Преобразование данных. Часть 2

Материалы > Анализ и обработка данных

Во второй части рассмотрим нелинейные преобразования количественных данных.

Продолжим работу в том же блокноте

Нелинейные преобразования

Нелинейные преобразования, как уже было сказано, меняют структуру распределения.

```
# вновь подгрузим полный датасет boston
boston = pd.read_csv('/content/boston.csv')
```

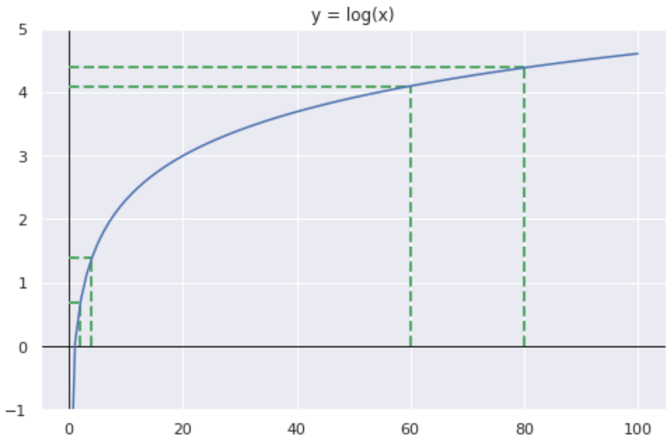
Логарифмическое преобразование

Смысл преобразования

Рассмотрим график логарифмической функции.

```
# построим график логарифмической функции
x = np.linspace(0.05, 100, 100)
y = np.log(x)

ax = plt.axes()

plt.xlim([-5, 105])
plt.ylim([-1, 5])

ax.hlines(y = 0, xmin = -5, xmax = 105, linewidth = 1, color = 'k')
ax.vlines(x = 0, ymin = -1, ymax = 5, linewidth = 1, color = 'k')

plt.plot(x, y)

# возьмем произвольные промежутки между малыми
ax.vlines(x = 2, ymin = 0, ymax = np.log(2), linewidth = 2, color = 'g', linestyles =
ax.vlines(x = 4, ymin = 0, ymax = np.log(4), linewidth = 2, color = 'g', linestyles =
ax.hlines(y = np.log(2), xmin = 0, xmax = 2, linewidth = 2, color = 'g', linestyles =
ax.hlines(y = np.log(4), xmin = 0, xmax = 4, linewidth = 2, color = 'g', linestyles =

# и большими значениями
ax.vlines(x = 60, ymin = 0, ymax = np.log(60), linewidth = 2, color = 'g', linestyles
ax.vlines(x = 80, ymin = 0, ymax = np.log(80), linewidth = 2, color = 'g', linestyles
ax.hlines(y = np.log(60), xmin = 0, xmax = 60, linewidth = 2, color = 'g', linestyles
ax.hlines(y = np.log(80), xmin = 0, xmax = 80, linewidth = 2, color = 'g', linestyles
```

```

plt.title('y = log(x)')

plt.show()
```



По оси x располагаются значения до трансформации, по оси y — после. Ценность логарифмического преобразования в том, что:

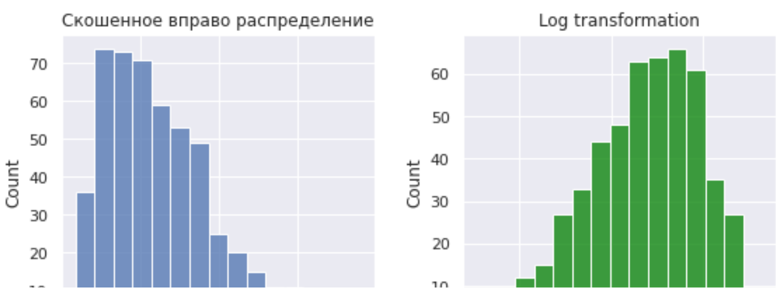
- расстояние между небольшими значениями увеличивается;
- расстояние между большими значениями наоборот уменьшается.

И, таким образом, это преобразование делает скошенное распределение более симметричным и приближенным к нормальному. Замечу, что, как видно из графика, в общем случае преобразование возможно только для положительных исходных значений.

Скошенное вправо распределение

В силу описанной выше особенности логарифмическое преобразование чаще применяют к **скошенным вправо** (right-skewed) распределениям. В этих распределениях бóльшая часть наблюдений находится как раз в диапазоне меньших значений.

```
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.histplot(x = boston.LSTAT, bins = 15, ax = ax[0])
ax[0].set_title('Скошенное вправо распределение')

sns.histplot(x = np.log(boston.LSTAT),
             bins = 15, color = 'green',
            ax = ax[1])
ax[1].set_title('Log transformation')

plt.tight_layout()
plt.show()
```



Количественно оценим изменения в скошенности и островершинности (то есть сосредоточенности значений вокруг среднего) распределения.

Коэффициент ассиметрии

Первое свойство измеряется **коэффициентом ассиметрии** (skewness), который в нормальном распределении должен быть равен нулю. При этом,

- положительные значения говорят о скошенности вправо (positively или right-skewed);
- отрицательные, о скошенности влево (negatively или left-skewed).

```
# импортируем необходимые функции
from scipy.stats import skew, kurtosis
```

```
# рассчитаем ассиметричность до и после преобразования
skew(boston.LSTAT), skew(np.log(boston.LSTAT))
```

```
(0.9037707431346133, -0.3192822699479382)
```

Выраженная скошенность вправо превратилась в меньшую скошенность влево.

Коэффициент эксцесса

Коэффициент эксцесса (kurtosis) измеряет островершинность распределения. Можно сказать, что эксцесс показывает сосредоточенность (плотность) значений вокруг среднего.

По *формуле Фишера* (Fisher's definition) для нормального распределения значение этого коэффициента также равно нулю. Одновременно,

- положительные значения говорят о большей сосредоточенности значений около среднего (острая вершина);
- отрицательные — о наличии более «тяжелых хвостов» (плоская вершина).

```
# рассчитаем коэффициент эксцесса до и после преобразования
kurtosis(boston.LSTAT), kurtosis(np.log(boston.LSTAT))
```

```
(0.476544755729746, -0.4390590293275558)
```

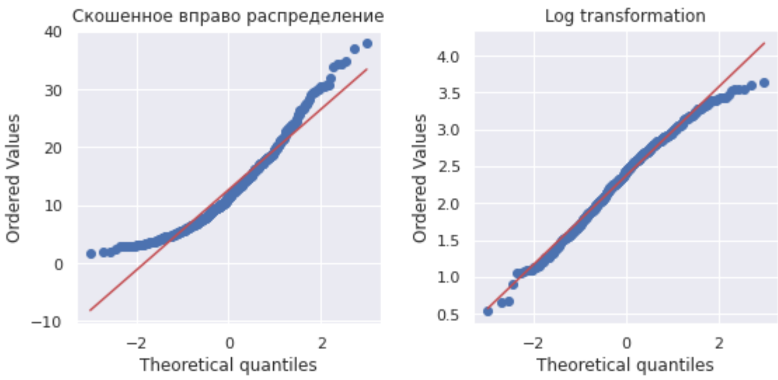
Вершина сменилась с более острой на более плоскую.

График нормальной вероятности

Наконец с помощью **графика нормальной вероятности** (normal probability plot) мы можем визуально оценить, приблизилось ли распредеделение к нормальному.

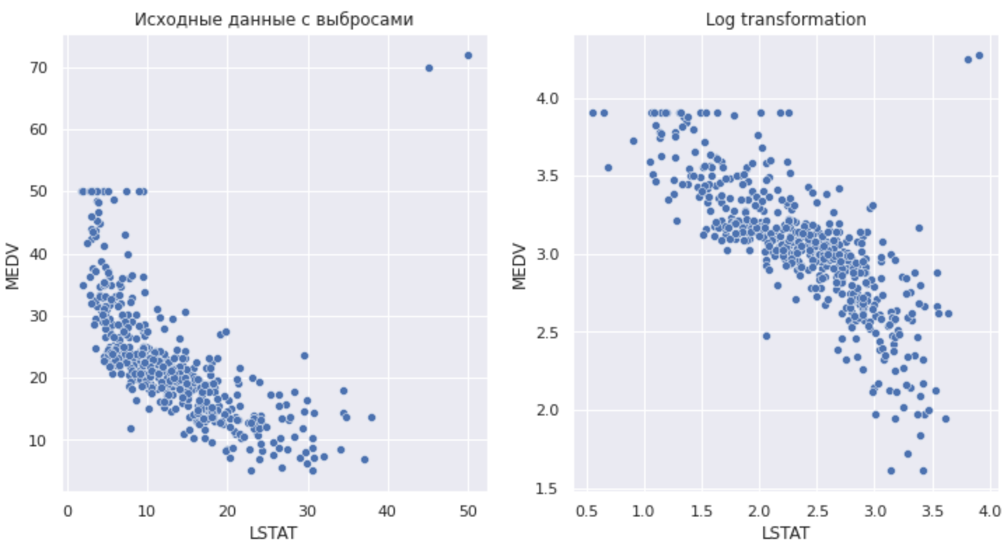
```
# построим графики нормальной вероятности
from scipy.stats import probplot

```

```
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

probplot(boston.LSTAT, dist = 'norm', plot = ax[0])
ax[0].set_title('Скошенное вправо распределение')

probplot(np.log(boston.LSTAT), dist = 'norm', plot = ax[1])
ax[1].set_title('Log transformation')

plt.tight_layout()
plt.show()
```



Разумеется, логарифмическое преобразование снижает эффект выбросов «справа».

```
fig, ax = plt.subplots(1, 2, figsize = (12,6))

sns.scatterplot(x = boston_outlier.LSTAT, y = boston_outlier.MEDV, ax = ax[0]).set(titl
sns.scatterplot(x = np.log(boston_outlier.LSTAT), y = np.log(boston_outlier.MEDV), ax =
```

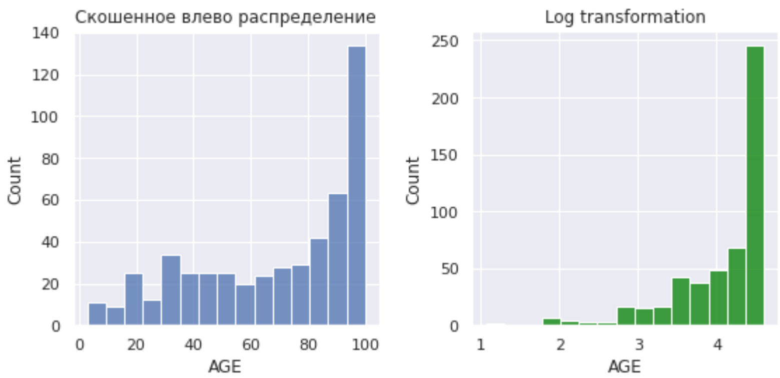


Скошенное влево распределение

Логарифмическое преобразование, скорее всего, не подойдет для скошенного влево распределения, потому что здесь наоборот бóльшая часть наблюдений находится в правой части диапазона. Применив лог-преобразование, мы только увеличим скошенность.

```
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.histplot(x = boston.AGE, bins = 15, ax = ax[0])
ax[0].set_title('Скошенное влево распределение')

```

```
sns.histplot(x = np.log(boston.AGE),
             bins = 15, color = 'green',
             ax = ax[1])
ax[1].set_title('Log transformation')

plt.tight_layout()
plt.show()
```



Посмотрим на коэффициент ассиметрии, коэффициент эксцесса и график нормальной вероятности.

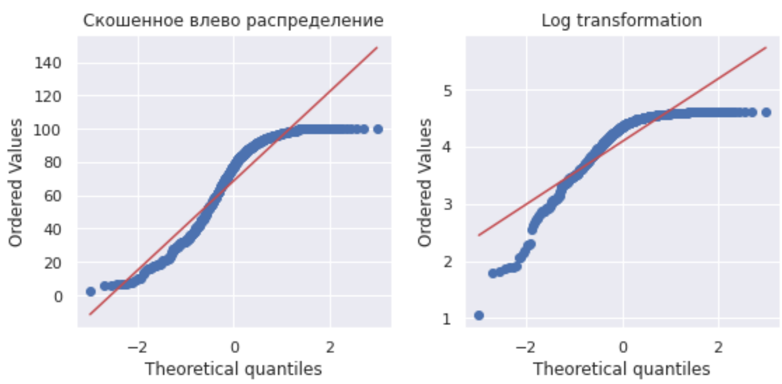
```
skew(boston.AGE), skew(np.log(boston.AGE))
```

```
(-0.5971855948016143, -1.6706835909283215)
```

```
kurtosis(boston.AGE), kurtosis(np.log(boston.AGE))
```

```
(-0.97001392664039, 2.907332087827127)
```

```
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

probplot(boston.AGE, dist = 'norm', plot = ax[0])
ax[0].set_title('Скошенное влево распределение')

probplot(np.log(boston.AGE), dist = 'norm', plot = ax[1])
ax[1].set_title('Log transformation')

plt.tight_layout()
plt.show()
```



Как мы видим, показатели только ухудшились.

Большие значения

Дополнительно замечу, что даже если распределение скошено вправо, но в нем присутствуют только большие (удаленные от нуля) значения, то логарифмическое преобразование может не приблизить распределение к нормальному, поскольку эффект расширения и сжатия на больших значениях менее заметен.

Логарифм нуля и отрицательных значений

Так как логарифм нуля и отрицательных значений неопределен, при наличии нулевых значений мы можем **добавить к значениям константу** (например, $\log(x + 0.001)$).

```
# в переменной ZN есть нулевые значения
# добавим небольшую константу
np.log(boston.ZN + 0.0001)[:5]
```

```
0    2.890377
1   -9.210340
2   -9.210340
3   -9.210340
4   -9.210340
Name: ZN, dtype: float64
```

Так как в данном случае мы все-таки произвольным образом меняем исходные данные, то можно использовать **преобразование обратного гиперболического синуса** (inverse hyperbolic sine (IHS) transformation).

$$IHS(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$

```
np.log(boston.ZN + np.sqrt(boston.ZN ** 2 + 1))[:5]
```

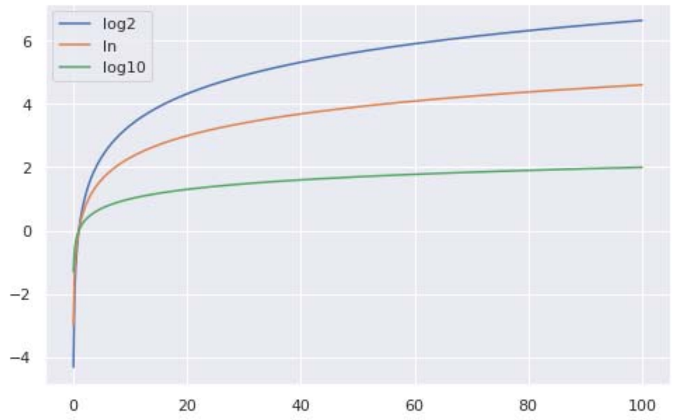
```
0    3.58429
1    0.00000
2    0.00000
3    0.00000
4    0.00000
Name: ZN, dtype: float64
```

Дополнительным преимуществом такого подхода является то, что мы можем преобразовывать любые действительные числа (а не только неотрицательные).

Основание логарифма

Чем меньше основание логарифма ($2 < e < 10$), тем больше диапазон преобразованных значений. Это можно наблюдать на графиках логарифмических функций, где функция с бо́льшим основанием становится «плоской» быстрее.

```
x = np.linspace(0.05, 100, 500)
y_2 = np.log2(x)
y_ln = np.log(x)
y_10 = np.log10(x)

plt.plot(x, y_2, label = 'log2')
plt.plot(x, y_ln, label = 'ln')
plt.plot(x, y_10, label = 'log10')

plt.legend()

plt.show()
```



При этом разумеется в целом логарифмическое преобразование с любым основанием действует примерно одинаково.

Линейная взаимосвязь

Многие процессы удобно моделировать с помощью линейных моделей. И хотя, как мы увидим в следующем разделе, можно использовать линейную модель с полиномиальными коэффициентам, зачастую проще наоборот «выправить» (straighten) сами данные.

```
# визуально оценим "выпрямление" данных
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.scatterplot(x = boston.LSTAT, y = boston.MEDV, ax = ax[0])
ax[0].set_title('Изначальное распределение')

sns.scatterplot(x = np.log(boston.LSTAT), y = boston.MEDV, ax = ax[1])
ax[1].set_title('Log transformation')

plt.tight_layout()

plt.show()
```

В качестве метрики линейности взаимосвязи можно использовать коэффициент *линейной* корреляции Пирсона (максимизация коэффициента корреляции означает максимизацию линейности).

```
# посмотрим, как изменится корреляция, если преобразовать
# одну, вторую или сразу обе переменные
boston['LSTAT_log'] = np.log(boston['LSTAT'])
boston['MEDV_log'] = np.log(boston['MEDV'])

boston[['LSTAT', 'LSTAT_log', 'MEDV', 'MEDV_log']].corr()
```

	LSTAT	LSTAT_log	MEDV	MEDV_log
LSTAT	1.000000	0.944031	-0.737663	-0.805034
LSTAT_log	0.944031	1.000000	-0.815442	-0.822960
MEDV	-0.737663	-0.815442	1.000000	0.953155
MEDV_log	-0.805034	-0.822960	0.953155	1.000000

Как мы видим, в данном случае коэффициент корреляции будет наибольшим в том случае, когда мы преобразовываем обе переменные ($-0{,}82$).

Если вы преобразовываете целевую переменную, важно выполнить обратное преобразования после формирования прогноза. Для операции взятия натурального логарифма обратным преобразованием будет возведение числа Эйлера в степень преобразованного числа.

$$y = ln(x) \to x = e^y$$

```
# сравним исходный датасет и лог-преобразование + обратную операцию
# (округлим значения, чтобы ошибка округления не мешала сравнению)
boston.MEDV.round(2).equals(np.exp(np.log(boston.MEDV)).round(2))
```

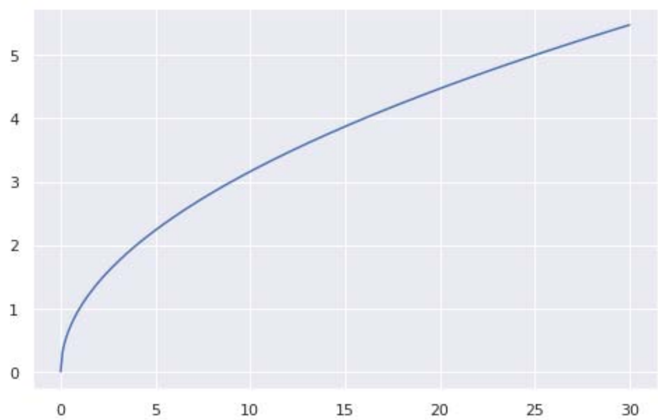
```
True
```

Примечание. Логарифмическое преобразование также легко интерпретировать. Например, снижение на $-0{,}162$ функции натурального логарифма свидетельствует о снижении на 15% в исходных данных вне зависимости от их масштаба.

Преобразование квадратного корня

Рассмотрим **преобразование с помощью квадратного корня** (square root transformation).

```
x = np.linspace(0, 30, 300)
y = np.sqrt(x)

plt.plot(x, y);
```



В целом, как видно из формы кривой, такое преобразование действует аналогично логарифмическому, однако менее агрессивно. С другой стороны, его без изменений можно применять к нулевым значениям.

Лестница степеней Тьюки

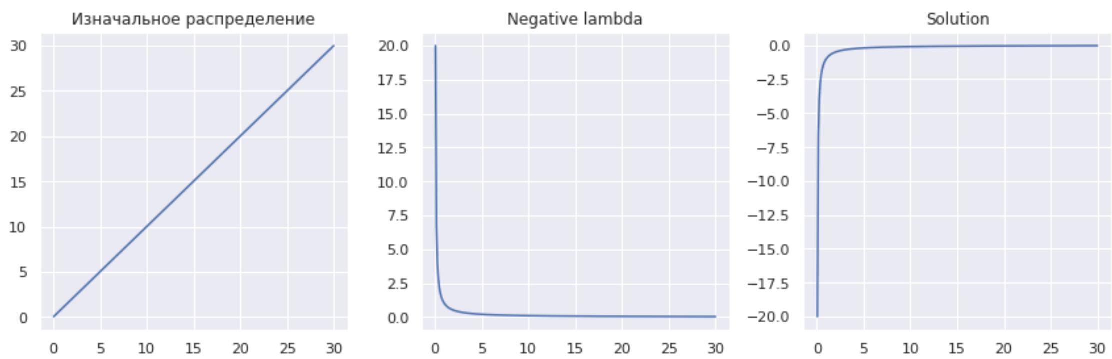
Помимо преобразования квадратного корня (то есть возведение в $\frac{1}{2}$ степени) можно пробовать и другие показатели. Для того чтобы понять, какое преобразование окажется наиболее удачным, используют **лестницу степеней Тьюки** (Tukey's Ladder of Powers).

λ	$\|\|$	-2	-1	$-\frac{1}{2}$	0	$\frac{1}{2}$	1	2
y	$\|\|$	$\frac{1}{x^2}$	$\frac{1}{x}$	$\frac{1}{\sqrt{x}}$	$\log x$	$\sqrt{x}$	x	x^2

По сути, мы по очереди применяем каждое из значений λ и смотрим, что получится. Здесь важно вставить несколько примечаний:

- вместо возведения в нулевую степень применяется логарифмическое преобразование;
- если переменная принимает отрицательные значения, то после, например, возведения в квадрат и обратного преобразования, такое значение потеряет смысл;
- если параметр λ принимает отрицательные значения, то зависимость переворачивается; например, если $y = x$ — это возрастающая зависимость, то $\frac{1}{x}$ — убывающая; преодолеть это можно, прописав что $-(x^{\lambda})$, если $\lambda < 0$.

```
x = np.linspace(0.05, 30, 300)

y0 = x
y1 = x ** (-1)
y2 = -(x ** (-1))

fig, ax = plt.subplots(nrows = 1, ncols = 3, figsize = (12,4))

ax[0].plot(x, y0);
ax[0].set_title('Изначальное распределение')

ax[1].plot(x, y1);
ax[1].set_title('Negative lambda')

ax[2].plot(x, y2);
ax[2].set_title('Solution')

plt.tight_layout()

plt.show()
```



Тогда,

$$y = \begin{cases} x^\lambda & \text{if } \lambda > 0 \\ \log x & \text{if } \lambda = 0 \\ -(x^\lambda) & \text{if } \lambda < 0 \end{cases}$$

λ	$\|\|$	-2	-1	$-\frac{1}{2}$	0	$\frac{1}{2}$	1	2
y	$\|\|$	$-\frac{1}{x^2}$	$-\frac{1}{x}$	$-\frac{1}{\sqrt{x}}$	$\log x$	$\sqrt{x}$	x	x^2

Этот инструмент удобно использовать, когда нужно «выправить» нелинейную взаимосвязь между переменными, например, в задаче линейной регрессии. Напишем функцию, которая на вход будет принимать признак и целевую переменную и выдавать оптимальную λ и соответствующий ей коэффициент корреляции.

```
def tukey(x, y):

  x, y = x.to_numpy(), y.to_numpy()

  # в lambdas поместим возможные степени
  lambdas = [-2, -1, -1/2, 0, 1/2, 1, 2]
  # в corrs будем записывать получающиеся корреляции
  corrs = []

  # в цикле последовательно применим каждую lambda
  for l in lambdas:

    if l < 0:
      # рассчитаем коэффициент корреляции Пирсона и добавим результат в corrs
      corrs.append(np.corrcoef(x ** l, y ** l)[0][1])

    elif l == 0:
      corrs.append(np.corrcoef(np.log(x + np.sqrt(x**2 + 1)),
                               np.log(y + np.sqrt(y**2 + 1)))[0][1])

    else:
      corrs.append(np.corrcoef(-(x ** l), -(y ** l))[0][1])

  # теперь найдем индекс наибольшего значения корреляции
  idx = np.argmax(np.abs(corrs))

  # выведем оптимальную lambda и соответствующую корреляцию
  return lambdas[idx], np.round(corrs[idx], 3)
```

```
tukey(boston.LSTAT, boston.MEDV)
```

```
(0, -0.824)
```

Оптимальным будет логарифмическое преобразование. Применим функцию к нескольким признакам с положительными значениями.

```
for col in boston[['CRIM', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'LSTAT']
  print(str(col) + '\t' + str(tukey(boston[col], boston.MEDV)))
```

```
CRIM    (0, -0.593)
NOX    (-0.5, -0.526)
RM    (2, 0.724)
AGE    (0.5, -0.402)
DIS    (-1, 0.489)
RAD    (0, -0.44)
TAX    (-0.5, -0.558)
PTRATIO    (0.5, -0.509)
LSTAT    (0, -0.824)
```

Посмотрим на корреляцию изначальных переменных.

```
boston[['CRIM', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'LSTAT', 'MEDV']].c
```

```
CRIM      -0.39
NOX       -0.43
RM         0.70
AGE       -0.38
DIS        0.25
RAD       -0.38
TAX       -0.47
PTRATIO   -0.51
LSTAT     -0.74
MEDV       1.00
Name: MEDV, dtype: float64
```

Для дальнейшей работы отберем RM, PTRATIO и LSTAT. Выполним необходимые преобразования.

```
boston_transformed = {}

boston_transformed['RM'] = boston.RM ** 2
boston_transformed['PTRATIO'] = np.sqrt(boston.PTRATIO)
boston_transformed['LSTAT'] = np.log(boston.LSTAT)
boston_transformed['MEDV'] = boston.MEDV

boston_transformed = pd.DataFrame(boston_transformed,
                                  columns = ['RM', 'PTRATIO', 'LSTAT', 'MEDV'])

boston_transformed.head()
```

	RM	PTRATIO	LSTAT	MEDV
0	43.230625	3.911521	1.605430	24.0
1	41.229241	4.219005	2.212660	21.6
2	51.624225	4.219005	1.393766	34.7
3	48.972004	4.324350	1.078410	33.4
4	51.079609	4.324350	1.673351	36.2

Построим модель линейной регрессии на исходных и преобразованных данных и рассчитаем коэффициент детерминации.

```
from sklearn.linear_model import LinearRegression
```

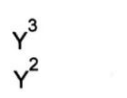
```
model = LinearRegression()
model.fit(boston[['RM', 'PTRATIO', 'LSTAT']], boston.MEDV)
model.score(boston[['RM', 'PTRATIO', 'LSTAT']], boston.MEDV)
```

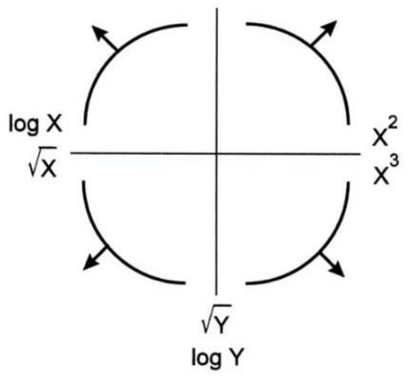
```
0.6786241601613112
```

```
model = LinearRegression()
model.fit(boston_transformed[['RM', 'PTRATIO', 'LSTAT']], boston_transformed.MEDV)
model.score(boston_transformed[['RM', 'PTRATIO', 'LSTAT']], boston_transformed.MEDV)
```

```
0.7446785206677597
```

Еще одним вариантом было бы применение различных преобразований к признаку и целевой переменной. В частности, Тьюки и Мостеллер предложили следующее правило выпуклости (**Tukey and Mosteller's Bulging Rule**).





В данном случае преобразования выбираются в зависимости от четырех приведнных на схеме форм зависимости данных.

Теперь рассмотрим более сложные, но близкие по смыслу к лестнице Тьюки степенные преобразования (power transformations) Бокса-Кокса и Йео-Джонсона.

Преобразование Бокса-Кокса

Приведем формулу **преобразования Бокса-Кокса** (Box-Cox transformation).

$$y_i^{(\lambda)} = \begin{cases} \frac{y_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \ln y_i & \text{if } \lambda = 0 \end{cases}$$

Очевидно, что преобразуемые значения y_i могут быть только положительными. Параметр λ выбирается методом наибольшего (максимального) правдоподобия (maximum likelihood method). Его рассмотрение выходит за рамки сегодняшнего занятия.

```
from sklearn.preprocessing import PowerTransformer

pt = PowerTransformer(method = 'box-cox')

# найдем оптимальный параметр лямбда
pt.fit(boston[['LSTAT']])

pt.lambdas_
```

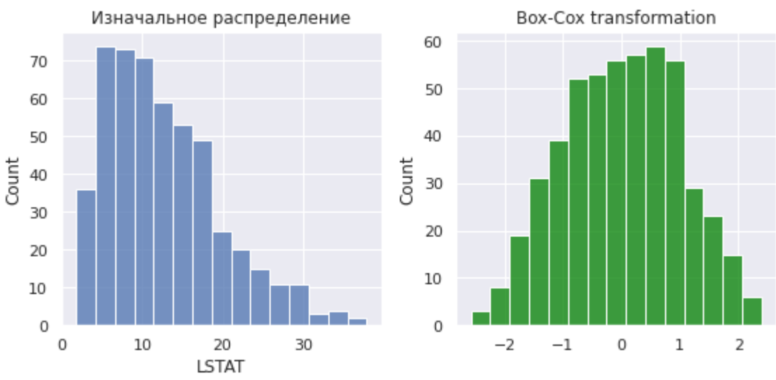
```
array([0.22776737])
```

```
# преобразуем данные
bc_pt = pt.transform(boston[['LSTAT']])

# метод .transform() возвращает двумерный массив
bc_pt.shape
```

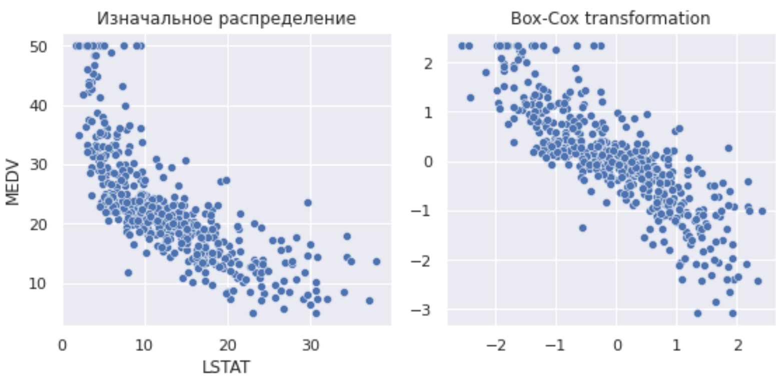
```
(506, 1)
```

```
# сравним изначальное распределение и распределение после преобразования Бокса-Кокса
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.histplot(x = boston.LSTAT, bins = 15, ax = ax[0])
ax[0].set_title('Изначальное распределение')

# так как на выходе метод .transform() выдает двумерный массив,
# его необходимо преобразовать в одномерный
sns.histplot(x = bc_pt.flatten(),
             bins = 15, color = 'green',
```

```
                 ax = ax[1])
ax[1].set_title('Box-Cox transformation')

plt.tight_layout()
plt.show()
```




```
# оценим изменение взаимосвязи после преобразования Бокса-Кокса
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.scatterplot(x = boston.LSTAT, y = boston.MEDV, ax = ax[0])
ax[0].set_title('Изначальное распределение')

# можно использовать функцию power_transform(),
# она действует аналогично классу, но без estimator
sns.scatterplot(x = power_transform(boston[['LSTAT']], method = 'box-cox').flatten(),
                y = power_transform(boston[['MEDV']], method = 'box-cox').flatten(),
                ax = ax[1])
ax[1].set_title('Box-Cox transformation')

plt.tight_layout()

plt.show()
```




```
# посмотрим на достигнутый коэффициент корреляции
pd.DataFrame(power_transform(boston[['LSTAT', 'MEDV']], method = 'box-cox'),
             columns = [['LSTAT', 'MEDV']]).corr()
```

	LSTAT	MEDV
LSTAT	1.000000	-0.830424
MEDV	-0.830424	1.000000

```
# сравним корреляцию признаков с целевой переменной
# после преобразования Бокса-Кокса
MEDV_bc = power_transform(boston[['MEDV']], method = 'box-cox').flatten()
```

```
for col in boston[['CRIM', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'LSTAT']
  col_bc = power_transform(boston[[col]], method = 'box-cox').flatten()
  print(col + '\t' + str(np.round(np.corrcoef(col_bc, MEDV_bc)[0][1], 3)))
```

```
CRIM	-0.528
NOX	-0.5
RM	0.64
AGE	-0.452
DIS	0.392
RAD	-0.403
TAX	-0.538
PTRATIO	-0.522
LSTAT	-0.83
```

```
# возьмем признаки RM, PTRATIO, LSTAT и целевую переменную MEDV
# и применим преобразование
pt = PowerTransformer(method = 'box-cox')
boston_bc = pt.fit_transform(boston[['RM', 'PTRATIO', 'LSTAT', 'MEDV']])
boston_bc = pd.DataFrame(boston_bc, columns = ['RM', 'PTRATIO', 'LSTAT', 'MEDV'])

# построим линейную регрессию
# в данном случае показатель чуть хуже, чем при лестнице Тьюки
model = LinearRegression()
model.fit(boston_bc[['RM', 'PTRATIO', 'LSTAT']], boston_bc.MEDV)
model.score(boston_bc[['RM', 'PTRATIO', 'LSTAT']], boston_bc.MEDV)
```

```
0.7331845210304999
```

```
# посмотрим на лямбды
pt.lambdas_
```

```
array([0.44895975, 4.35021552, 0.22776736, 0.2166209 ])
```

```
# выполним обратное преобразование
pd.DataFrame(pt.inverse_transform(boston_bc),
             columns = ['RM', 'PTRATIO', 'LSTAT', 'MEDV']).head()
```

	RM	PTRATIO	LSTAT	MEDV
0	6.575	15.3	4.98	24.0
1	6.421	17.8	9.14	21.6
2	7.185	17.8	4.03	34.7
3	6.998	18.7	2.94	33.4
4	7.147	18.7	5.33	36.2

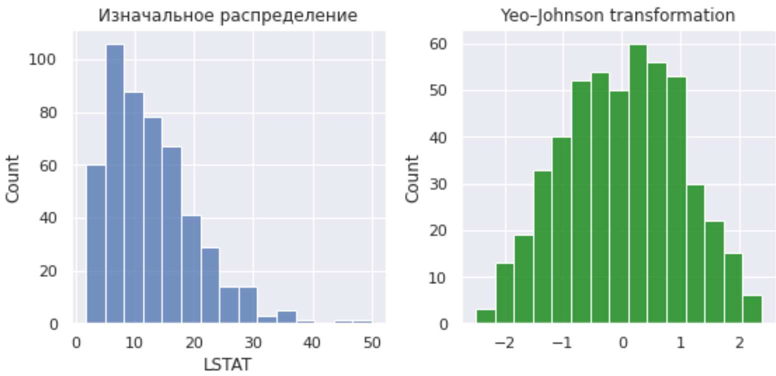
Преобразование Йео-Джонсона

Преобразование Йео-Джонсона (Yeo-Johnson transformation) позволяет работать с нулевыми и отрицательными значениями.

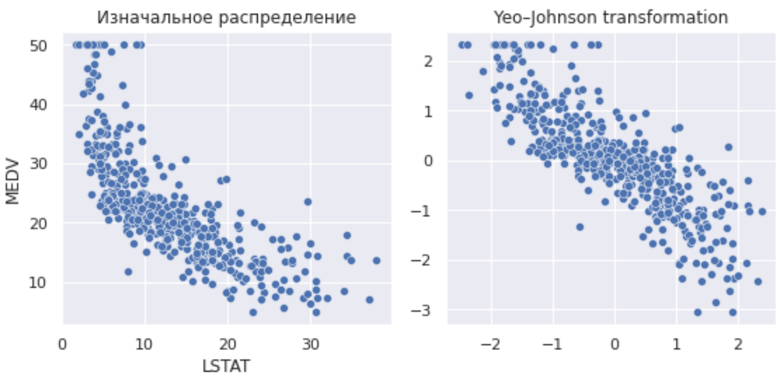
$$y = \begin{cases} ((y+1)^\lambda - 1)/\lambda & \text{if } \lambda \neq 0, y \geq 0 \\ \log(y+1) & \text{if } \lambda = 0, y \geq 0 \\ -[(-y+1)^{(2-\lambda)} - 1]/(2-\lambda) & \text{if } \lambda \neq 2, y < 0 \\ -\log(-y+1) & \text{if } \lambda = 2, y < 0 \end{cases}$$

Попробуем преобразование Йео-Джонсона.

```
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.histplot(x = boston_outlier.LSTAT, bins = 15, ax = ax[0])
ax[0].set_title('Изначальное распределение')

sns.histplot(x = power_transform(boston[['LSTAT']], method = 'yeo-johnson').flatten(),
             bins = 15, color = 'green',
            ax = ax[1])
ax[1].set_title('Yeo-Johnson transformation')

plt.tight_layout()
plt.show()
```




```
# посмотрим, как изменится линейность взаимосвязи
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.scatterplot(x = boston.LSTAT, y = boston.MEDV, ax = ax[0])
ax[0].set_title('Изначальное распределение')

sns.scatterplot(x = power_transform(boston[['LSTAT']], method = 'yeo-johnson').flatter
                y = power_transform(boston[['MEDV']], method = 'yeo-johnson').flatten(
                ax = ax[1])
ax[1].set_title('Yeo-Johnson transformation')

plt.tight_layout()

plt.show()
```




```
# возьмем те же признаки и целевую переменную, преобразуем их
# преобразование Йео-Джонсона является методом по умолчанию
pt = PowerTransformer()
boston_yj = pt.fit_transform(boston[['RM', 'PTRATIO', 'LSTAT', 'MEDV']])
boston_yj = pd.DataFrame(boston_yj, columns = ['RM', 'PTRATIO', 'LSTAT', 'MEDV'])

# построим модель
model = LinearRegression()
model.fit(boston_yj.iloc[:, :3], boston_yj.iloc[:, -1])
```

```
model.score(boston_yj.iloc[:, :3], boston_yj.iloc[:, -1])
```

```
0.7333775808517045
```

QuantileTransformer

При **квантильном преобразовании** (quantile transformation) исходным данным присваивается квантильный ранг в целевом (равномерном или нормальном) распределении. Этот ранг и есть новая преобразованная оценка.

Особенность такого преобразования заключается в том, что новое распределение никак не связано с исходным. Рассмотрим пример преобразования данных с выбросами.

```
from sklearn.preprocessing import QuantileTransformer

# приведем переменные с выбросами (!) к нормальному распределению
# с помощью квантиль-функции
qt = QuantileTransformer(n_quantiles = len(boston_outlier),
                         output_distribution = 'normal',
                         random_state = 42)

# для каждого из столбцов вычислим квантили нормального распределения,
# соответствующие заданному выше количеству квантилей (n_quantiles)
# и преобразуем (map) данные к нормальному распределению
boston_qt = pd.DataFrame(qt.fit_transform(boston_outlier),
                         columns = boston_outlier.columns)

# посмотрим на значения, на основе которых будут рассчитаны квантили
qt.quantiles_[-5:]
```

```
array([[34.77, 50.  ],
       [36.98, 50.  ],
       [37.97, 50.  ],
       [45.  , 70.  ],
       [50.  , 72.  ]])
```

```
# посмотрим на соответствующие им квантили нормального распределения
qt.references_[-5:]
```

```
array([0.99211045, 0.99408284, 0.99605523, 0.99802761, 1.        ])
```

```
# рассчитаем предпоследнее значение с помощью библиотеки scipy
from scipy.stats import norm

norm.ppf(0.99802761, loc = 0, scale = 1)
```

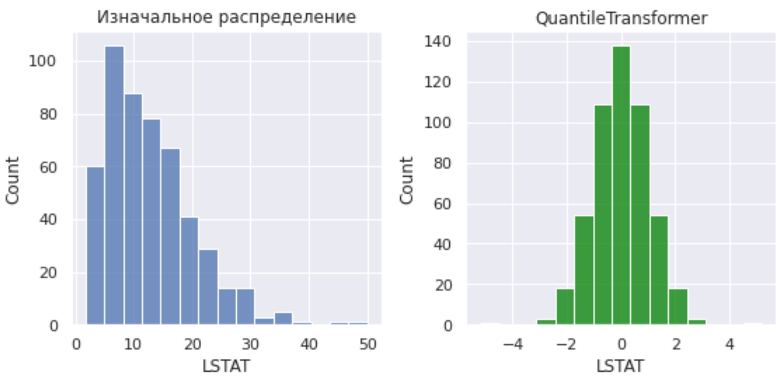
```
2.8825440308212347
```

```
# сравним с преобразованными значениями
boston_qt.LSTAT.sort_values()[-5:]
```

```
373    2.413985
414    2.517047
374    2.656761
506    2.882545
507    5.199338
Name: LSTAT, dtype: float64
```

```
# выведем результат
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (8,4))

sns.histplot(x = boston_outlier.LSTAT, bins = 15, ax = ax[0])
ax[0].set_title('Изначальное распределение')

```

```python
sns.histplot(x = boston_qt.LSTAT,
             bins = 15, color = 'green',
            ax = ax[1])
ax[1].set_title('QuantileTransformer')

plt.tight_layout()
plt.show()
```




```python
# посмотрим, выправилась ли взаимосвязь
plt.scatter(boston_qt.LSTAT, boston_qt.MEDV);
```

Исходя из преобразованных значений (посмотрите на значение с индексом 507) и точечной диаграммы мы видим, что эффект выбросов сохранился.

```python
max(boston_qt.LSTAT), max(boston_qt.MEDV)
```

```
(5.19933758270342, 5.19933758270342)
```

```python
# сравним исходную корреляцию
boston_outlier[['LSTAT', 'MEDV']].corr().iloc[0,1]
```

```
-0.5772033139947359
```

```python
# с корреляцией после преобразования
boston_qt.corr().iloc[0,1]
```

```
-0.7037287662365327
```

```python
# посмотрим на корреляцию после преобразования Йео-Джонсона
boston_yj = pd.DataFrame(power_transform(boston_outlier, method = 'yeo-johnson'),
                         columns = boston_outlier.columns)

boston_yj.corr().iloc[0,1]
```

```
-0.7661930913306837
```

Подведем итог

На сегодняшнем занятии мы рассмотрели линейные и нелинейные способы преобразования количественных данных.

Ответы на вопросы

Вопрос. Не очень понял применение формулы евклидова расстояния на занятии по кластерному анализу.

Ответ. На занятии по кластеризации мы измеряли с помощью евклидова расстояния дистанцию между двумя векторами, здесь же мы использовали формулу для расчета длины одного и того же вектора.