

sklearn и линейная регрессия

На этом семинаре мы:

- Узнаем о том, что такое переобучение, что такое линейная регрессия и как не допустить переобучения в линейных моделях;
- Обучим линейную регрессию для предсказания цены дома;
- Поймем, в чем отличие между разными регуляризаторами;
- Немного поработаем с `sklearn`.

```
In [28]: import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import warnings
warnings.simplefilter("ignore")
%matplotlib inline

sns.set(style="darkgrid")
```

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 — $\{x, \dots, x^{20}\}$.

```
In [29]: from sklearn.linear_model import LinearRegression

# Сгенерируем данные и будем это считать генеральной совокупностью
np.random.seed(36)
x = np.linspace(0, 1, 100)
y = np.cos(1.5 * np.pi * x)

# Так как в реальной жизни у нас обычно нет генеральных совокупностей, возьмем кусочек
# (это позволит нам смоделировать реальную жизнь, где мы имеем дело только с ограниченн
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, si

from sklearn.preprocessing import PolynomialFeatures # за нас сделает полиномы

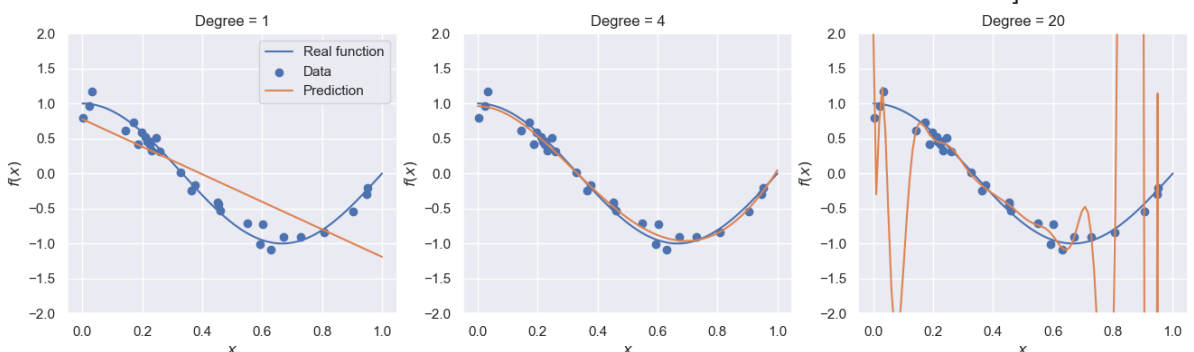
fig, axs = plt.subplots(figsize=(16, 4), ncols=3)
for i, degree in enumerate([1, 4, 20]):
    X_objects = PolynomialFeatures(degree, include_bias=False).fit_transform(x_objects)
    X = PolynomialFeatures(degree, include_bias=False).fit_transform(x[:, None])
    regr = LinearRegression().fit(X_objects, y_objects)
    y_pred = regr.predict(X)
    print(f"Для полинома {degree} степени", regr.coef_)
    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)
```

Для полинома 1 степени [-1.96635465]

Для полинома 4 степени [0.0697848 -14.40575795 18.10888186 -4.68598787]

Для полинома 20 степени [-7.10352482e+02 6.99927626e+04 -2.86707960e+06 6.26409468e+07

-8.46749684e+08 7.72357817e+09 -5.01094981e+10 2.38782706e+11
-8.50765434e+11 2.27914549e+12 -4.55144036e+12 6.55553039e+12
-6.16294972e+12 2.25687178e+12 3.16447002e+12 -6.29877800e+12
5.54630107e+12 -2.86671685e+12 8.42042605e+11 -1.09320878e+11]



Вопрос: Почему первая модель получилась плохой, а третья 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 ℓ^2 -нормы вектора w :

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

Это слагаемое называют ℓ_2 -регуляризатором, а коэффициент λ — коэффициентом регуляризации. Более подробно о регуляризации и переобучении вы узнаете из лекций.

Загружаем данные

Мы будем работать с данными из соревнования [House Prices: Advanced Regression Techniques](#), в котором требовалось предсказать стоимость жилья. Давайте сначала загрузим и немного изучим данные (можно взять `train.csv` со страницы соревнования или из нашего репозитория на github).

```
In [30]: data = pd.read_csv("train.csv")
data.head()
```

```
Out[30]:
```

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

5 rows × 81 columns

```
In [31]: data.shape
```

```
Out[31]: (1460, 81)
```

```
In [32]: data.columns
```

```
Out[32]: Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
              'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
              'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
              'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAd
d',
              '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', 'FullBat
h',
              'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
              'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageTyp
e',
              'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQua
l',
              '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.

Разделим данные на обучающую и тестовую выборки. Для простоты не будем выделять дополнительно валидационную выборку (хотя это обычно стоит делать, она нужна для подбора гиперпараметров модели, то есть параметров, которые нельзя подбирать по обучающей выборке). Дополнительно нам придется отделить значения целевой переменной от данных.

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

```
In [33]: from sklearn.model_selection import train_test_split

data = data.drop(columns=["Id"])

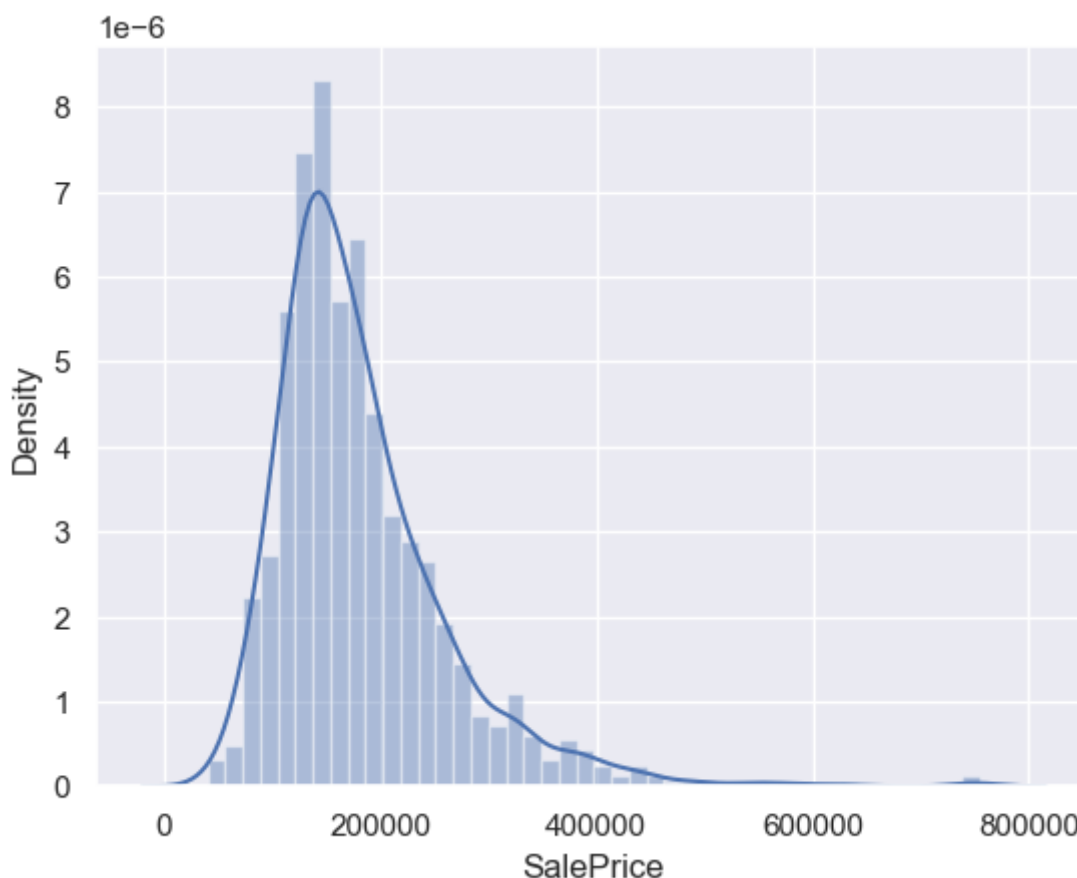
y = data["SalePrice"]
X = data.drop(columns=["SalePrice"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ran
```

Посмотрим сначала на значения целевой переменной.

```
In [34]: sns.distplot(y_train)
```

```
Out[34]: <AxesSubplot: xlabel='SalePrice', ylabel='Density'>
```



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.

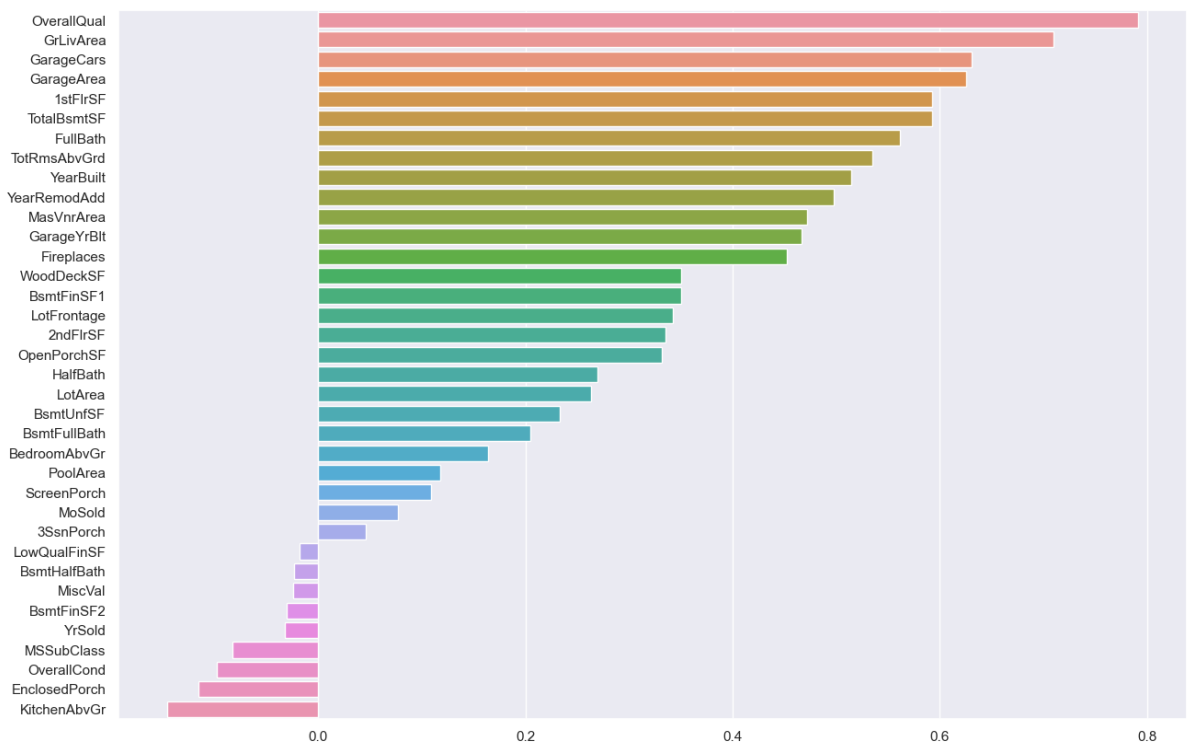
Вопрос: Что означает, что коэффициент корреляции Пирсона между двумя случайными величинами равен 1? -1? 0?

```
In [35]: numeric_data = X_train.select_dtypes([np.number])
numeric_data_mean = numeric_data.mean()
numeric_features = numeric_data.columns

X_train = X_train.fillna(numeric_data_mean)
X_test = X_test.fillna(numeric_data_mean)

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

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



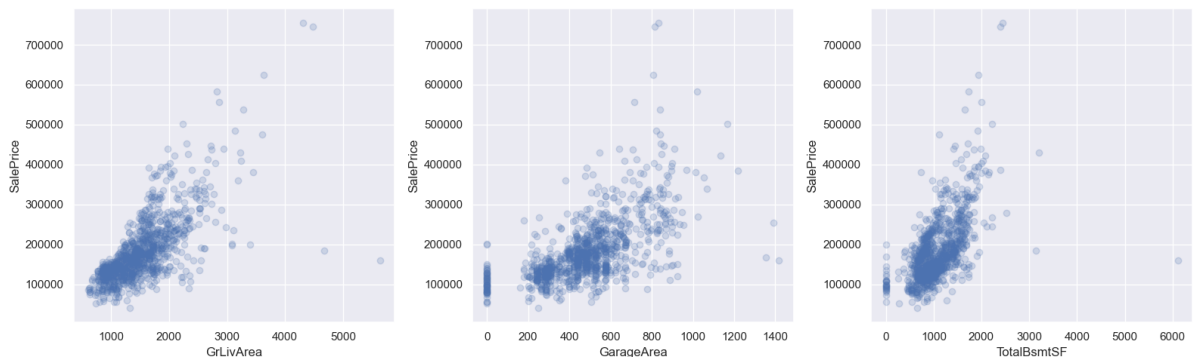
Посмотрим на признаки из начала списка. Для этого нарисуем график зависимости целевой переменной от каждого из признаков. На этом графике каждая точка соответствует паре признак-таргет (такие графики называются scatter-plot).

```
In [36]: 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.

Первая модель

Немного об обучении моделей. В арсенале дата-саентиста 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`, `CatBoost` и пр. для градиентного бустинга над деревьями, `PyTorch`, `Tensorflow` 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 ℓ_2 -регуляризацией
- `Lasso` — линейная регрессия с оптимизацией MSE и ℓ_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:

```

In [37]: from sklearn.linear_model import Ridge
         from sklearn.metrics import mean_squared_error

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

```

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

```
Test RMSE = 32085.7681
Train RMSE = 35631.3131
```

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

Вопрос: Чему равна оптимальная константа для RMSE?

```
In [42]: best_constant = y_train.mean()
print("Test RMSE with best constant = %.4f" % mean_squared_error(y_test, best_constant, squared=False))
print("Train RMSE with best constant = %.4f" % mean_squared_error(y_train, best_constant, squared=False))
```

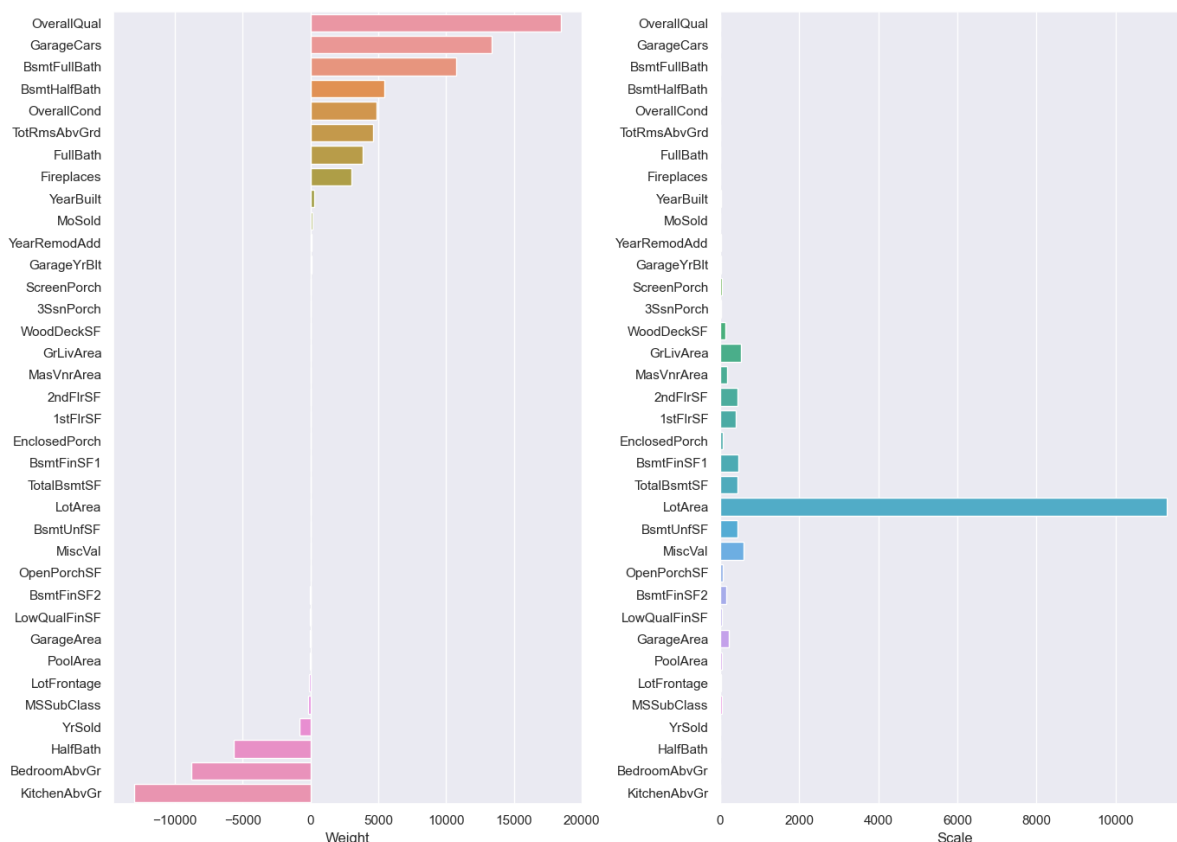
```
Test RMSE with best constant = 76636.5188
Train RMSE with best constant = 80588.6191
```

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?

```
In [39]: 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()
```

```
In [40]: 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.

```
In [44]: from sklearn.preprocessing import StandardScaler

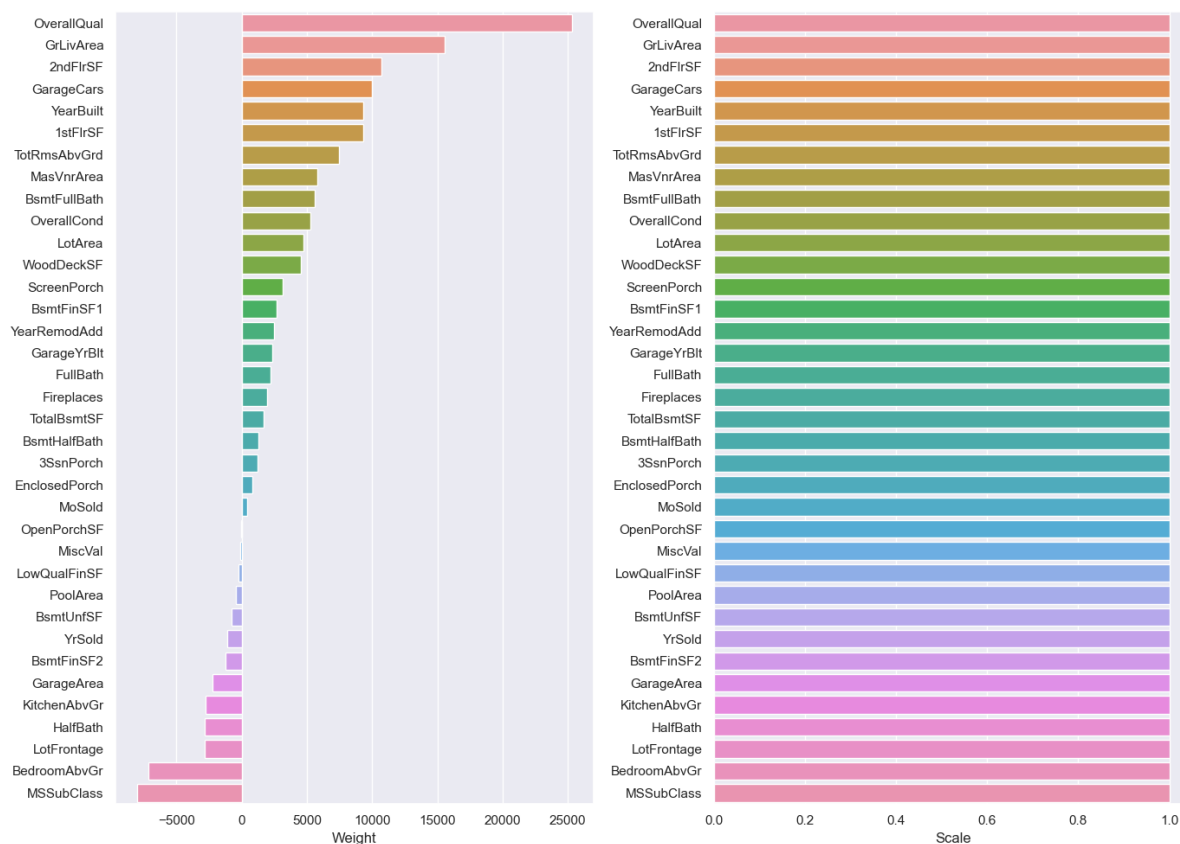
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train[numeric_features])
X_test_scaled = scaler.transform(X_test[numeric_features])

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

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

Test RMSE = 32085.3150
Train RMSE = 35631.2160
```

```
In [45]: 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 (заметьте, что мы перебираем `alpha` по логарифмической сетке, чтобы узнать оптимальный порядок величины).

```
In [46]: from sklearn.model_selection import GridSearchCV

alphas = np.logspace(-2, 3, 20)
searcher = GridSearchCV(Ridge(), [{"alpha": alphas}], scoring="neg_root_mean")
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 = 297.6351

```
Out[46]: Text(0, 0.5, 'CV score')
```

