

## ✓ Линейная регрессия и sklearn

В этой домашней работе:

- Обучим линейную регрессию для предсказания цены дома;
- Научимся работать с разными типами признаков;
- Поймем, в чем отличие между разными регуляризаторами;
- Научимся пользоваться основными инструментами в sklearn: моделями, трансформерами и pipeline;
- Обсудим преобразования признаков и целевой переменной, которые могут помочь в обучении линейных моделей.

Скачайте тренировочную и тестовую выборку из соревнования на kaggle: [House Prices: Advanced Regression Techniques](https://www.kaggle.com/datasets/robikscube/house-prices-advanced-regression-techniques). Разместите данные рядом с тетрадкой или поправьте пути при их чтении.

# Чтобы не перегружать ячейки с кодом, вынесем сюда все импорты

```
import warnings

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import Lasso, LinearRegression, Ridge
from sklearn.model_selection import GridSearchCV, cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import KBinsDiscretizer, OneHotEncoder, PolynomialFeatures, StandardScaler

sns.set_theme(style="darkgrid")
warnings.simplefilter("ignore")
%matplotlib inline
```

## ✓ Часть 0. Введение в линейные модели

Напомним, что линейная регрессия — это модель следующего вида:

$$a(x) = \langle w, x \rangle + w_0$$

где  $w \in \mathbb{R}^d$ ,  $w_0 \in \mathbb{R}$ . Обучить линейную регрессию — значит найти  $w$  и  $w_0$ .

В машинном обучении часто говорят об *обобщающей способности модели*, то есть о способности модели работать на новых, тестовых данных хорошо. Если модель будет идеально предсказывать выборку, на которой она обучалась, но при этом просто ее запомнит, не "вытащив" из данных никакой закономерности, от нее будет мало толку. Такую модель называют *переобученной*: она слишком подстроилась под обучающие примеры, не выявив никакой полезной закономерности, которая позволила бы ей совершать хорошие предсказания на данных, которые она ранее не видела.

Рассмотрим следующий пример, на котором будет хорошо видно, что значит переобучение модели. Для этого нам понадобится сгенерировать синтетические данные. Рассмотрим зависимость

$$y(x) = \cos(1.5\pi x)$$

$y$  — целевая переменная, а  $x$  — объект (просто число от 0 до 1). В жизни мы наблюдаем какое-то конечное количество пар объект-таргет, поэтому смоделируем это, взяв 30 случайных точек  $x_i$  в отрезке  $[0; 1]$ . Более того, в реальной жизни целевая переменная может быть зашумленной (измерения в жизни не всегда точны), смоделируем это, зашумив значение функции нормальным шумом:  $\tilde{y}_i = y(x_i) + \mathcal{N}(0, 0.01)$ .

Попытаемся обучить три разных линейных модели: признаки для первой —  $\{x\}$ , для второй —  $\{x, x^2, x^3, x^4\}$ , для третьей —  $\{x, \dots, x^{20}\}$ .

```
np.random.seed(36)
x = np.linspace(0, 1, 100)
y = np.cos(1.5 * np.pi * x)

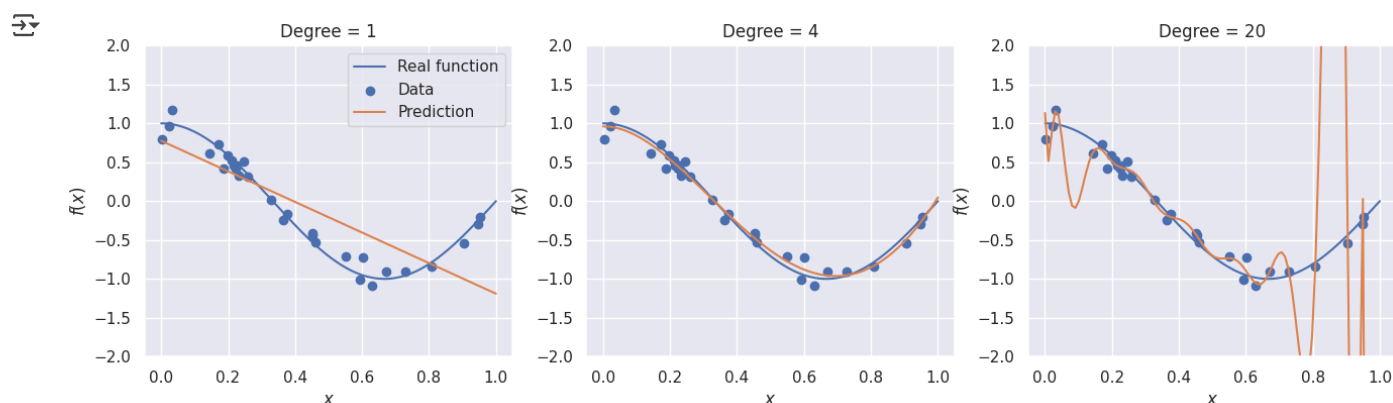
x_objects = np.random.uniform(0, 1, size=30)
y_objects = np.cos(1.5 * np.pi * x_objects) + np.random.normal(scale=0.1, size=x_objects.shape)

fig, axs = plt.subplots(figsize=(16, 4), ncols=3)
for i, degree in enumerate([1, 4, 20]):
    X_objects = PolynomialFeatures(degree, include_bias=False).fit_transform(x_objects[:, None])
    X = PolynomialFeatures(degree, include_bias=False).fit_transform(x[:, None])
    regr = LinearRegression().fit(X_objects, y_objects)
    y_pred = regr.predict(X)
    axs[i].plot(x, y, label="Real function")
    axs[i].scatter(x_objects, y_objects, label="Data")
    axs[i].plot(x, y_pred, label="Prediction")
```

```

if i == 0:
    axs[i].legend()
axs[i].set_title("Degree = %d" % degree)
axs[i].set_xlabel("$x$")
axs[i].set_ylabel("$f(x)$")
axs[i].set_ylim(-2, 2)

```



**Вопрос 1:** Почему первая модель получилась плохой, а третья переобучилась?

**Ответ:** Для первой модели учитываются  $X$  только в первой степени, получается линейная зависимость, когда она на самом деле таковой не является. В третьей модели степени слишком высокие: модель подобрала веса таким образом, чтобы наиболее точно попадать в ответ на тренировочной выборке — переобучилась

Чтобы избежать переобучения, модель регуляризуют. Обычно переобучения в линейных моделях связаны с большими весами, а поэтому модель часто штрафуют за большие значения весов, добавляя к функционалу качества, например, квадрат  $\ell^2$ -нормы вектора  $w$ :

$$Q_{reg}(X, y, a) = Q(X, y, a) + \lambda \|w\|_2^2$$

Это слагаемое называют  $\ell_2$ -регуляризатором, а коэффициент  $\lambda$  — коэффициентом регуляризации.

**Вопрос 2:** Почему большие веса в линейной модели — плохо?

**Ответ:** Большие веса показывают, что модель слишком пригнорилась к обучающей выборке и будет показывать низкие метрики на тестовой, она слишком чувствительна к шуму

**Вопрос 3:** Почему регуляризовать  $w_0$  — плохая идея?

**Ответ:**  $w_0$  отвечает за смещение, оно не связано с признаками и позволяет более точно подстоить предсказание. При регуляризации  $w_0$  график предсказания сместится на начало координат, т.к. модель будет штрафиться за любое его отклонение от нуля. Это снизит точность

**Вопрос 4:** На что влияет коэффициент  $\lambda$ ? Что будет происходить с моделью, если  $\lambda$  начать уменьшать? Что будет, если  $\lambda$  сделать слишком большим?

**Ответ:** Чем меньше  $\lambda$  тем более большие значения весов может выбрать модель, что приведет к ее переобучению. Если же поставить слишком большое  $\lambda$ , модели более выгодно просто занулить все веса и останется только  $w_0$ , т.е. модель будет всегда предсказывать среднее значение — недообучится

## ✓ Часть 1. Загружаем данные

```

train_data = pd.read_csv("src/homeworks/homework3/train.csv")
train_data.head()

```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	F
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	

5 rows × 15 columns

```
train_data.shape
```

```
(1460, 81)
```

```
train_data.columns
```

```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

Первое, что стоит заметить — у нас в данных есть уникальное для каждого объекта поле `id`. Обычно такие поля только мешают и способствуют переобучению. Удалим это поле из данных.

Выделим валидационную выборку, а также отделим значения целевой переменной от данных.

**Вопрос 1:** Почему поля типа `id` могут вызвать переобучение модели (не обязательно линейной)?

**Ответ:** `'id'` указывает лишь на номер и никак не связано с признаками. Они уникальны, их наличие в тренировочной выборке бессмысленно. При его наличии модель может запомнить конкретные записи, а не общую закономерность

**Вопрос 2:** Почему стоит дополнительно отделять валидационную выборку?

**Ответ:** Для подбора гиперпараметров, чтобы недопустить переобучение модели. Без нее подберутся параметры для наиболее точного угадывания значения на тестовой выборке, а при попадании в реальные условия модель может показывать неудовлетворительные результаты

**Вопрос 3:** Обратите внимание на фиксацию `random_state` при сплите данных. Почему это важно?

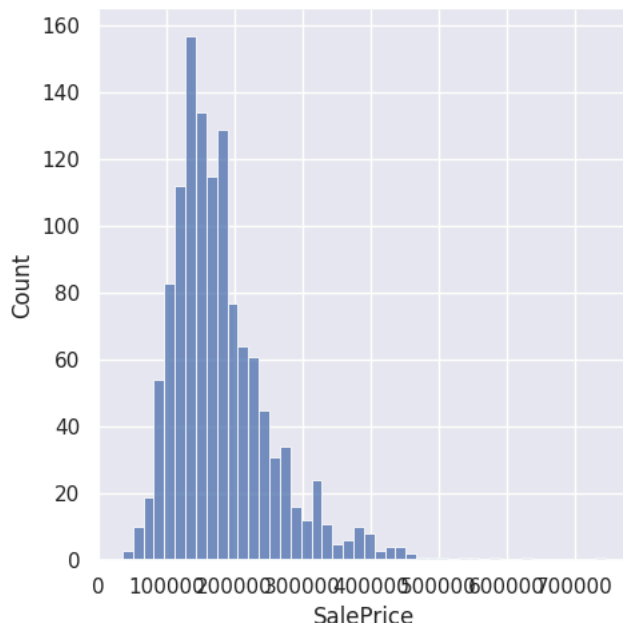
**Ответ:** Значение задает воспроизводимость обучения. При указанном `'random_state'` датасет всегда будет разделяться одинаково

```
y = train_data["SalePrice"]
X = train_data.drop(["SalePrice", "Id"], axis=1)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.15, random_state=42, shuffle=True)
```

Посмотрим сначала на значения целевой переменной.

```
sns.displot(y_train)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f95a0dc08f0>
```




Судя по гистограмме, у нас есть примеры с нетипично большой стоимостью, что может помешать нам, если наша функция потерь слишком чувствительна к выбросам. В дальнейшем мы рассмотрим способы, как минимизировать ущерб от этого.

Так как для решения нашей задачи мы бы хотели обучить линейную регрессию, было бы хорошо найти признаки, "наиболее линейно" связанные с целевой переменной, иначе говоря, посмотреть на коэффициент корреляции Пирсона между признаками и целевой переменной. Заметим, что не все признаки являются числовыми, пока что мы не будем рассматривать такие признаки.

**Вопрос:** Что означает, что коэффициент корреляции Пирсона между двумя случайными величинами равен 1? -1? 0?

**Ответ:** 1: при возрастании одной — пропорционально возрастает другая, -1: при возрастании одной — вторая пропорционально убывает, 0: нет линейной зависимости

```
numeric_data = X_train.select_dtypes([np.number])
numeric_features = numeric_data.columns
numeric_data.head()
```



	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF
984	90	75.0	10125	5	5	1977	1977	0.0	0	
777	20	100.0	13350	5	5	1974	1974	0.0	762	
707	120	48.0	6240	8	5	2006	2006	176.0	863	
598	20	80.0	12984	5	6	1977	1977	459.0	1283	14
874	50	52.0	5720	5	6	1941	1950	0.0	0	

5 rows × 36 columns

Заметим, что в данных присутствуют пропуски, заполним их средним значением по признаку.

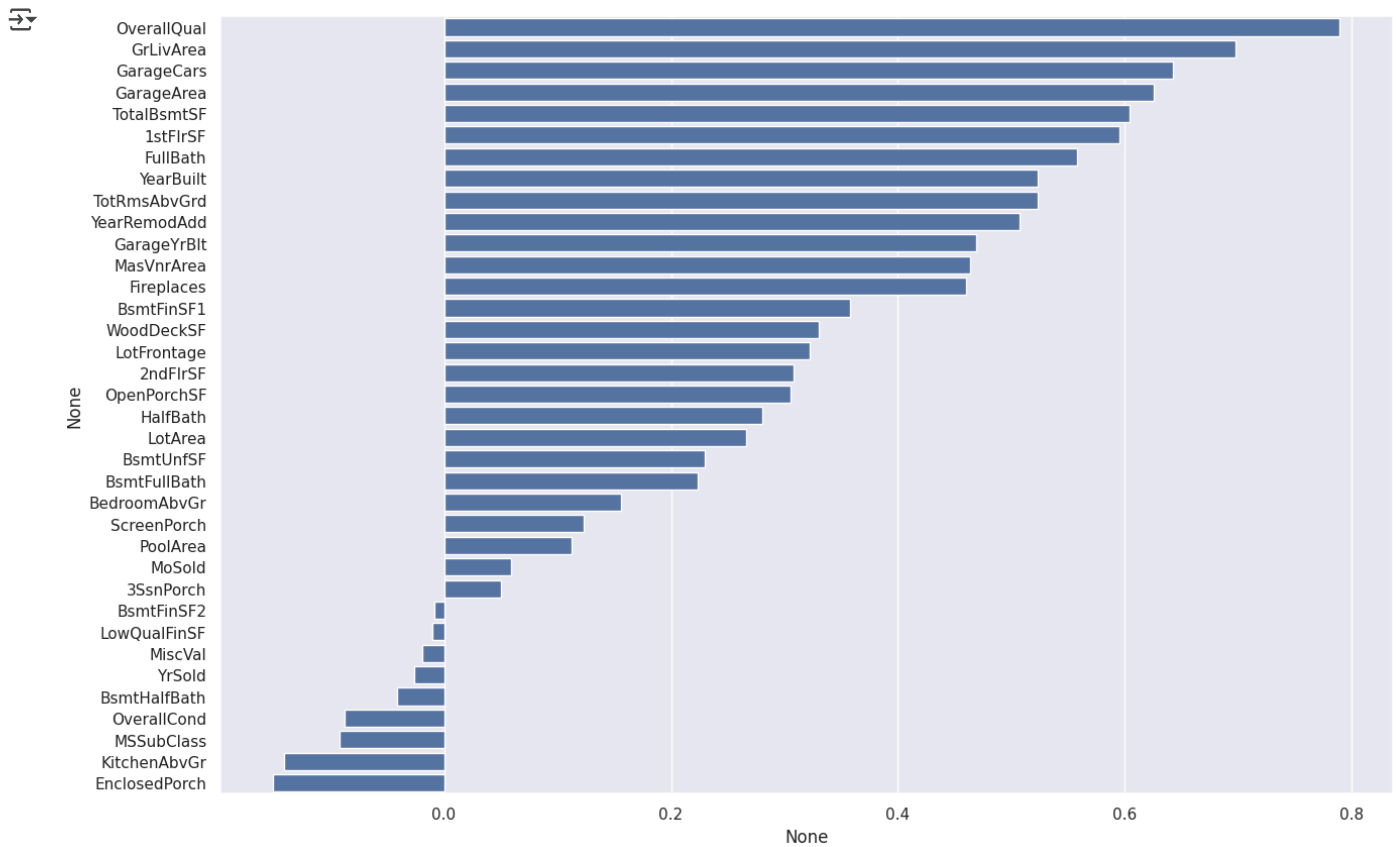
**Вопрос:** Как правильно заполнять пропуски для валидационной и тестовой выборки?

**Ответ:** Заполнять их средним или часто встречающимся по обучающей выборке, чтобы избежать переобучения, показав среднее валидационных/тестовых данные модели

```
X_train[numeric_features] = X_train[numeric_features].fillna(X_train[numeric_features].mean())
X_val[numeric_features] = X_val[numeric_features].fillna(X_train[numeric_features].mean())
```

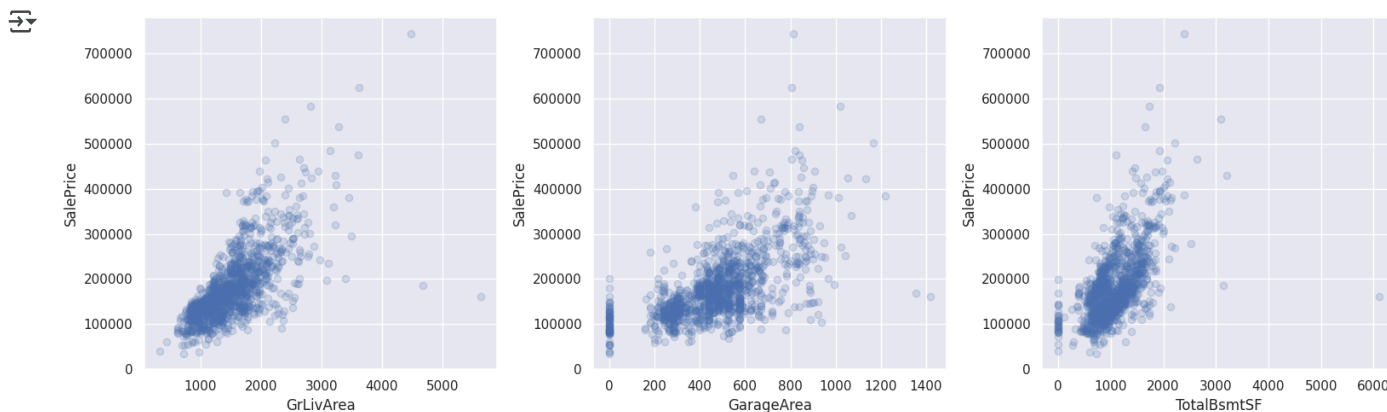
```
correlations = X_train[numeric_features].corrwith(y_train).sort_values(ascending=False)
```

```
plot = sns.barplot(y=correlations.index, x=correlations)
plot.figure.set_size_inches(15, 10)
```



Посмотрим на признаки из начала списка. Для этого нарисуем график зависимости целевой переменной от каждого из признаков. На этом графике каждая точка соответствует паре признак-таргет (такие графики называются scatter-plot).

```
fig, axs = plt.subplots(figsize=(16, 5), ncols=3)
for i, feature in enumerate(["GrLivArea", "GarageArea", "TotalBsmntSF"]):
    axs[i].scatter(X_train[feature], y_train, alpha=0.2)
    axs[i].set_xlabel(feature)
    axs[i].set_ylabel("SalePrice")
plt.tight_layout()
```



Видим, что между этими признаками и целевой переменной действительно наблюдается линейная зависимость.

## ✓ Часть 2. Первая модель

*Немного об обучении моделей.* В арсенале ML-специалиста кроме `pandas` и `matplotlib` должны быть библиотеки, позволяющие обучать модели. Для простых моделей (линейные модели, решающее дерево, ...) отлично подходит `sklearn`: в нем очень понятный и простой интерфейс. Несмотря на то, что в `sklearn` есть реализация бустинга и простых нейронных сетей, ими все же не пользуются и предпочитают специализированные библиотеки: `XGBoost`, `LightGBM` и пр. для градиентного бустинга над деревьями, `PyTorch`, и пр. для нейронных сетей. Так как мы будем обучать линейную регрессию, нам подойдет реализация из `sklearn`.

Попробуем обучить линейную регрессию на числовых признаках из нашего датасета. В `sklearn` есть несколько классов, реализующих линейную регрессию:

- [LinearRegression](#) — "классическая" линейная регрессия с оптимизацией MSE. Веса находятся как точное решение:  

$$w^* = (X^T X)^{-1} X^T y$$
- [Ridge](#) — линейная регрессия с оптимизацией MSE и  $\ell_2$ -регуляризацией
- [Lasso](#) — линейная регрессия с оптимизацией MSE и  $\ell_1$ -регуляризацией

У моделей из `sklearn` есть методы `fit` и `predict`. Первый принимает на вход обучающую выборку и вектор целевых переменных и обучает модель, второй, будучи вызванным после обучения модели, возвращает предсказание на выборке.

Попробуем обучить нашу первую модель на числовых признаках, которые у нас сейчас есть:

```
model = Ridge()
model.fit(X_train[numeric_features], y_train)
y_pred = model.predict(X_val[numeric_features])
y_train_pred = model.predict(X_train[numeric_features])
```

Стандартный способ оценить качество регрессии — **MSE**

```
def mean_squared_error(y_true: np.ndarray, y_pred: np.ndarray, squared: bool = True) -> float:
    """Calculate Mean Squared Error

    Args:
        y_true: array with ground truth values, [n_samples,]
        y_pred: array with predicted values, [n_samples,]
        squared: whether to return squared MSE or not

    Returns:
        number, calculated error
    """
    if len(y_true) != len(y_pred):
        raise ValueError("Length of arrays 'y_true' and 'y_pred' must be equal")

    if np.isnan(y_true).any() or np.isnan(y_pred).any():
        raise ValueError("Arrays must not contain None values.")
```

```

mse = np.mean((y_pred - y_true) ** 2)

if squared:
    return mse
else:
    return np.sqrt(mse)

print("Val RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
print("Train RMSE = %.4f" % mean_squared_error(y_train, y_train_pred, squared=False))

```

Val RMSE = 38413.6660  
Train RMSE = 33716.3830

Мы обучили первую модель и даже посчитали ее качество на отложенной выборке! Давайте теперь посмотрим на то, как можно оценить качество модели с помощью кросс-валидации. Принцип кросс-валидации изображен на рисунке



При кросс-валидации мы делим обучающую выборку на  $n$  частей (fold). Затем мы обучаем  $n$  моделей: каждая модель обучается при отсутствии соответствующего фолда, то есть  $i$ -ая модель обучается на всей обучающей выборке, кроме объектов, которые попали в  $i$ -ый фолд (out-of-fold). Затем мы измеряем качество  $i$ -ой модели на  $i$ -ом фолде. Так как он не участвовал в обучении этой модели, мы получим "честный результат". После этого, для получения финального значения метрики качества, мы можем усреднить полученные нами  $n$  значений.

```

cv_scores = cross_val_score(model, X_train[numeric_features], y_train, cv=10, scoring="neg_root_mean_squared_error")
print("Cross validation scores:\n\t", "\n\t".join("%.4f" % x for x in cv_scores))
print("Mean CV MSE = %.4f" % np.mean(-cv_scores))

```

Cross validation scores:

```

-28779.2227
-48361.2216
-28832.5605
-42870.6846
-67722.1283
-35268.1657
-29020.3239
-22814.2673
-27948.8006
-26959.4486
Mean CV MSE = 35857.6824

```

Обратите внимание на то, что результаты `cv_scores` получились отрицательными. Это соглашение в `sklearn` (скоринговую функцию нужно максимизировать). Поэтому все стандартные скореры называются `neg_*`, например, `neg_root_mean_squared_error`.

Обратите внимание, что по отложенной выборке и при кросс-валидации мы считаем RMSE (Root Mean Squared Error), хотя в функционале ошибки при обучении модели используется MSE.

$$\text{RMSE}(X, y, a) = \sqrt{\frac{1}{\ell} \sum_{i=1}^{\ell} (y_i - a(x_i))^2}$$

**Вопрос:** Почему оптимизация RMSE эквивалентна оптимизации MSE?

**Ответ:** Потому что корень монотонно возрастающая функция

Для того, чтобы иметь некоторую точку отсчета, удобно посчитать оптимальное значение функции потерь при константном предсказании.

**Вопрос:** Чему равна оптимальная константа для RMSE?

**Ответ:** Она равна оптимальной константе для MSE, а она в свою очередь считается как производная:

$$-\frac{2}{n} \sum_{i=1}^n (y_i - c) = 0 \Rightarrow c = \frac{\sum y_i}{n}$$

```

best_constant = np.mean(y_val)
best_constant

```

np.float64(180373.51598173517)

```

print(
    "Test RMSE with best constant = %.4f"
    % mean_squared_error(y_val, best_constant * np.ones(y_val.shape), squared=False)
)
print(
    "Train RMSE with best constant = %.4f"

```

```
% mean_squared_error(y_train, best_constant * np.ones(y_train.shape), squared=False)
)
```

```
↗ Test RMSE with best constant = 90103.0896
Train RMSE with best constant = 77378.3943
```

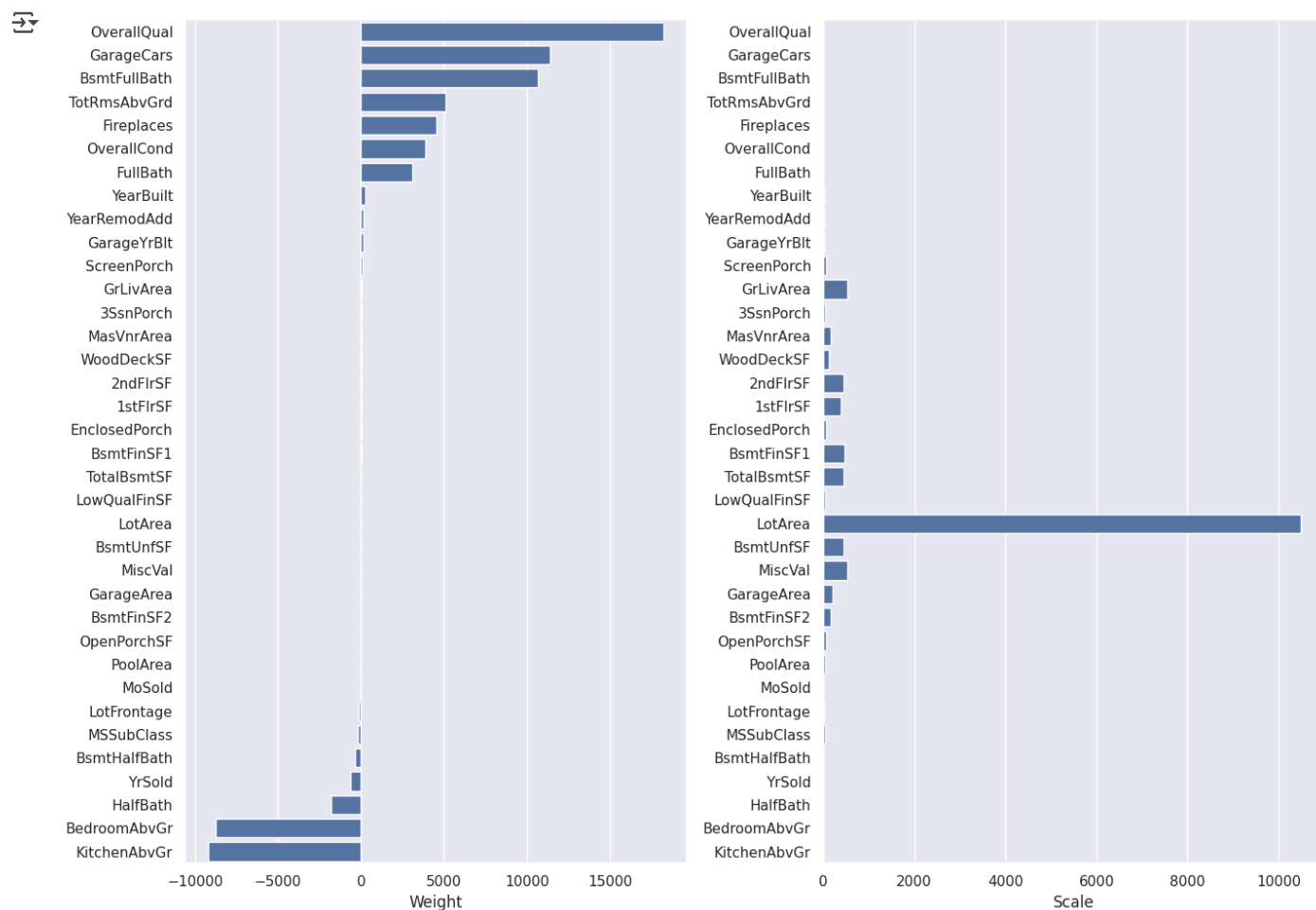
Давайте посмотрим на то, какие же признаки оказались самыми "сильными". Для этого визуализируем веса, соответствующие признакам. Чем больше вес — тем более сильным является признак.

**Вопрос:** Почему это не совсем правда?

**Ответ:** Признаки не отмасштабированы

```
def show_weights(features, weights, scales):
    fig, axs = plt.subplots(figsize=(14, 10), ncols=2)
    sorted_weights = sorted(zip(weights, features, scales), reverse=True)
    weights = [x[0] for x in sorted_weights]
    features = [x[1] for x in sorted_weights]
    scales = [x[2] for x in sorted_weights]
    sns.barplot(y=features, x=weights, ax=axs[0])
    axs[0].set_xlabel("Weight")
    sns.barplot(y=features, x=scales, ax=axs[1])
    axs[1].set_xlabel("Scale")
    plt.tight_layout()
```

```
show_weights(numeric_features, model.coef_, X_train[numeric_features].std())
```





Будем масштабировать наши признаки перед обучением модели. Это, среди прочего, сделает нашу регуляризацию более честной: теперь все признаки будут регуляризоваться в равной степени.

Для этого воспользуемся трансформером [StandardScaler](#). Трансформеры в `sklearn` имеют методы `fit` и `transform` (а еще `fit_transform`). Метод `fit` принимает на вход обучающую выборку и считает по ней необходимые значения (например статистики, как `StandardScaler`: среднее и стандартное отклонение каждого из признаков). `transform` применяет преобразование к переданной выборке.

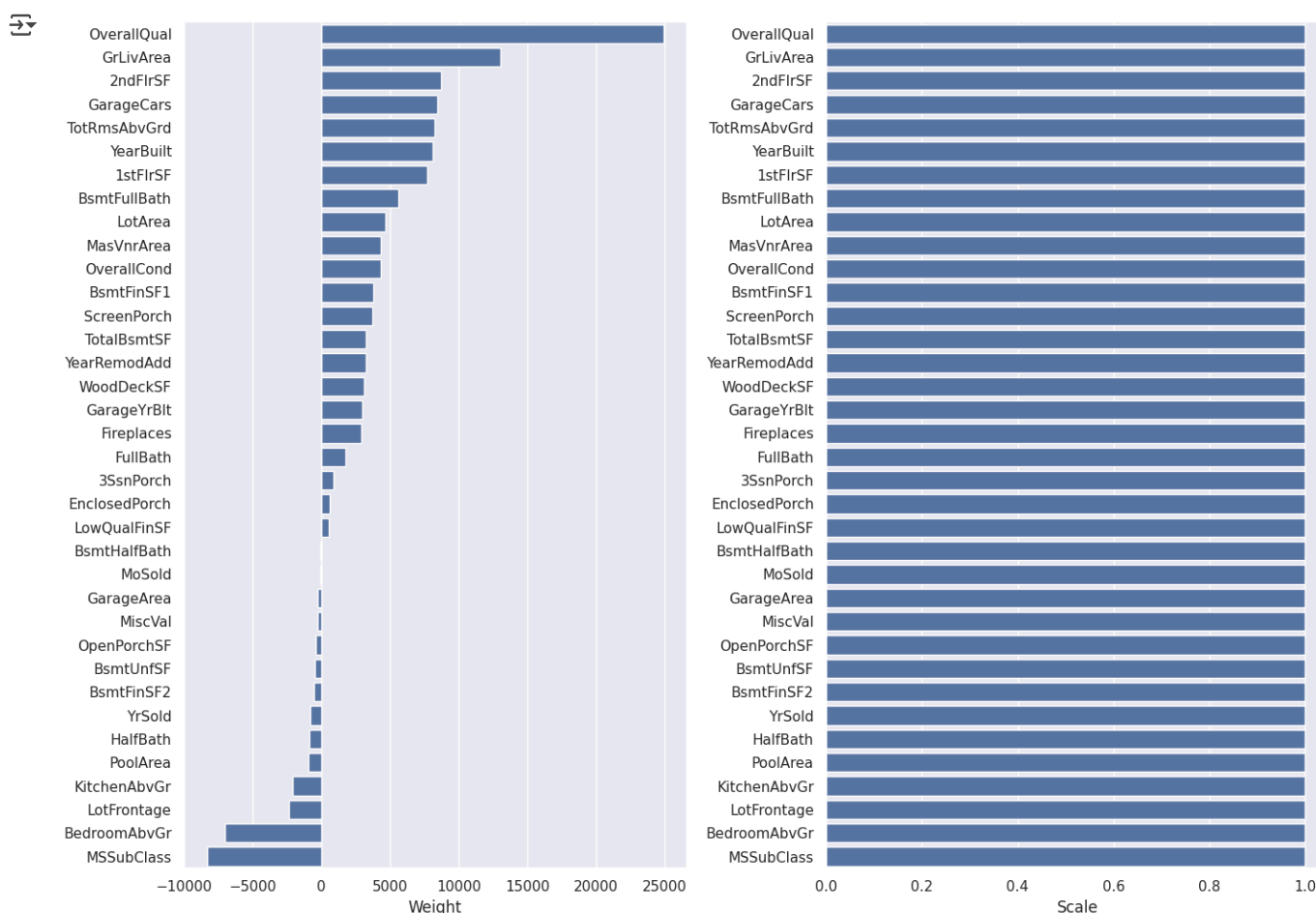
```
scaler = StandardScaler().fit(X_train[numeric_features])
X_train_scaled = scaler.transform(X_train[numeric_features])
X_val_scaled = scaler.transform(X_val[numeric_features])

model = Ridge()
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_val_scaled)
y_train_pred = model.predict(X_train_scaled)

print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
print("Train RMSE = %.4f" % mean_squared_error(y_train, y_train_pred, squared=False))
```

```
↗ Test RMSE = 38413.2544
   Train RMSE = 33716.3558
```

```
scales = pd.Series(data=X_train_scaled.std(axis=0), index=numeric_features)
show_weights(numeric_features, model.coef_, scales)
```



Наряду с параметрами (веса  $w$ ,  $w_0$ ), которые модель оптимизирует на этапе обучения, у модели есть и гиперпараметры. У нашей модели это  $\alpha$  — коэффициент регуляризации. Подбирают его обычно по сетке, измеряя качество на валидационной (не тестовой) выборке или с помощью кросс-валидации. Посмотрим, как это можно сделать.

Для начала зададим возможные значения гиперпараметра, воспользуемся `np.logspace`, чтобы узнать оптимальный порядок величины. Ограничим допустимые значения  $10^{-2}$  и  $10^3$ , возьмем 20 точек.

```
alphas = np.logspace(-2, 3, 20)
assert alphas[0] == 1e-2
assert alphas[-1] == 1e3
assert len(alphas) == 20

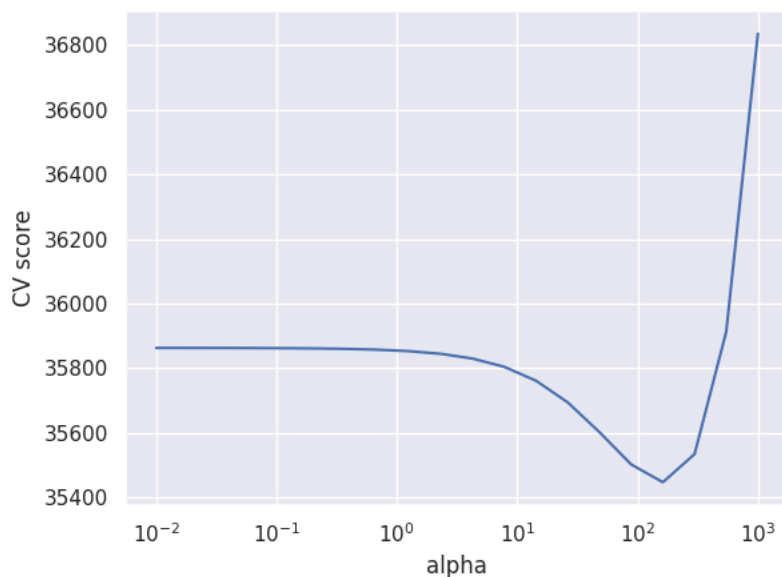
alphas
array([1.00000000e-02, 1.83298071e-02, 3.35981829e-02, 6.15848211e-02,
       1.12883789e-01, 2.06913808e-01, 3.79269019e-01, 6.95192796e-01,
       1.27427499e+00, 2.33572147e+00, 4.28133240e+00, 7.84759970e+00,
       1.43844989e+01, 2.63665090e+01, 4.83293024e+01, 8.85866790e+01,
       1.62377674e+02, 2.97635144e+02, 5.4559478e+02, 1.00000000e+03])

searcher = GridSearchCV(Ridge(), [{"alpha": alphas}], scoring="neg_root_mean_squared_error", cv=10)
searcher.fit(X_train_scaled, y_train)

best_alpha = searcher.best_params_["alpha"]
print("Best alpha = %.4f" % best_alpha)

plt.plot(alphas, -searcher.cv_results_["mean_test_score"])
plt.xscale("log")
plt.xlabel("alpha")
plt.ylabel("CV score")

Best alpha = 162.3777
Text(0, 0.5, 'CV score')
```



**Вопрос:** Почему мы не подбираем коэффициент регуляризации по обучающей выборке? По тестовой выборке?

**Ответ:** Тестовую выборку модель не должна видеть на этапе обучения, потому что используется для оценки уже обученной модели. Тренировочную выборку при обучении видит модель, если подбирать на ней, модель подстроится под нее. А также коэффициент регуляризации является гиперпараметром, которые подбираются как раз на валидационной выборке.

Попробуем обучить модель с подобранным коэффициентом регуляризации. Заодно воспользуемся очень удобным классом [Pipeline](#): обучение модели часто представляется как последовательность некоторых действий с обучающей и тестовой выборками (например, сначала нужно отмасштабировать выборку (причем для обучающей выборки нужно применить метод `fit`, а для тестовой — `transform`), а затем обучить/применить модель (для обучающей `fit`, а для тестовой — `predict`). Pipeline позволяет хранить эту последовательность шагов и корректно обрабатывает разные типы выборок: и обучающую, и тестовую.

```
simple_pipeline = Pipeline([("scaling", StandardScaler()), ("regression", Ridge(best_alpha))])

model = simple_pipeline.fit(X_train[numeric_features], y_train)
y_pred = model.predict(X_val[numeric_features])
print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
```

```
↗ Test RMSE = 38765.2831
```

### ✓ Часть 3. Работаем с категориальными признаками

Сейчас мы явно вытягиваем из данных не всю информацию, что у нас есть, просто потому, что мы не используем часть признаков. Эти признаки в датасете закодированы строками, каждый из них обозначает некоторую категорию. Такие признаки называются категориальными. Давайте выделим такие признаки и сразу заполним пропуски в них специальным значением (то, что у признака пропущено значение, само по себе может быть хорошим признаком).

```
categorical = list(X_train.dtypes[X_train.dtypes == "object"].index)
X_train[categorical].head()
```

```
↗
```

	MSZoning	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	...	Garage
984	RL	Pave	NaN	Reg	Lvl	AllPub	Inside	Gtl	Mitchel	Norm	...	A
777	RL	Pave	NaN	IR1	Lvl	AllPub	Inside	Gtl	Sawyer	Norm	...	A
707	RL	Pave	NaN	Reg	Lvl	AllPub	Inside	Gtl	NridgHt	Norm	...	A
598	RL	Pave	NaN	Reg	Bnk	AllPub	Inside	Gtl	Crawfor	Norm	...	A
874	RM	Pave	NaN	Reg	Lvl	AllPub	Inside	Gtl	OldTown	Artery	...	De

5 rows × 43 columns

В категориальных данных также есть пропуски, заполним их отдельной новой категорией NotGiven

```
X_train[categorical] = X_train[categorical].fillna("NotGiven")
X_val[categorical] = X_val[categorical].fillna("NotGiven")
```

```
X_train[categorical].sample(5)
```

```
↗
```

	MSZoning	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	...	Garage
596	RM	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	OldTown	Norm	...	
1413	RL	Pave	NotGiven	IR1	Lvl	AllPub	Corner	Gtl	SawyerW	Norm	...	
108	RM	Pave	NotGiven	Reg	Lvl	AllPub	Corner	Gtl	IDOTRR	Artery	...	1
851	RL	Pave	NotGiven	Reg	Lvl	AllPub	Inside	Gtl	Blmngtn	Norm	...	
134	RL	Pave	NotGiven	IR1	Lvl	AllPub	Inside	Gtl	Sawyer	Norm	...	

5 rows × 43 columns

Сейчас нам нужно как-то закодировать эти категориальные признаки числами, ведь линейная модель не может работать с такими абстракциями. Два стандартных трансформера из `sklearn` для работы с категориальными признаками — [OrdinalEncoder](#) (просто перенумеровывает значения признака натуральными числами) и [OneHotEncoder](#).

`OneHotEncoder` ставит в соответствие каждому признаку целый вектор, состоящий из нулей и одной единицы (которая стоит на месте, соответствующем принимаемому значению, таким образом кодируя его).

**Вопрос:** Проинтерпретируйте, что означают веса модели перед `OneHot`-кодированными признаками. Почему пользоваться `OrdinalEncoder` в случае линейной модели — скорее плохой вариант? Какие недостатки есть у `OneHot`-кодирования?

**Ответ:** Веса перед `OneHot`-кодированными признаками показывают насколько увеличивается или уменьшается предсказание модели, если объект принадлежит данной категории. `OrdinalEncoder` задает порядок, даже если его нет. Основной недостаток `OneHotEncoder` — увеличение числа признаков

```
column_transformer = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)

pipeline = Pipeline(steps=[("ohe_and_scaling", column_transformer), ("regression", Ridge())])

model = pipeline.fit(X_train, y_train)
y_pred = model.predict(X_val)
print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
```

```
↗ Test RMSE = 30608.8288
```

**Вопрос:** Как вы думаете, почему мы не производим скейлинг OneHot-кодированных признаков?

**Ответ:** Такие признаки принимают значения только 0 или 1, т.е. они уже отмасштабированы

Посмотрим на размеры матрицы после OneHot-кодирования:

```
print("Size before OneHot:", X_train.shape)
print("Size after OneHot:", column_transformer.transform(X_train).shape)
```

```
↗ Size before OneHot: (1241, 79)
  Size after OneHot: (1241, 301)
```

Как видим, количество признаков увеличилось более, чем в 3 раза. Это может повысить риски переобучиться: соотношение количества объектов к количеству признаков сильно сократилось.

Попытаемся обучить линейную регрессию с  $\ell_1$ -регуляризатором.

**Вопрос:** Каким полезным свойством обладает такой регуляризатор?

**Ответ:** Происходит отбор признаков — некоторые веса зануляются.

```
column_transformer = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)
```

```
lasso_pipeline = Pipeline(steps=[("ohe_and_scaling", column_transformer), ("regression", Lasso())])
```

```
model = lasso_pipeline.fit(X_train, y_train)
y_pred = model.predict(X_val)
print("RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
```

```
↗ RMSE = 31592.3587
```

```
ridge_zeros = np.sum(pipeline.steps[-1][-1].coef_ == 0)
lasso_zeros = np.sum(lasso_pipeline.steps[-1][-1].coef_ == 0)
print("Zero weights in Ridge:", ridge_zeros)
print("Zero weights in Lasso:", lasso_zeros)
```

```
↗ Zero weights in Ridge: 0
  Zero weights in Lasso: 34
```

Подберем для нашей модели оптимальный коэффициент регуляризации. Обратите внимание, как перебираются параметры у Pipeline.

```
alphas = np.logspace(-2, 4, 20)
searcher = GridSearchCV(
    lasso_pipeline, [{"regression__alpha": alphas}], scoring="neg_root_mean_squared_error", cv=10, n_jobs=-1
)
searcher.fit(X_train, y_train)
```

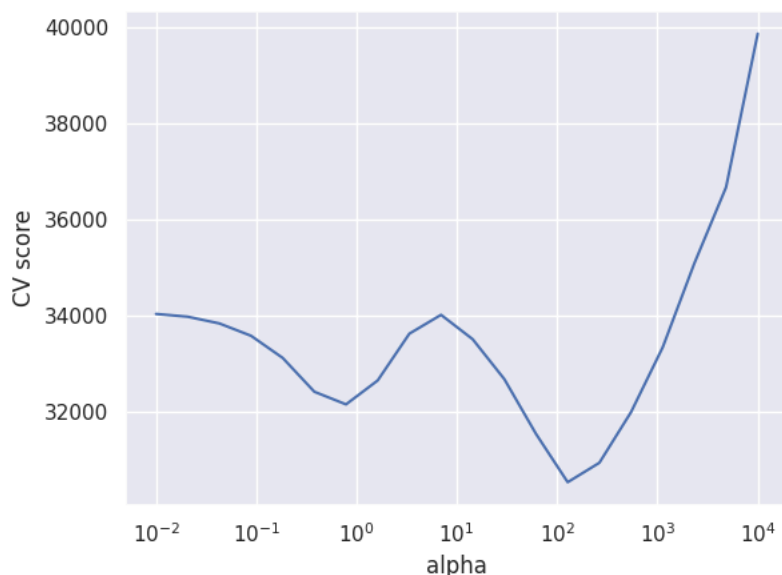
```
best_alpha = searcher.best_params_["regression__alpha"]
print("Best alpha = %.4f" % best_alpha)
```

```
plt.plot(alphas, -searcher.cv_results_["mean_test_score"])
plt.xscale("log")
plt.xlabel("alpha")
plt.ylabel("CV score")
```

13/19

14/19

```
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
/home/sashka/python_spbu_TP23_sem4/.venv/lib/python3.12/site-packages/sklearn/linear_model/_coordinate_descent.py:656:
model = cd_fast.sparse_enet_coordinate_descent(
Best alpha = 127.4275
Text(0, 0.5, 'CV score')
```



```
column_transformer = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)

pipeline = Pipeline(steps=[("ohe_and_scaling", column_transformer), ("regression", Lasso(best_alpha))])

model = pipeline.fit(X_train, y_train)
y_pred = model.predict(X_val)
print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
```

↗ Test RMSE = 29931.7715

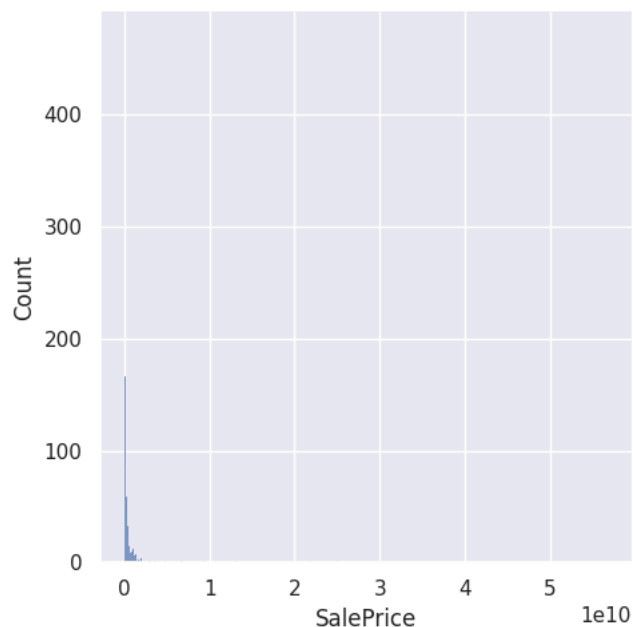
```
lasso_zeros = np.sum(pipeline.steps[-1][-1].coef_ == 0)
print("Zero weights in Lasso:", lasso_zeros)
```

↗ Zero weights in Lasso: 199

Иногда очень полезно посмотреть на распределение остатков. Нарисуем гистограмму распределения квадратичной ошибки на обучающих объектах:

```
error = (y_train - model.predict(X_train)) ** 2
sns.displot(error)
```

↗ <seaborn.axisgrid.FacetGrid at 0x7f95a1012f30>



Как видно из гистограммы, есть примеры с очень большими остатками. Попробуем их выбросить из обучающей выборки. Например, выбросим примеры, остаток у которых больше 0.95-квантили.

```
mask = error < np.quantile(error, 0.95)
```

```
column_transformer = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)

pipeline = Pipeline(steps=[("ohe_and_scaling", column_transformer), ("regression", Lasso(best_alpha))])

model = pipeline.fit(X_train[mask], y_train[mask])
y_pred = model.predict(X_val)
print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
```

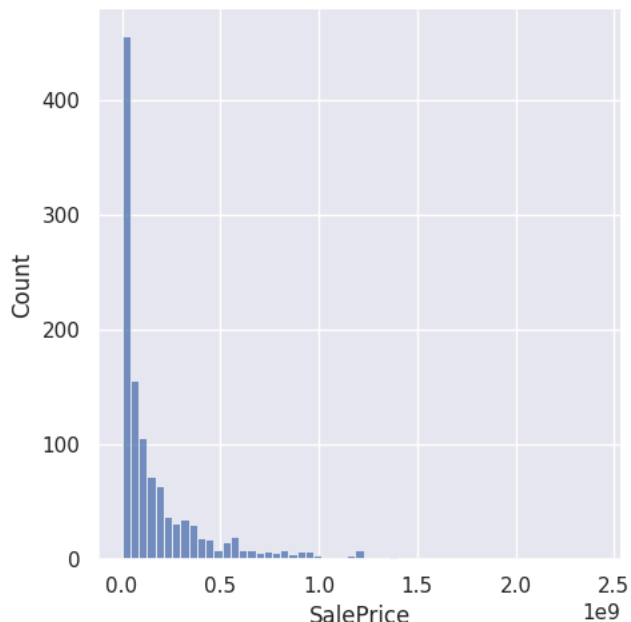
↗ Test RMSE = 29614.2239

```
X_train = X_train[mask]
y_train = y_train[mask]
```

```
error = (y_train[mask] - model.predict(X_train[mask])) ** 2
sns.displot(error)
```



```
<seaborn.axisgrid.FacetGrid at 0x7f95a0ba6660>
```



Видим, что качество модели заметно улучшилось! Также бывает очень полезно посмотреть на примеры с большими остатками и попытаться понять, почему же модель на них так сильно ошибается: это может дать понимание, как модель можно улучшить.

#### ✓ Часть 4. Подготовка данных для линейных моделей

Есть важное понятие, связанное с применением линейных моделей, — *спрямляющее пространство*. Под ним понимается такое признаковое пространство для наших объектов, в котором линейная модель хорошо описывает данные, даёт хорошее качество прогнозов.

Не существует общих рекомендаций о том, как найти спрямляющее пространство для произвольной выборки. Есть лишь некоторые общие советы — например, если добавить в выборку полиномиальных признаков, то скорее всего модель станет работать лучше (если не переобучится). Есть и другие трюки.

У линейных моделей есть огромное преимущество: они имеют мало параметров, а поэтому их можно обучить даже на небольшой выборке. Если выборка большая, то параметры модели получится оценить более надёжно — но в то же время есть риск, что данные будут слишком разнообразными, чтобы линейная модель могла уловить все закономерности в них. Иногда можно улучшить ситуацию путём разбиения признакового пространства на несколько областей и построения своей модели в каждой из них.

Попробуем для примера в нашей задаче разделить выборку на две части по признаку OverallQual. Это один из самых сильных признаков, и, возможно, разбиение по нему даст нам две выборки с заведомо разными ценами на дома.

Для начала вспомним, какое качество получается у обычной гребневой регрессии.

```
column_transformer = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)

pipeline = Pipeline(steps=[("ohe_and_scaling", column_transformer), ("regression", Ridge())])

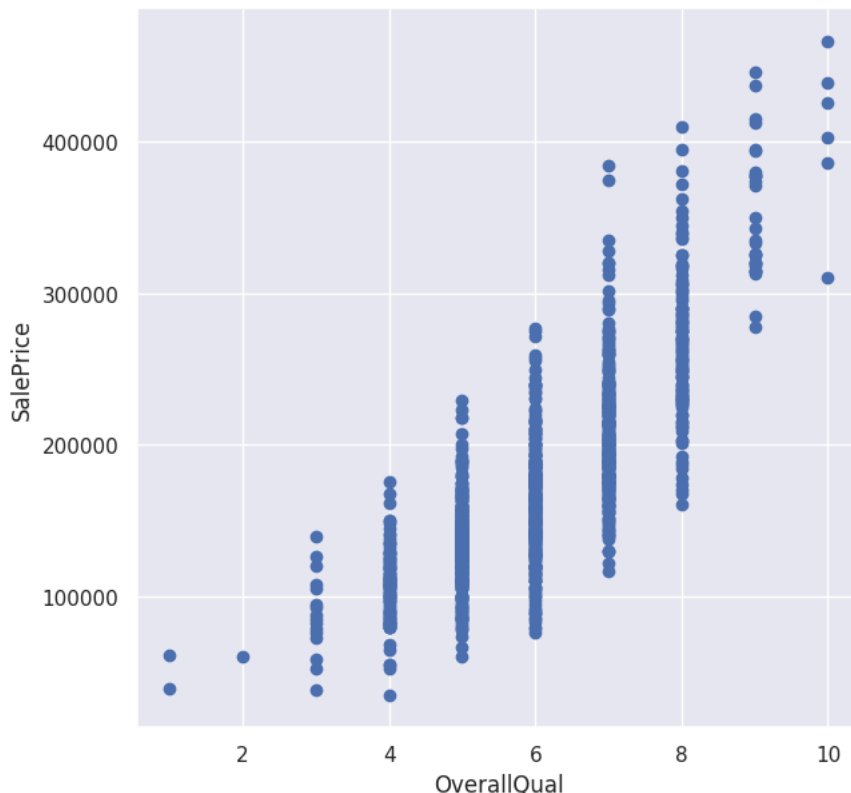
model = pipeline.fit(X_train, y_train)
y_pred = model.predict(X_val)
print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))
```

```
Test RMSE = 29142.7148
```

Посмотрим на связь OverallQual и целевой переменной.

```
plt.figure(figsize=(7, 7))
plt.scatter(X_train["OverallQual"], y_train)
plt.xlabel("OverallQual")
plt.ylabel("SalePrice")
```

↗ Text(0, 0.5, 'SalePrice')



```
threshold = 5
mask = X_train["OverallQual"] <= threshold
X_train_1 = X_train[mask]
y_train_1 = y_train[mask]
X_train_2 = X_train[~mask]
y_train_2 = y_train[~mask]

column_transformer1 = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)

pipeline1 = Pipeline(steps=[("ohe_and_scaling", column_transformer1), ("regression", Ridge())])

column_transformer2 = ColumnTransformer(
    [("ohe", OneHotEncoder(handle_unknown="ignore"), categorical), ("scaling", StandardScaler(), numeric_features)]
)

pipeline2 = Pipeline(steps=[("ohe_and_scaling", column_transformer2), ("regression", Ridge())])

model1 = pipeline1.fit(X_train_1, y_train_1)
model2 = pipeline2.fit(X_train_2, y_train_2)

y_pred_1 = model1.predict(X_val)
y_pred_2 = model2.predict(X_val)
mask_test = X_val["OverallQual"] <= threshold
y_pred = y_pred_1.copy()
y_pred[~mask_test] = y_pred_2[~mask_test]

print("Test RMSE = %.4f" % mean_squared_error(y_val, y_pred, squared=False))

↗ Test RMSE = 28363.1211
```

Получилось лучше! И это при практически случайном выборе разбиения. Если бы мы поработали над этим получше, то и качество, скорее всего, получилось бы выше.

Перейдём к следующему трюку – бинаризации признаков. Мы выбираем  $n$  порогов  $t_1, \dots, t_n$  для признака  $x_j$  и генерируем  $n + 1$  новый признак:  $[x_j \leq t_1], [t_1 < x_j \leq t_2], \dots, [t_{n-1} < x_j \leq t_n], [x_j > t_n]$ . Такое преобразование может неплохо помочь в случае, если целевая переменная нелинейно зависит от одного из признаков. Рассмотрим синтетический пример.

```
x_plot = np.linspace(0, 1, 10000)

X = np.random.uniform(0, 1, size=30)
```

```

y = np.cos(1.5 * np.pi * X) + np.random.normal(scale=0.1, size=X.shape)

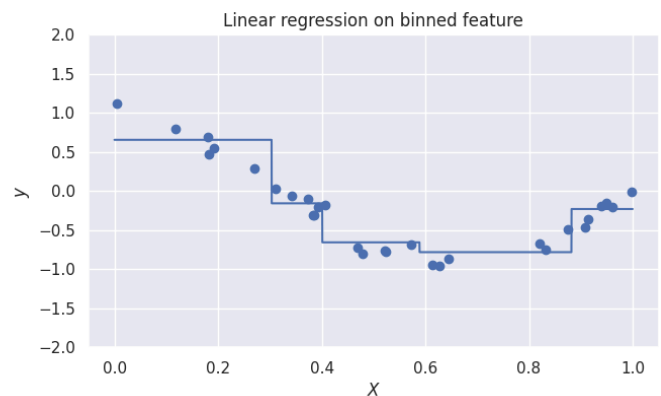
fig, axs = plt.subplots(figsize=(16, 4), ncols=2)

regr = LinearRegression()
regr.fit(X[:, np.newaxis], y)
y_pred_regr = regr.predict(x_plot[:, np.newaxis])
axs[0].scatter(X[:, np.newaxis], y, label="Data")
axs[0].plot(x_plot, y_pred_regr, label="Predictions")
axs[0].legend()
axs[0].set_title("Linear regression on original feature")
axs[0].set_xlabel("$X$")
axs[0].set_ylabel("$y$")
axs[0].set_ylim(-2, 2)

binner = KBinsDiscretizer(n_bins=5, strategy="quantile")
pipeline = Pipeline(steps=[("binning", binner), ("regression", LinearRegression())])
pipeline.fit(X[:, np.newaxis], y)
y_pred_binned = pipeline.predict(x_plot[:, np.newaxis])
axs[1].scatter(X[:, np.newaxis], y, label="Data")
axs[1].plot(x_plot, y_pred_binned, label="Predictions")
axs[1].set_title("Linear regression on binned feature")
axs[1].set_xlabel("$X$")
axs[1].set_ylabel("$y$")
axs[1].set_ylim(-2, 2)

```

↔ (-2.0, 2.0)



Видно, что качество модели существенно возросло. С другой стороны, увеличилось и количество параметров модели (из-за увеличения числа признаков), поэтому при бинаризации важно контролировать переобучение.

Иногда может помочь преобразование целевой переменной. Может оказаться, что по мере роста признаков целевая переменная меняется экспоненциально. Например, может оказаться, что при линейном уменьшении продолжительности видео число его просмотров растёт экспоненциально. Учесть это можно с помощью логарифмирования целевой переменной — ниже синтетический пример с такой ситуацией.

```

X = np.random.exponential(1, size=30)
y = np.exp(X) + np.random.normal(scale=0.1, size=X.shape)

x_plot = np.linspace(np.min(X), np.max(X), 10000)

fig, axs = plt.subplots(figsize=(16, 4), ncols=2)

regr = LinearRegression()
regr.fit(X[:, np.newaxis], y)
y_pred_regr = regr.predict(x_plot[:, np.newaxis])
axs[0].scatter(X[:, np.newaxis], y, label="Data")
axs[0].plot(x_plot, y_pred_regr, label="Predictions")
axs[0].legend()
axs[0].set_title("Linear regression on original target")
axs[0].set_xlabel("$X$")
axs[0].set_ylabel("$y$")

y_log = np.log(y)
regr.fit(X[:, np.newaxis], y_log)
y_pred_log = np.exp(regr.predict(x_plot[:, np.newaxis]))
axs[1].scatter(X[:, np.newaxis], y, label="Data")
axs[1].plot(x_plot, y_pred_log, label="Predictions")
axs[1].set_title("Linear regression on log target")

```