраст	Опыт	ЗП
25	7	50
27	3	90
29	5	80
31	7	90
33	9	100
29	11	130

−

Возраст	Опыт	ЗП
25	0,98	50
27	3	70
29	5	80
31	7	90
33	9	100
34,99	11	130

=

Возраст	Опыт	ЗП
25	6,02	50
27	3	20
29	5	80
31	7	90
33	9	100
−5,99	11	130

Получившиеся разницы $-5{,}99$, $6{,}02$ и 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.

Возраст	Опыт	ЗП
25	0,98	50
27	3	70
29	5	80
31	7	90
33	9	100
34,99	11	130

=

Возраст	Опыт	ЗП
25	0,975	50
27	3	70
29	5	80
31	7	90
33	9	100
34,95	11	130

=

Возраст	Опыт	ЗП
25	0,005	50
27	3	0
29	5	80
31	7	90
33	9	100
0,004	11	130

Перейдем к практике.

Реализация на Питоне через класс IterativeImputer

```
1 # сделаем копию датасета для работы с методом MICE
2 mice = titanic.copy()
```

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

```
1 # предварительно нам нужно "включить" класс IterativeImputer,
2 from sklearn.experimental import enable_iterative_imputer
3 # затем импортировать его
4 from sklearn.impute import IterativeImputer
```

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

```
1 # в сегодняшнем примере ограничимся использованием линейной регрессии
2 from sklearn.linear_model import LinearRegression
3 from sklearn.linear_model import BayesianRidge
4 from sklearn.ensemble import RandomForestRegressor
```

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

```
1 # создадим объект класса StandardScaler
2 scaler = StandardScaler()
3
4 # стандартизируем данные и сразу поместим их в датафрейм
5 mice = pd.DataFrame(scaler.fit_transform(mice), columns = mice.columns)
```

Создадим объект класса IterativeImputer и заполним пропуски.

```
# создадим объект класса IterativeImputer 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)
```

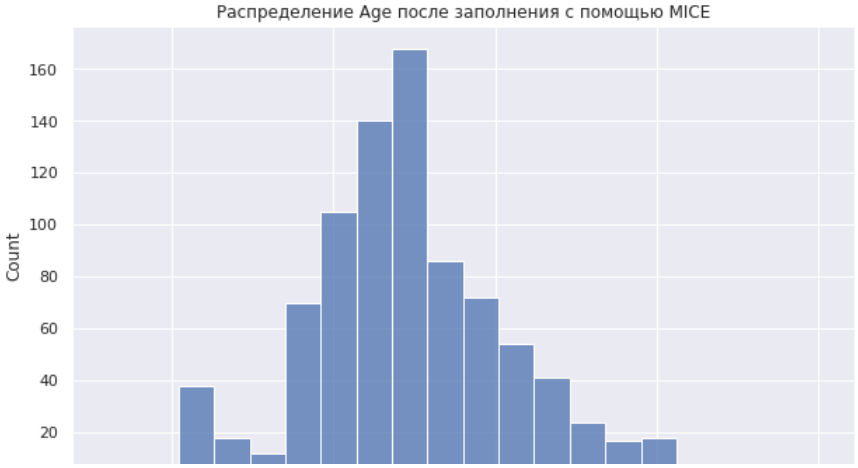
	Pclass	Sex	SibSp	Parch	Fare	Age
0	3.0	0.0	1.0	0.0	7.2500	22.0
1	1.0	1.0	1.0	0.0	71.2833	38.0
2	3.0	1.0	0.0	0.0	7.9250	26.0
3	1.0	1.0	1.0	0.0	53.1000	35.0
4	3.0	0.0	0.0	0.0	8.0500	35.0
5	3.0	0.0	0.0	0.0	8.4583	28.3
6	1.0	0.0	0.0	0.0	51.8625	54.0

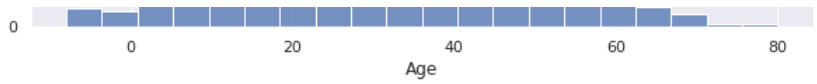
```
# убедимся, что пропусков не осталось
mice.Age.isna().sum(), mice.shape
```

```
(0, (891, 6))
```

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.

```
1  # обрежем нулевые и отрицательные значения
2  mice.Age.clip(lower = 0.5, inplace = True)
```

Количественно оценим получившееся распределение.

```
1  # оценим среднее арифметическое и медиану
2  mice.Age.mean().round(1), mice.Age.median()
```

```
1              (29.3, 28.3)
```

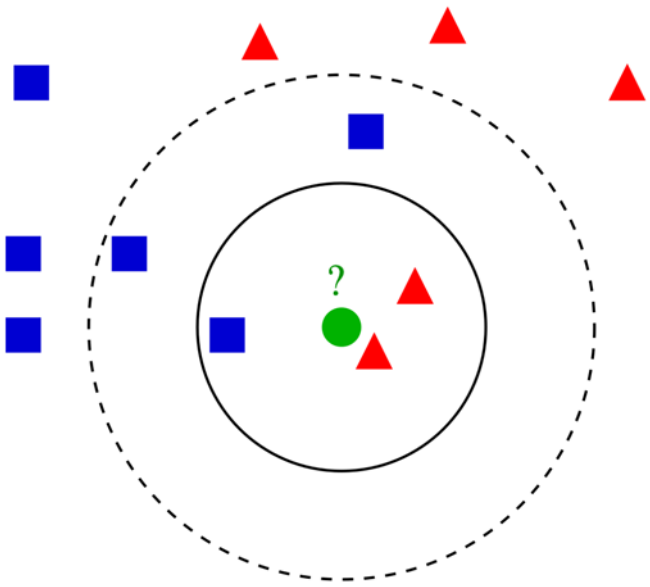
```
1 # сравним СКО исходного датасета и данных после алгоритма MICE
2 titanic.Age.std().round(2), mice.Age.std().round(2)
```

```
1           (14.53, 13.54)
```

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

Метод k-ближайших соседей

Мы уже использовали **алгоритм k-ближайших соседей** (k-nearest neighbors algorithm, k-NN) для создания рекомендательной системы. Еще раз рассмотрим работу этого алгоритма на примере *задачи классификации*.



Источник: Википедия

На рисунке выше 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 $k = 3$ бóльшая 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

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
```

```
(0, (891, 6))
```

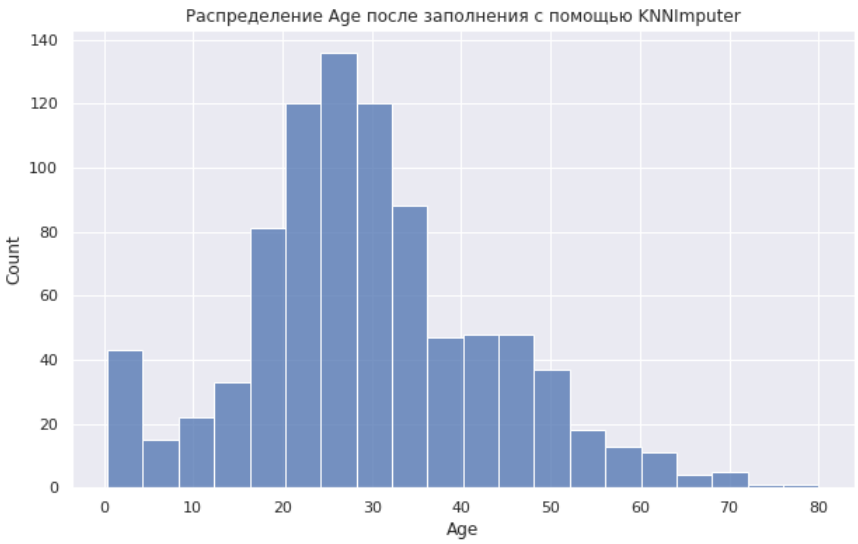
Вернем исходный масштаб данных.

```
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)
```

	Pclass	Sex	SibSp	Parch	Fare	Age
0	3.0	0.0	1.0	0.0	7.2500	22.0
1	1.0	1.0	1.0	0.0	71.2833	38.0
2	3.0	1.0	0.0	0.0	7.9250	26.0
3	1.0	1.0	1.0	0.0	53.1000	35.0
4	3.0	0.0	0.0	0.0	8.0500	35.0
5	3.0	0.0	0.0	0.0	8.4583	24.2
6	1.0	0.0	0.0	0.0	51.8625	54.0

Осталось взглянуть на получившееся распределение.

```
sns.histplot(knn['Age'], bins = 20)
plt.title('Распределение Age после заполнения с помощью KNNImputer');
```



Распределение близко к нормальному.

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 векторы (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).

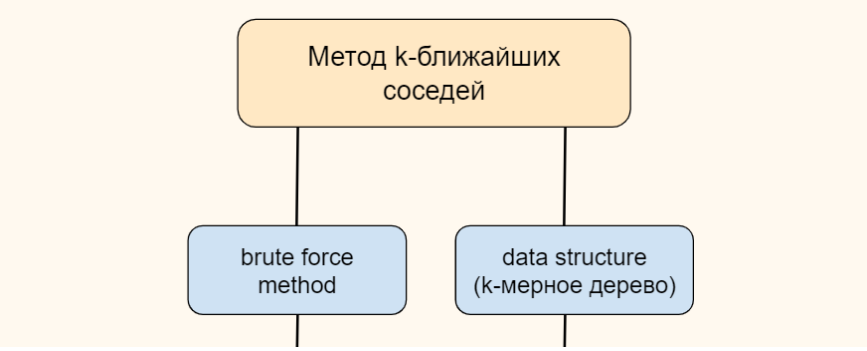
Алгоритм k-ближайших соседей потребует двух циклов:

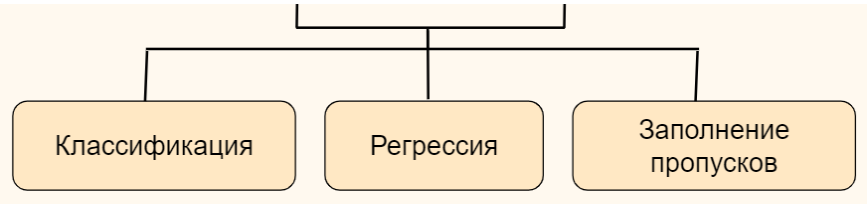
1. в *первом* цикле мы будем поочередно брать по одному вектору запроса;
2. во *втором* вложенном в него цикле мы будем для каждого вектора запроса находить расстояние до всех векторов сравнения;
3. наконец, найдя и отсортировав векторы сравнения по расстоянию, выберем для каждого вектора запроса k-ближайших соседей;
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) исполнение кода 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 векторы запроса).

Подобной 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.

Сравнение методов

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

1. возьмем получившиеся в результате применения каждого из методов *одинаковые* датасеты (различаются только значения, которыми были заполнены пропуски);
2. 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('/content/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)}')
```

```
Method: constant, accuracy: 0.79
Method: median, accuracy: 0.795
Method: binned median, accuracy: 0.808
Method: linear regression, accuracy: 0.808
Method: stochastic linear regression, accuracy: 0.796
Method: MICE, accuracy: 0.808
Method: KNNImputer, accuracy: 0.802
```

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 такое hot-deck imputation?

Ответ. *Hot-deck imputation* или **метод горячей колоды** предполагает, что мы используем данные того же датасета для заполнения пропусков. Внутри hot-deck imputation есть несколько различных способов заполнения пропусков:

- в простейшем варианте мы случайным образом берем непропущенное наблюдение и используем его для заполнения пропуска;
- кроме этого, мы можем отсортировать данные по имеющимся признакам и заполнить пропуски предыдущим значением (last observation carried forward, LOCF); читается, что таким образом мы заполняем пропуски наиболее близкими значениями;
- подход LOCF можно модифицировать и предварительно разбить данные на подгруппы.

Помимо этого существует и *cold-deck imputation* или **метод холодной колоды**, когда данные для заполнения пропусков берутся из другого датасета.

Терминологически «колодой» (deck of cards) называли стопку *перфокарт* (punched cards), на которые записывались данные или программа. При этом горячей называлась используемая сейчас «колода», холодной — та, которая была отложена.

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.

```
df = pd.DataFrame([[1, 2, 3],
                   ['?', 5, 6],
                   [7, '?', 9]])

df
```

	0	1	2
0	1	2	3
1	?	5	6
2	7	?	9

```
1  df[df == '?'] = np.nan
2  df
```

	0	1	2
0	1	2	3
1	NaN	5	6
2	7	NaN	9

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,

P (П | Н, О) = f(О)

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.

Вопрос. Какие библиотеки для заполнения пропусков существуют на R?

Ответ. Посмотрите imputeTS⧉.
