

Преобразование данных. Часть 1

[Материалы](#) > [Анализ и обработка данных](#)


Для многих моделей машинного обучения важно, чтобы количественные данные имели **одинаковый масштаб** (same scale). Это справедливо для:

- алгоритмов, рассчитывающих *расстояние* (например, алгоритма k-ближайших соседей или метода k-средних); а также
- моделей, оптимизирующих веса методом градиентного спуска и использующих *регуляризацию* (в частности, это линейная или логистическая регрессия).

Кроме того, в некоторых случаях нам может понадобиться **преобразовать данные** так, чтобы привести их к распределению, более похожему на нормальное. Это может быть важно для:

- проведения статистических тестов;
- превращения нелинейной зависимости в линейную (что важно, в частности, для вычисления линейной корреляции или использования линейной модели);
- стабилизации дисперсии, например, при выявлении гетероскедастичности остатков модели линейной регрессии.

Сегодняшнее занятие в большей степени посвящено способам и инструментам преобразования данных. Практикой их применения мы займемся на последующих занятиях.

Откроем блокнот к этому занятию 

Подготовка данных

Скачаем и подгрузим данные о недвижимости в Бостоне.

boston.csv

[Скачать](#)

Подготовим данные.

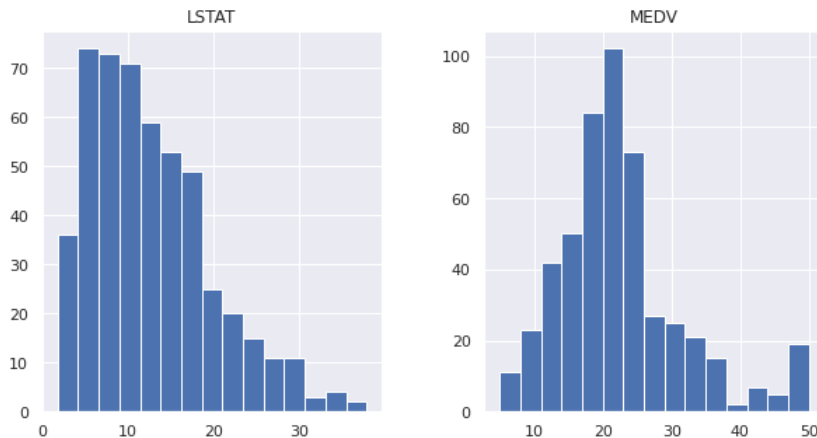
```
1 # возьмем признак LSTAT (процент населения с низким социальным статусом)
2 # и целевую переменную MEDV (медианная стоимость жилья)
```

```
3 | boston = pd.read_csv('/content/boston.csv')[['LSTAT', 'MEDV']]
4 | boston.shape
```

```
1 | (506, 2)
```

Визуализируем распределения с помощью гистограммы.

```
1 | boston.hist(bins = 15, figsize = (10, 5));
```



Посмотрим на основные статистические показатели.

```
1 | boston.describe()
```

	LSTAT	MEDV
count	506.000000	506.000000
mean	12.653063	22.532806
std	7.141062	9.197104
min	1.730000	5.000000
25%	6.950000	17.025000
50%	11.360000	21.200000
75%	16.955000	25.000000
max	37.970000	50.000000

Линейные и нелинейные преобразования

Ключевое отличие масштабирования от приближения одного распределения к другому, например, нормальному распределению (а именно эти два типа преобразований мы в основном будем рассматривать на сегодняшнем занятии), заключается в том, что первая трансформация линейна, вторая — нелинейна.

Линейные преобразования (linear transformation) не меняют структуру распределения, **нелинейные преобразования** (non-linear transformation) — меняют.

Возьмем скошенное вправо распределение LSTAT и применим к нему один из способов масштабирования, стандартизацию, и нелинейное преобразование Бокса-Кокса (смысл этих преобразований мы подробно рассмотрим ниже).

```

1 # создадим сетку подграфиков 1 x 3
2 fig, ax = plt.subplots(nrows = 1, ncols = 3, figsize = (12,4))
3
4 # на первом графике разместим изначальное распределение
5 sns.histplot(data = boston, x = 'LSTAT',
6               bins = 15,
7               ax = ax[0])
8 ax[0].set_title('Изначальное распределение')
9
10 # на втором - данные после стандартизации
11 sns.histplot(x = (boston.LSTAT - np.mean(boston.LSTAT)) / np.std(boston.LSTAT),
12              bins = 15, color = 'green',
13              ax = ax[1])
14 ax[1].set_title('Стандартизация')
15
16 # наконец скачаем функцию степенного преобразования power_transform()
17 from sklearn.preprocessing import power_transform
18
19 # и на третьем графике покажем преобразование Бокса-Кокса
20 sns.histplot(x = power_transform(boston[['LSTAT']]),
21              method = 'box-cox').flatten(),
22              bins = 12, color = 'orange',
23              ax = ax[2])
24 ax[2].set(title = 'Степенное преобразование', xlabel = 'LSTAT')
25
26 plt.tight_layout()
27 plt.show()

```



Как вы видите, в первом случае скошенность (skewness) и в целом форма распределения сохранилась, изменился только масштаб (среднее значение сместилось к нулю, изменился разброс). Во втором случае, изменилась сама структура распределения, то есть соотношение расстояний между точками.

Замечу, что в основном сегодня мы будем пользоваться модулем `preprocessing` библиотеки `sklearn` или трансформационными инструментами `Scipy`.

Добавление выбросов

Как уже было сказано выше, выбросы очень сильно влияют на качество данных. Для того чтобы посмотреть, как рассматриваемые нами инструменты справляются с выбросами, добавим несколько сильно отличающихся от общей массы наблюдений.

```

1 # создадим два отличающихся наблюдения
2 outliers = pd.DataFrame({
3     'LSTAT': [45, 50],

```

```

4     'MEDV': [70, 72]
5     })
6
7     # добавим их в исходный датафрейм
8     boston_outlier = pd.concat([boston, outliers], ignore_index = True)
9
10    # посмотрим на размерность нового датафрейма
11    boston_outlier.shape

```

```
1 (508, 2)
```

```

1 # убедимся, что наблюдения добавились
2 boston_outlier.tail()

```

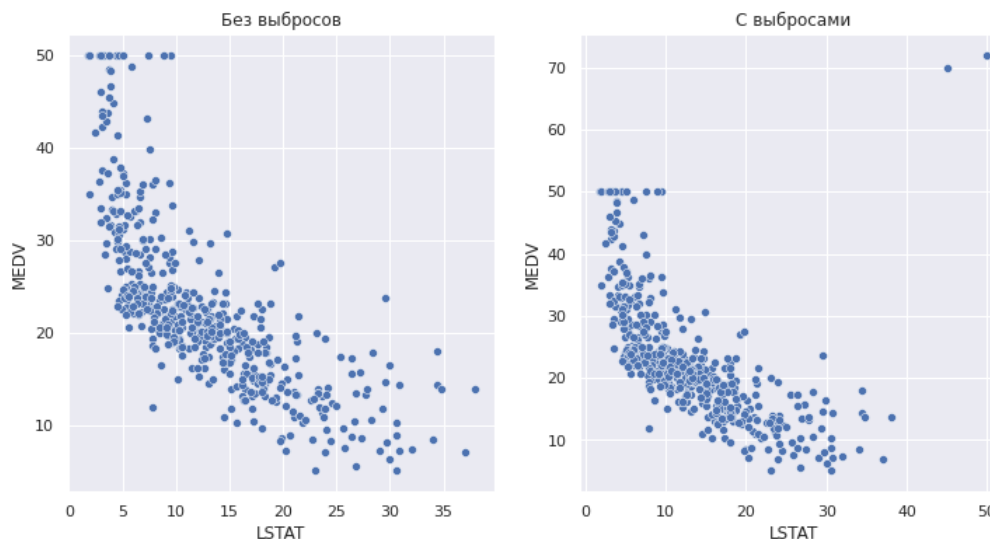
	LSTAT	MEDV
503	5.64	23.9
504	6.48	22.0
505	7.88	11.9
506	45.00	70.0
507	50.00	72.0

Посмотрим на данные с выбросами и без.

```

1 fig, ax = plt.subplots(1, 2, figsize = (12,6))
2
3 sns.scatterplot(data = boston, x = 'LSTAT', y = 'MEDV', ax = ax[0]).set(title = 'Без вы
4 sns.scatterplot(data = boston_outlier, x = 'LSTAT', y = 'MEDV', ax = ax[1]).set(title =

```



Линейные преобразования (масштабирование)

Рассмотрим несколько способов **масштабирования признаков** (feature scaling).

Стандартизация

Если данные следуют нормальному или близкому к нормальному распределению (что

желательно для многих моделей ML), имеет смысл прибегнуть к **стандартизации** (standartazation): то есть *приведению к нулевому среднему значению и единичному СКО* (так называемое стандартное нормальное распределение). Приведем формулу.

$$x' = \frac{x - \mu}{\sigma}$$

На практике стандартизация оказывается полезна и в тех случаях, когда данные не следуют нормальному распределению.

Стандартизация вручную

Замечу, что часто бывает удобно стандартизировать данные без использования класса sklearn.

```
1 | ((boston - boston.mean()) / boston.std()).head(3)
```

	LSTAT	MEDV
0	-1.074499	0.159528
1	-0.491953	-0.101424
2	-1.207532	1.322937

StandardScaler

Преобразование данных

Точно такой же результат можно получить через класс **StandardScaler** модуля preprocessing библиотеки sklearn. Создадим объект этого класса и применим **метод .fit()**.

```
1 | # из модуля preprocessing импортируем класс StandardScaler
2 | from sklearn.preprocessing import StandardScaler
3 |
4 | # создадим объект класса StandardScaler и применим метод .fit()
5 | st_scaler = StandardScaler().fit(boston)
6 | st_scaler
```

```
1 | StandardScaler()
```

При вызове метода **.fit()** алгоритм рассчитывает среднее арифметическое и СКО каждого из столбцов. Их можно посмотреть через соответствующие атрибуты.

```
1 | # выведем среднее арифметическое
2 | st_scaler.mean_
```

```
1 | array([12.65306324, 22.53280632])
```

```
1 | # и СКО каждого из столбцов
2 | array([7.13400164, 9.18801155])
```

Метод .transform() соответственно использует рассчитанные значения среднего и СКО для стандартизации данных.

```
1 | # метод .transform() возвращает массив Numpy с преобразованными значениями
2 | boston_scaled = st_scaler.transform(boston)
```

```
3
4 # превратим массив в датафрейм с помощью функции pd.DataFrame()
5 pd.DataFrame(boston_scaled, columns = boston.columns).head(3)
```

	LSTAT	MEDV
0	-1.075562	0.159686
1	-0.492439	-0.101524
2	-1.208727	1.324247

Метод `.fit_transform()` сразу вычисляет статистические показатели и применяет их для масштабирования данных.

```
1 boston_scaled = pd.DataFrame(StandardScaler().fit_transform(boston),
2                               columns = boston.columns)

1 # аналогичным образом стандартизируем данные с выбросами
2 boston_outlier_scaled = pd.DataFrame(StandardScaler().fit_transform(boston_outlier),
3                                   columns = boston_outlier.columns)
```

Визуализация преобразования

Объявим две вспомогательные функции, которые помогут нам визуализировать эту и ряд последующих трансформаций для данных с выбросами и без. Первая функция будет выводить точечные диаграммы.

```
1 # первая функция будет принимать на вход четыре датафрейма
2 # и визуализировать изменения с помощью точечной диаграммы
3 def scatter_plots(df, df_outlier, df_scaled, df_outlier_scaled, title):
4
5     fig, ax = plt.subplots(2, 2, figsize = (12,12))
6
7     sns.scatterplot(data = df, x = 'LSTAT', y = 'MEDV', ax = ax[0, 0])
8     ax[0, 0].set_title('Изначальный без выбросов')
9
10    sns.scatterplot(data = df_outlier, x = 'LSTAT', y = 'MEDV', color = 'green', ax = ax
11    ax[0, 1].set_title('Изначальный с выбросами')
12
13    sns.scatterplot(data = df_scaled, x = 'LSTAT', y = 'MEDV', ax = ax[1, 0])
14    ax[1, 0].set_title('Преобразование без выбросов')
15
16    sns.scatterplot(data = df_outlier_scaled, x = 'LSTAT', y = 'MEDV', color = 'green',
17    ax[1, 1].set_title('Преобразование с выбросами')
18
19    plt.suptitle(title)
20    plt.show()
```

```
1 # вторая функция будет визуализировать изменения с помощью гистограммы
2 def hist_plots(df, df_outlier, df_scaled, df_outlier_scaled, title):
3
4     fig, ax = plt.subplots(2, 2, figsize = (12,12))
5
6     sns.histplot(data = df, x = 'LSTAT', ax = ax[0, 0])
7     ax[0, 0].set_title('Изначальный без выбросов')
8
9     sns.histplot(data = df_outlier, x = 'LSTAT', color = 'green', ax = ax[0, 1])
10    ax[0, 1].set_title('Изначальный с выбросами')
11
12    sns.histplot(data = df_scaled, x = 'LSTAT', ax = ax[1, 0])
```

```

13 ax[1, 0].set_title('Преобразование без выбросов')
14
15 sns.histplot(data = df_outlier_scaled, x = 'LSTAT', color = 'green', ax = ax[1, 1])
16 ax[1, 1].set_title('Преобразование с выбросами')
17
18 plt.suptitle(title)
19 plt.show()

```

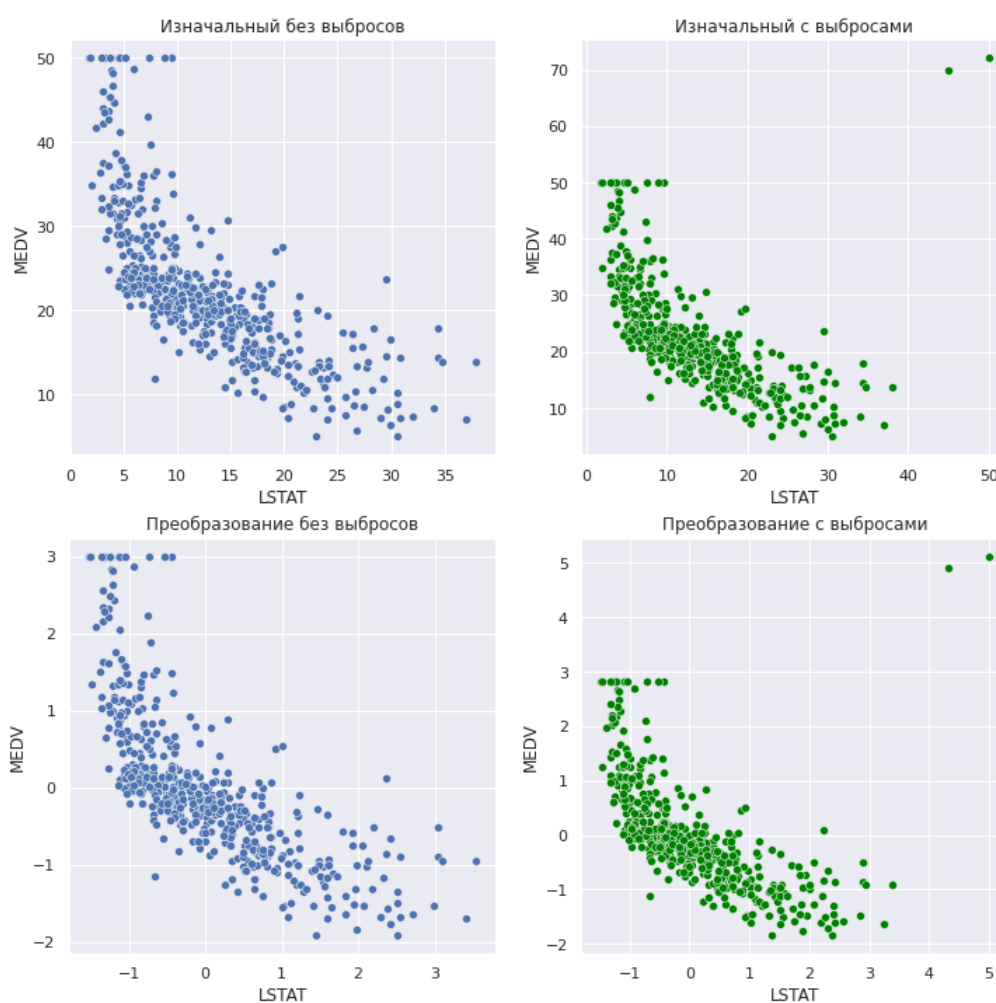
Применим эти функции к стандартизованным данным.

```

1 scatter_plots(boston,
2               boston_outlier,
3               boston_scaled,
4               boston_outlier_scaled,
5               title = 'Стандартизация данных')

```

Стандартизация данных



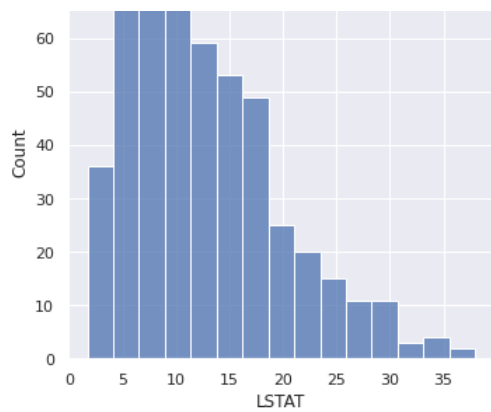
```

1 hist_plots(boston,
2            boston_outlier,
3            boston_scaled,
4            boston_outlier_scaled,
5            title = 'Стандартизация данных')

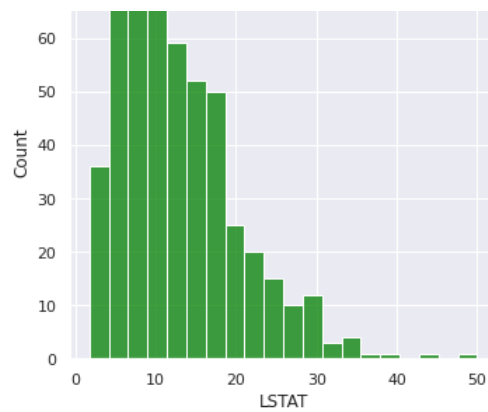
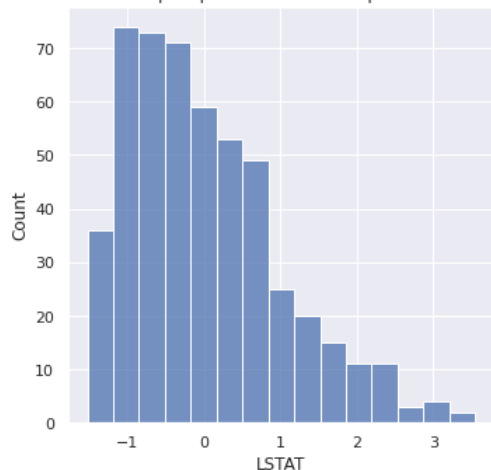
```

Стандартизация данных

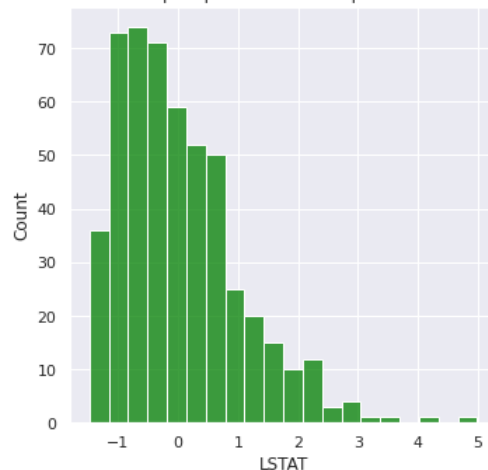




Преобразование без выбросов



Преобразование с выбросами



Обратите внимание, что стандартизация не ограничивает данные определенным диапазоном и допускает отрицательные значения. Кроме того, этот метод чувствителен к выбросам в том смысле, что влияет на расчет СКО, и диапазон двух признаков с выбросами после стандартизации все равно будет различаться.

Как следствие, при наличии выбросов стандартизация не гарантирует одинаковый масштаб признаков.

Хорошая иллюстрация этого факта есть на сайте [библиотеки sklearn](#).

Обратное преобразование

Вернуть исходный масштаб можно с помощью **метода .inverse_transform()**.

```
1 boston_inverse = pd.DataFrame(st_scaler.inverse_transform(boston_scaled),
2                               columns = boston.columns)
```

Иногда возникает вопрос, почему исходные и преобразованные к исходному виду данные не будут идентичными.

```
1 # используем метод .equals(), чтобы выяснить, одинаковы ли датафреймы
2 boston.equals(boston_inverse)
```

```
1 False
```

Это связано лишь с особенностями округления (как видно ниже различия минимальны).

```
1 # вычтем значения одного датафрейма из значений другого
```



```
2 | (boston - boston_inverse).head(3)
```

	LSTAT	MEDV
0	0.000000e+00	0.0
1	0.000000e+00	0.0
2	-8.881784e-16	0.0

Оценить приблизительное равенство можно так.

```
1 | np.all(np.isclose(boston.to_numpy(), boston_inverse.to_numpy()))
```

```
1 | True
```

Проблема утечки данных

В ответах на вопросы к занятию по классификации мы уже обсуждали применение стандартизации к обучающей и тестовой выборкам и упомянули проблему **утечки данных** (data leakage). Рассмотрим этот вопрос еще раз.

Если мы сразу отмасштабируем все данные (и обучающую, и тестовую выборки), то информация из тестовой части «утечет» в обучающую просто потому, что в случае стандартизации среднее и СКО будут рассчитываться на основе всех данных. Как следствие, модель на этапе обучения уже «увидит» тестовые данные, а значит качество модели «на тесте» может быть неоправданно завышено.

Для того чтобы тестовые данные никак не влияли на обучающую часть, нужно:

- рассчитать среднее и СКО обучающей выборки;
- отмасштабировать обучающие данные;
- обучить на них модель;
- использовать ранее рассчитанные среднее и СКО для масштабирования тестовых данных;
- сделать прогноз на отмасштабированных тестовых данных и оценить качество модели.

Именно такое разделение и обеспечивают методы `.fit()` и `.transform()`.

Перейдем к практике.

```
1 | # импортируем данные о недвижимости в Калифорнии
2 | from sklearn.datasets import fetch_california_housing
3 |
4 | # при return_X_y = True вместо объекта Bunch возвращаются признаки (X) и целевая переменная (y)
5 | # параметр as_frame = True возвращает датафрейм и Series вместо массивов Numpy
6 | X, y = fetch_california_housing(return_X_y = True, as_frame = True)
7 |
8 | # убедимся, что данные в нужном нам формате
9 | type(X), type(y)
```

```
1 | (pandas.core.frame.DataFrame, pandas.core.series.Series)
```

```
1 # посмотрим на признаки
2 X.head(3)
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24

```
1 # разделим данные на обучающую и тестовую выборки
2 from sklearn.model_selection import train_test_split
3
4 X_train, X_test, y_train, y_test = train_test_split(X, y,
5                                                    random_state = 42)
```

```
1 # импортируем класс для стандартизации данных
2 from sklearn.preprocessing import StandardScaler
3 # и создания модели линейной регрессии
4 from sklearn.linear_model import LinearRegression
5
6 # создадим объект класса StandardScaler
7 scaler = StandardScaler()
8 scaler
```

```
1 StandardScaler()
```

```
1 # масштабируем признаки обучающей выборки
2 X_train_scaled = scaler.fit_transform(X_train)
3
4 # убедимся, что объект scaler запомнил значения среднего и СКО
5 # для каждого признака
6 scaler.mean_, scaler.scale_
```

```
1 (array([ 3.87831412e+00,  2.85959948e+01,  5.43559839e+00,  1.09688116e+00,
2          1.42749729e+03,  3.10665968e+00,  3.56467196e+01, -1.19583736e+02]),
3  array([1.90372658e+00, 1.26109222e+01, 2.42157219e+00, 4.38789636e-01,
4          1.14289394e+03, 1.19554480e+01, 2.13388067e+00, 2.00237697e+00]))
```

```
1 # применим масштабированные данные для обучения модели линейной регрессии
2 model = LinearRegression().fit(X_train_scaled, y_train)
3 model
```

```
1 LinearRegression()
```

```
1 # преобразуем тестовые данные с использованием среднего и СКО, рассчитанных на обучающ
2 # так тестовые данные не повлияют на обучение модели, и мы избежим утечки данных
3 X_test_scaled = scaler.transform(X_test)
4
5 # сделаем прогноз на стандартизированных тестовых данных
6 y_pred = model.predict(X_test_scaled)
7 y_pred[:5]
```

```
1 array([0.72412832, 1.76677807, 2.71151581, 2.83601179, 2.603755  ])
```

```
1 # и оценим R-квадрат (метрика (score) по умолчанию для класса LinearRegression)
2 model.score(X_test_scaled, y_test)
```

```
1 0.5910509795491351
```

Применение пайплайна

По мере увеличения количества этапов работы с данными, количество кода также увеличилось. Сделать запись более экономной можно с помощью пайплайна.

Пайплайн (pipeline) последовательно применяет заданные преобразования данных (**transformer**) и выдает прогноз или метрику последнего по порядку инструмента, как правило, класса модели (**estimator**).

В нашем случае инструмента два: StandardScaler (transformer) и LinearRegression (estimator). Вначале рассмотрим более простой с точки зрения синтаксиса способ.

Класс make_pipeline

Создадим с помощью класса make_pipeline объект-контейнер, в который поместим необходимые нам инструменты.

```
1 # импортируем класс make_pipeline (упрощенный вариант класса Pipeline) из модуля pipeline
2 from sklearn.pipeline import make_pipeline
3
4 # создадим объект pipe, в который поместим объекты классов StandardScaler и LinearRegression
5 pipe = make_pipeline(StandardScaler(), LinearRegression())
6 pipe
```

```
1 Pipeline(steps=[('standardscaler', StandardScaler()),
2                  ('linearregression', LinearRegression())])
```

Применим **метод .fit()** к объекту pipe и используем обучающую выборку. Этот метод:

- вызывает соответствующие методы **.fit()** и **.transform()** класса StandardScaler, т.е. рассчитывает среднее и СКО и масштабирует данные;
- затем вызовет метод **.fit()** класса LinearRegression, обучит модель на преобразованных данных и «запомнит» коэффициенты.

```
1 pipe.fit(X_train, y_train)
```

```
1 Pipeline(steps=[('standardscaler', StandardScaler()),
2                  ('linearregression', LinearRegression())])
```

Теперь если мы применим к объекту pipe **метод .predict()** и передадим ему признаки тестовой части, то пайплайн вначале стандартизирует выборку с помощью рассчитанных ранее среднего и СКО обучающей выборки, а затем сделает прогноз.

```
1 pipe.predict(X_test)
```

```
1 array([0.72412832, 1.76677807, 2.71151581, ..., 1.72382152, 2.34689276,
2        3.52917352])
```

Обратите внимание, что пайплайн также позволил избежать утечки данных.

Аналогично, мы можем применить **метод .score()** и передать ему тестовую выборку. Этот метод выполнит масштабирование, обучит модель, сделает прогноз и посчитает метрику качества.

```
1 pipe.score(X_test, y_test)
```

```
1 0.5910509795491351
```

Замечу, что метод **.score()** применим только в том случае, если последний класс внутри

пайплайна располагает таким методом. В нашем случае, для класса `LinearRegression` метод `.score()` задан и выдает коэффициент детерминации R^2 .

Сделать масштабирование данных и прогноз или оценку качества модели можно в одну строчку.

```
1 | make_pipeline(StandardScaler(), LinearRegression()).fit(X_train, y_train).predict(X_test)
1 | array([0.72412832, 1.76677807, 2.71151581, ..., 1.72382152, 2.34689276,
2 |         3.52917352])
1 | make_pipeline(StandardScaler(), LinearRegression()).fit(X_train, y_train).score(X_test,
1 | 0.5910509795491351
```

Класс `make_pipeline` является упрощенной версией класса `Pipeline`.

```
1 | type(pipe)
1 | sklearn.pipeline.Pipeline
```

Класс Pipeline

Для того чтобы создать объект класса `Pipeline`, этому классу нужно передать кортежи из названия инструмента и соответствующего класса.

```
1 | # импортируем класс Pipeline
2 | from sklearn.pipeline import Pipeline
3 |
4 | # задаем названия и создаем объекты используемых классов
5 | pipe = Pipeline(steps = [('scaler', StandardScaler()),
6 |                          ('lr', LinearRegression())],
7 |                 verbose = True)
```

Обратите внимание на параметр **verbose**. Он используется во многих классах и функциях. Изначально, `verbose` по-английский означает «склонный к многословности, болтливый [человек]», применительно к программированию — это детальный вывод хода выполнения программы.

```
1 | # рассчитаем коэффициент детерминации
2 | pipe.fit(X_train, y_train).score(X_test, y_test)
1 | [Pipeline] ..... (step 1 of 2) Processing scaler, total= 0.0s
2 | [Pipeline] ..... (step 2 of 2) Processing lr, total= 0.0s
3 | 0.5910509795491351
```

Нормализация среднего

Нормализация среднего (mean normalization) предполагает деление разности между значением и средним признака не на СКО, а на диапазон от минимального до максимального значения.

$$x' = \frac{x - \mu}{x_{max} - x_{min}}$$

Перейдем к другим способам масштабирования.

Приведение к диапазону

Приведение признаков к заданному диапазону (scaling features to a range) является альтернативой стандартизации в тех случаях, когда нормальное распределение не является условием для обучения алгоритма. Рассмотрим два инструмента: `MinMaxScaler` и `MaxAbsScaler`.

MinMaxScaler

`MinMaxScaler` приводит данные к заданному диапазону (по умолчанию к промежутку от 0 до 1). Приведем формулу.

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Если мы хотим привести данные к произвольному диапазону $[a, b]$, то можем воспользоваться общей формулой.

$$x' = a + \frac{(x - x_{\min})(b - a)}{x_{\max} - x_{\min}}$$

Дополнительно замечу, что при работе с изображениями, если скажем на черно-белой фотографии каждый пиксель имеет диапазон от 0 до 255, то для приведения всех пикселей к диапазону от 0 до 1 достаточно разделить каждое значение на 255.

Применим класс `MinMaxScaler` к нашим данным.

```
1 # импортируем класс MinMaxScaler
2 from sklearn.preprocessing import MinMaxScaler
3
4 # создаем объект этого класса,
5 # в параметре feature_range оставим диапазон по умолчанию
6 minmax = MinMaxScaler(feature_range = (0, 1))
7 minmax

1 MinMaxScaler()

1 # применим метод .fit() и
2 minmax.fit(boston)
3
4 # найдем минимальные и максимальные значения
5 minmax.data_min_, minmax.data_max_

1 (array([1.73, 5.  ]), array([37.97, 50.  ]))

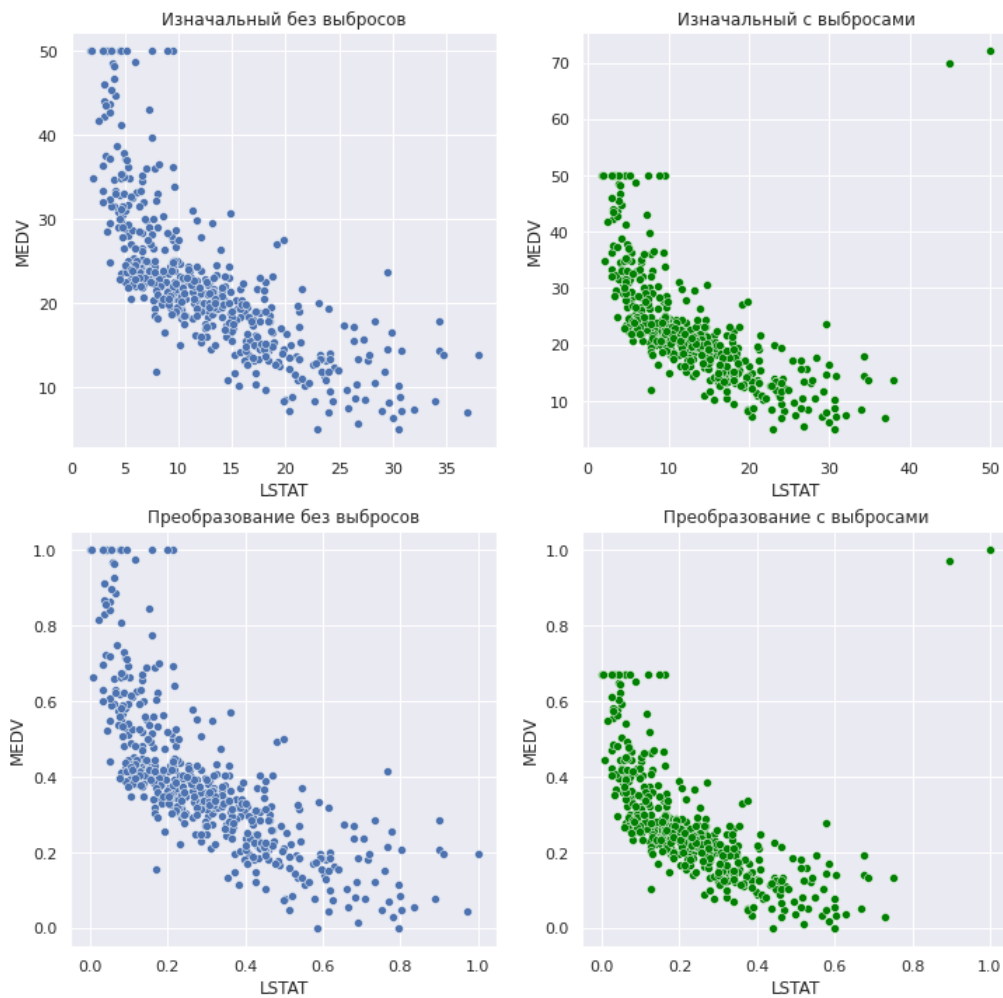
1 # приведем данные без выбросов (достаточно метода .transform())
2 boston_scaled = minmax.transform(boston)
3 # и с выбросами к заданному диапазону
4 boston_outlier_scaled = minmax.fit_transform(boston_outlier)
5
6 # преобразуем результаты в датафрейм
7 boston_scaled = pd.DataFrame(boston_scaled, columns = boston.columns)
8 boston_outlier_scaled = pd.DataFrame(boston_outlier_scaled, columns = boston.columns)
```

Визуально оценим результат.

```
1 # построим точечные диаграммы
2 scatter_plots(boston,
3               boston_outlier,
```

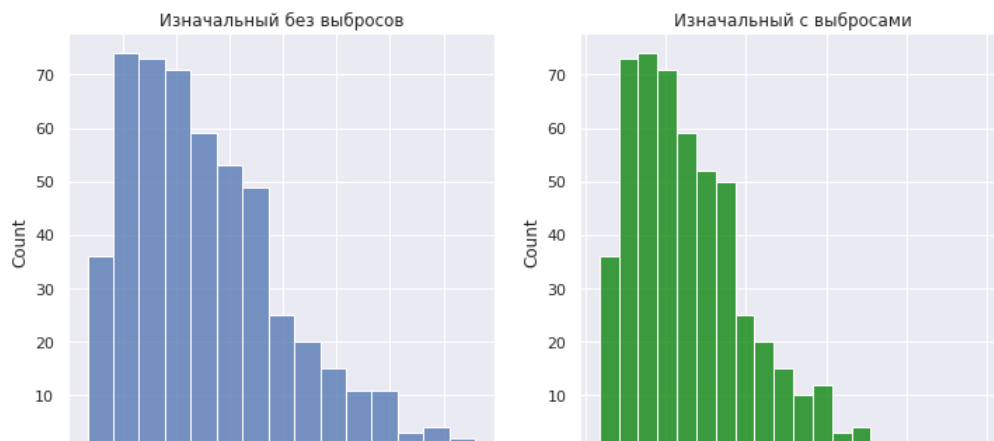
```
4 boston_scaled,  
5 boston_outlier_scaled,  
6 title = 'MinMaxScaler')
```

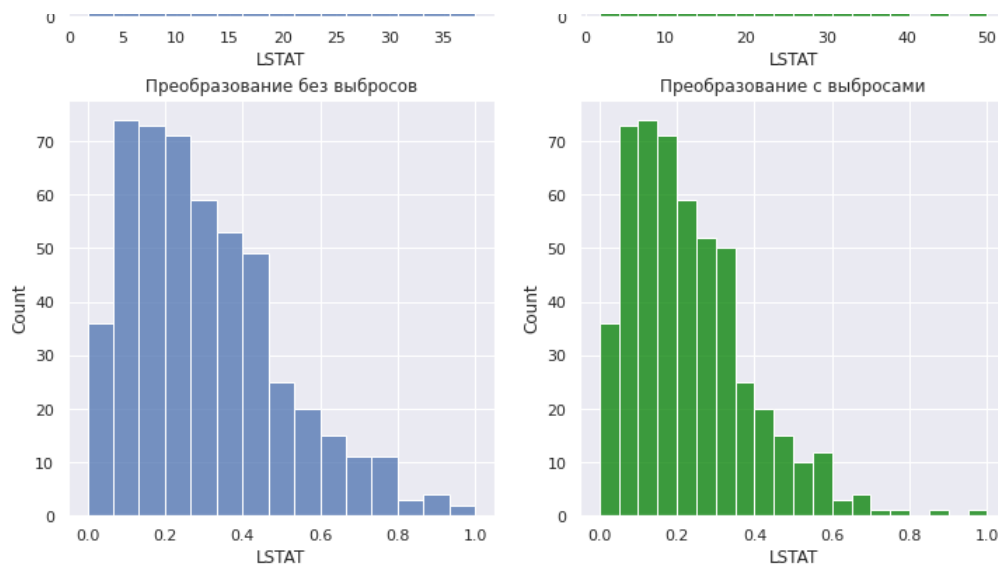
MinMaxScaler



```
1 # и гистограммы  
2 hist_plots(boston,  
3           boston_outlier,  
4           boston_scaled,  
5           boston_outlier_scaled,  
6           title = 'MinMaxScaler')
```

MinMaxScaler





Этот метод также чувствителен к выбросам и при их наличии не обеспечивает единого масштаба признаков.

MaxAbsScaler

Стандартизация разреженных данных

Работая с рекомендательными системами, мы увидели, что данные могут храниться в разреженных матрицах (sparse matrices). Приведем простой пример.

```
1 # создадим разреженную матрицу с пятью признаками
2 sparse_data = {}
3
4 sparse_data['F1'] = [0, 0, 1.25, 0, 2.15, 0, 0, 0, 0, 0, 0]
5 sparse_data['F2'] = [0, 0, 0, 0.45, 0, 1.20, 0, 0, 0, 1.28, 0]
6 sparse_data['F3'] = [0, 0, 0, 0, 2.15, 0, 0, 0, 0.33, 0, 0]
7 sparse_data['F4'] = [0, -6.5, 0, 0, 0, 0, 8.25, 0, 0, 0, 0]
8 sparse_data['F5'] = [0, 0, 0, 0, 0, 3.17, 0, 0, 0, 0, -1.85]
9
10 sparse_data = pd.DataFrame(sparse_data)
11 sparse_data
```

	F1	F2	F3	F4	F5
0	0.00	0.00	0.00	0.00	0.00
1	0.00	0.00	0.00	-6.50	0.00
2	1.25	0.00	0.00	0.00	0.00
3	0.00	0.45	0.00	0.00	0.00
4	2.15	0.00	2.15	0.00	0.00
5	0.00	1.20	0.00	0.00	3.17
6	0.00	0.00	0.00	8.25	0.00
7	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.33	0.00	0.00
9	0.00	1.28	0.00	0.00	0.00

10	0.00	0.00	0.00	0.00	0.00
11	0.00	0.00	0.00	0.00	-1.85

Если применить, например, стандартизацию, то в соответствии с формулой этого преобразования нули заполнятся другими отличными от нуля значениями.

```
1 pd.DataFrame(StandardScaler().fit_transform(sparse_data),
2               columns = sparse_data.columns).round(2)
```

	F1	F2	F3	F4	F5
0	-0.43	-0.53	-0.35	-0.05	-0.10
1	-0.43	-0.53	-0.35	-2.19	-0.10
2	1.47	-0.53	-0.35	-0.05	-0.10
3	-0.43	0.45	-0.35	-0.05	-0.10
4	2.83	-0.53	3.28	-0.05	-0.10
5	-0.43	2.07	-0.35	-0.05	2.90
6	-0.43	-0.53	-0.35	2.68	-0.10
7	-0.43	-0.53	-0.35	-0.05	-0.10
8	-0.43	-0.53	0.21	-0.05	-0.10
9	-0.43	2.24	-0.35	-0.05	-0.10
10	-0.43	-0.53	-0.35	-0.05	-0.10
11	-0.43	-0.53	-0.35	-0.05	-1.86

Таким образом мы испортим наши данные. Для того чтобы этого избежать можно использовать MaxAbsScaler.

Примечание. MinMaxScaler, в чем вы можете убедиться самостоятельно, справится с сохранением нулей в столбцах, где есть только положительные значения, и не справится со столбцами с отрицательными значениями.

Формула и простой пример

Приведем формулу.

$$x' = \frac{x}{|x_{max}|}$$

В данном случае мы делим каждое значение на модуль максимального значения признака. Посмотрим на простом примере, что в этом случае происходит с данными.

```
1 # создадим двумерный массив
2 arr = np.array([[ 1., -1., -2.],
3                 [ 2.,  0.,  0.],
4                 [ 0.,  1.,  1.]])
```

```
1 # применим MaxAbsScaler
2 from sklearn.preprocessing import MaxAbsScaler
3 maxabs = MaxAbsScaler()
```



```
4 |
5 | maxabs.fit_transform(arr)

1 | array([[ 0.5, -1. , -1. ],
2 |        [ 1. ,  0. ,  0. ],
3 |        [ 0. ,  1. ,  0.5]])

1 | # выведем модуль максимального значения каждого столбца
2 | maxabs.scale_

1 | array([2., 1., 2.])
```

В качестве примера разберем первый столбец. Максимальным значением по модулю будет «два» и именно на это число мы делим каждое значение признака.

Отметим некоторые особенности преобразования MaxAbsScaler:

- нулевые значения сохраняются;
- только положительные значения столбца приводятся к диапазону от 0 до 1 и здесь MaxAbsScaler работает так же, как и MinMaxScaler;
- только отрицательные значения приводятся к диапазону от -1 до 0;
- положительные и отрицательные значения — к диапазону от -1 до 1.

Разреженная матрица и MaxAbsScaler

Применим MaxAbsScaler к разреженной матрице.

```
1 | pd.DataFrame(MaxAbsScaler().fit_transform(sparse_data),
2 |              columns = sparse_data.columns).round(2)
```

	F1	F2	F3	F4	F5
0	0.00	0.00	0.00	0.00	0.00
1	0.00	0.00	0.00	-0.79	0.00
2	0.58	0.00	0.00	0.00	0.00
3	0.00	0.35	0.00	0.00	0.00
4	1.00	0.00	1.00	0.00	0.00
5	0.00	0.94	0.00	0.00	1.00
6	0.00	0.00	0.00	1.00	0.00
7	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.15	0.00	0.00
9	0.00	1.00	0.00	0.00	0.00
10	0.00	0.00	0.00	0.00	0.00
11	0.00	0.00	0.00	0.00	-0.58

Матрица csr и MaxAbsScaler

Как мы знаем, разреженные матрицы удобно хранить в формате сжатого хранения строкой

(compressed sparse row, csr), и мы можем применить `MaxAbsScaler` непосредственно к данным в этом формате.

```
1 # создадим матрицу в формате сжатого хранения строкой
2 from scipy.sparse import csr_matrix
3 csr_data = csr_matrix(sparse_data.values)
4 print(csr_data)
```

```
1 (1, 3)      -6.5
2 (2, 0)       1.25
3 (3, 1)       0.45
4 (4, 0)       2.15
5 (4, 2)       2.15
6 (5, 1)       1.2
7 (5, 4)       3.17
8 (6, 3)       8.25
9 (8, 2)       0.33
10 (9, 1)      1.28
11 (11, 4)     -1.85
```

```
1 # применим MaxAbsScaler
2 csr_data_scaled = MaxAbsScaler().fit_transform(csr_data)
3 print(csr_data_scaled)
```

```
1 (1, 3)      -0.7878787878787878
2 (2, 0)       0.5813953488372093
3 (3, 1)       0.3515625
4 (4, 0)       1.0
5 (4, 2)       1.0
6 (5, 1)       0.9375
7 (5, 4)       0.9999999999999999
8 (6, 3)       1.0
9 (8, 2)       0.15348837209302327
10 (9, 1)      1.0
11 (11, 4)     -0.583596214511041
```

```
1 # восстановим плотную матрицу
2 csr_data_scaled.todense().round(2)
```

```
1 array([[ 0. ,  0. ,  0. ,  0. ,  0. ],
2        [ 0. ,  0. ,  0. , -0.79,  0. ],
3        [ 0.58,  0. ,  0. ,  0. ,  0. ],
4        [ 0. ,  0.35,  0. ,  0. ,  0. ],
5        [ 1. ,  0. ,  1. ,  0. ,  0. ],
6        [ 0. ,  0.94,  0. ,  0. ,  1. ],
7        [ 0. ,  0. ,  0. ,  1. ,  0. ],
8        [ 0. ,  0. ,  0. ,  0. ,  0. ],
9        [ 0. ,  0. ,  0.15,  0. ,  0. ],
10       [ 0. ,  1. ,  0. ,  0. ,  0. ],
11       [ 0. ,  0. ,  0. ,  0. ,  0. ],
12       [ 0. ,  0. ,  0. ,  0. , -0.58]])
```

`MaxAbsScaler` также чувствителен к выбросам.

RobustScaler

В отличие от приведенных выше инструментов, `RobustScaler` более устойчив к выбросам в силу того, что усреднение происходит по разнице между третьим и первым квартилями, то есть робастными статистическими показателями.

$$x = \frac{x - Q_1(x)}{Q_3(x) - Q_1(x)}$$

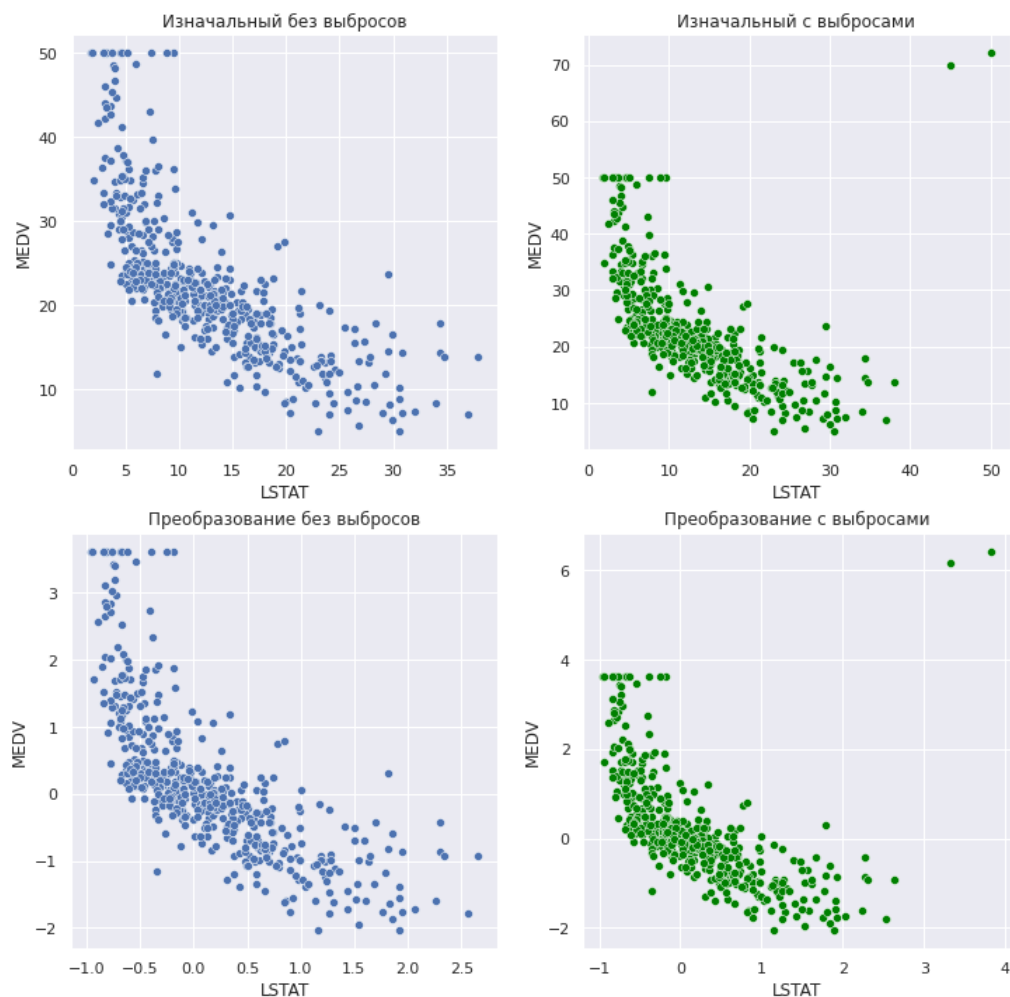
Применим класс RobustScaler.

```
1 from sklearn.preprocessing import RobustScaler
2
3 boston_scaled = RobustScaler().fit_transform(boston)
4 boston_outlier_scaled = RobustScaler().fit_transform(boston_outlier)
5
6 boston_scaled = pd.DataFrame(boston_scaled, columns = boston.columns)
7 boston_outlier_scaled = pd.DataFrame(boston_outlier_scaled, columns = boston.columns)
```

Посмотрим на преобразование на графике.

```
1 scatter_plots(boston,
2               boston_outlier,
3               boston_scaled,
4               boston_outlier_scaled,
5               title = 'RobustScaler')
```

RobustScaler

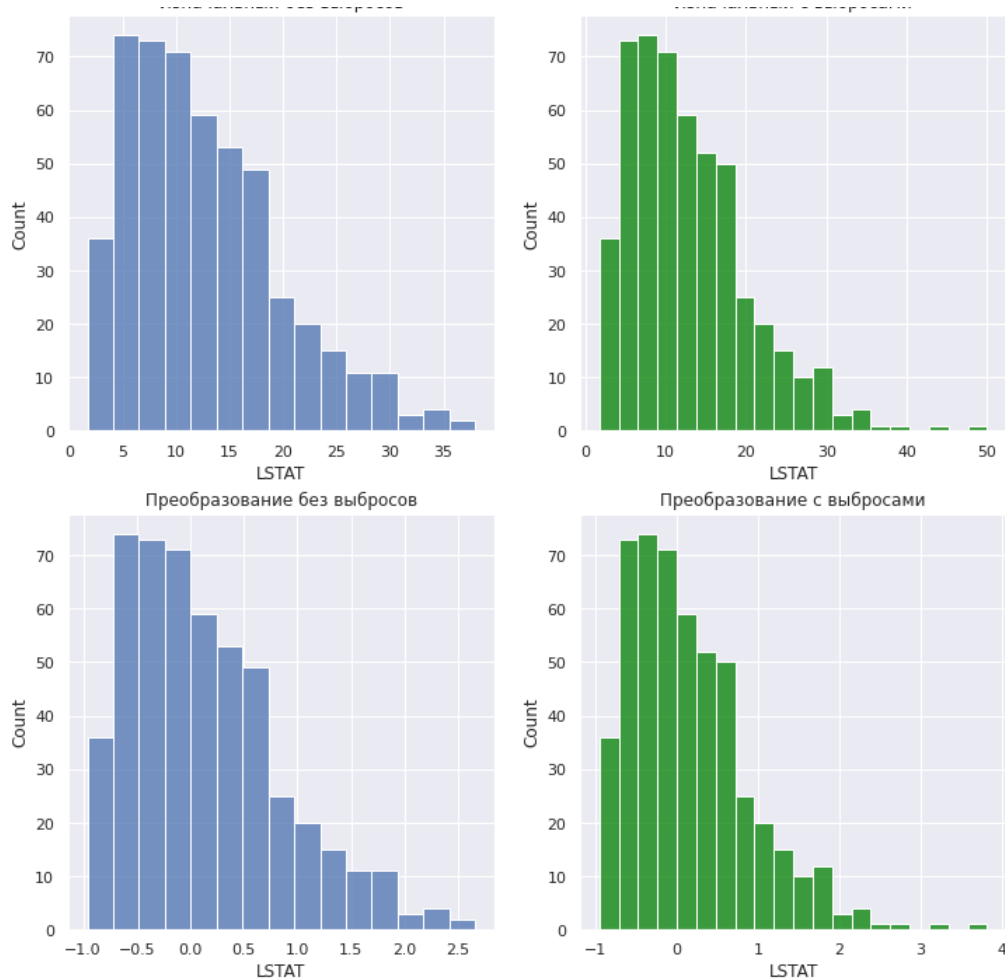


```
1 hist_plots(boston,
2             boston_outlier,
3             boston_scaled,
4             boston_outlier_scaled,
5             title = 'RobustScaler')
```

RobustScaler

Изначальный без выбросов

Изначальный с выбросами



RobustScaler не приводит данные строго к одному диапазону и не меняет структуру распределения (и в частности не изменяет расстояние между основной массой данных и выбросами).

Класс Normalizer

Класс Normalizer, в отличие от предыдущих инструментов, по умолчанию приводит наблюдения (то есть строки, а не столбцы датафрейма) к **единичной норме** (длине вектора, равной единице; unit norm, unit vector) или **нормализует** (normalizes) их.

В рамках вводного курса мы уже говорили, что каждое наблюдение можно представить в качестве вектора в n -мерном пространстве.

Понятие нормы вектора

Прежде чем говорить про нормализацию разберём понятие нормы вектора. Под нормой понимается такая функция, которая ставит в соответствие вектору в n -мерном пространстве некоторое число.

Это число часто рассматривают как длину вектора от начала координат до конца вектора. Причем эту длину или расстояние можно измерять по-разному.

Рассмотрим вероятно наиболее распространенное **Евклидово расстояние** (Euclidean distance) или как ещё говорят **L2 норму** (L2 norm) вектора \mathbf{x} с координатами x_1, x_2, \dots, x_n (термины длины, расстояния и нормы часто оказываются взаимозаменяемы).

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

Другими словами, мы смотрим, на какое расстояние нам необходимо сместиться в каждом из измерений, возводим это расстояние в квадрат, суммируем и извлекаем квадратный корень. Рассмотрим простой пример.

```
1 # возьмем вектор с координатами [4, 3]
2 v = np.array([4, 3])
3
4 # и найдем его длину или L2 норму
5 l2norm = np.sqrt(v[0]**2 + v[1]**2)
6 l2norm
```

```
1 5.0
```

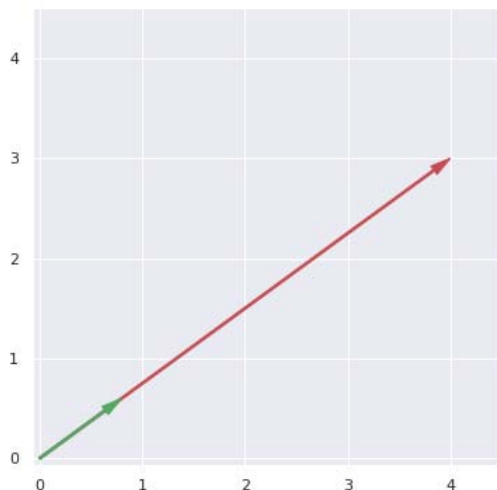
Если каждый компонент вектора разделить на L2 норму, то его длина или расстояние *по прямой* от начала координат до конца вектора было бы равно единице.

```
1 # разделим каждый компонент вектора на его норму
2 v_normalized = v/l2norm
3 v_normalized
```

```
1 array([0.8, 0.6])
```

Это и есть L2 нормализация.

```
1 # выведем оба вектора на графике
2 plt.figure(figsize = (6, 6))
3
4 ax = plt.axes()
5
6 plt.xlim([-0.07, 4.5])
7 plt.ylim([-0.07, 4.5])
8
9 ax.arrow(0, 0, v[0], v[1], width = 0.02, head_width = 0.1, head_length = 0.2, length_includes_head=True)
10 ax.arrow(0, 0, v_normalized[0], v_normalized[1], width = 0.02, head_width = 0.1, head_length = 0.2, length_includes_head=True)
11
12 plt.show()
```



L2 нормализация

Возьмём простой двумерный массив данных, вручную выполним построчную L2 нормализацию, а затем с помощью класса Normalizer проверим результат.

```
1 # каждая строка - это вектор
2 arr = np.array([[45, 30],
3                 [12, -340],
4                 [-125, 4]])
```

```
1 # найдем L2 норму первого вектора
2 np.sqrt(arr[0][0] ** 2 + arr[0][1] ** 2)
```

```
1 54.08326913195984
```

```
1 # в цикле пройдемся по строкам
2 for row in arr:
3     # найдем L2 норму каждого вектора-строки
4     l2norm = np.sqrt(row[0] ** 2 + row[1] ** 2)
5     # и разделим на нее каждый из компонентов вектора
6     print((row[0]/l2norm).round(8), (row[1]/l2norm).round(8))
```

```
1 0.83205029 0.5547002
2 0.03527216 -0.99937774
3 -0.99948839 0.03198363
```

```
1 # убедимся, что L2 нормализация выполнена верно,
2 # подставив в формулу Евклидова расстояния новые координаты
3 np.sqrt(0.83205029 ** 2 + 0.5547002 ** 2).round(3)
```

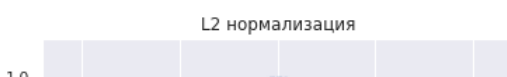
```
1 1.0
```

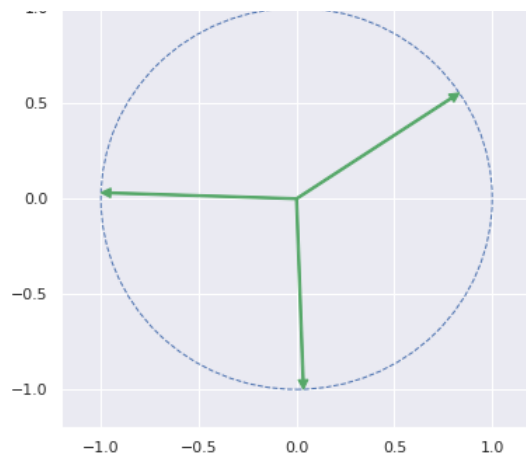
```
1 # выполним ту же операцию с помощью класса Normalizer
2 from sklearn.preprocessing import Normalizer
3
4 Normalizer().fit_transform(arr)
```

```
1 array([[ 0.83205029,  0.5547002 ],
2        [ 0.03527216, -0.99937774],
3        [-0.99948839,  0.03198363]])
```

Графически конец каждого L2 нормализованного вектора оказывается на единичной окружности (то есть окружности с радиусом, равным единице).

```
1 plt.figure(figsize = (6, 6))
2
3 ax = plt.axes()
4
5 # в цикле нормализуем каждый из векторов
6 for v in Normalizer().fit_transform(arr):
7     # и выведем его на графике в виде стрелки
8     ax.arrow(0, 0, v[0], v[1], width = 0.01, head_width = 0.05, head_length = 0.05, length = 1)
9
10 # добавим единичную окружность
11 circ = plt.Circle((0, 0), radius = 1, edgecolor = 'b', facecolor = 'None', linestyle = None)
12 ax.add_patch(circ)
13
14 plt.xlim([-1.2, 1.2])
15 plt.ylim([-1.2, 1.2])
16
17 plt.title('L2 нормализация')
18
19 plt.show()
```





Этот метод подходит для алгоритмов, основанных на расстоянии между векторами. В частности, вспомним формулу косинусного сходства.

$$\cos \theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2}$$

В знаменателе уже заложена L2 нормализация. По сути, мы вначале приводим каждый вектор к L2 норме, равной одному, а затем с помощью скалярного произведения находим косинус угла между ними.

Опасность нормализации по строкам

С точки зрения масштабирования данных у такого подхода есть один недостаток. Нормализация по строкам разрушает связи внутри признаков. Рассмотрим массив, в котором строки это люди, а столбцы — данных о них (в частности, рост, вес и возраст).

```
1 people = np.array([[180, 80, 50],  
2                   [170, 73, 50]])
```

Как мы видим, обоим людям по 50 лет. Проведем L2 нормализацию по строкам.

```
1 Normalizer().fit_transform(people)
```

```
1 array([[0.8857221 , 0.39365427, 0.24603392],  
2        [0.88704238, 0.38090643, 0.26089482]])
```

Как мы видим, после нормализации получается, что возраст у них разный.

По этой причине проводить масштабирование по строкам можно только в том случае, если связи между наблюдениями внутри признаков не имеют значения (другими словами, вам не важно, что в новых данных люди с одинаковым возрастом получают разные значения).

L1 нормализация

Как уже было сказано, длину вектора не обязательно измерять по формуле Евклидова расстояния. Можно воспользоваться формулой **расстояния городских кварталов** (Manhattan distance, taxicab distance) или **L1 нормой** (L1 norm).

$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \dots + |x_n|$$

По большому счету, вместо возведения в квадрат мы находим модуль каждой координаты вектора. При этом извлекать квадратный корень для возвращения к исходным единицам измерения уже нет необходимости.

```

1 # возьмем тот же массив
2 arr

1 array([[ 45,   30],
2        [ 12, -340],
3        [-125,   4]])

1 # рассчитаем L1 норму для первой строки
2 np.abs(arr[0][0]) + np.abs(arr[0][1])

1 75

1 # вновь пройдемся по каждому вектору
2 for row in arr:
3     # найдем соответствующую L1 норму
4     l1norm = np.abs(row[0]) + np.abs(row[1])
5     # и нормализуем векторы
6     print((row[0]/l1norm).round(8), (row[1]/l1norm).round(8))

1 0.6 0.4
2 0.03409091 -0.96590909
3 -0.96899225 0.03100775

```

Теперь к единичной норме приведена сумма модулей координат вектора.

```

1 # убедимся в том, что вторая вектор-строка имеет единичную L1 норму
2 np.abs(0.03409091) + np.abs(-0.96590909)

1 1.0

```

Сравним результат с объектом класса Normalizer.

```

1 # через параметр norm = 'l1' укажем,
2 # что хотим провести L1 нормализацию
3 Normalizer(norm = 'l1').fit_transform(arr)

1 array([[ 0.6,   0.4],
2        [ 0.03409091, -0.96590909],
3        [-0.96899225, 0.03100775]])

```

Теперь выведем L1 нормализованные векторы на графике и посмотрим, как рассчитывалось расстояние до первого вектора.

```

1 plt.figure(figsize = (6, 6))
2 ax = plt.axes()
3
4 # выведем L1 нормализованные векторы
5 for v in Normalizer(norm = 'l1').fit_transform(arr):
6     ax.arrow(0, 0, v[0], v[1], width = 0.01, head_width = 0.05, head_length = 0.05, length_includes_head = True)
7
8 # то, как рассчитывалось расстояние до первого вектора
9 ax.arrow(0, 0, 0.6, 0, width = 0.005, head_width = 0.03, head_length = 0.05, length_includes_head = True)
10 ax.arrow(0.6, 0, 0, 0.4, width = 0.005, head_width = 0.03, head_length = 0.05, length_includes_head = True)
11
12 # а также границы единичных векторов при L1 нормализации
13 points = [[1, 0], [0, 1], [-1, 0], [0, -1]]
14 polygon= plt.Polygon(points, fill = None, edgecolor = 'b', linestyle = '--')
15 ax.add_patch(polygon)
16
17 plt.xlim([-1.2, 1.2])
18 plt.ylim([-1.2, 1.2])
19

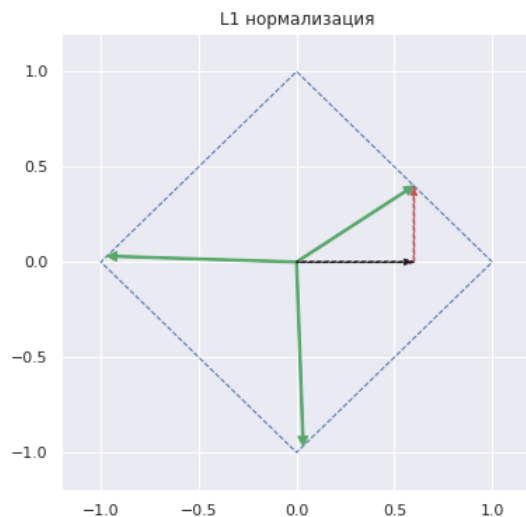
```



```

20 plt.title('L1 нормализация')
21
22 plt.show()

```



Из графика выше становится очевидно, почему это расстояние L1 (то есть сумма черного и красного векторов, равная единице) называется Manhattan distance или taxicab distance. Водителю такси на Манхэттене, основанном на гипподамовой системе с прямоугольными кварталами, чтобы попасть из точки А в точку Б пришлось бы двигаться строго перпендикулярными отрезками.

Расстояние Минковского

Обобщением Евклидова расстояния и расстояния городских кварталов будет **расстояние Минковского** (Minkowski distance).

$$||\mathbf{x}||_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

где $p = 1$ и $p = 2$ — это соответственно метрика городских кварталов и Евклидово расстояние.

Расстояние Чебышёва

Что интересно, если p стремится к бесконечности, то формула расстояния принимает вид

$$\lim_{p \rightarrow \infty} \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}} = \max_{i=1}^n |x_i|$$

Такое расстояние называется **расстоянием Чебышёва** (Chebyshev distance). По сути для расчета этого расстояния мы берем наибольшую по модулю координату вектора.

```

1 # найдем расстояние Чебышёва для первого вектора
2 max(np.abs(arr[0][0]), np.abs(arr[0][1]))

```

```

1 45

```

```
1 # теперь для всего массива
2 for row in arr:
3     # найдем соответствующую норму Чебышёва
4     l_inf = max(np.abs(row[0]), np.abs(row[1]))
5     # и нормализуем векторы
6     print((row[0]/l_inf).round(8), (row[1]/l_inf).round(8))
```

```
1 1.0 0.66666667
2 0.03529412 -1.0
3 -1.0 0.032
```

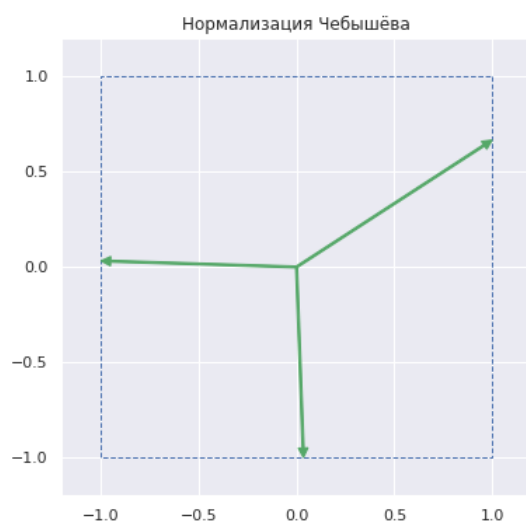
Для нормализации векторов по расстоянию Чебышёва классу Normalizer нужно передать параметр `norm = 'max'`.

```
1 Normalizer(norm = 'max').fit_transform(arr)
```

```
1 array([[ 1.          ,  0.66666667],
2        [ 0.03529412, -1.          ],
3        [-1.          ,  0.032       ]])
```

Графически в двумерном пространстве нормализованные таким образом векторы располагаются на квадрате, стороны которого имеют длину, равную двум, и параллельны осям координат.

```
1 plt.figure(figsize = (6, 6))
2 ax = plt.axes()
3
4 # выведем нормализованные по расстоянию Чебышёва векторы,
5 for v in Normalizer(norm = 'max').fit_transform(arr):
6     ax.arrow(0, 0, v[0], v[1], width = 0.01, head_width = 0.05, head_length = 0.05, length_includes_head=True)
7
8 # а также границы единичных векторов при такой нормализации
9 points = [[1, 1], [1, -1], [-1, -1], [-1, 1]]
10 polygon= plt.Polygon(points, fill = None, edgecolor = 'b', linestyle = '--')
11 ax.add_patch(polygon)
12
13 plt.xlim([-1.2, 1.2])
14 plt.ylim([-1.2, 1.2])
15
16 plt.title('Нормализация Чебышёва')
17
18 plt.show()
```



Про терминологию

Терминология способов преобразования количественных данных пока окончательно не устоялась. Например, часто под нормализацией понимают приведение данных к диапазону от 0 до 1 или в целом весь процесс масштабирования данных.

На практике бывает полезно сообщать, какую именно линейную трансформацию вы применяете к данным (а не только ее название), либо приводить название используемого инструмента.
