Объяснение кода для предсказания цен на дома (House Prices)

Введение

Задача: Предсказать стоимость дома (`SalePrice`) на основе различных его характеристик (площадь, количество комнат, район, год постройки и т.д.), используя данные из датасета Kaggle "House Prices".

Инструмент: Мы будем использовать модель *Линейной Регрессии* и её вариации из библиотеки `scikit-learn` в Python. Линейная регрессия пытается найти простую математическую формулу (линию или плоскость), которая наилучшим образом связывает характеристики дома с его ценой.

Цель кода:

1. Загрузить данные.
2. Подготовить данные для модели (очистка, преобразование).
3. Обучить несколько вариантов линейной регрессии.
4. Оценить, какая модель работает лучше (хотя бы на обучающих данных).
5. Сделать предсказания для тестового набора данных (для которого цены неизвестны).
6. Сохранить предсказания в формате, готовом для отправки на Kaggle.

---

1. Загрузка данных и начальная настройка

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
# ... другие импорты sklearn ...
from sklearn.metrics import mean_squared_log_error

#%matplotlib inline # Строка для Jupyter, чтобы графики отображались в
блокноте
plt.style.use('seaborn-v0_8-darkgrid') # Стиль для графиков
pd.set_option('display.max_columns', None) # Настройка Pandas, чтобы
показывать все столбцы DataFrame

# --- 1. Загрузка данных ---
print("1. Загрузка данных...")
train_df = pd.read_csv('train_hw.csv', sep=',') # Загружаем обучающий
набор
test_df = pd.read_csv('test_hw.csv', sep=',') # Загружаем тестовый набор
(без цен)
```

```
# Сохраняем ID, они понадобятся для файла с ответами
train_ids = train_df['Id']
test_ids = test_df['Id']

# Удаляем столбец 'Id' из данных, т.к. он не несет информации для
предсказания цены
train_df = train_df.drop('Id', axis=1)
test_df = test_df.drop('Id', axis=1)

print(f"Размер трейна: {train_df.shape}") # Показывает (кол-во строк, кол-
во столбцов)
print(f"Размер теста: {test_df.shape}")
```

Объяснение:

1. **Импорты:** Мы подключаем библиотеки:
   - `pandas` (`pd`): Основной инструмент для работы с табличными данными (как Excel, но в коде).
   - `numpy` (`np`): Для математических операций, особенно с массивами чисел.
   - `matplotlib.pyplot` (`plt`) и `seaborn` (`sns`): Для построения графиков.
   - `sklearn`: Главная библиотека для машинного обучения в Python. Из неё мы берем:
     - `LinearRegression`, `RidgeCV`, `LassoCV`: Сами модели линейной регрессии (Ridge и Lasso - это вариации с регуляризацией, о них позже).
     - `StandardScaler`: Инструмент для масштабирования данных.
     - `OneHotEncoder`: Инструмент для кодирования текстовых данных.
     - `mean_squared_log_error`: Функция для оценки качества модели (близкая к той, что используется на Kaggle).
2. **Настройки:** Задаем стиль графиков и говорим Pandas показывать все столбцы при выводе таблиц (удобно для анализа).
3. **Загрузка CSV:** С помощью `pd.read_csv` читаем два файла:
   - `train_hw.csv`: Содержит характеристики домов *и* их цены (`SalePrice`). На этих данных модель будет учиться.
   - `test_hw.csv`: Содержит только характеристики домов. Для них мы должны предсказать цены.
4. **Работа с `Id`:** Уникальный идентификатор (`Id`) нужен только для того, чтобы потом правильно сопоставить наши предсказания с домами в тестовом наборе при формировании файла для Kaggle. Для самого обучения модели он бесполезен (это просто номер строки), поэтому мы сохраняем `Id` в отдельные переменные (`train_ids`, `test_ids`), а затем удаляем столбец `Id` из наших рабочих таблиц (`train_df`, `test_df`).
5. **Вывод размеров:** `df.shape` показывает количество строк и столбцов в таблице. Это помогает понять объем данных.

---

2. Подготовка данных

Это самый важный этап. Модели машинного обучения требуют данные в определенном числовом формате и без пропусков.

```
# --- 2. Подготовка данных ---
print("\n2. Подготовка данных...")

# Логарифмируем целевую переменную (SalePrice)
train_df['SalePrice'] = np.log1p(train_df['SalePrice'])
y_train_log = train_df['SalePrice'] # Сохраняем логарифмированную цену
отдельно
train_features = train_df.drop('SalePrice', axis=1) # Оставляем только
признаки для обучения
test_features = test_df.copy() # Копируем тестовые признаки

# Объединяем трейн и тест для ОДИНАКОВОЙ обработки признаков
all_features = pd.concat((train_features,
test_features)).reset_index(drop=True)
print(f"Размер объединенных данных: {all_features.shape}")
```

Объяснение:

1. **Логарифмирование `SalePrice`:**
   - **Зачем?** Цены на дома часто имеют "скошенное" распределение: много домов со средней ценой и мало очень дорогих. Линейные модели лучше работают с данными, распределение которых ближе к нормальному ("колокольчику"). Логарифмирование (`np.log1p` - это log(1+x), безопасный способ взять логарифм) сжимает большие значения сильнее, чем маленькие, делая распределение более симметричным. Кроме того, метрика оценки на Kaggle (RMSLE) по сути работает с логарифмами цен.
   - **Что делаем?** Заменяем столбец `SalePrice` на его логарифм. Сохраняем этот логарифмированный столбец в `y_train_log` (это наша цель для обучения). Из `train_df` удаляем столбец `SalePrice`, оставляя только входные признаки (`train_features`).
2. **Объединение `train` и `test`:**
   - **Зачем?** Чтобы применить все шаги предобработки (заполнение пропусков, кодирование категорий, масштабирование) *единообразно* к обоим наборам данных. Если обрабатывать их раздельно, можно получить разное количество признаков или использовать разные значения для заполнения пропусков, что сломает модель.
   - **Что делаем?** С помощью `pd.concat` "склеиваем" таблицы `train_features` и `test_features` одну под другой. `reset_index(drop=True)` обновляет индексы строк.

2.1 Обработка пропусков

В данных часто есть пропущенные значения (`NaN`). Модели не умеют с ними работать, их нужно заполнить.

```
```

```
# --- 2.1 Обработка пропусков ---
print("\n2.1 Обработка пропусков...")

# Определяем типы колонок
numeric_cols =
all_features.select_dtypes(include=np.number).columns.tolist() # Числовые
categorical_cols =
all_features.select_dtypes(include='object').columns.tolist() # Текстовые
(категориальные)

# Для некоторых категориальных фичей NaN имеет смысл 'None'
cols_fill_none = [
    'PoolQC', 'MiscFeature', 'Alley', 'Fence', 'MasVnrType',
'FireplaceQu',
    'GarageType', 'GarageFinish', 'GarageQual', 'GarageCond',
    'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'
]
for col in cols_fill_none:
    if col in all_features.columns: # Проверяем, есть ли такой столбец
        all_features[col] = all_features[col].fillna('None') # Заполняем
пропуски словом 'None'

# Оставшиеся категориальные пропуски заполняем модой (самым частым
значением)
for col in categorical_cols:
    if all_features[col].isnull().any(): # Если есть пропуски в столбце
        mode_val = all_features[col].mode()[0] # Находим самое частое
значение
        all_features[col] = all_features[col].fillna(mode_val) # Заполняем
им пропуски
        # print(f"   - Категориальный столбец '{col}' заполнен модой
('{mode_val}')")

# Числовые пропуски заполняем медианой (средним по порядку значением)
for col in numeric_cols:
    if all_features[col].isnull().any():
        median_val = all_features[col].median() # Находим медиану
        all_features[col] = all_features[col].fillna(median_val) #
Заполняем ей пропуски
        # print(f"   - Числовой столбец '{col}' заполнен медианой
({median_val:.2f})")

print(f"\nПропусков после заполнения:
{all_features.isnull().sum().sum()}") # Проверяем, что пропусков не
осталось
```

Объяснение:

1. **Разделение колонок:** Сначала определяем, какие столбцы содержат числа (`numeric_cols`), а какие - текст (`categorical_cols`). Это важно, так как к ним применяются разные стратегии

заполнения.

2. **Категории: `NaN` как 'None':** Для некоторых признаков (качество бассейна `PoolQC`, тип гаража `GarageType` и т.д.) пропуск скорее всего означает отсутствие этого объекта (нет бассейна, нет гаража). В таких случаях логично заменить `NaN` на строковое значение 'None'. Мы проходимся по списку таких колонок и используем `.fillna('None')`.
3. **Категории: `NaN` как Мода:** Для остальных текстовых колонок, где `NaN` не означает "отсутствие", мы заполняем пропуски *модой* - самым часто встречающимся значением в этом столбце. Это наиболее вероятное значение. `.mode()[0]` используется, так как мод может быть несколько, мы берем первую.
4. **Числа: `NaN` как Медиана:** Для числовых признаков (площадь, год постройки) пропуски заполняем *медианой*. Медиана - это значение, которое находится ровно посередине отсортированного списка всех значений. Она менее чувствительна к экстремальным значениям (выбросам), чем среднее арифметическое, поэтому часто предпочтительнее.
5. **Проверка:** В конце убеждаемся, что сумма всех пропусков (`all_features.isnull().sum().sum()`) равна нулю.

2.2 Логарифмирование числовых признаков с высокой асимметрией

Подобно целевой переменной `SalePrice`, некоторые числовые признаки тоже могут быть сильно "скошены". Их логарифмирование может помочь модели.

```
# --- 2.2 Логарифмирование числовых признаков с высокой асимметрией ---
print("\n2.2 Логарифмирование асимметричных числовых признаков...")
# Считаем асимметрию (skewness) для всех числовых колонок
skewness = all_features[numeric_cols].apply(lambda x:
x.skew()).sort_values(ascending=False)
# Выбираем признаки с асимметрией больше 0.75 по модулю
high_skew = skewness[abs(skewness) > 0.75]
skewed_features = high_skew.index # Получаем список названий этих
признаков

print(f"   - Найдено {len(skewed_features)} асимметричных признаков для
логарифмирования.")
# Применяем log1p к выбранным признакам
for col in skewed_features:
    all_features[col] = np.log1p(all_features[col])
```

Объяснение:

1. **Асимметрия (`skewness`):** Это мера того, насколько распределение признака несимметрично. Значение 0 означает идеальную симметрию. Большие положительные значения означают длинный "хвост" вправо (много мелких значений, мало крупных), большие отрицательные - хвост влево.
2. **Выбор признаков:** Мы вычисляем асимметрию для всех числовых признаков (`.skew()`) и отбираем те, у которых абсолютное значение асимметрии больше порога (здесь 0.75). Этот порог - эвристика, его можно подбирать.

3. **Логарифмирование:** К отобранным сильно асимметричным признакам применяем ту же функцию `np.log1p`, что и для `SalePrice`, чтобы сделать их распределение более симметричным.

2.3 Кодирование категориальных признаков

Модели машинного обучения понимают только числа. Текстовые категории (названия районов, типы крыш и т.д.) нужно преобразовать в числовой вид.

```
# --- 2.3 Кодирование категориальных признаков (One-Hot Encoding) ---
print("\n2.3 Кодирование категориальных признаков (One-Hot Encoding)...")
# Применяем One-Hot Encoding ко всем категориальным столбцам
all_features_encoded = pd.get_dummies(all_features,
columns=categorical_cols, drop_first=True, dtype=int)
print(f"   - Размер данных после OHE: {all_features_encoded.shape}")
```

Объяснение:

1. **One-Hot Encoding (OHE):** Это самый распространенный способ кодирования категорий. Для каждой категории в исходном столбце создается новый столбец (dummy-переменная). В этом новом столбце ставится 1, если объект принадлежит к этой категории, и 0 - если нет.
    - *Пример:* Если столбец `ТипКрыши` имел значения `['Плоская', 'Скатная', 'Мансарда']`, OHE создаст два новых столбца (почему два, см. `drop_first`): `ТипКрыши_Скатная` и `ТипКрыши_Мансарда`.
        - Для дома с плоской крышей оба столбца будут 0.
        - Для дома со скатной крышей `ТипКрыши_Скатная` будет 1, `ТипКрыши_Мансарда` будет 0.
        - Для дома с мансардой `ТипКрыши_Скатная` будет 0, `ТипКрыши_Мансарда` будет 1.
2. **`pd.get_dummies`:** Функция Pandas, которая делает OHE.
    - `columns=categorical_cols`: Указываем, какие столбцы кодировать.
    - `drop_first=True`: Удаляет первую категорию из каждого признака. Это делается для избежания *мультиколлинеарности* (когда один признак можно выразить через другие). Если мы знаем значения для всех категорий кроме одной, значение последней уже предопределено. Модели не любят такую избыточность.
    - `dtype=int`: Задает тип данных для новых столбцов (0 и 1) как целые числа.
3. **Результат:** Количество столбцов в таблице значительно увеличивается, но теперь вся информация представлена в числовом виде.

2.4 Разделение обратно на трейн и тест

Теперь, когда вся обработка завершена, нужно разделить объединенную таблицу обратно на обучающий и тестовый наборы.

```
```

```
# --- 2.4 Разделение обратно на трейн и тест ---
# Используем исходное количество строк в train_df для разделения
X_train_processed = all_features_encoded[:len(train_df)]
X_test_processed = all_features_encoded[len(train_df):]

print(f"\nРазмер обработанного трейна: {X_train_processed.shape}")
print(f"Размер обработанного теста: {X_test_processed.shape}")
```

Объяснение:

- Мы знаем, сколько строк было в исходном `train_df`. Используя срезы Python (`[:N]` - взять первые N строк, `[N:]` - взять строки начиная с N-й), мы разделяем `all_features_encoded` на `X_train_processed` (обработанные признаки для обучения) и `X_test_processed` (обработанные признаки для предсказания).

2.5 Масштабирование числовых признаков

Признаки могут иметь разный масштаб (например, площадь дома измеряется тысячами, а количество комнат - единицами). Линейные модели (особенно с регуляризацией) работают лучше, когда все признаки приведены к одному масштабу.

```
# --- 2.5 Масштабирование числовых признаков (StandardScaler) ---
print("\n2.5 Масштабирование признаков (StandardScaler)...")
scaler = StandardScaler() # Создаем объект StandardScaler
# Обучаем scaler на ТРЕНИРОВОЧНЫХ данных и сразу применяем масштабирование
X_train_scaled = scaler.fit_transform(X_train_processed)
# Применяем ТО ЖЕ САМОЕ масштабирование к ТЕСТОВЫМ данным (НЕ обучаем
заново!)
X_test_scaled = scaler.transform(X_test_processed)

# Преобразуем обратно в DataFrame для удобства (не обязательно для модели)
X_train_scaled_df = pd.DataFrame(X_train_scaled,
index=X_train_processed.index, columns=X_train_processed.columns)
X_test_scaled_df = pd.DataFrame(X_test_scaled,
index=X_test_processed.index, columns=X_test_processed.columns)
```

Объяснение:

1. **StandardScaler:** Этот метод масштабирует данные так, чтобы у каждого признака среднее значение стало равно 0, а стандартное отклонение - 1.
2. `fit_transform` **на трейне:**
    - `fit`: Scaler "изучает" среднее и стандартное отклонение *только* на обучающих данных (`X_train_processed`). Это важно, чтобы информация из тестовых данных не "протекла" в процесс обучения.
    - `transform`: Scaler применяет выученное преобразование к данным. `fit_transform` делает оба шага сразу.

3. **`transform` на тесте:** К тестовым данным (`X_test_processed`) мы применяем *только* `transform`, используя параметры (среднее и стд. отклонение), выученные на *трейне*. Это гарантирует, что и трейн, и тест масштабируются одинаково. **Никогда нельзя делать `fit` или `fit_transform` на тестовых данных!**
4. **Результат:** Теперь все признаки (и исходные числовые, и созданные OHE) имеют схожий масштаб, что помогает модели правильно взвесить их вклад.

---

3. Обучение моделей

Теперь, когда данные готовы, можно обучать модели.

```
# --- 3. Обучение моделей ---
print("\n3. Обучение моделей...")

# --- 3.1 Простая Линейная Регрессия ---
print("\n   - Обучение LinearRegression...")
lr = LinearRegression() # Создаем модель
lr.fit(X_train_scaled, y_train_log) # Обучаем на масштабированных
признаках и лог. цене

# --- 3.2 Ridge Регрессия (L2 регуляризация) ---
print("   - Обучение RidgeCV...")
# Список возможных значений для параметра регуляризации alpha
alphas_ridge = [0.01, 0.1, 1.0, 5.0, 10.0, 20.0, 50.0, 100.0]
# RidgeCV автоматически подберет лучший alpha с помощью кросс-валидации
(cv=5)
ridge_cv = RidgeCV(alphas=alphas_ridge, cv=5)
ridge_cv.fit(X_train_scaled, y_train_log) # Обучаем
print(f"     - Лучший alpha для Ridge: {ridge_cv.alpha_}") # Выводим
найденный лучший параметр

# --- 3.3 Lasso Регрессия (L1 регуляризация) ---
print("   - Обучение LassoCV...")
alphas_lasso = [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1] # Lasso
обычно требует меньшие alpha
# LassoCV также подбирает alpha через кросс-валидацию
lasso_cv = LassoCV(alphas=alphas_lasso, cv=5, max_iter=5000,
random_state=42)
lasso_cv.fit(X_train_scaled, y_train_log) # Обучаем
print(f"     - Лучший alpha для Lasso: {lasso_cv.alpha_}")
```

Объяснение:

1. **Модель:** Мы создаем объект модели (например, `lr = LinearRegression()`).
2. **Обучение (`fit`):** Метод `.fit(X, y)` "учит" модель находить зависимость между признаками `X` (`X_train_scaled`) и целевой переменной `y` (`y_train_log`). Модель подбирает внутренние параметры (коэффициенты для каждого признака), чтобы минимизировать ошибку предсказания на обучающих данных.

3. **Линейная регрессия (`LinearRegression`):** Самая простая форма. Может быть склонна к *переобучению* (overfitting) на данных с большим количеством признаков, т.е. слишком хорошо запоминать обучающие данные и плохо работать на новых.
4. **Ridge и Lasso Регрессия:** Это улучшенные версии линейной регрессии с *регуляризацией*. Регуляризация - это техника, которая штрафует модель за слишком большие коэффициенты при признаках. Это помогает предотвратить переобучение и делает модель более устойчивой к новым данным.
    - **Ridge (L2):** Штрафует за сумму квадратов коэффициентов. Склонен уменьшать коэффициенты, но редко делает их ровно нулём.
    - **Lasso (L1):** Штрафует за сумму абсолютных значений коэффициентов. Может "обнулять" коэффициенты для неважных признаков, тем самым выполняя автоматический отбор признаков.
5. **`RidgeCV` и `LassoCV`:** Суффикс `CV` означает Cross-Validation. Эти версии моделей автоматически перебирают несколько значений параметра регуляризации (`alpha` - чем он больше, тем сильнее штраф) и с помощью кросс-валидации (деления обучающих данных на части и проверки на них) выбирают тот `alpha`, который дает наилучшее качество. Это избавляет от необходимости подбирать `alpha` вручную. `cv=5` означает 5-кратную кросс-валидацию.

---

4. Оценка моделей (на трейне)

Посмотрим, насколько хорошо модели справились с предсказанием на тех же данных, на которых обучались. **Важно:** это не показатель того, как модель будет работать на *новых* данных, но позволяет сравнить модели между собой.

```
# --- 4. Оценка моделей (на трейне) ---
print("\n4. Оценка моделей (MSLE на трейне)...")

# Функция для расчета RMSLE (через MSLE)
def rmsle(y_true_log, y_pred_log):
    # Сначала преобразуем логи обратно в цены с помощью expm1 (exp(x) - 1)
    y_true = np.expm1(y_true_log)
    y_pred = np.expm1(y_pred_log)
    # Считаем Mean Squared Log Error
    return mean_squared_log_error(y_true, y_pred)

# Делаем предсказания на ТРЕНИРОВОЧНЫХ данных для каждой модели
pred_lr_log = lr.predict(X_train_scaled)
pred_ridge_log = ridge_cv.predict(X_train_scaled)
pred_lasso_log = lasso_cv.predict(X_train_scaled)

# Считаем ошибку для каждой модели
msle_lr = rmsle(y_train_log, pred_lr_log)
msle_ridge = rmsle(y_train_log, pred_ridge_log)
msle_lasso = rmsle(y_train_log, pred_lasso_log)

# Выводим ошибки (MSLE и RMSLE - корень из MSLE)
```

```
print(f"   - Linear Regression MSLE (train): {msle_lr:.5f} (RMSLE:
{np.sqrt(msle_lr):.5f})")
print(f"   - RidgeCV MSLE (train):           {msle_ridge:.5f} (RMSLE:
{np.sqrt(msle_ridge):.5f})")
print(f"   - LassoCV MSLE (train):           {msle_lasso:.5f} (RMSLE:
{np.sqrt(msle_lasso):.5f})")

# Выбираем лучшую модель (здесь просто выбираем Ridge как часто надежный
вариант)
best_model = ridge_cv
print(f"\n   - Выбрана модель: RidgeCV (alpha={best_model.alpha_})")
```

Объяснение:

1. **Предсказание на трейне (`predict`):** Используем обученные модели, чтобы сделать предсказания для `X_train_scaled`. Результат (`pred_..._log`) будет в логарифмической шкале.
2. **Метрика RMSLE/MSLE:**
   - `mean_squared_log_error` (MSLE) из `sklearn` вычисляет среднее значение квадратов разностей логарифмов: `mean( (log(pred+1) - log(true+1))^2 )`.
   - RMSLE (Root Mean Squared Log Error), которая используется на Kaggle, это просто квадратный корень из MSLE: `sqrt(MSLE)`.
   - Наша функция `rmsle` принимает на вход логарифмы (`y_true_log`, `y_pred_log`), преобразует их обратно в исходную шкалу цен с помощью `np.expm1` (операция, обратная `np.log1p`), а затем использует `mean_squared_log_error`. Это позволяет получить оценку, близкую к Kaggle.
3. **Сравнение:** Сравниваем значения MSLE (или RMSLE). Чем меньше ошибка, тем лучше модель справилась *на обучающих данных*. Модели с регуляризацией (Ridge, Lasso) часто показывают чуть большую ошибку на трейне, чем простая линейная регрессия (потому что регуляризация не дает им идеально подстроиться под трейн), но при этом лучше работают на новых данных.
4. **Выбор модели:** На основе ошибок (или опыта) выбираем лучшую модель для финальных предсказаний. Здесь выбран `RidgeCV`, так как он часто дает хороший баланс между качеством и стабильностью.

---

5. Создание предсказаний для Kaggle

Используем выбранную лучшую модель для предсказания цен на тестовом наборе данных и сохраняем результат в нужном формате.

```
# --- 5. Создание предсказаний для Kaggle ---
print("\n5. Создание предсказаний для Kaggle...")
# Делаем предсказания на обработанных и масштабированных ТЕСТОВЫХ данных
test_predictions_log = best_model.predict(X_test_scaled)

# Преобразуем логарифмические предсказания обратно в обычные цены
test_predictions = np.expm1(test_predictions_log)
```

```
# На всякий случай, заменяем возможные отрицательные предсказания на 0
test_predictions[test_predictions < 0] = 0

# Создаем DataFrame для отправки
submission_df = pd.DataFrame({
    'Id': test_ids, # Берем сохраненные ранее ID тестовых домов
    'SalePrice': test_predictions # Добавляем предсказанные цены
})

# Сохраняем в CSV файл без индекса строки
submission_filename = 'submission_simplified_ridge.csv'
submission_df.to_csv(submission_filename, index=False)

print(f"   - Файл предсказаний '{submission_filename}' успешно сохранен.")
print(f"   - Пример предсказаний:\n{submission_df.head()}") # Показываем
первые 5 строк файла
```

Объяснение:

1. **Предсказание на тесте:** Используем `best_model.predict()` на подготовленных тестовых данных `X_test_scaled`. Получаем `test_predictions_log`.
2. **Обратное преобразование:** Применяем `np.expm1()` к логарифмическим предсказаниям, чтобы получить цены в их исходном масштабе (`test_predictions`).
3. **Очистка:** Хотя маловероятно, но на всякий случай проверяем, нет ли отрицательных цен, и заменяем их на 0.
4. **Формирование файла:** Создаем `pandas` DataFrame с двумя колонками:
    - `Id`: Используем `test_ids`, которые мы сохранили в самом начале.
    - `SalePrice`: Используем полученные `test_predictions`.
5. **Сохранение:** С помощью `.to_csv()` сохраняем DataFrame в файл. `index=False` указывает, что не нужно записывать индекс строк DataFrame в файл (Kaggle ожидает только колонки `Id` и `SalePrice`).

Дополнительно: Важность признаков (Lasso)

Если в качестве лучшей модели была выбрана Lasso, можно посмотреть, какие признаки она посчитала важными (т.е. присвоила им ненулевые коэффициенты).

```
# --- Дополнительно: Визуализация важности признаков (для Lasso) ---
if best_model == lasso_cv: # Если лучшая модель - Lasso
    print("\nДополнительно: Важность признаков (Lasso)")
    # Получаем коэффициенты модели для каждого признака
    lasso_coefs = pd.Series(lasso_cv.coef_,
index=X_train_processed.columns)
    # Отбираем только те признаки, у которых коэффициент не равен 0
    important_coefs = lasso_coefs[lasso_coefs != 0].sort_values()
    if not important_coefs.empty: # Если есть ненулевые коэффициенты
        plt.figure(figsize=(10, max(6, len(important_coefs) // 4))) #
```

```
Задаем размер графика
        important_coefs.plot(kind='barh') # Строим горизонтальную
гистограмму
        plt.title(f'Lasso Coefficients (alpha={lasso_cv.alpha_:.5f}) -
{len(important_coefs)} non-zero features')
        plt.xlabel('Coefficient Value')
        plt.ylabel('Feature')
        plt.tight_layout() # Улучшаем расположение элементов графика
        plt.show() # Показываем график
    else:
        print("   - Lasso обнулил все коэффициенты.")
```

Объяснение:

1. **Проверка:** Условие `if best_model == lasso_cv` выполняется, только если мы ранее выбрали Lasso как лучшую модель.
2. **Коэффициенты:** `lasso_cv.coef_` содержит массив коэффициентов, которые модель присвоила каждому признаку после обучения. `pd.Series(...)` создает удобный объект, где индексами служат названия признаков.
3. **Отбор:** `lasso_coefs[lasso_coefs != 0]` выбирает только те коэффициенты (и соответствующие им признаки), которые не равны нулю. Lasso "зануляет" неважные признаки. `.sort_values()` сортирует их по величине коэффициента.
4. **Визуализация:** Строится горизонтальная столбчатая диаграмма (`barh`), показывающая величину и знак коэффициента для каждого "важного" признака. Это дает представление о том, какие характеристики дома модель считает наиболее влияющими на цену (и в какую сторону - положительный коэффициент увеличивает цену, отрицательный уменьшает).