Машинное обучение

Кодирование категориальных переменных

Материалы > Анализ и обработка данных

Алгоритмы машинного обучения, как мы знаем, не умеют работать с категориальными данными, выраженными с помощью строковых значений. Для этого строки необходимо **закодировать** (encode) числами. Сегодня мы рассмотрим основные способы такой кодировки.

Откроем блокнот к этому занятию

Подготовим простые учебные данные кредитного скоринга.

```
scoring = {
    'Name' : ['Иван', 'Николай', 'Алексей', 'Александра', 'Евгений', 'Елена'],
    'Age' : [35, 43, 21, 34, 24, 27],
    'City' : ['Москва', 'Нижний Новгород', 'Санкт-Петербург', 'Владивосток', 'Москва',
    'Experience' : [7, 13, 2, 8, 4, 12],
    'Salary' : [95, 135, 73, 100, 78, 110],
    'Credit_score' : ['Good', 'Good', 'Bad', 'Medium', 'Medium', 'Good'],
    'Outcome' : [1, 1, 0, 1, 0, 1]
}

df = pd.DataFrame(scoring)
df
```

	Name	Age	City	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	7	95	Good	Вернул
1	Николай	43	Нижний Новгород	13	135	Good	Вернул
2	Алексей	21	Санкт-Петербург	2	73	Bad	Не вернул
3	Александра	34	Владивосток	8	100	Medium	Вернул
4	Евгений	24	Москва	4	78	Medium	Не вернул
5	Елена	27	Екатеринбург	12	110	Good	Вернул

Про категориальные переменные

Вначале в целом повторим, как выявлять и исследовать категориальные переменные.

Методы .info(), .unique(), value_counts()

Начать исследование категориальных переменных можно с изучения типа данных. Для этого

подойдут метод **.info()** или атрибут dtypes.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   Name          6 non-null      object
 1   Age           6 non-null      int64 
 2   City          6 non-null      object
 3   Experience    6 non-null      int64 
 4   Salary        6 non-null      int64 
 5   Credit_score  6 non-null      object
 6   Outcome       6 non-null      object
dtypes: int64(3), object(4)
memory usage: 464.0+ bytes
```

```
df.dtypes
```

```
Name            object
Age              int64
City            object
Experience       int64
Salary           int64
Credit_score    object
Outcome         object
dtype: object
```

При этом категориальные признаки часто могут «прятаться» в типах int и float. В этом случае для их выявления можно изучить распределение данных.

Отдельные категории можно посмотреть с помощью метода **.unique()**.

```
df.City.unique()
```

```
array(['Москва', 'Нижний Новгород', 'Санкт-Петербург', 'Владивосток',
       'Екатеринбург'], dtype=object)
```

С помощью методов **.values_counts()** библиотеки Pandas и **np.unique()** библиотеки Numpy можно посмотреть и количество объектов в каждой категории.

```
# метод .value_counts() сортирует категории по количеству объектов
# в убывающем порядке
df.City.value_counts()
```

```
Москва             2
Нижний Новгород    1
Санкт-Петербург    1
Владивосток        1
Екатеринбург       1
Name: City, dtype: int64
```

```
np.unique(df.City, return_counts = True)
```

```
(array(['Владивосток', 'Екатеринбург', 'Москва', 'Нижний Новгород',
        'Санкт-Петербург'], dtype=object), array([1, 1, 2, 1, 1]))
```

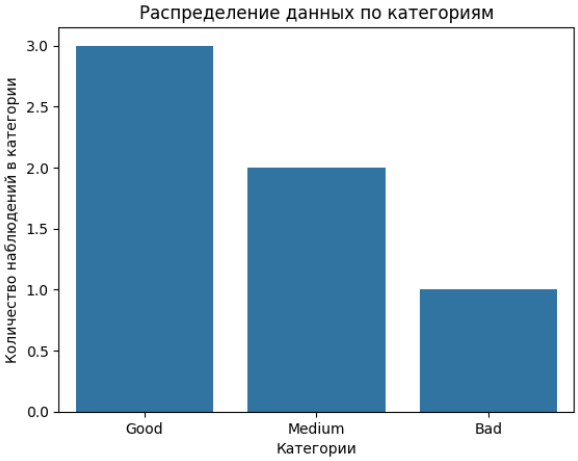
Последовательное применение методов **.value_counts()** и **.count()** выведет общее количество уникальных категорий.

```
# посмотрим на общее количество уникальных категорий
df.City.value_counts().count()
```

```
5
```

Выведем категории на графике.

```
score_counts = df.Credit_score.value_counts()
sns.barplot(x = score_counts.index, y = score_counts.values)
plt.title('Распределение данных по категориям')
plt.ylabel('Количество наблюдений в категории')
plt.xlabel('Категории');
```



Тип данных category

Хорошая практика — перевести категориальную переменную в тип данных category. Зачастую (например, если много категорий) это ускоряет работу с категориями и уменьшает использование памяти.

Можно воспользоваться уже знакомым нам **методом .astype()**.

```
df = df.astype({'City' : 'category', 'Outcome' : 'category'})
```

Функция pd.Categorical() позволяет прописать, в частности, сами категории, а также указать, есть ли в переданных категориях порядок или нет.

```
df.Credit_score = pd.Categorical(df.Credit_score,
                                 categories = ['Bad', 'Medium', 'Good'],
                                 ordered = True)
```

Воспользуемся атрибутами categories и dtype.

```
df.Credit_score.cat.categories
```

```
Index(['Bad', 'Medium', 'Good'], dtype='object')
```

```
df.Credit_score.dtype
```

```
CategoricalDtype(categories=['Bad', 'Medium', 'Good'], ordered=True, categories_dtype=o
```

Атрибут codes выводит коды каждой из категорий (мы воспользуемся этим в дальнейшем при кодировании).

```
df.Credit_score.cat.codes
```

```
0    2
1    2
2    0
3    1
4    1
```

```
6    5    2
7    dtype: int8
```

Категории можно переименовать.

```
df.Outcome = df.Outcome.cat.rename_categories(new_categories =
                                              {'Вернул': 'Yes', 'Не вернул': 'No'})

df
```

	Name	Age	City	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	7	95	Good	Yes
1	Николай	43	Нижний Новгород	13	135	Good	Yes
2	Алексей	21	Санкт-Петербург	2	73	Bad	No
3	Александра	34	Владивосток	8	100	Medium	Yes
4	Евгений	24	Москва	4	78	Medium	No
5	Елена	27	Екатеринбург	12	110	Good	Yes

Убедимся, что нужные нам признаки преобразованы в тип category.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Name          6 non-null      object
 1   Age           6 non-null      int64
 2   City          6 non-null      category
 3   Experience    6 non-null      int64
 4   Salary        6 non-null      int64
 5   Credit_score  6 non-null      category
 6   Outcome       6 non-null      category
dtypes: category(3), int64(3), object(1)
memory usage: 806.0+ bytes
```

Кардинальность данных

Большое количество уникальных категорий в столбце называется **высокой кардинальностью** (high cardinality) признака. В частности, потенциально (если бы у нас было больше данных) признак City мог бы обладать высокой кардинальностью.

Высокая кардинальность, среди прочего, может привести к разреженности данных. Одним из способов преодоления этой сложности могло бы стать создание нового признака, например, региона, группирующего несколько городов.

```
region = np.where(((df.City == 'Екатеринбург') | (df.City == 'Владивосток')), 0, 1)

df.insert(loc = 3, column= 'Region', value = region)

df
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	1	7	95	Good	Yes

1	Николай	43	Нижний Новгород	1	13	135	Good	Yes
2	Алексей	21	Санкт-Петербург	1	2	73	Bad	No
3	Александра	34	Владивосток	0	8	100	Medium	Yes
4	Евгений	24	Москва	1	4	78	Medium	No
5	Елена	27	Екатеринбург	0	12	110	Good	Yes

Дополнительным полезным свойством нового признака будет то, что на основе изначальных данных алгоритм бы не увидел разницы между Москвой и Владивостоком или Москвой и Екатеринбургом (а вполне вероятно, что в данных она есть). В новом же признаке, по сути делящем города по принадлежности к европейской и азиатской частям России, такую разницу выявить получится.

Базовые методы кодирования

Кодирование через cat.codes

Как уже было сказано выше, кодировать категориальную переменную можно через атрибут cat.codes.

```
df_cat = df.copy()
df_cat.Credit_score.cat.codes
```

```
0    2
1    2
2    0
3    1
4    1
5    2
dtype: int8
```

```
df_cat.Credit_score = df_cat.Credit_score.astype('category').cat.codes
df_cat
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	1	7	95	2	Yes
1	Николай	43	Нижний Новгород	1	13	135	2	Yes
2	Алексей	21	Санкт-Петербург	1	2	73	0	No
3	Александра	34	Владивосток	0	8	100	1	Yes
4	Евгений	24	Москва	1	4	78	1	No
5	Елена	27	Екатеринбург	0	12	110	2	Yes

Mapping

Этот способ мы уже применяли на прошлых занятиях. Суть его заключается в том, чтобы передать схему кодирования в виде словаря в **функцию map()** и применить к соответствующему столбцу.

```
df_map = df.copy()

# ключами будут старые значения признака
```

```
# значениями словаря - новые значения признака
map_dict = {'Bad' : 0,
            'Medium' : 1,
            'Good': 2}

df_map['Credit_score'] = df_map['Credit_score'].map(map_dict)
df_map
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	1	7	95	2	Yes
1	Николай	43	Нижний Новгород	1	13	135	2	Yes
2	Алексей	21	Санкт-Петербург	1	2	73	0	No
3	Александра	34	Владивосток	0	8	100	1	Yes
4	Евгений	24	Москва	1	4	78	1	No
5	Елена	27	Екатеринбург	0	12	110	2	Yes

Словарь в функцию **map()** можно передать и так.

```
# сделаем еще одну копию датафрейма
df_map = df.copy()

df_map.Credit_score = df_map.Credit_score.map(dict(Bad = 0, Medium = 1, Good = 2))
df_map
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	1	7	95	2	Yes
1	Николай	43	Нижний Новгород	1	13	135	2	Yes
2	Алексей	21	Санкт-Петербург	1	2	73	0	No
3	Александра	34	Владивосток	0	8	100	1	Yes
4	Евгений	24	Москва	1	4	78	1	No
5	Елена	27	Екатеринбург	0	12	110	2	Yes

Label Encoder

Рассмотрим класс LabelEncoder библиотеки sklearn. Этот класс преобразует n категорий в числа от 1 до n. Применим его к целевой переменной (бинарная категориальная переменная).

На вход LabelEncoder принимает только одномерные массивы (например, Series)

```
from sklearn.preprocessing import LabelEncoder

labelencoder = LabelEncoder()

df_le = df.copy()

df_le.loc[:, 'Outcome'] = labelencoder.fit_transform(df_le.loc[:, 'Outcome'])
df_le
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	1	7	95	Good	1

1	Николай	43	Нижний Новгород	1	13	135	Good	1
2	Алексей	21	Санкт-Петербург	1	2	73	Bad	0
3	Александра	34	Владивосток	0	8	100	Medium	1
4	Евгений	24	Москва	1	4	78	Medium	0
5	Елена	27	Екатеринбург	0	12	110	Good	1

Для категорий, в которых больше двух классов, но нет внутренней иерархии (номинальные данные), этот encoder подходит хуже, потому что построенная на основе преобразованных данных модель может подумать, что между категориями есть иерархия, когда в действительности ее нет.

```
# применим LabelEncoder к номинальной переменной City
df_le.loc[:, 'City'] = labelencoder.fit_transform(df_le.loc[:, 'City'])
df_le
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	2	1	7	95	Good	1
1	Николай	43	3	1	13	135	Good	1
2	Алексей	21	4	1	2	73	Bad	0
3	Александра	34	0	0	8	100	Medium	1
4	Евгений	24	2	1	4	78	Medium	0
5	Елена	27	1	0	12	110	Good	1

При этом даже для порядковых категориальных данных этот способ вряд ли подойдет, потому что LabelEncoder не видит порядка в данных.

```
# применим LabelEncoder к номинальной переменной Credit_score
df_le.loc[:, 'Credit_score'] = labelencoder.fit_transform(df_le.loc[:, 'Credit_score'])
df_le
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	2	1	7	95	1	1
1	Николай	43	3	1	13	135	1	1
2	Алексей	21	4	1	2	73	0	0
3	Александра	34	0	0	8	100	2	1
4	Евгений	24	2	1	4	78	2	0
5	Елена	27	1	0	12	110	1	1

```
labelencoder.classes_
```

```
array(['Bad', 'Good', 'Medium'], dtype=object)
```

Как мы видим, на второе место в иерархии категорий LabelEncoder поместил Good, что конечно является ошибкой. Таким образом, можно сказать, что LabelEncoder лучше всего справляется с бинарными категориальными данными.

Ordinal Encoder

С порядковыми категориальными данными справится OrdinalEncoder, которому при создании

объекта класса можно передать иерархию категорий.

На вход OrdinalEncoder принимает только двумерные массивы.

```
from sklearn.preprocessing import OrdinalEncoder

ordinalencoder = OrdinalEncoder(categories = [['Bad', 'Medium', 'Good']])

df_oe = df.copy()

# используем метод .to_frame() для преобразования Series в датафрейм
df_oe.loc[:, 'Credit_score'] = ordinalencoder.fit_transform(df_oe.loc[:, 'Credit_score'
df_oe
```

	Name	Age	City	Region	Experience	Salary	Credit_score	Outcome
0	Иван	35	Москва	1	7	95	2.0	Yes
1	Николай	43	Нижний Новгород	1	13	135	2.0	Yes
2	Алексей	21	Санкт-Петербург	1	2	73	0.0	No
3	Александра	34	Владивосток	0	8	100	1.0	Yes
4	Евгений	24	Москва	1	4	78	1.0	No
5	Елена	27	Екатеринбург	0	12	110	2.0	Yes

Убедимся, что иерархия категорий не нарушена.

```
ordinalencoder.categories_
```

```
[array(['Bad', 'Medium', 'Good'], dtype=object)]
```

OneHotEncoding

Как уже было сказано, номинальные данные нельзя заменять числами $1, 2, 3, \ldots$, так как алгоритм ML на этапе обучения подумает, что речь идет о порядковых данных. Нужно использовать one-hot encoder. С этим инструментом мы уже познакомились, когда рассматривали основы нейронных сетей.

Класс OneHotEncoder

Вначале применим класс OneHotEncoder библиотеки sklearn.

```
df_onehot = df.copy()

from sklearn.preprocessing import OneHotEncoder

# создадим объект класса OneHotEncoder
# параметр sparse = True выдал бы результат в сжатом формате
onehotencoder = OneHotEncoder(sparse_output = False)

encoded_df = pd.DataFrame(onehotencoder.fit_transform(df_onehot[['City']]))
encoded_df
```

	0	1	2	3	4

0	0.0	0.0	1.0	0.0	0.0
1	0.0	0.0	0.0	1.0	0.0
2	0.0	0.0	0.0	0.0	1.0
3	1.0	0.0	0.0	0.0	0.0
4	0.0	0.0	1.0	0.0	0.0
5	0.0	1.0	0.0	0.0	0.0

Выведем новые признаки с помощью **метода .get_feature_names_out()**.

```
onehotencoder.get_feature_names_out()
```

```
array(['City_Владивосток', 'City_Екатеринбург', 'City_Москва',
       'City_Нижний Новгород', 'City_Санкт-Петербург'], dtype=object)
```

Используем вывод этого метода, чтобы добавить названия столбцов.

```
encoded_df.columns = onehotencoder.get_feature_names_out()
encoded_df
```

	City_Владивосток	City_Екатеринбург	City_Москва	City_Нижний Новгород	City_Санкт-Петербург
0	0.0	0.0	1.0	0.0	0.0
1	0.0	0.0	0.0	1.0	0.0
2	0.0	0.0	0.0	0.0	1.0
3	1.0	0.0	0.0	0.0	0.0
4	0.0	0.0	1.0	0.0	0.0
5	0.0	1.0	0.0	0.0	0.0

Присоединим новые признаки к исходному датафрейму, удалив, разумеется, признак City.

```
df_onehot = df_onehot.join(encoded_df)
df_onehot.drop('City', axis = 1, inplace = True)
```

Обратите внимание, на самом деле нам не нужен первый признак (в данном случае, Владивосток). Если его убрать, при «срабатывании» этого признака (наблюдение с индексом три) все остальные признаки будут иметь нули (так мы поймем, что речь идет именно об этом отсутствующем признаке).

```
df_onehot = df.copy()

# чтобы удалить первый признак, используем параметр drop = 'first'
onehot_first = OneHotEncoder(drop = 'first', sparse = False)

encoded_df = pd.DataFrame(onehot_first.fit_transform(df_onehot[['City']]))
encoded_df.columns = onehot_first.get_feature_names_out()

df_onehot = df_onehot.join(encoded_df)
df_onehot.drop('Outcome', axis = 1, inplace = True)
df_onehot
```

	Name	Age	City	Region	Experience	Salary	Credit_score	City_Екатеринбург	City_Москва	City_Нижний Новгород	City_Санкт-Петербург
0	Иван	35	Москва	1	7	95	Good	0.0	1.0	0.0	0.0
1	Николай	43	Нижний Новгород	1	13	135	Good	0.0	0.0	1.0	0.0

2	Алексей	21	Санкт-Петербург	1	2	73	Bad	0.0	0.0	0.0	1.0
3	Александра	34	Владивосток	0	8	100	Medium	0.0	0.0	0.0	0.0
4	Евгений	24	Москва	1	4	78	Medium	0.0	1.0	0.0	0.0
5	Елена	27	Екатеринбург	0	12	110	Good	1.0	0.0	0.0	0.0

Функция pd.get_dummies()

Еще один способ — использовать **функцию pd.get_dummies()** библиотеки Pandas. Применим функцию к столбцу City.

```
df_dum = df.copy()
pd.get_dummies(df_dum, columns = ['City'])
```

City_Владивосток	City_Екатеринбург	City_Москва	City_Нижний Новгород	City_Санкт-Петербург
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
1	0	0	0	0
0	0	1	0	0
0	1	0	0	0

Уменьшить ширину новых столбцов можно через параметры prefix и prefix_sep.

```
pd.get_dummies(df_dum, columns = ['City'], prefix = '', prefix_sep = '')
```

Владивосток	Екатеринбург	Москва	Нижний Новгород	Санкт-Петербург
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
1	0	0	0	0
0	0	1	0	0
0	1	0	0	0

Опять же, можно не использовать первую dummy-переменную.

```
pd.get_dummies(df_dum, columns = ['City'],
               prefix = '',
               prefix_sep = '',
               drop_first = True)
```

Екатеринбург	Москва	Нижний Новгород	Санкт-Петербург
0	1	0	0
0	0	1	0
0	0	0	1

0	0	0	0
0	1	0	0
1	0	0	0

Библиотека category_encoders

Рассмотрим еще один способ выполнить one-hot encoding через соответствующий инструмент очень полезной библиотеки category_encoders.

```
# установим библиотеку
!pip install category_encoders
```

Импортируем библиотеку и применим класс OneHotEncoder.

```
df_catenc = df.copy()

import category_encoders as ce

# в параметр cols передадим столбцы, которые нужно преобразовать
ohe_encoder = ce.OneHotEncoder(cols = ['City'])
# в метод .fit_transform() мы передадим весь датафрейм целиком
df_catenc = ohe_encoder.fit_transform(df_catenc)
df_catenc
```

	Name	Age	City_1	City_2	City_3	City_4	City_5	Region	Experience	Salary
0	Иван	35	1	0	0	0	0	1	7	95
1	Николай	43	0	1	0	0	0	1	13	135
2	Алексей	21	0	0	1	0	0	1	2	73
3	Александра	34	0	0	0	1	0	0	8	100
4	Евгений	24	1	0	0	0	0	1	4	78
5	Елена	27	0	0	0	0	1	0	12	110

Что очень удобно, класс OneHotEncoder библиотеки category_encoders вставил новые столбцы сразу в изначальный датафрейм и удалил исходный признак.

Сравнение инструментов

Создадим два очень простых датасета из одного признака: один обучающий, второй — тестовый. В первом датасете в этом признаке (назовем его recom) будет три категории: yes, no, maybe. Во втором, только две, yes и no.

```
train = pd.DataFrame({'recom' : ['yes', 'no', 'maybe']})
train
```

	recom
0	yes
1	no
2	maybe

```
test = pd.DataFrame({'recom' : ['yes', 'no', 'yes']})
test
```

	recom
0	yes
1	no
2	yes

Теперь применим каждый из приведенных выше инструментов к этим датасетам (напомню, что обучать кодировщик мы должны на обущающей выборке, чтобы избежать утечки данных).

pd.get_dummies()

Функция **pd.get_dummies()** не «запоминает» категории при обучении.

```
pd.get_dummies(train)
```

	recom_maybe	recom_no	recom_yes
0	False	False	True
1	False	True	False
2	True	False	False

```
pd.get_dummies(test)
```

	recom_no	recom_yes
0	False	True
1	True	False
2	False	True

При попытке обучить модель будет ошибка.

OHE sklearn

Посмотрим, как с этим справится класс OneHotEncoder библиотеки sklearn.

```
ohe = OneHotEncoder()
ohe_model = ohe.fit(train)
ohe_model.categories_
```

```
[array(['maybe', 'no', 'yes'], dtype=object)]
```

```
train_arr = ohe_model.transform(train).toarray()
pd.DataFrame(train_arr, columns = ['maybe', 'no', 'yes'])
```

	maybe	no	yes
0	0.0	0.0	1.0
1	0.0	1.0	0.0
2	1.0	0.0	0.0

2	1.0	0.0	0.0

```
test_arr = ohe_model.transform(test).toarray()
pd.DataFrame(test_arr, columns = ['maybe', 'no', 'yes'])
```

	maybe	no	yes
0	0.0	0.0	1.0
1	0.0	1.0	0.0
2	0.0	0.0	1.0

Мы видим, что этот кодировщик учел отсутствующую в тестовой выборке категорию. Впрочем, в обратном случае, когда категория отсутствует в обучающей выборке, OneHotEncoder не будет иметь возможности правильно закодировать датасеты.

```
ohe = OneHotEncoder()
ohe_model = ohe.fit(test)
ohe_model.categories_
```

```
[array(['no', 'yes'], dtype=object)]
```

OHE category_encoders

Попробуем инструмент из библиотеки category_encoders.

```
ohe_encoder = ce.OneHotEncoder()
ohe_encoder.fit(train)
```

```
OneHotEncoder(cols=['recom'])
```

```
# категория maybe стоит на последнем месте
ohe_encoder.transform(test)
```

	recom_1	recom_2	recom_3
0	1	0	0
1	0	1	0
2	1	0	0

```
# убедимся в этом, добавив названия столбцов
test_df = ohe_encoder.transform(test)
test_df.columns = ohe_encoder.category_mapping[0]['mapping'].index[:3]
test_df
```

	yes	no	maybe
0	1	0	0
1	0	1	0
2	1	0	0

Binning

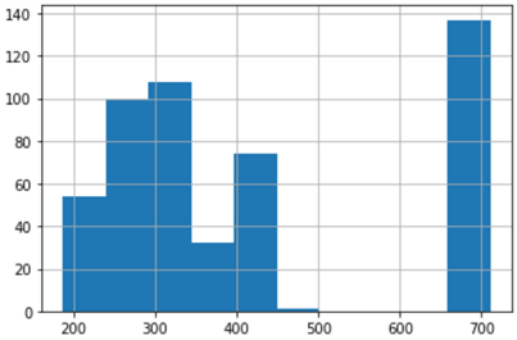
Некоторые (в частности, мультимодальные) количественные распределения не поддаются трансформации и приведению, например, к нормальному распределению.

Для того чтобы извлечь ценную информацию из таких признаков можно попробовать сделать переменные категориальными, разбив данные на интервалы, которые и будут классами нового признака. Такой подход называется **binning** или **bucketing**.

Вновь обратимся к датасету о недвижимости в Бостоне и, в частности, рассмотрим переменную TAX.

boston.csv **Скачать**

```
boston = pd.read_csv('/content/boston.csv')
boston.TAX.hist();
```



Как мы видим, распределение вряд ли можно трансформировать, используя какое-либо преобразование. Применим binning.

На равные интервалы

Подход **binning на равные интервалы** (binning with equally spaced boundaries) предполагает, что мы берем диапазон от минимального до максимального значений и делим его на нужное нам количество *равных* частей (если мы хотим получить три интервала, то нам нужно четыре границы).

```
min_value = boston.TAX.min()
max_value = boston.TAX.max()

bins = np.linspace(min_value, max_value, 4)
bins
```

```
array([187.        , 361.66666667, 536.33333333, 711.        ])
```

Создадим названия категорий.

```
labels = ['low', 'medium', 'high']
```

Применим **функцию pd.cut()**. В параметр bins мы передадим интервалы, в labels — названия категорий.

```
boston['TAX_binned'] = pd.cut(boston.TAX,
                              bins = bins,
```

```
                                labels = labels,
                                # уточним, что первый интервал должен включать
                                # нижнюю границу (значение 187)
                                include_lowest = True)
```

Посмотрим на результат.

```
boston[['TAX', 'TAX_binned']].sample(5, random_state = 42)
```

	TAX	TAX_binned
173	296.0	low
274	254.0	low
491	711.0	high
72	305.0	low
452	666.0	high

Границы и количество элементов в них можно получить с помощью метода **.value_counts()**.

```
boston.TAX.value_counts(bins = 3, sort = False)
```

```
(186.475, 361.667]    273
(361.667, 536.333]     96
(536.333, 711.0]      137
Name: TAX, dtype: int64
```

Результат этого метода позволяет выявить недостаток подхода binning на равные интервалы. Количество объектов внутри интервалов сильно различается. Преодолеть эту особенность можно с помощью деления по квантилям.

По квантилям

Binning по квантилям (quantile binning) позволяет разделить наблюдения не по значениям признака, а по количеству объектов в интервале. Например, выберем разделение на три части.

```
# для наглядности вначале найдем интересующие нас квантили
np.quantile(boston.TAX, q = [1/3, 2/3])
```

```
array([300., 403.])
```

Применим **функцию pd.qcut()**.

```
boston['TAX_qbinned'], boundaries = pd.qcut(boston.TAX,
                                            q = 3,
                                            # precision определяет округление
                                            precision = 1,
                                            labels = labels,
                                            retbins = True)

boundaries
```

```
array([187., 300., 403., 711.])
```

	TAX	TAX_qbinned
173	296.0	low

274	254.0	low
491	711.0	high
72	305.0	medium
452	666.0	high

```
boston.TAX_qbinned.value_counts()
```

```
low       172
high      168
medium    166
Name: TAX_qbinned, dtype: int64
```

Как вы видите, в данном случае количество объектов примерно одинаковое. Наглядную иллюстрацию двух подходов можно посмотреть здесь.

KBinsDiscretizer

Эти же задачи можно решить с помощью класса KBinsDiscretizer библиотеки sklearn. Рассмотрим три основных параметра класса:

- параметр **strategy** определяет, как будут делиться интервалы:
  - на равные части (uniform);
  - по квантилям (quantile); или
  - так, чтобы значения в каждом кластере относились к центроиду (kmeans);
- параметр **encode** определяет, как закодировать интервалы:
  - ordinal, т.е. числами от 1 до n интервалов; или
  - one-hot encoding;
- количество интервалов **n_bins**.

Применим каждую из стратегий. Так как в категориях заложен порядок, выберем ordinal кодировку.

```
from sklearn.preprocessing import KBinsDiscretizer
```

strategy = uniform

```
est = KBinsDiscretizer(n_bins = 3, encode = 'ordinal',
                       strategy = 'uniform', subsample = None)

est.fit(boston[['TAX']])
est.bin_edges_
```

```
array([array([187.        , 361.66666667, 536.33333333, 711.        ])],
      dtype=object)
```

```
np.unique(est.transform(boston[['TAX']]), return_counts = True)
```

```
(array([0., 1., 2.]), array([273,  96, 137]))
```

strategy = quantile

```
est = KBinsDiscretizer(n_bins = 3, encode = 'ordinal', strategy = 'quantile')
est.fit(boston[['TAX']])
est.bin_edges_
```

```
array([array([187., 300., 403., 711.])], dtype=object)
```

```
np.unique(est.transform(boston[['TAX']]), return_counts = True)
```

```
(array([0., 1., 2.]), array([165, 143, 198]))
```

strategy = kmeans

```
est = KBinsDiscretizer(n_bins = 3, encode = 'ordinal',
                       strategy = 'kmeans', subsample = None)

est.fit(boston[['TAX']])
est.bin_edges_
```

```
array([array([187.        , 338.7198937 , 535.07350433, 711.        ])],
      dtype=object)
```

```
np.unique(est.transform(boston[['TAX']]), return_counts = True)
```

```
(array([0., 1., 2.]), array([262, 107, 137]))
```

Еще одно сравнение стратегий разделения на интервалы можно посмотреть здесь.

С помощью статистических показателей

Дополнительно замечу, что интервалы можно заполнить каким-либо статистическим показателем. Например, медианой. Для наглядности снова создадим только три интервала и найдем медианное значение внутри каждого из них.

Воспользуемся **функцией binned_statistic()** модуля scipy.stats. Функция возвращает медианы каждого из интервалов, границы интервалов, а также к какому из интервалов относится каждое из наблюдений.

Нас будут интересовать метрики и границы интервалов.

```
from scipy.stats import binned_statistic

medians, bin_edges, _ = binned_statistic(boston.TAX,
                                         np.arange(0, len(boston)),
                                         statistic = 'median',
                                         bins = 3)

medians, bin_edges
```

```
(array([216. , 147.5, 424. ]),
 array([187.        , 361.66666667, 536.33333333, 711.        ]))
```

Подставим эти значения в функцию **pd.cut()**.

```
boston['TAX_binned_median'] = pd.cut(boston.TAX,
                                     bins = bin_edges,
                                     labels = medians,
                                     include_lowest = True)
```

```
boston['TAX_binned_median'].value_counts()
```

```
216.0    273
424.0    137
147.5     96
Name: TAX_binned_median, dtype: int64
```

Алгоритм Дженкса

Алгоритм естественных границ Дженкса (Jenks natural breaks optimization) делит данные на группы (кластеры) таким образом, чтобы минимизировать отклонение наблюдений от среднего каждого класса (дисперсию внутри классов) и максимизировать отклонение среднего каждого класса от среднего других классов (дисперсию между классами).

Этот алгоритм также можно использовать для определения границ интервалов. Установим библиотеку jenkspy.

```
!pip install jenkspy
```

Найдем оптимальные границы. Количество интервалов (n_classes) нужно по-прежнему указывать вручную.

```
import jenkspy

breaks = jenkspy.jenks_breaks(boston.TAX, n_classes = 3)
breaks
```

```
[187.0, 337.0, 469.0, 711.0]
```

Подставим интервалы в функцию **pd.cut()**.

```
boston['TAX_binned_jenks'] = pd.cut(boston.TAX,
                                    bins = breaks,
                                    labels = labels,
                                    include_lowest = True)

boston['TAX_binned_jenks'].value_counts()
```

```
low       262
high      137
medium    107
Name: TAX_binned_jenks, dtype: int64
```

Подведем итог

Сегодня мы рассмотрели базовые методы кодирования категориальных переменных, а также стратегии binning/bucketing.