House Appraisal & Price Prediction

Project’s Outlines

01 Introduction

02 EDA and Preprocessing

03 House Price Predicting (Supervised Learning)

04 House Clustering and SalePrice Predicting (Unsupervised Learning)

About

Source: **https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/overview**

Objective:
With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this dataset requires us to predict the final price of each home.

Expected results:

	Id	SalePrice
0	1461	123554.718750
1	1462	153939.062500
2	1463	176793.765625
3	1464	183828.296875
4	1465	193644.484375

Introduction

Features Description

1. SalePrice - the property's sale price in dollars.
2. MSSubClass: The building class
3. MSZoning: The general zoning classification
4. LotFrontage: Linear feet of street connected to property
5. LotArea: Lot size in square feet
6. Street: Type of road access
7. Alley: Type of alley access
8. LotShape: General shape of property
9. LandContour: Flatness of the property
10. Utilities: Type of utilities available
11. LotConfig: Lot configuration
12. LandSlope: Slope of property
13. Neighborhood: Physical locations within Ames city limits
14. Condition1: Proximity to main road or railroad
15. Condition2: Proximity to main road or railroad (if a second is present)
16. BldgType: Type of dwelling
17. HouseStyle: Style of dwelling
18. OverallQual: Overall material and finish quality
19. OverallCond: Overall condition rating
20. YearBuilt: Original construction date
21. YearRemodAdd: Remodel date
22. RoofStyle: Type of roof
23. RoofMatl: Roof material
24. Exterior1st: Exterior covering on house
25. Exterior2nd: Exterior covering on house (if more than one material)
26. MasVnrType: Masonry veneer type
27. MasVnrArea: Masonry veneer area in square feet
28. ExterQual: Exterior material quality
29. ExterCond: Present condition of the material on the exterior
30. Foundation: Type of foundation
31. BsmtQual: Height of the basement
32. BsmtCond: General condition of the basement
33. BsmtExposure: Walkout or garden level basement walls
34. BsmtFinType1: Quality of basement finished area
35. BsmtFinSF1: Type 1 finished square feet
36. BsmtFinType2: Quality of second finished area (if present)
37. BsmtFinSF2: Type 2 finished square feet
38. BsmtUnfSF: Unfinished square feet of basement area
39. TotalBsmtSF: Total square feet of basement area
40. Heating: Type of heating
41. HeatingQC: Heating quality and condition
42. CentralAir: Central air conditioning
43. Electrical: Electrical system
44. 1stFlrSF: First Floor square feet
45. 2ndFlrSF: Second floor square feet
46. LowQualFinSF: Low quality finished square feet (all floors)
47. GrLivArea: Above grade (ground) living area square feet
48. BsmtFullBath: Basement full bathrooms
49. BsmtHalfBath: Basement half bathrooms
50. FullBath: Full bathrooms above grade
51. HalfBath: Half baths above grade
52. Bedroom: Number of bedrooms above basement level
53. Kitchen: Number of kitchens
54. KitchenQual: Kitchen quality
55. TotRmsAbvGrd: Total rooms above grade (does not include bathrooms)
56. Functional: Home functionality rating
57. Fireplaces: Number of fireplaces
58. FireplaceQu: Fireplace quality
59. GarageType: Garage location
60. GarageYrBlt: Year garage was built
61. GarageFinish: Interior finish of the garage
62. GarageCars: Size of garage in car capacity
63. GarageArea: Size of garage in square feet
64. GarageQual: Garage quality
65. GarageCond: Garage condition
66. PavedDrive: Paved driveway
67. WoodDeckSF: Wood deck area in square feet
68. OpenPorchSF: Open porch area in square feet
69. EnclosedPorch: Enclosed porch area in square feet
70. 3SsnPorch: Three season porch area in square feet
71. ScreenPorch: Screen porch area in square feet
72. PoolArea: Pool area in square feet
73. PoolQC: Pool quality
74. Fence: Fence quality
75. MiscFeature: Miscellaneous feature not covered in other categories
76. MiscVal: $Value of miscellaneous feature
77. MoSold: Month Sold
78. YrSold: Year Sold
79. SaleType: Type of sale
80. SaleCondition: Condition of sale

Original data is split into:

- Train set (81 features, with House ID), and
- Test set (80 features, with House ID)

The train dataset has the column SalePrice with the objective checking validation. This column doesn't exist in the test dataset and that's our ultimate goal → ***predict SalePrice for each House ID in the test set.***

EDA

Descriptive Statistics

	count	mean	std	min	25%	50%	75%	max
Id	1460.0	730.500000	421.610009	1.0	365.75	730.5	1095.25	1460.0
MSSubClass	1460.0	56.897260	42.300571	20.0	20.00	50.0	70.00	190.0
LotFrontage	1201.0	70.049958	24.284752	21.0	59.00	69.0	80.00	313.0
LotArea	1460.0	10516.828082	9981.264932	1300.0	7553.50	9478.5	11601.50	215245.0
OverallQual	1460.0	6.099315	1.382997	1.0	5.00	6.0	7.00	10.0
OverallCond	1460.0	5.575342	1.112799	1.0	5.00	5.0	6.00	9.0
YearBuilt	1460.0	1971.267808	30.202904	1872.0	1954.00	1973.0	2000.00	2010.0
YearRemodAdd	1460.0	1984.865753	20.645407	1950.0	1967.00	1994.0	2004.00	2010.0
MasVnrArea	1452.0	103.685262	181.066207	0.0	0.00	0.0	166.00	1600.0
BsmtFinSF1	1460.0	443.639726	456.098091	0.0	0.00	383.5	712.25	5644.0
BsmtFinSF2	1460.0	46.549315	161.319273	0.0	0.00	0.0	0.00	1474.0
BsmtUnfSF	1460.0	567.240411	441.866955	0.0	223.00	477.5	808.00	2336.0
TotalBsmtSF	1460.0	1057.429452	438.705324	0.0	795.75	991.5	1298.25	6110.0
1stFlrSF	1460.0	1162.626712	386.587738	334.0	882.00	1087.0	1391.25	4692.0
2ndFlrSF	1460.0	346.992466	436.528436	0.0	0.00	0.0	728.00	2065.0
LowQualFinSF	1460.0	5.844521	48.623081	0.0	0.00	0.0	0.00	572.0
GrLivArea	1460.0	1515.463699	525.480383	334.0	1129.50	1464.0	1776.75	5642.0
BsmtFullBath	1460.0	0.425342	0.518911	0.0	0.00	0.0	1.00	3.0
BsmtHalfBath	1460.0	0.057534	0.238753	0.0	0.00	0.0	0.00	2.0
FullBath	1460.0	1.565068	0.550916	0.0	1.00	2.0	2.00	3.0
HalfBath	1460.0	0.382877	0.502885	0.0	0.00	0.0	1.00	2.0

	count	mean	std	min	25%	50%	75%	max
BedroomAbvGr	1460.0	2.866438	0.815778	0.0	2.00	3.0	3.00	8.0
KitchenAbvGr	1460.0	1.046575	0.220338	0.0	1.00	1.0	1.00	3.0
TotRmsAbvGrd	1460.0	6.517808	1.625393	2.0	5.00	6.0	7.00	14.0
Fireplaces	1460.0	0.613014	0.644666	0.0	0.00	1.0	1.00	3.0
GarageYrBlt	1379.0	1978.506164	24.689725	1900.0	1961.00	1980.0	2002.00	2010.0
GarageCars	1460.0	1.767123	0.747315	0.0	1.00	2.0	2.00	4.0
GarageArea	1460.0	472.980137	213.804841	0.0	334.50	480.0	576.00	1418.0
WoodDeckSF	1460.0	94.244521	125.338794	0.0	0.00	0.0	168.00	857.0
OpenPorchSF	1460.0	46.660274	66.256028	0.0	0.00	25.0	68.00	547.0
EnclosedPorch	1460.0	21.954110	61.119149	0.0	0.00	0.0	0.00	552.0
3SsnPorch	1460.0	3.409589	29.317331	0.0	0.00	0.0	0.00	508.0
ScreenPorch	1460.0	15.060959	55.757415	0.0	0.00	0.0	0.00	480.0
PoolArea	1460.0	2.758904	40.177307	0.0	0.00	0.0	0.00	738.0
MiscVal	1460.0	43.489041	496.123024	0.0	0.00	0.0	0.00	15500.0
MoSold	1460.0	6.321918	2.703626	1.0	5.00	6.0	8.00	12.0
YrSold	1460.0	2007.815753	1.328095	2006.0	2007.00	2008.0	2009.00	2010.0
SalePrice	1460.0	180921.195890	79442.502883	34900.0	129975.00	163000.0	214000.00	755000.0

EDA

Classify Columns

```
num_cols = [c for c in train.select_dtypes(exclude='object').columns if c != 'SalePrice' and c != 'Id']
print('Numerical columns :' ,num_cols ,"Number of numerical columns:" ,len(num_cols))

cat_cols = train.select_dtypes(include= 'object').columns
print('\nCategorical columns :' ,cat_cols,"\nNumber of categorical columns:" ,len(cat_cols) )
```

EDA

Classify Columns

```
num_cols = [c for c in train.select_dtypes(exclude='object').columns if c != 'SalePrice' and c != 'Id']
print('Numerical columns :' ,num_cols ,"Number of numerical columns:" ,len(num_cols))

cat_cols = train.select_dtypes(include= 'object').columns
print('\nCategorical columns :' ,cat_cols,"\nNumber of categorical columns:" ,len(cat_cols) )
```

Numerical columns:

- 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold’
- Number of numerical columns: **36**

EDA

Classify Columns

```
num_cols = [c for c in train.select_dtypes(exclude='object').columns if c != 'SalePrice' and c != 'Id']
print('Numerical columns :' ,num_cols ,"Number of numerical columns:" ,len(num_cols))

cat_cols = train.select_dtypes(include= 'object').columns
print('\nCategorical columns :' ,cat_cols,"\nNumber of categorical columns:" ,len(cat_cols) )
```

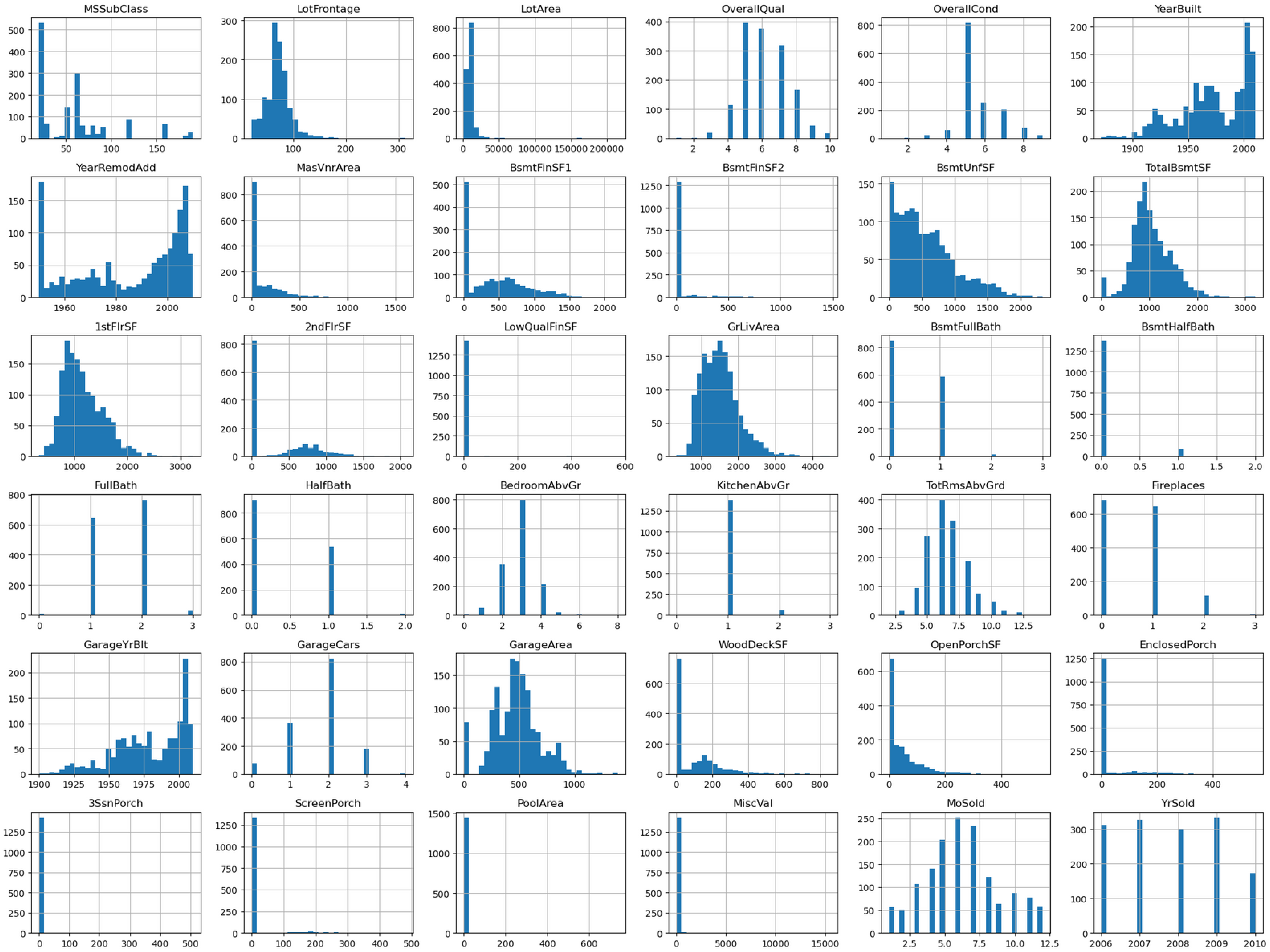
Numerical columns:

- 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold'
- Number of numerical columns: **36**

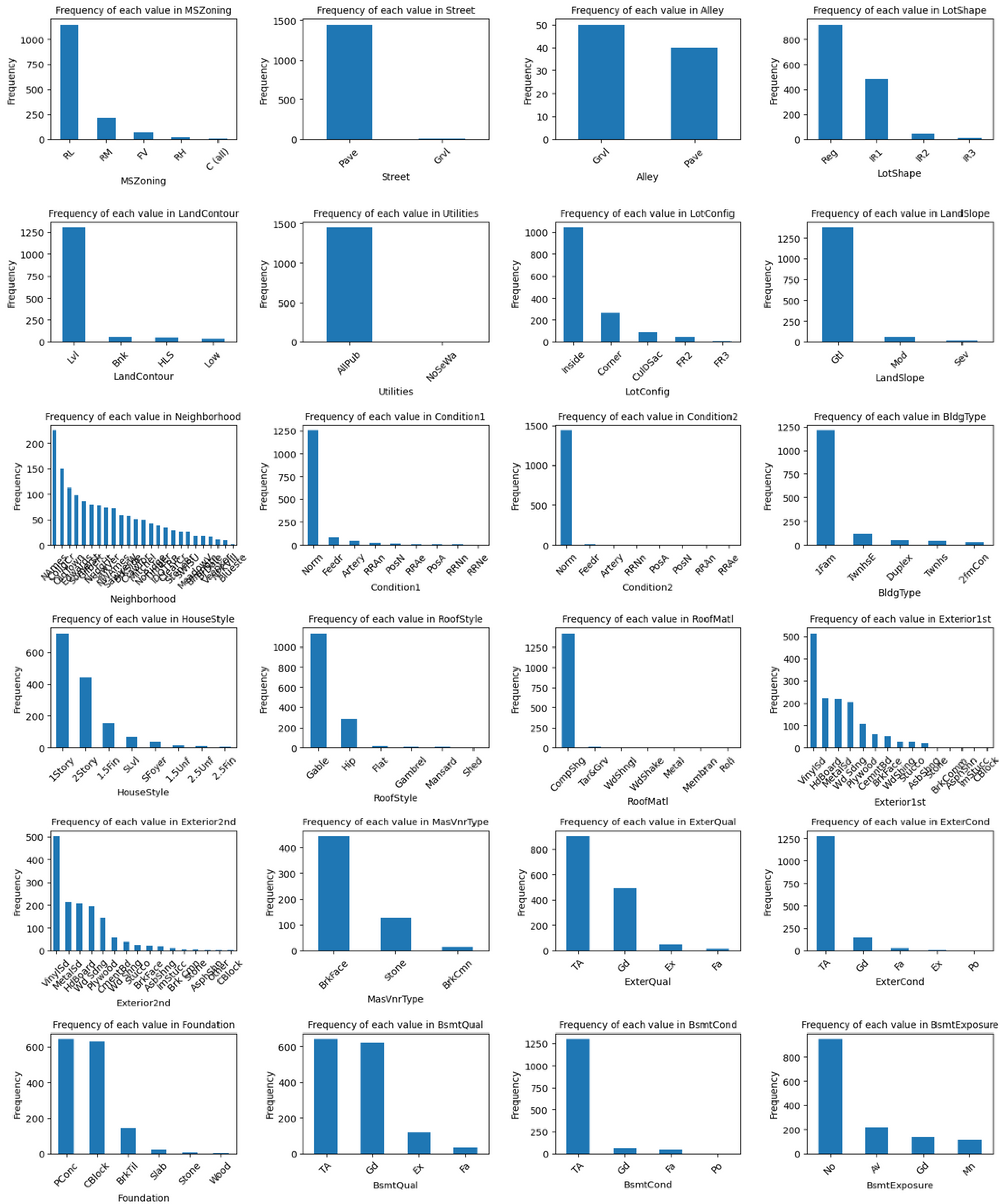
Categorical columns:

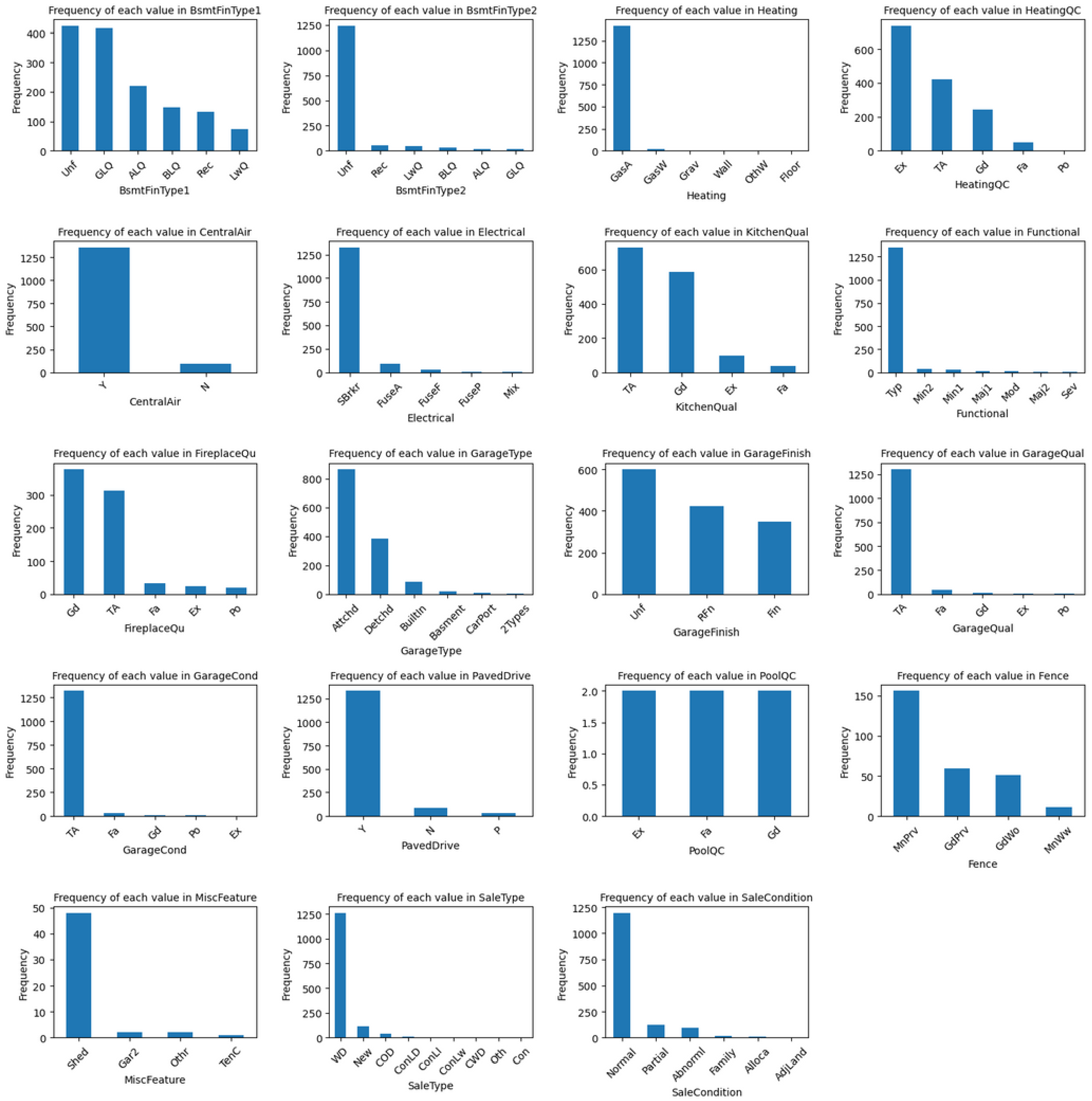
- 'MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual', 'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual', 'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature', 'SaleType', 'SaleCondition'
- Number of categorical columns: **43**

EDA Numerical Features



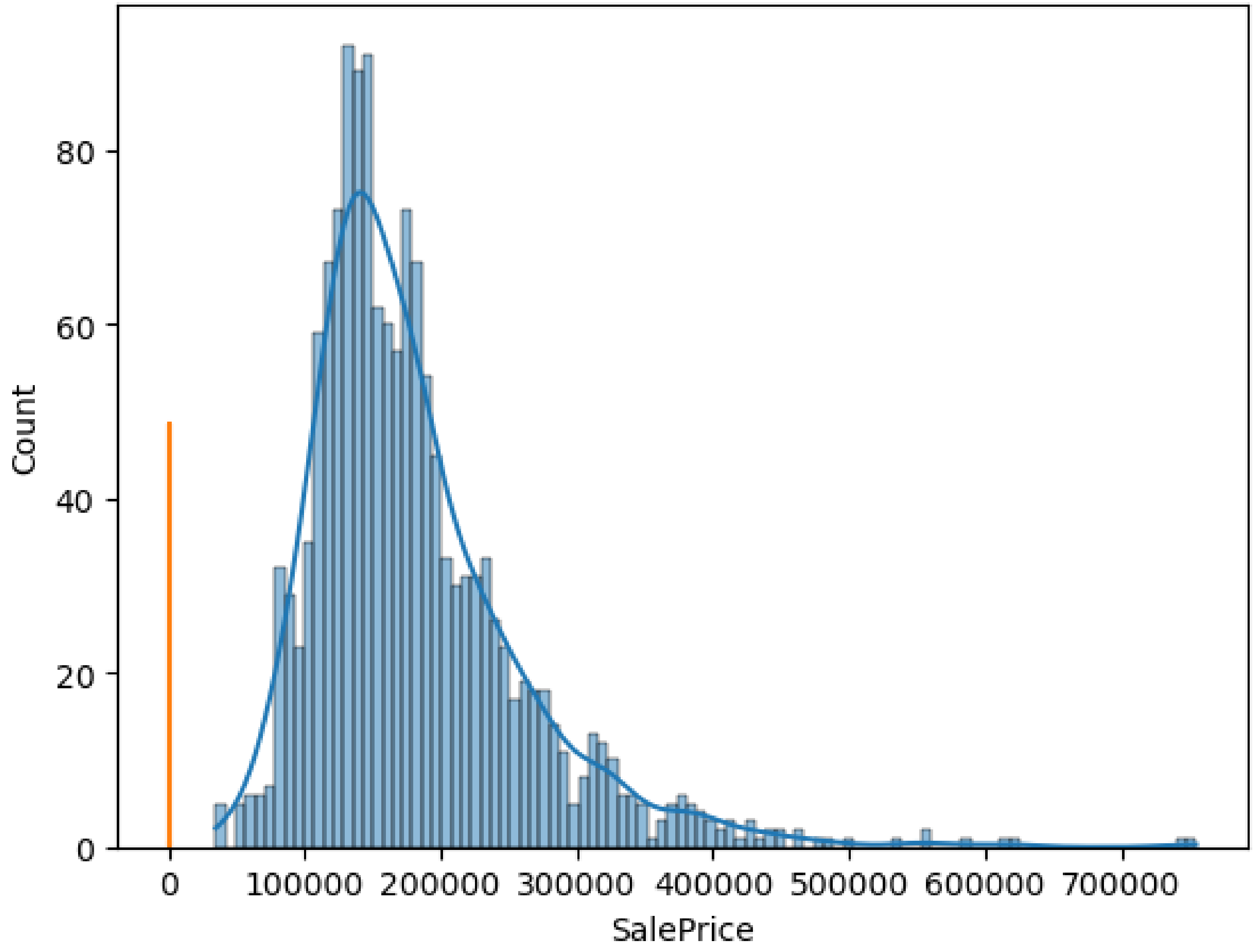
EDA Categorical Features



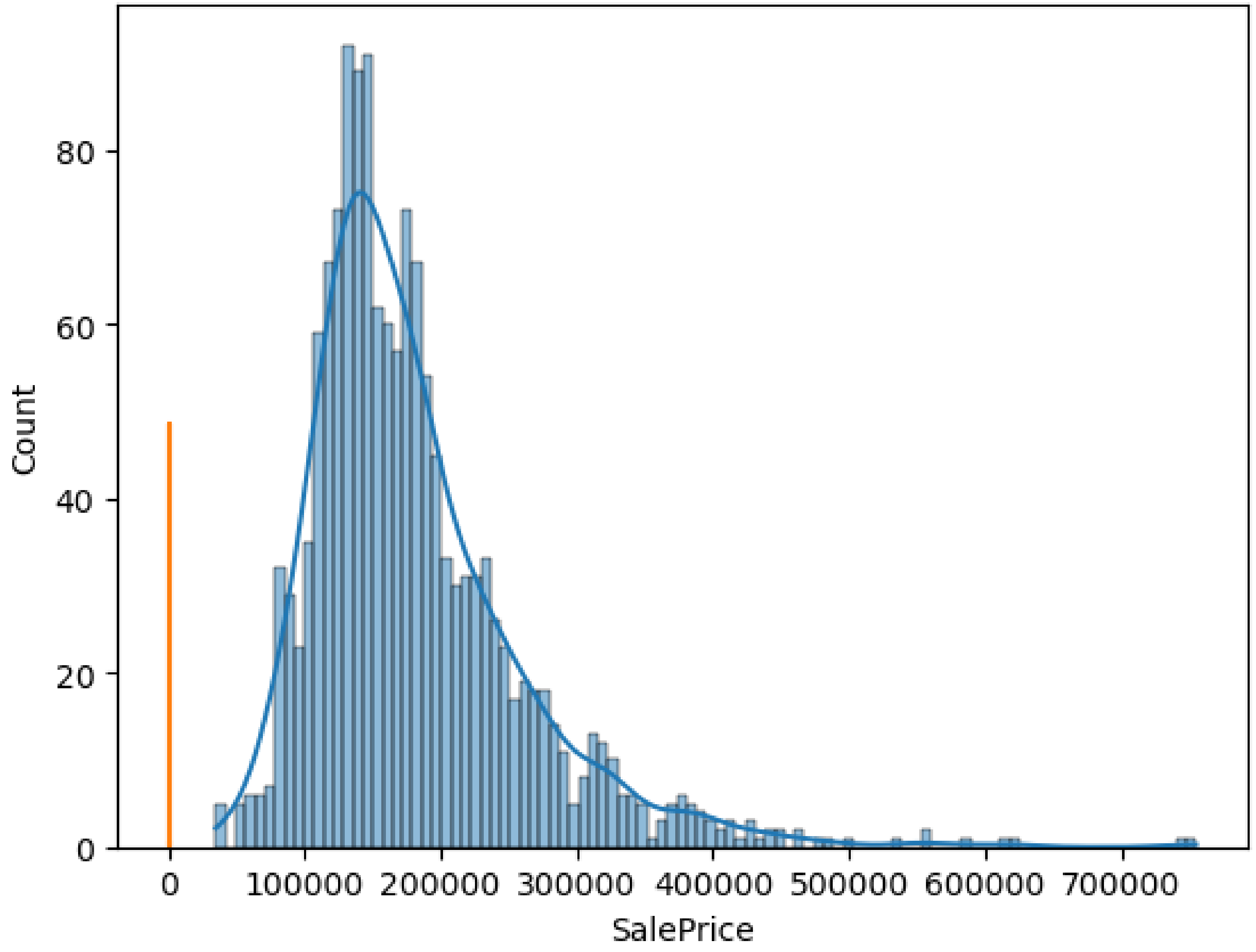


EDA

'SalePrice' distribution

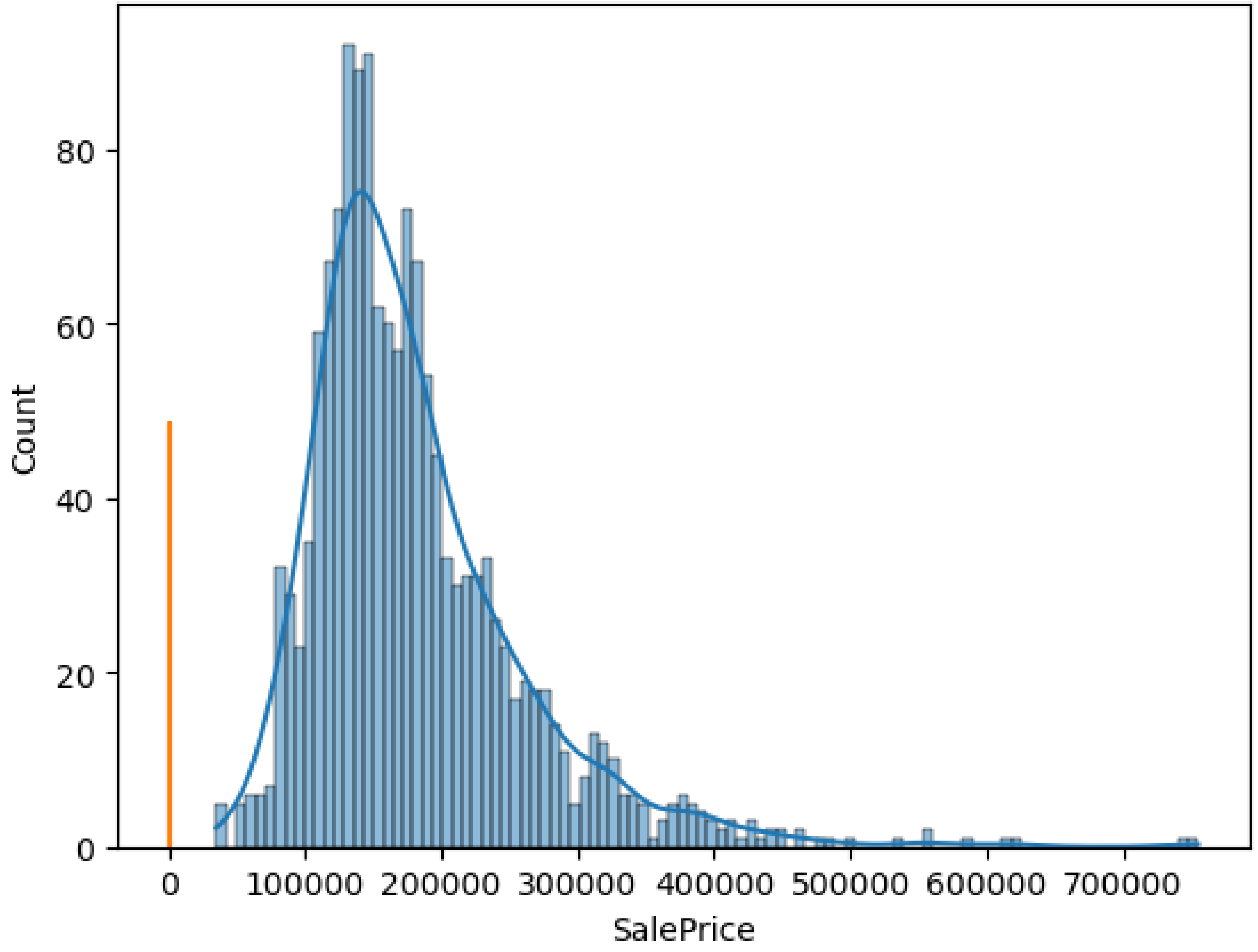


EDA 'SalePrice' distribution



- **Distribution Shape:**
  - The distribution of SalePrice remains **right-skewed** (positively skewed), with the **majority of home prices falling between 100,000 and 250,000**.
  - The data shows **a single strong peak** (mode), with frequency gradually tapering off as prices increase, indicating that **lower-to-mid-priced** houses are **most common** in the dataset.
- **Outliers and Spread:**
  - There is a **long right tail**, though now the artificial spike at zero is gone.
  - **Some homes** are significantly **more expensive** (above **USD400K**), but these are **rare** compared to the overall sample.
- **Central Tendency:**
  - **Most values** are concentrated **between USD120K and USD200K**, suggesting that this is the **primary price segment** in the data.
  - The **mean is likely higher than the mode**, due to the presence of high-value properties pulling the average upwards.

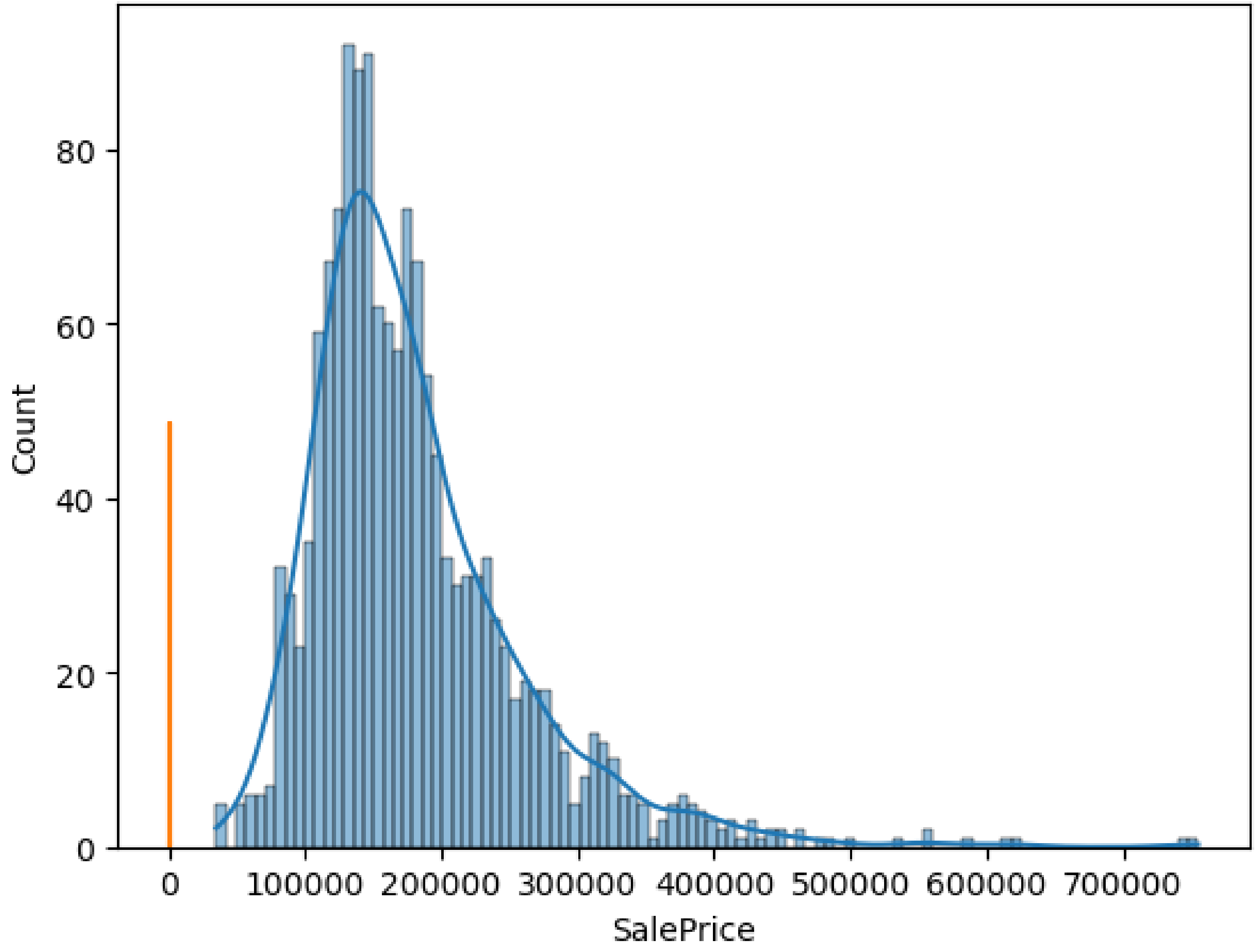
EDA 'SalePrice' distribution




```
SalePrice's Skew:   1.8828757597682129

SalePrice's Kurtosis:   6.536281860064529

SalePrice's Quantitle:
 0.001      36499.351
 0.010      61815.970
 0.050      88000.000
 0.250     129975.000
 0.500     163000.000
 0.750     214000.000
 0.950     326100.000
 0.990     442567.010
 0.999     689920.000
 Name: SalePrice, dtype: float64
```

- **Skew = 1.8829**: The distribution is **right-skewed** with a noticeable degree of skewness. While not extreme (>3), it is sufficient to cause some models—particularly linear models with homoscedastic error assumptions—to be unduly influenced by larger values.
- **Kurtosis = 6.5363**: This indicates **heavy tails**, meaning there are **more outliers in the right tail** than expected under a normal distribution. The risk of significant outliers is higher than the standard level.

EDA ‘SalePrice’ distribution

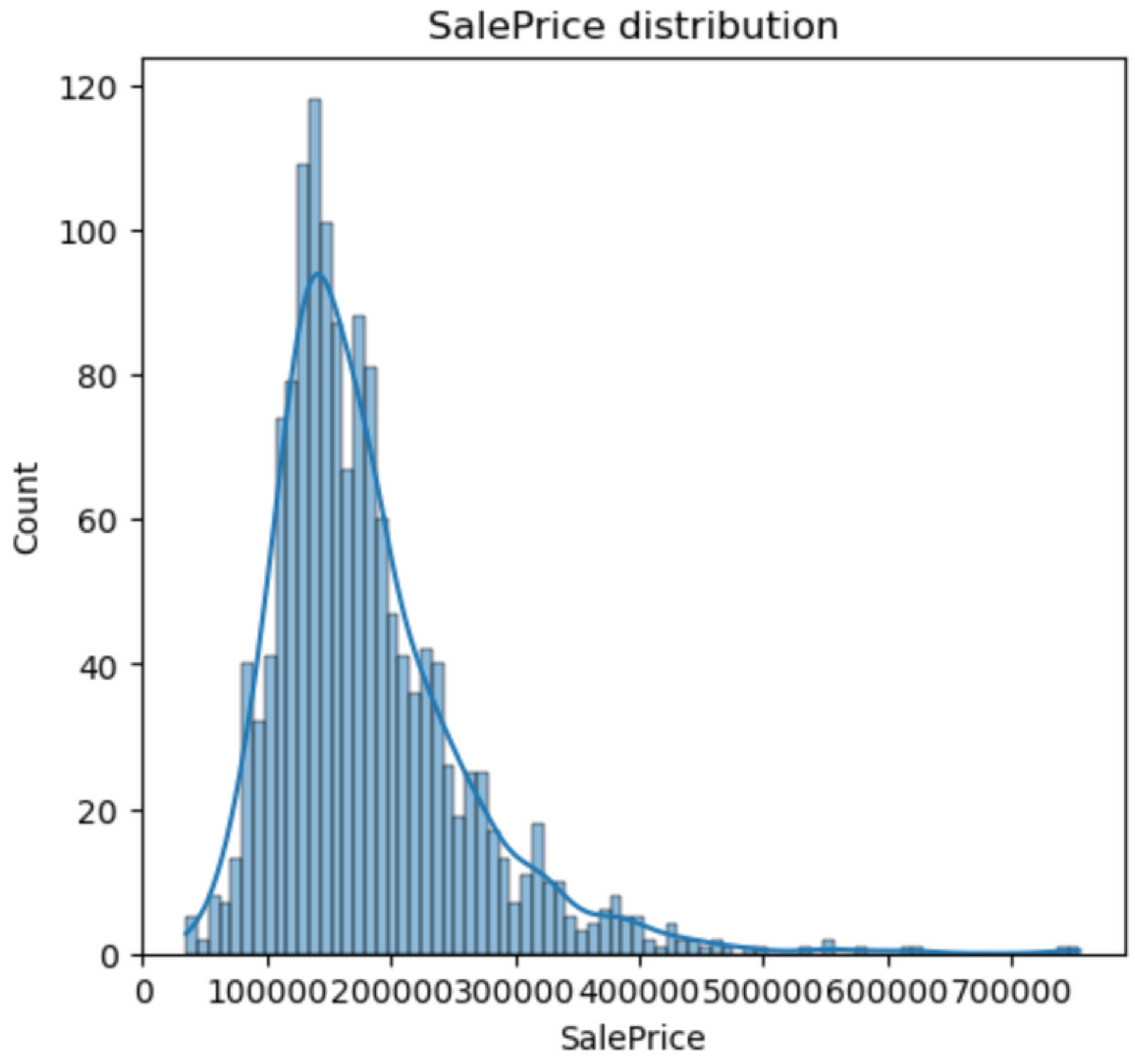



```
SalePrice's Skew:   1.8828757597682129

SalePrice's Kurtosis:   6.536281860064529

SalePrice's Quantitle:
 0.001     36499.351
 0.010     61815.970
 0.050     88000.000
 0.250    129975.000
 0.500    163000.000
 0.750    214000.000
 0.950    326100.000
 0.990    442567.010
 0.999    689920.000
 Name: SalePrice, dtype: float64
```

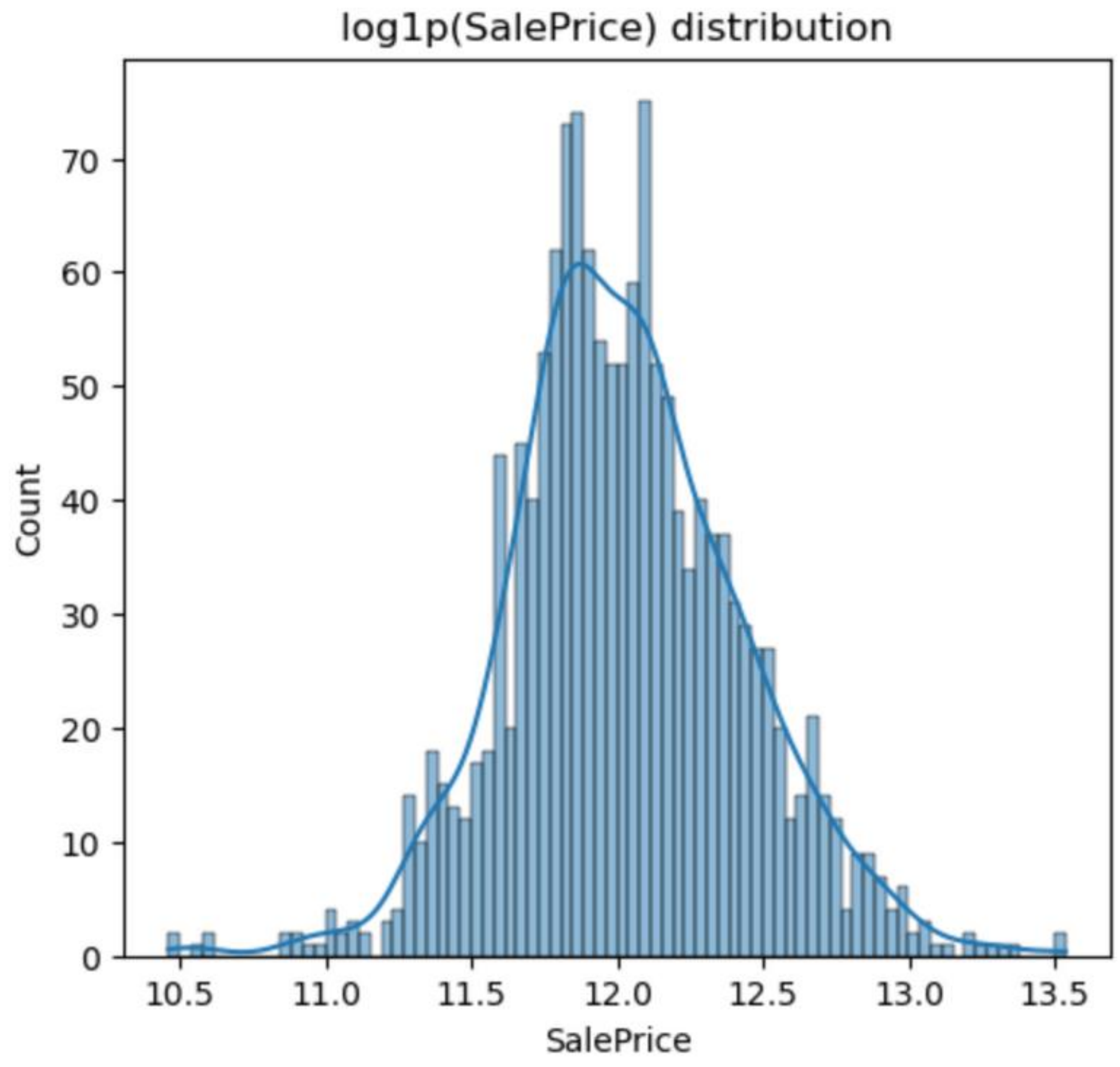
- **Skew = 1.8829**: The distribution is **right-skewed** with a noticeable degree of skewness. While not extreme (>3), it is sufficient to cause some models—particularly linear models with homoscedastic error assumptions—to be unduly influenced by larger values.
- **Kurtosis = 6.5363**: This indicates **heavy tails**, meaning there are **more outliers in the right tail** than expected under a normal distribution. The risk of significant outliers is higher than the standard level.

→ Try a log transformation (e.g., log1p) on SalePrice to reduce both the skewness and the heavy tails before training a model

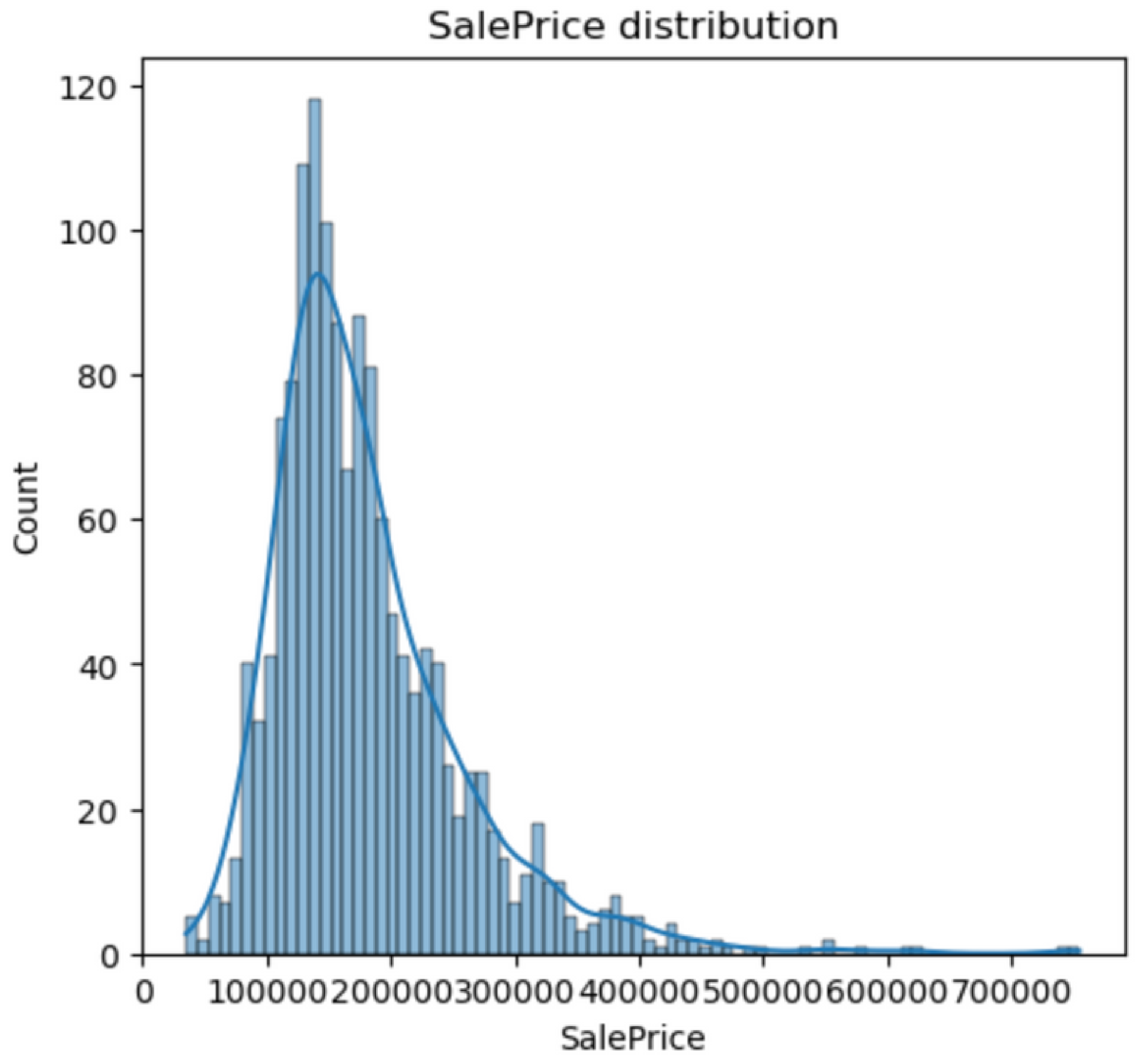
Preprocessing 'SalePrice' Log distribution



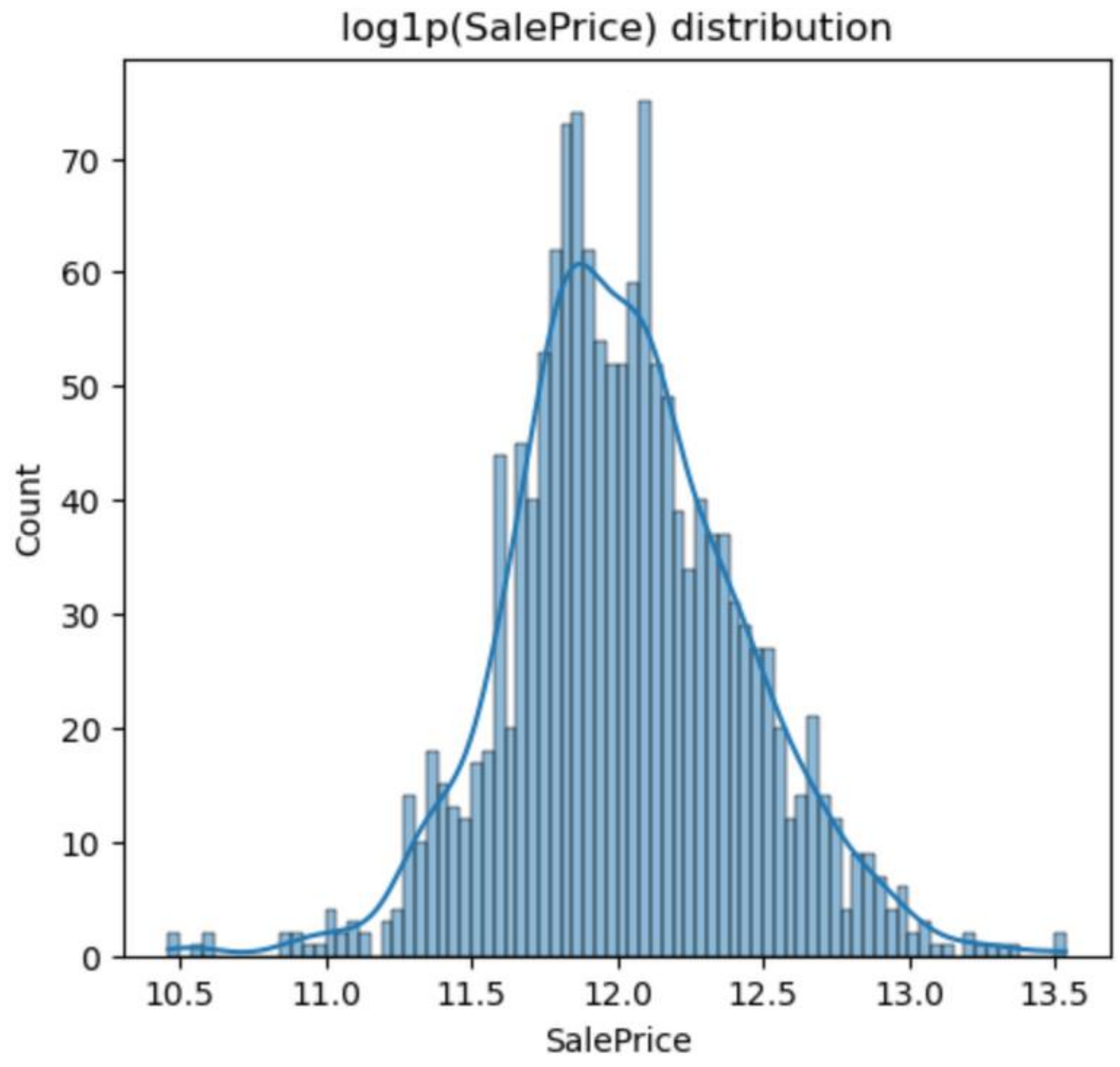
np.log1p(
train['SalePrice'
]
)



Preprocessing 'SalePrice' Log distribution



np.log1p(
train['SalePrice'
]
)



log1p *transformation* ***reshapes*** *the distribution to be* ***more symmetrical****, significantly* ***reducing*** *the* ***skewness***

Preprocessing Log'SalePrice' Diagnostic

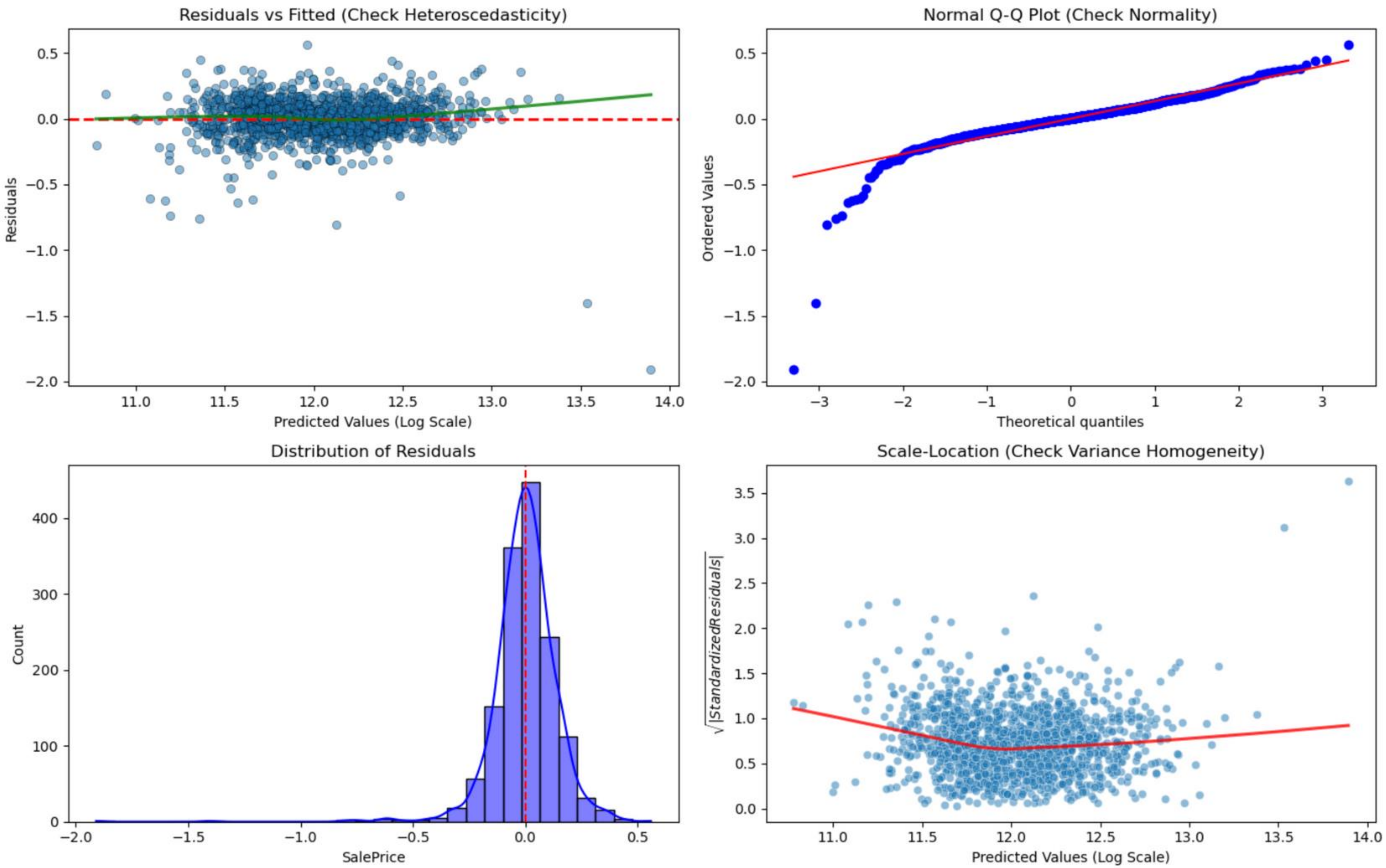
To **check the effectiveness** of the log transformation onto 'SalePrice' column, we visualize some plots and check:

- ***Heteroscedasticity***
- ***Normality***
- ***Variance Homogeneity***
- ***Distribution of Residuals***

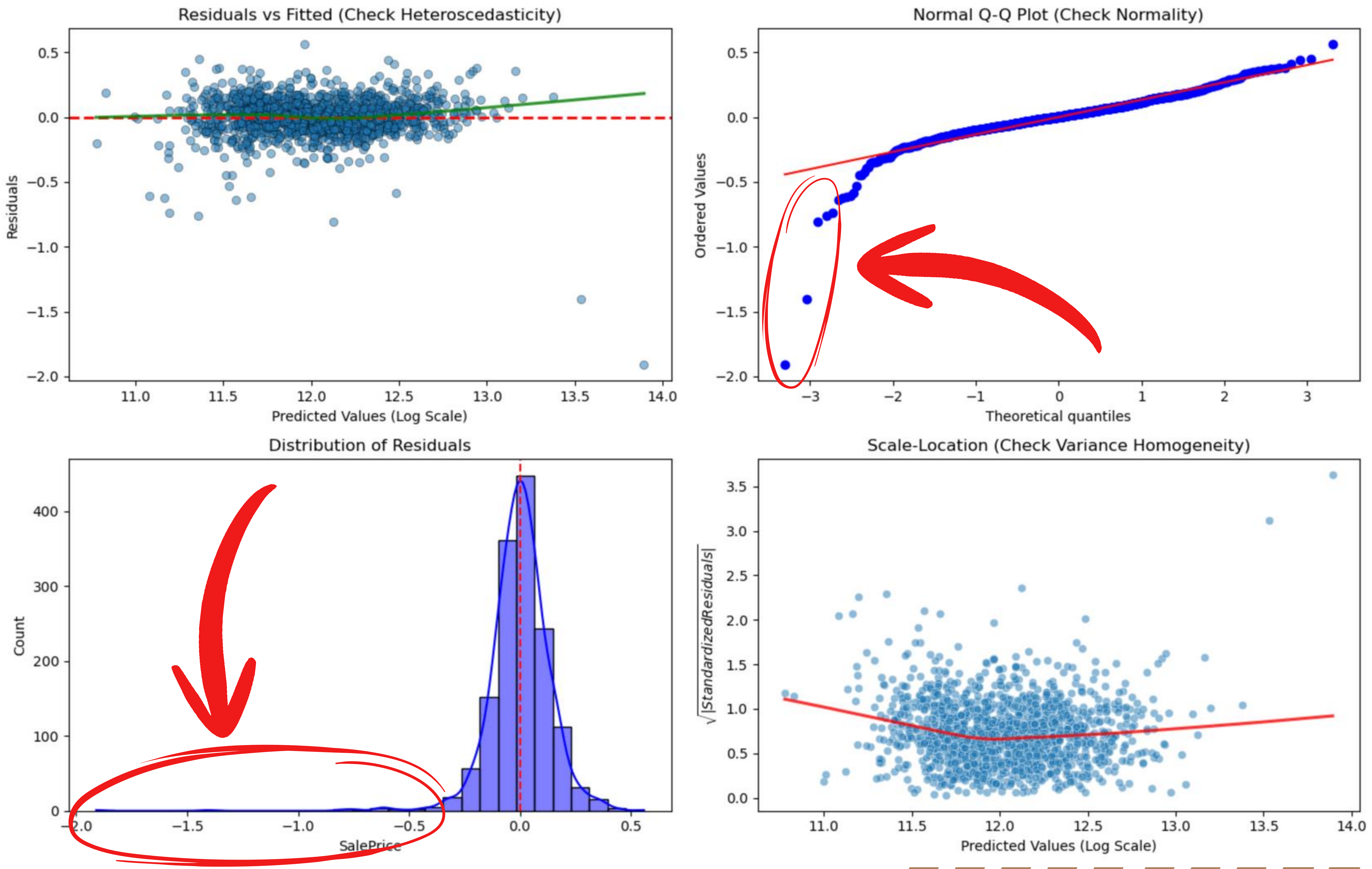
Preprocessing Log'SalePrice' Diagnostic

To **check the effectiveness** of the log transformation onto 'SalePrice' column, we visualize some plots and check:

- ***Heteroscedasticity***
- ***Normality***
- ***Variance Homogeneity***
- ***Distribution of Residuals***



Preprocessing Log'SalePrice' Diagnostic



Preprocessing Log'SalePrice' Diagnostic

Outliers Detection and Removal

```
bad_outliers = df[residuals < -0.5]
```

	Id	SalePrice	GrLivArea	SaleCondition	OverallQual
30	31	40000	1317	Normal	4
410	411	60000	1276	Abnorml	5
462	463	62383	864	Normal	5
495	496	34900	720	Abnorml	4
523	524	184750	4676	Partial	10
632	633	82500	1411	Family	7
812	813	55993	1044	Alloca	5
916	917	35311	480	Abnorml	2
968	969	37900	968	Abnorml	3
1298	1299	160000	5642	Partial	10
1324	1325	147000	1795	Partial	8

1. The "Internal/Abnormal Transactions" Group

- ID **633** (SalePrice 82,500 | Qual 7 | Size 1411 | Family)
  - A good quality house (7/10), large area, but sold at an extremely low price.
  - SaleCondition = Family. This indicates a sale between family members (e.g., parents to children).
- IDs **496, 917, 969** (SaleCondition = Abnorml)
  - "Abnorml" often indicates foreclosure, distress sales, or cancelled transactions. Their abnormally low prices are not due to poor house quality, but because the owner needed urgent cash.

2. The "Statistical Noise" Group

- ID **30** (SalePrice 40,000 | Size 1317)
  - Imagine a 1317 sqft house selling for only 40k, while ID 917 (a tiny 480 sqft house) sold for 35k.
  - Business perspective: could be a house in a demolition zone or with severe foundation damage not captured in the data.
- ID **462** (SalePrice 62,383 | Size 864)
  - The price is oddly specific (62,383): result of a specific auction or bank foreclosure formula rather than a negotiated market price.
- ID **1325** (SalePrice 147,000 | Qual 8 | Partial)
  - "Partial" usually refers to new construction not yet completed. A quality rating of 8 is very high (Luxury).
- ....

Preprocessing Log'SalePrice' Diagnostic

Outliers Detection and Removal

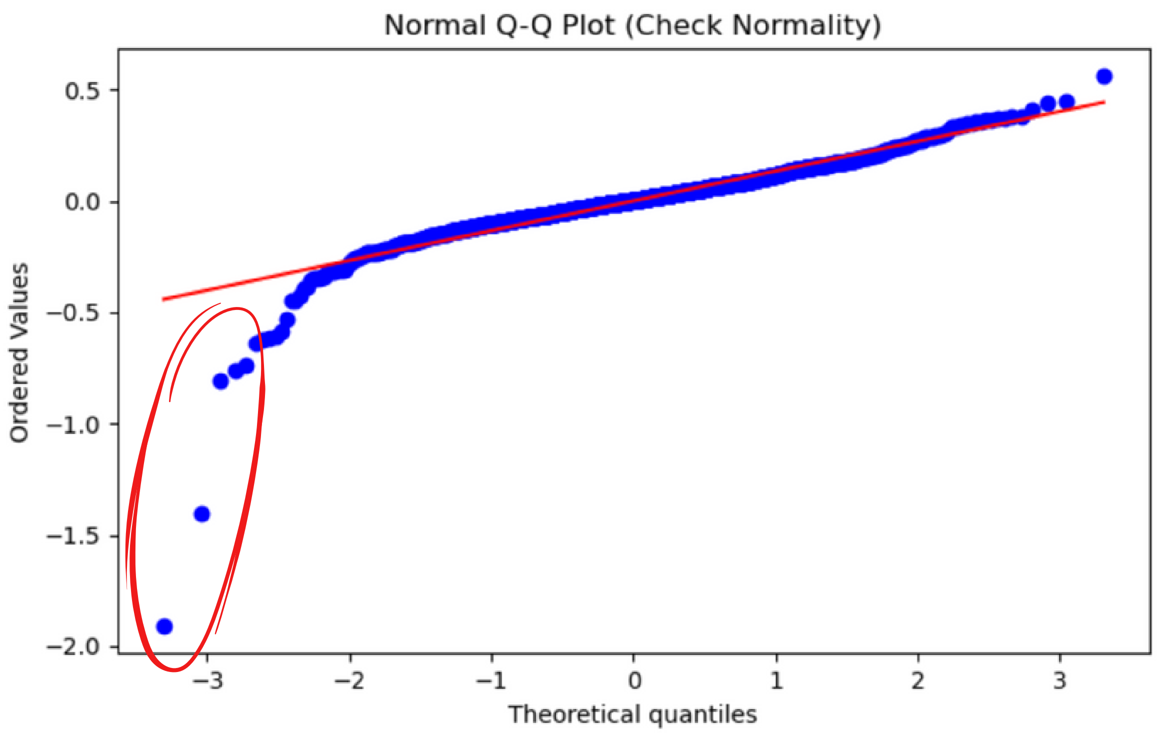
```
bad_outliers = df[residuals < -0.5]
```

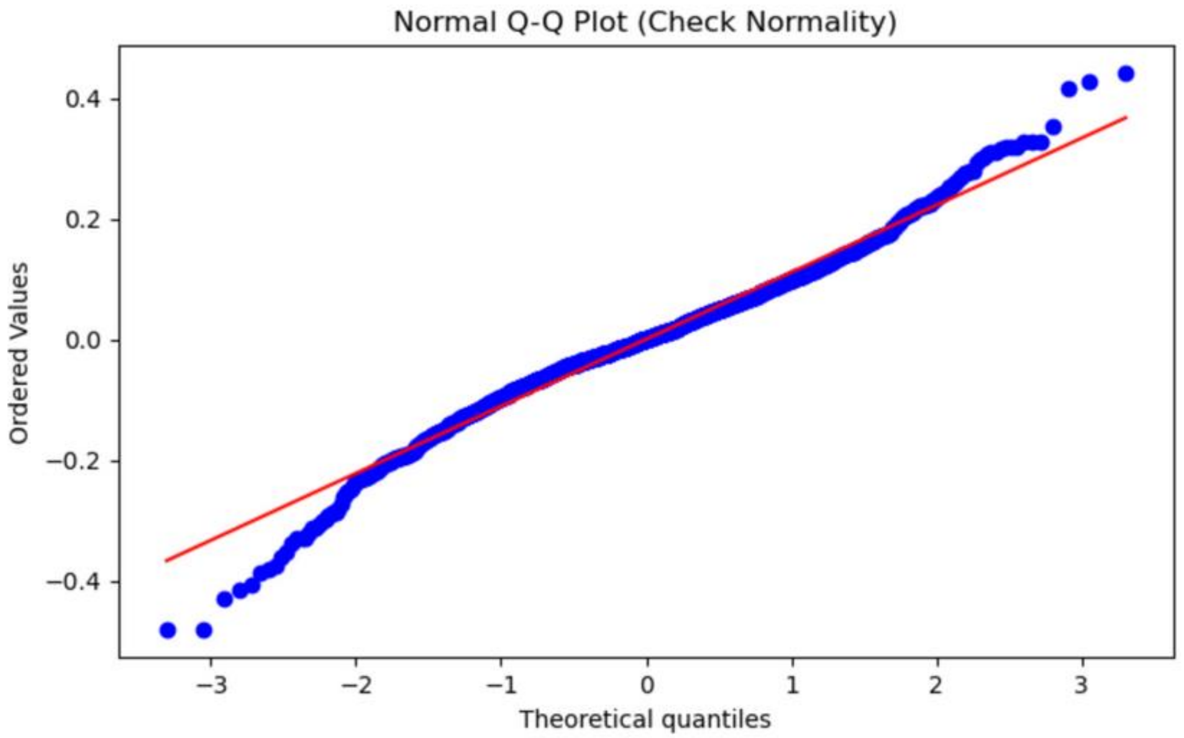
	Id	SalePrice	GrLivArea	SaleCondition	OverallQual
30	31	40000	1317	Normal	4
410	411	60000	1276	Abnorml	5
462	463	62383	864	Normal	5
495	496	34900	720	Abnorml	4
523	524	184750	4676	Partial	10
632	633	82500	1411	Family	7
812	813	55993	1044	Alloca	5
916	917	35311	480	Abnorml	2
968	969	37900	968	Abnorml	3
1298	1299	160000	5642	Partial	10
1324	1325	147000	1795	Partial	8

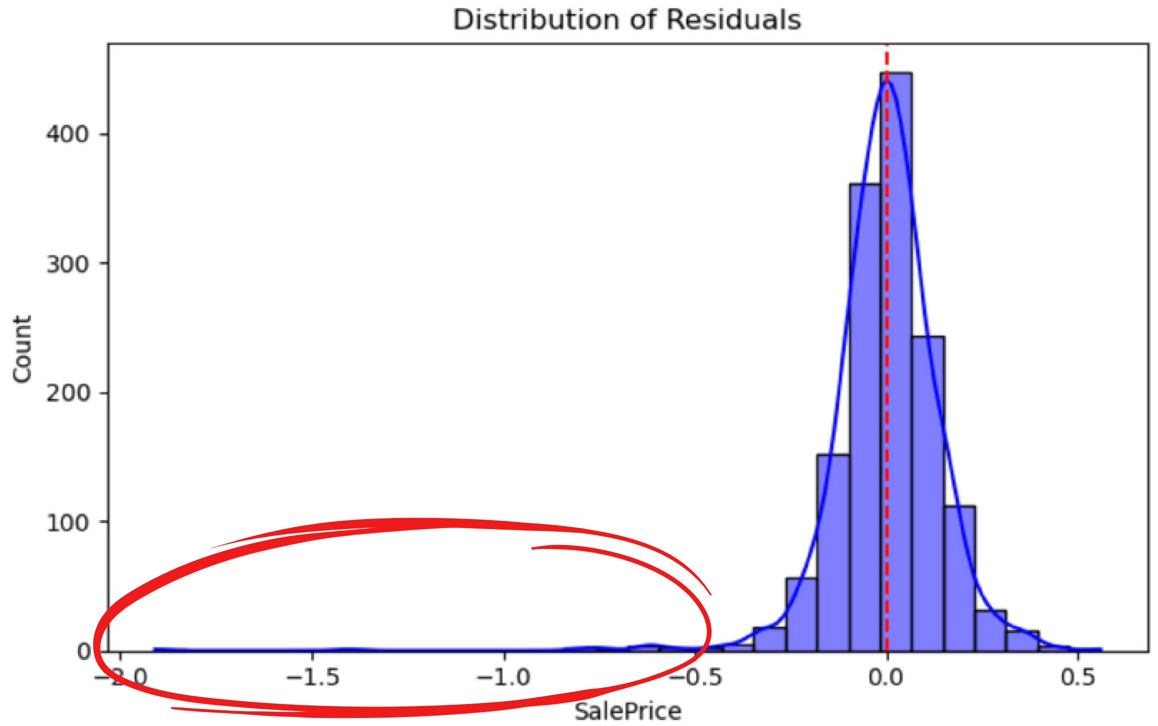
```
outliers_to_drop = [
31, 411, 463, 496, 524, 633,
813, 917, 969, 1299, 1325
]
```

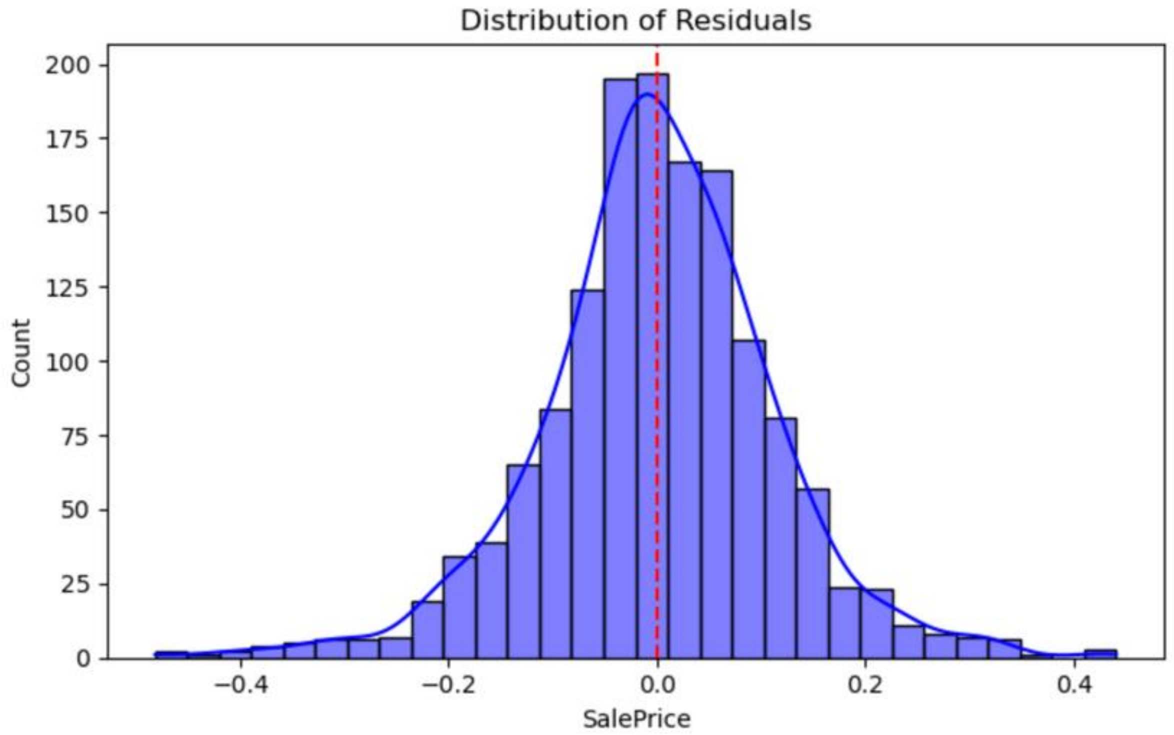
```
Training set size before dropping: (1460, 81)
Training set size after dropping: (1449, 81)
```

Preprocessing Log'SalePrice' Diagnostic









Preprocessing Ordinal and Nominal Features

For **ordinal columns**, we **convert text to number**:

1. Standard scale
quality_map = {'Ex': 5, 'Gd': 4, 'Ta': 3, 'Fa': 2, 'Po': 1, 'None': 0}

2. Basement Exposure scale
exposure_map = {'Gd': 4, 'Av': 3, 'Mn': 2, 'No': 1, 'None': 0}

3. Basement Finish scale (GLQ -> Unf)
bsmt_fin_map = {'GLQ': 6, 'ALQ': 5, 'BLQ': 4, 'Rec': 3, 'LwQ': 2, 'Unf': 1, 'None': 0}

4. Garage Finish scale
garage_fin_map = {'Fin': 3, 'RFn': 2, 'Unf': 1, 'None': 0}

5. Functional scale
func_map = {'Typ': 7, 'Min1': 6, 'Min2': 5, 'Mod': 4, 'Maj1': 3, 'Maj2': 2, 'Sev': 1, 'Sal': 0}

6. Fence scale
fence_map = {'GdPrv': 4, 'MnPrv': 3, 'GdWo': 2, 'MnWw': 1, 'None': 0}

7. LotShape
 # Reg (Regular) > IR1 > IR2 > IR3 (Irregular)
lot_shape_map = {'Reg': 4, 'IR1': 3, 'IR2': 2, 'IR3': 1}

8. LandSlope
 # Gtl (Gentle) > Mod > Sev (Severe)
slope_map = {'Gtl': 3, 'Mod': 2, 'Sev': 1}

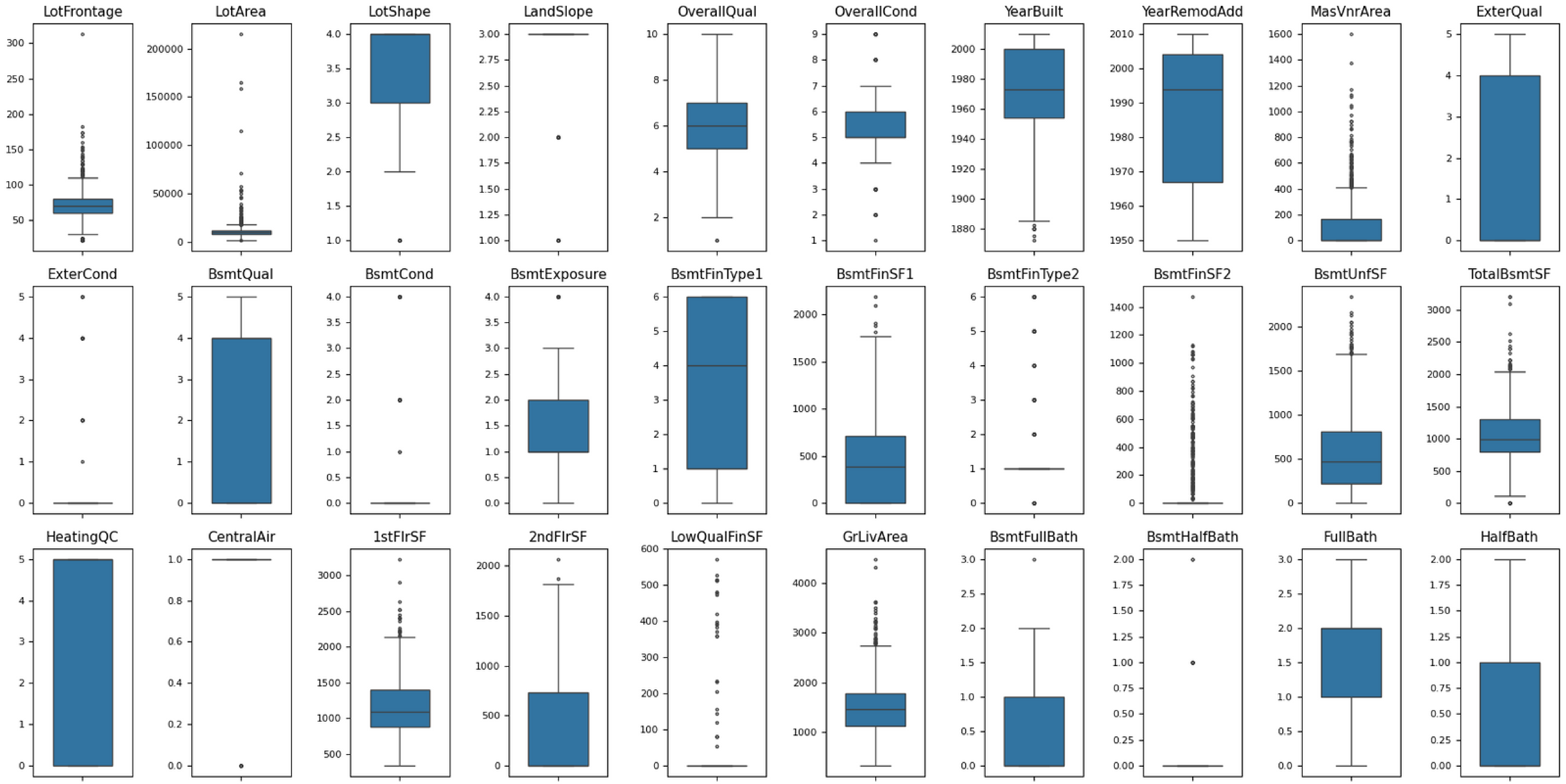
9. PavedDrive
 # Y (Paved) > P (Partial) > N (Dirt)
paved_map = {'Y': 3, 'P': 2, 'N': 1}

Ordinal and Nominal Features

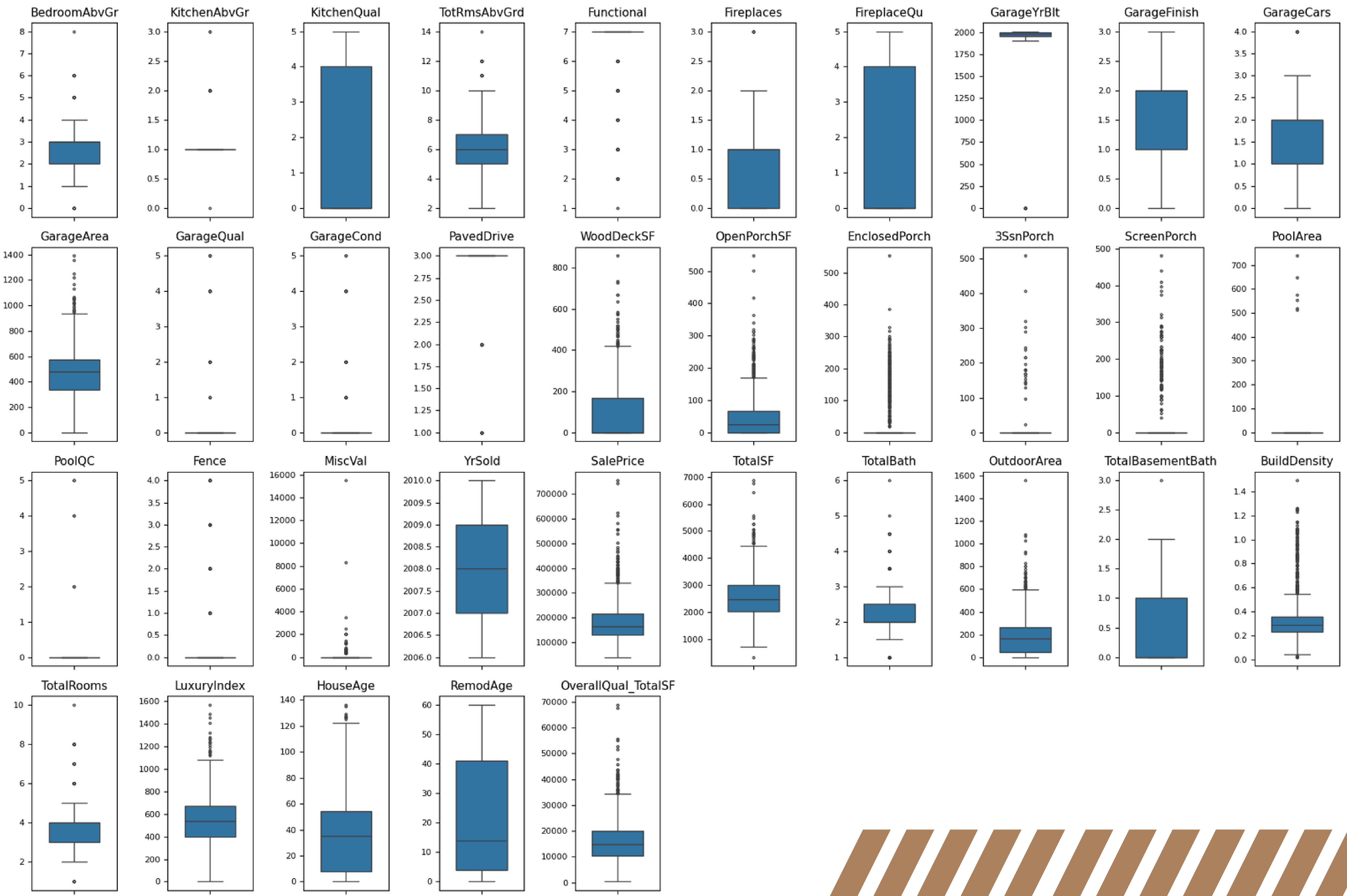
For **two nominal columns**, we **convert numbers to strings** for One-Hot Encoding:

MoSold	MSSubClass
2	60
5	20
9	60
2	70
12	60

Preprocessing Outliers Detection



Preprocessing Outliers Detection



Preprocessing

Outliers Processing

For features with skew > 0.75, we deploy log transformation and create additional logged columns ending with '_log'

```
def add_log_transformed_features(df, threshold=0.75, exclude_cols=['Id', 'SalePrice']):
    df_processed = df.copy()

    # 1. Identify numerical columns (excluding ID and Target)
    numeric_feats = df_processed.select_dtypes(include=['number']).columns
    numeric_feats = [col for col in numeric_feats if col not in exclude_cols]

    # 2. Calculate skewness
    skewed_feats = df_processed[numeric_feats].apply(lambda x: skew(x.dropna())).sort_values(ascending=False)
    skewness_df = pd.DataFrame({'Skew': skewed_feats})

    # 3. Filter columns exceeding skew threshold
    high_skew_cols = skewness_df[abs(skewness_df['Skew']) > threshold].index.tolist()
    print(f"Detected {len(high_skew_cols)} skewed features (skew > {threshold})")

    # 4. Create new columns
    new_log_cols = []
    for col in high_skew_cols:
        new_col_name = f"{col}_log"
        # Use safe log1p transformation
        df_processed[new_col_name] = np.log1p(df_processed[col])
        new_log_cols.append(new_col_name)

    return df_processed, new_log_cols

train, log_cols = add_log_transformed_features(train)
```

```
Detected 39 skewed features (skew > 0.75)
```

Preprocessing Binary Flags

For columns which have values of 0 means they don't have it. We will create columns starting with 'Has_' displaying 1 if they have it and otherwise 0.

```python
zero_inflated_cols = [
    'PoolArea',      # Has swimming pool?
    '2ndFlrSF',      # Has 2nd floor?
    'GarageArea',    # Has garage?
    'TotalBsmtSF',   # Has basement?
    'Fireplaces',    # Has fireplace?
    'MasVnrArea',    # Has exterior masonry veneer?
    'WoodDeckSF',    # Has wood deck?
    'OpenPorchSF',   # Has open porch?
    'EnclosedPorch',
    '3SsnPorch',
    'ScreenPorch',
    'MiscVal'        # Has other miscellaneous features of value?
]
for col in zero_inflated_cols:
    if col in df.columns:
        # Create new column name, e.g., Has_PoolArea
        new_col = f'Has_{col}'

        # Logic: If > 0 then 1 (Has), otherwise 0 (Does not have)
        df[new_col] = (df[col] > 0).astype(int)

return df
```

Preprocessing One-hot Encoding

Before deploying OHE method for those columns categorical, we remove ‘SalePrice’, and ‘Id’ columns

```
X_train = train.drop(['SalePrice', 'Id'], axis=1)
X_train_ohe = pd.get_dummies(X_train, columns=categorical_cols, drop_first=True)
```

```
Final number of features: 285
```

Preprocessing Correlation Detection and Removal

We will run two groups of models, one significantly affected by multicollinearity (1) and one less be (2). Therefore, we will remove features with highly exclusive correlation (prepared for 1), but keep remaining them and dropping duplicates for 2.

Preprocessing Correlation Detection and Removal

We will run two groups of models, one significantly affected by multicollinearity (1) and one less be (2). Therefore, we will remove features with highly exclusive correlation (prepared for 1), but keep remaining them and dropping duplicates for 2.

```python
cols_to_drop_linear = [
    # --- 1. COMPLETELY DUPLICATE (r ~ 1.0) ---
    # Remove Log of Flag/Ordinal columns (they only increase VIF)
    'Has_TotalBsmtSF_log', 'Has_EnclosedPorch_log', 'Has_ScreenPorch_log',
    'CentralAir_log', 'Has_PoolArea_log', 'Has_MiscVal_log', 'Has_3SsnPorch_log',
    'Has_GarageArea_log', 'PavedDrive_log', 'ExterCond_log', 'FireplaceQu_log',
    'LandSlope_log', 'PoolQC_log', 'BsmtCond_log', 'LotShape_log',
    'Functional_log', 'GarageQual_log', 'GarageCond_log', 'Fence_log',
    'BsmtExposure_log', 'BsmtFinType2_log', 'BsmtFinType1_log',

    # Remove OHE columns with 100% duplication
    'GarageType_None', 'Exterior2nd_CBlock', 'MSSubClass_90', 'BldgType_Duplex',
    'Exterior1st_CBlock', 'MSSubClass_190', 'BldgType_2fmCon', 'SaleType_New',

    # --- 2. ORIGINAL VS LOG/AGE/INTERACTION (Keep only one version) ---
    # Keep AGE version instead of YEAR
    'YearBuilt', 'YearRemodAdd',

    # Keep LOG/FLAG version instead of ORIGINAL
    'GrLivArea', 'TotalSF', '1stFlrSF', '2ndFlrSF',
    'BsmtFinSF2', 'BsmtFinSF1', 'BsmtUnfSF', 'TotalBsmtSF',
    'PoolArea', 'MiscVal', '3SsnPorch', 'ScreenPorch', 'EnclosedPorch',
    'WoodDeckSF', 'LotFrontage', 'GarageYrBlt', 'MasVnrArea',

    # Keep Scored version (Ordinal)
    'KitchenAbvGr', 'BedroomAbvGr',

    # Keep Aggregated/Interaction version
    'OverallQual', 'GarageArea', 'TotalRooms', 'TotRmsAbvGrd',

    # --- 3. REDUNDANT CALCULATED COLUMNS ---
    'TotalBasementBath',
]

# 4. Create list for Linear Model
X_linear_features = [col for col in all_features if col not in cols_to_drop_linear]
```

```python
cols_to_drop_tree = [
    'GarageType_None', 'Exterior2nd_CBlock', 'MSSubClass_90', 'BldgType_Duplex',
    'Exterior1st_CBlock', 'MSSubClass_190', 'BldgType_2fmCon', 'SaleType_New'
]

X_tree_features = [col for col in all_features if col not in cols_to_drop_tree]
```

Preprocessing Inf and High VIF Removal

We additionally **check VIF** of features from the **X_linear_features**. This ultimately **remains 38 features** with the value of **VIF < 10**.

Preprocessing Inf and High VIF Removal

We additionally **check VIF** of features from the **X_linear_features**. This ultimately **remains 38 features** with the value of **VIF < 10**. They are:

'LotArea', 'LotShape', 'LandSlope', 'OverallCond', 'ExterQual', 'ExterCond', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'HeatingQC', 'CentralAir', 'LowQualFinSF', 'KitchenQual', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageFinish', 'GarageCars', 'GarageQual', 'GarageCond', 'PavedDrive', 'Fence', 'YrSold', 'LuxuryIndex', 'HouseAge', 'RemodAge', 'Has_TotalBsmtSF', 'Has_Fireplaces'

'LowQualFinSF_log', 'KitchenAbvGr_log', 'BsmtFinSF2_log', 'OutdoorArea_log', 'LotFrontage_log', 'BsmtUnfSF_log', 'BsmtFinSF1_log', 'TotalRooms_log'

Training

Linear Regression & Ridge & Lasso

At this stage, we run three linear-sensitive models: **OLS**, **Ridge**, and **Lasso** on the **internal train** dataset without test.

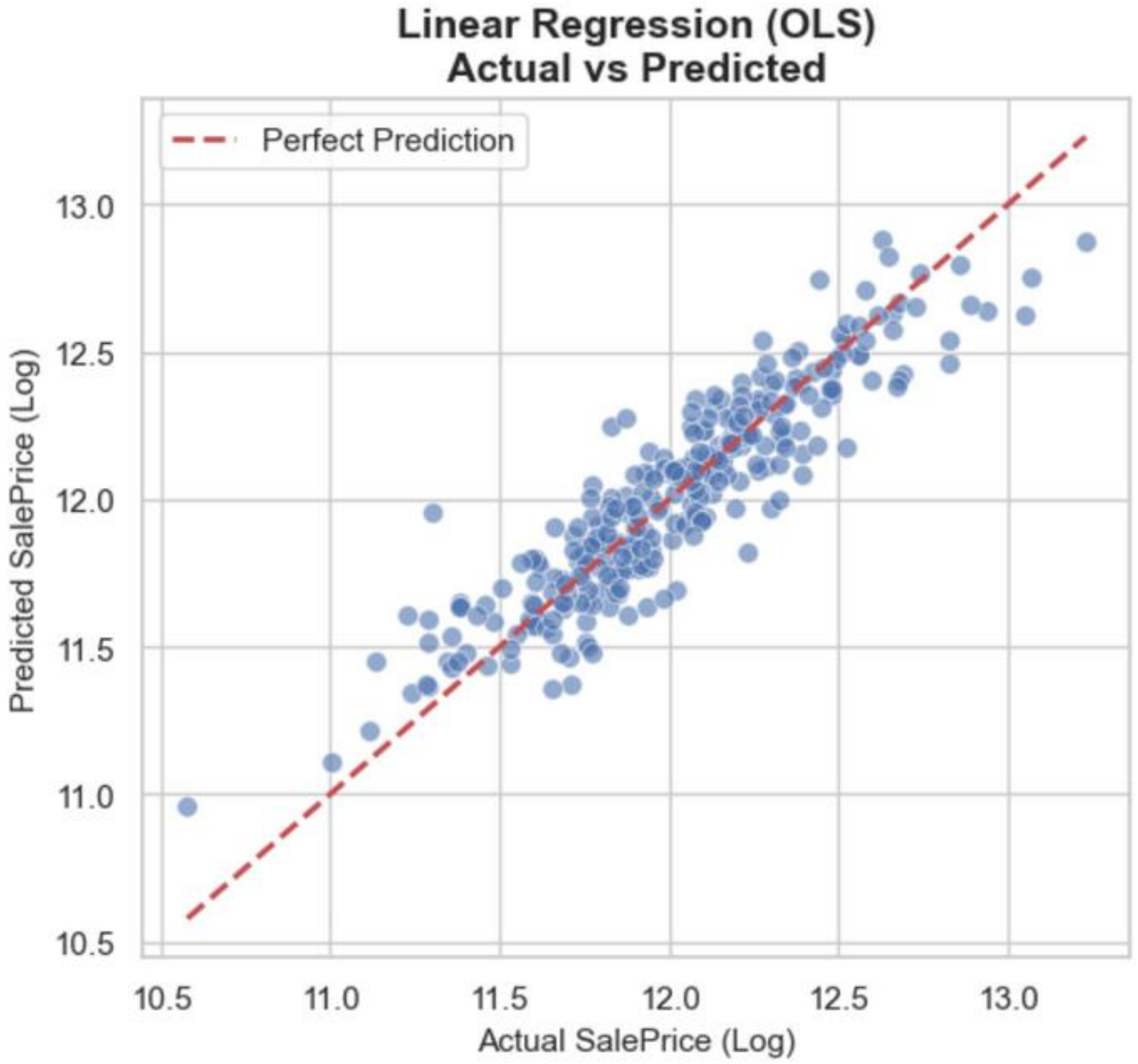
Training Linear Regression & Ridge & Lasso

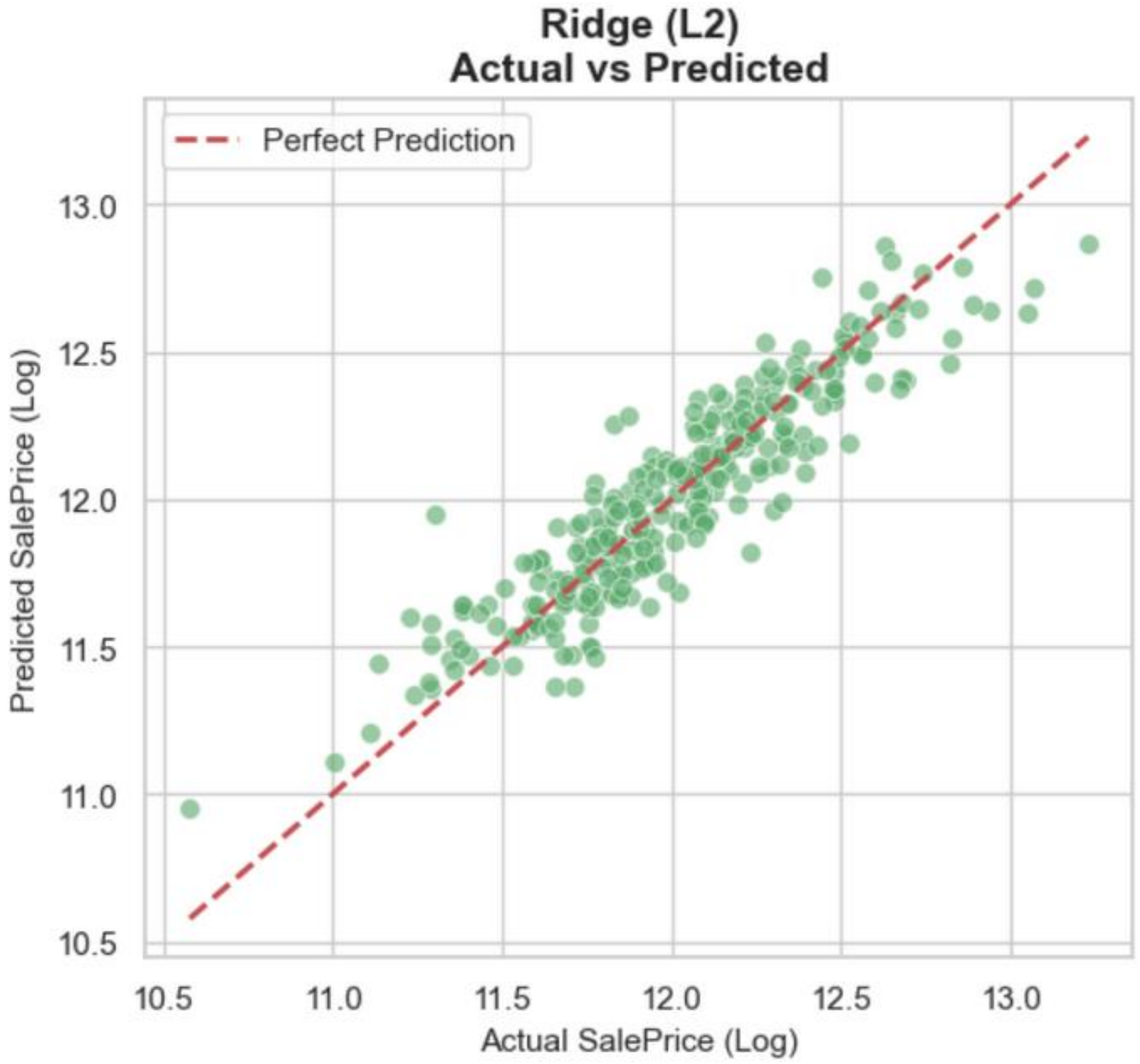
At this stage, we run three linear-sensitive models: **OLS**, **Ridge**, and **Lasso** on the **internal train** dataset without test.

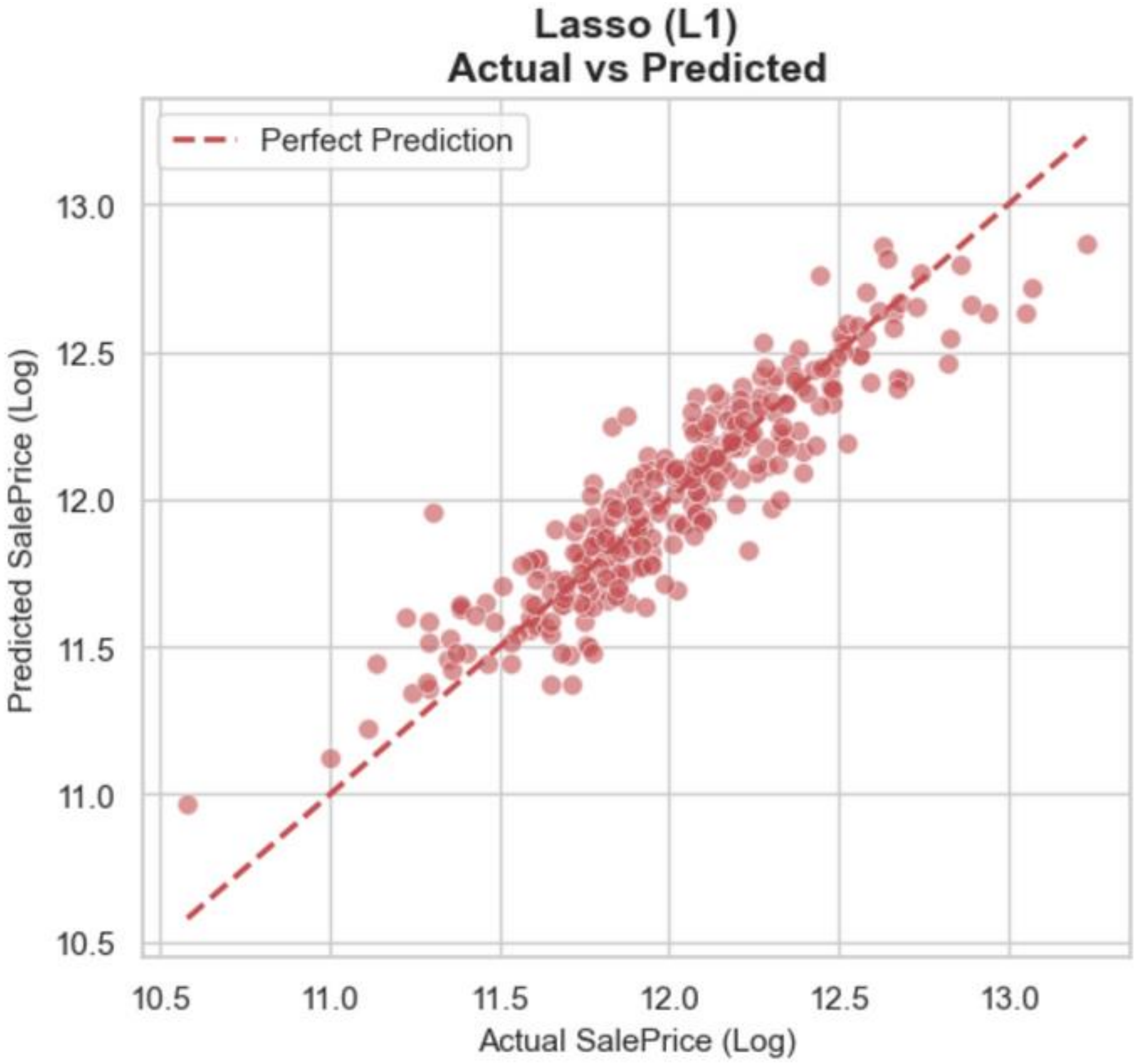
```
# --- 1. Train OLS Model (Linear Regression) ---
ols = LinearRegression()
ols.fit(X_train_int, y_train_int)
y_pred_ols = ols.predict(X_val)
rmse_ols = np.sqrt(mean_squared_error(y_val, y_pred_ols))

# --- 2. Train Ridge Model (L2 Regularization) ---
# Alpha = 10 (stable level)
ridge = Ridge(alpha=10, random_state=42)
ridge.fit(X_train_int, y_train_int)
y_pred_ridge = ridge.predict(X_val)
rmse_ridge = np.sqrt(mean_squared_error(y_val, y_pred_ridge))

# --- 3. Train Lasso Model (L1 Regularization - Feature Selection) ---
# Alpha = 0.0005 (determined optimal level)
lasso = Lasso(alpha=0.0005, random_state=42, max_iter=10000)
lasso.fit(X_train_int, y_train_int)
y_pred_lasso = lasso.predict(X_val)
rmse_lasso = np.sqrt(mean_squared_error(y_val, y_pred_lasso))
```

Training Linear Regression & Ridge & Lasso

At this stage, we run three linear-sensitive models: **OLS**, **Ridge**, and **Lasso** on the **internal train** dataset without test.



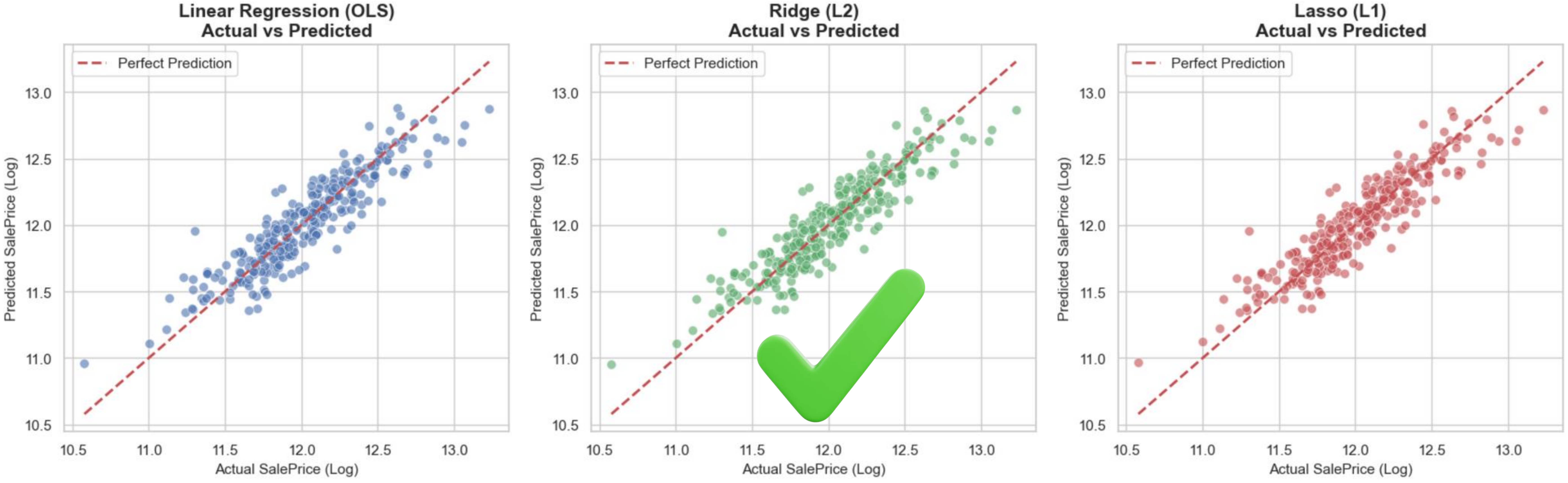





```
--- LINEAR MODELS PERFORMANCE REPORT (LOG SCALE) ---
1. Linear Regression (OLS) RMSE: 0.1583
2. Ridge Regression (alpha=10) RMSE: 0.1576
3. Lasso Regression (alpha=0.0005) RMSE: 0.1577
```

Training Linear Regression & Ridge & Lasso

At this stage, we run three linear-sensitive models: **OLS**, **Ridge**, and **Lasso** on the **internal train** dataset without test.



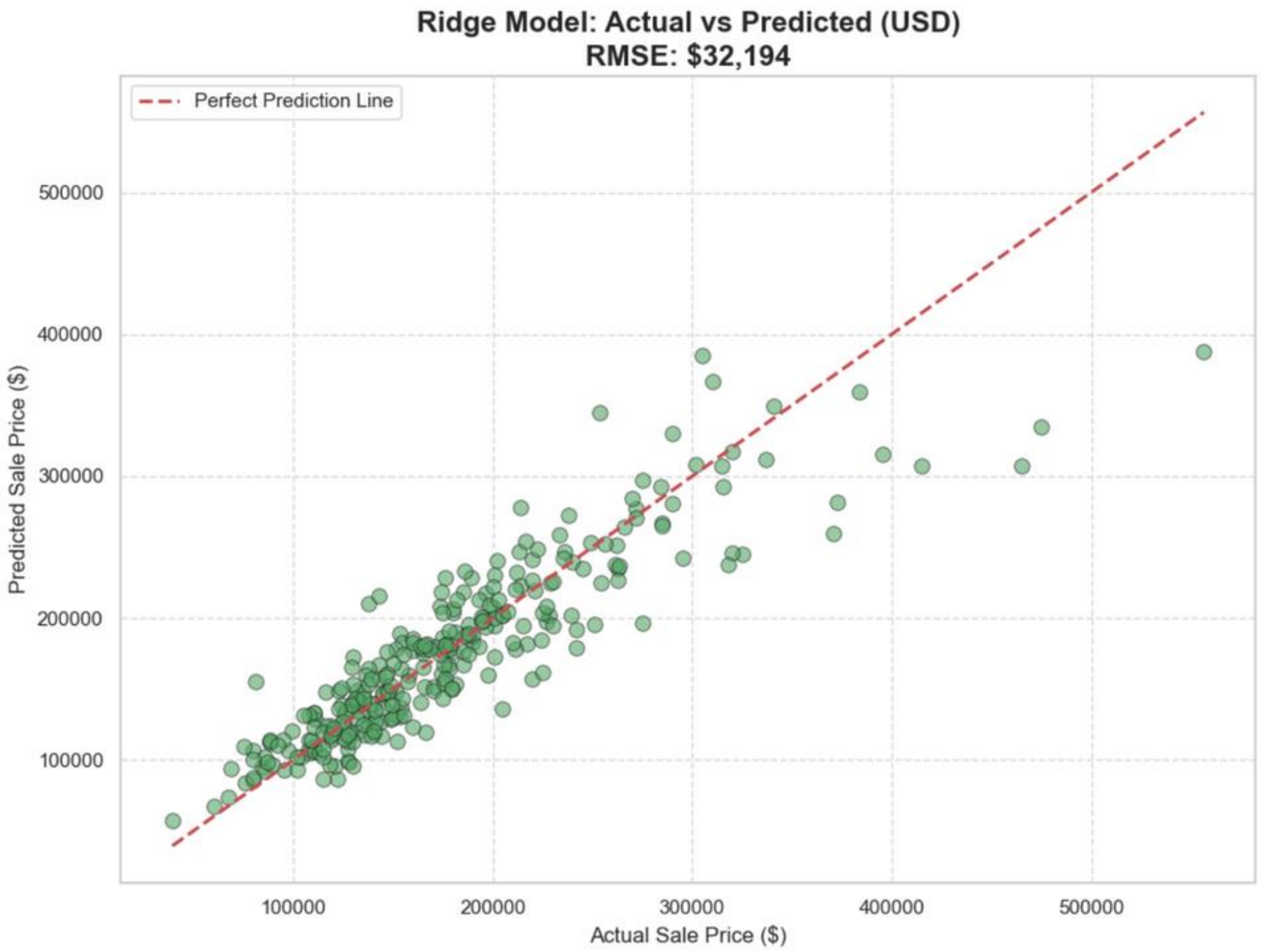

```
--- LINEAR MODELS PERFORMANCE REPORT (LOG SCALE) ---
1. Linear Regression (OLS) RMSE: 0.1583
2. Ridge Regression (alpha=10) RMSE: 0.1576
3. Lasso Regression (alpha=0.0005) RMSE: 0.1577
```

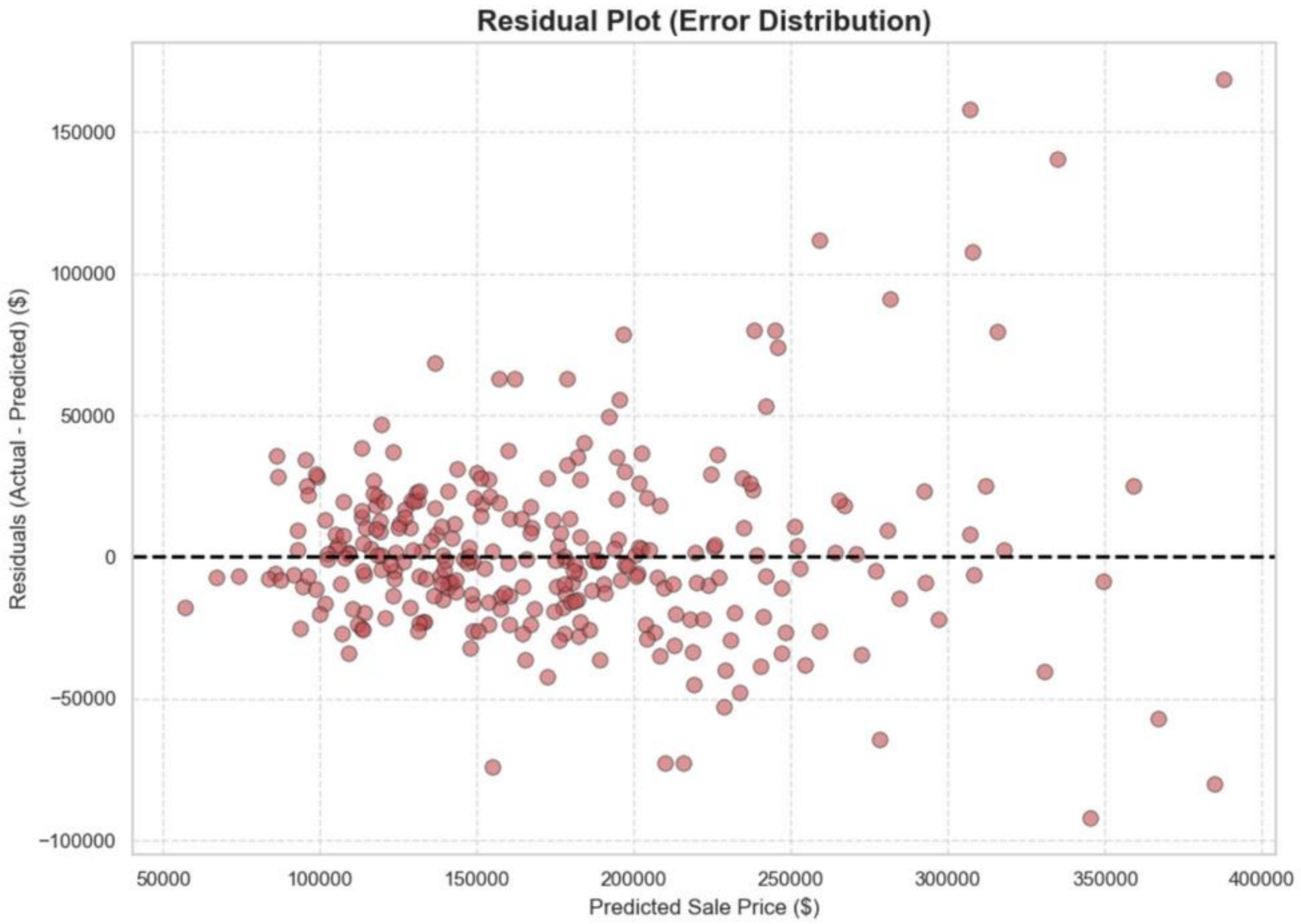
Training

SalePrice Prediction by Ridge

We try printing out the first 10 House price between Actual and Prediction generated by Ridge Regression.

	Actual_Price	Predicted_Price
538	133000.0	141931.656730
754	127500.0	99194.858214
49	177000.0	178074.535106
1380	124000.0	138987.004157
141	166000.0	119258.456424
614	305000.0	385005.163512
1050	220000.0	157041.839526
793	175000.0	186580.888758
1006	227000.0	196898.157181
1323	125000.0	131566.275138



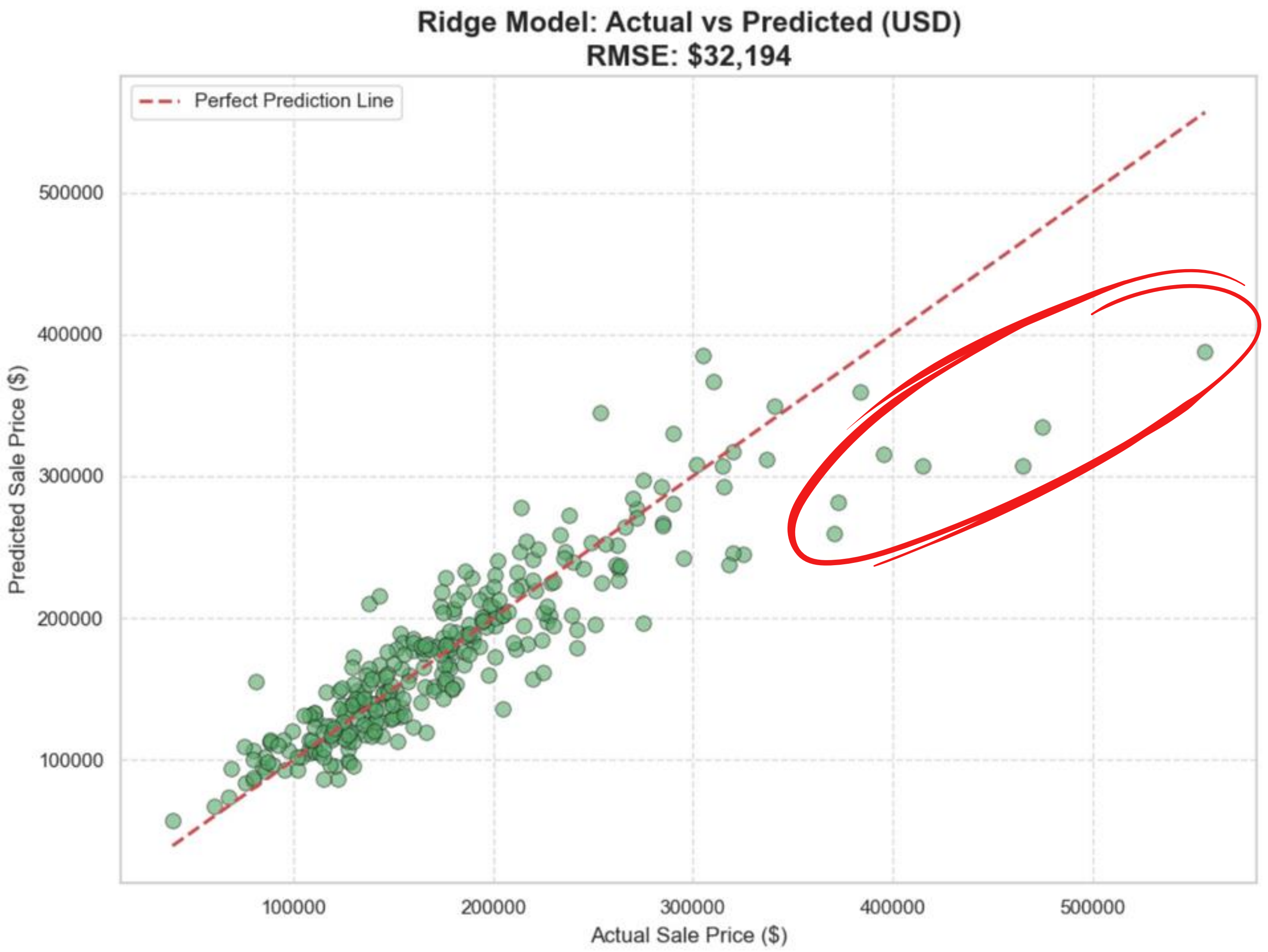


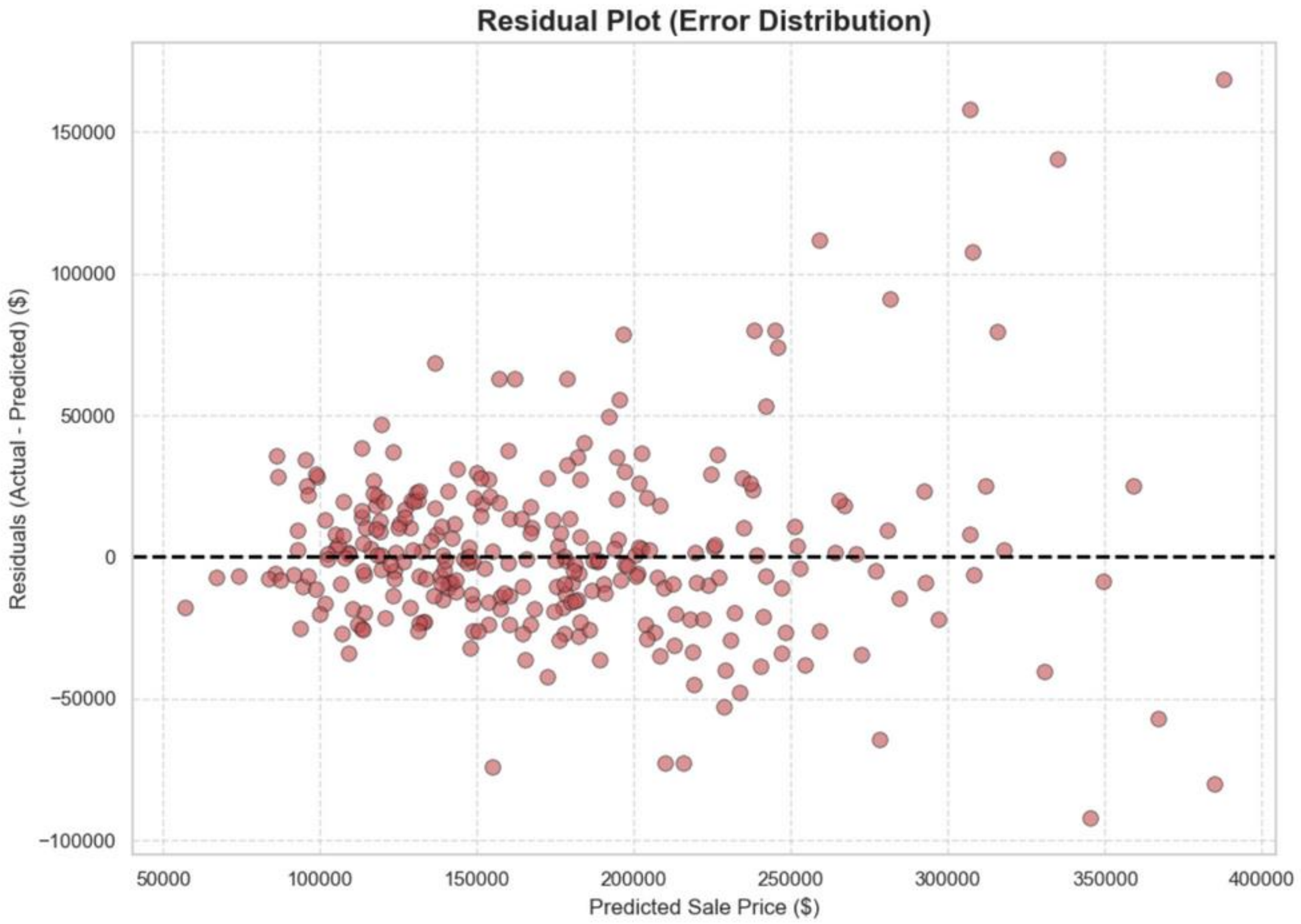
Training

SalePrice Prediction by Ridge

Limitations of Ridge model:

- The Ridge Regression model performs well in the mainstream market segment but is **under-fitting** in the high-end luxury segment.
- Where the actual price is **> $400,000**, the green data points consistently fall **below the red line**



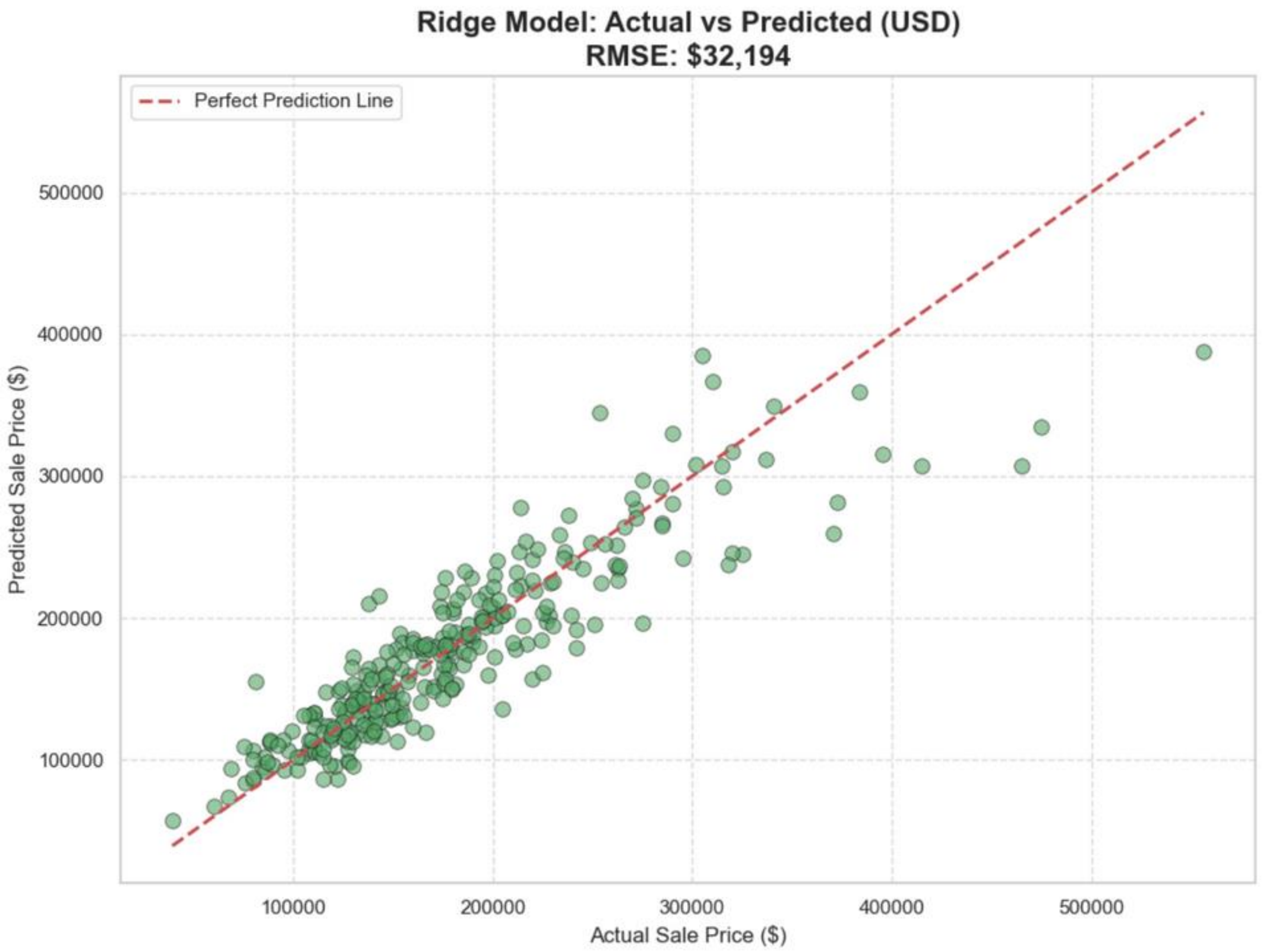


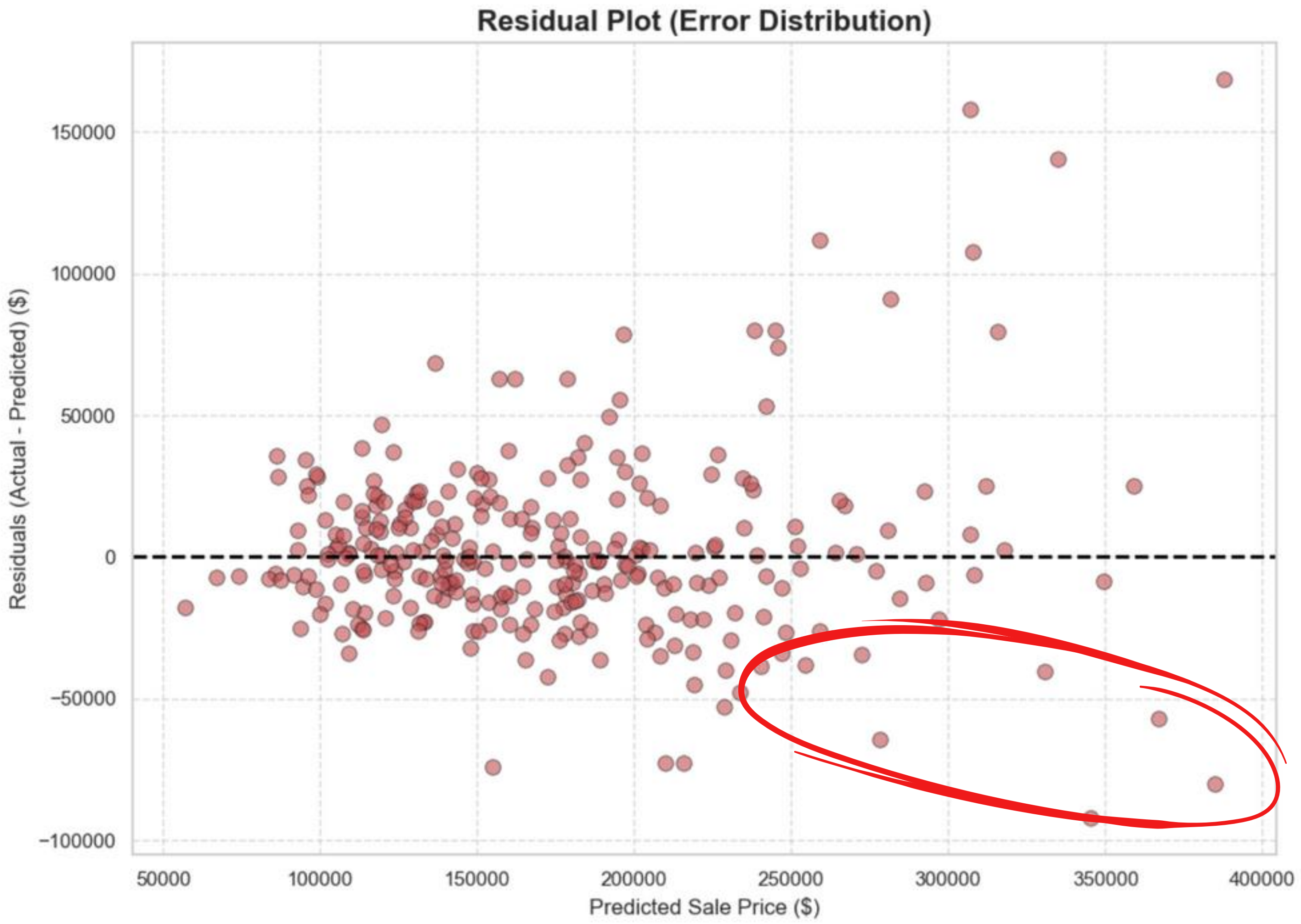
Training

SalePrice Prediction by Ridge

Limitations of Ridge model:

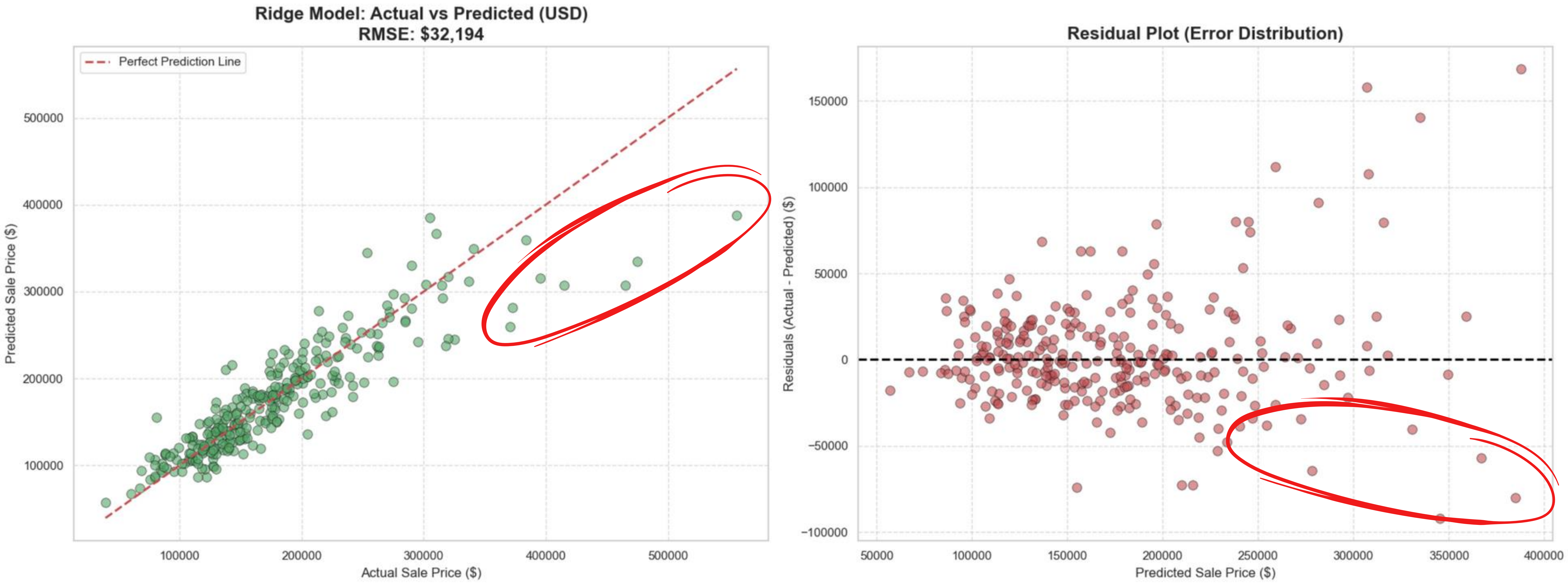
- The Residual Plot (right) clearly shows a funnel shape, widening significantly toward the right side. This confirms that the **model's error (variance) increases sharply as the home value rises**





Training

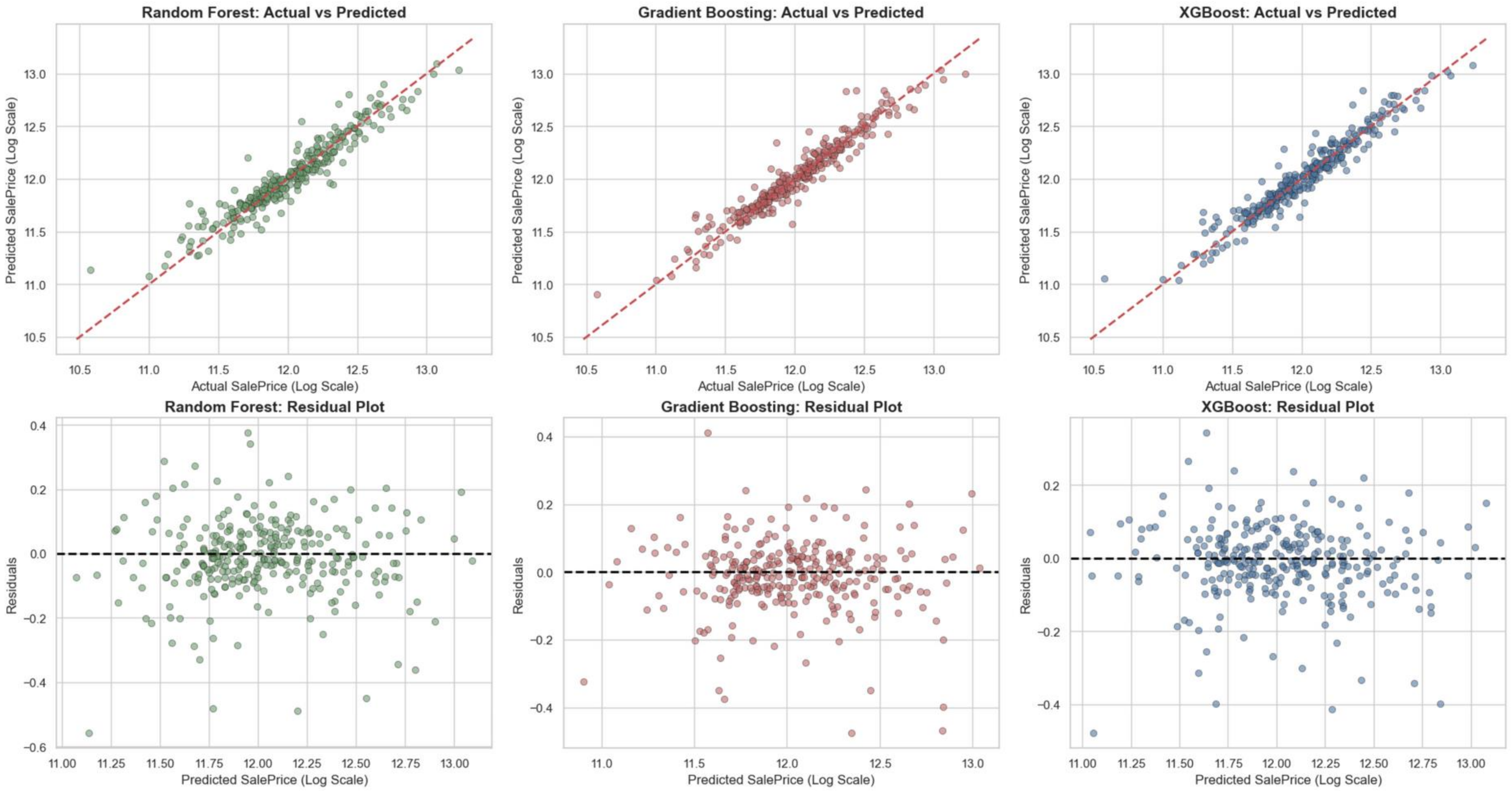
SalePrice Prediction by Ridge



next step: **deploy Tree-based models such as Random Forest or XGBoost.**
These algorithms are inherently capable of automatically learning complex non-linear interactions and are expected to significantly reduce the prediction error, especially within the high-value range.

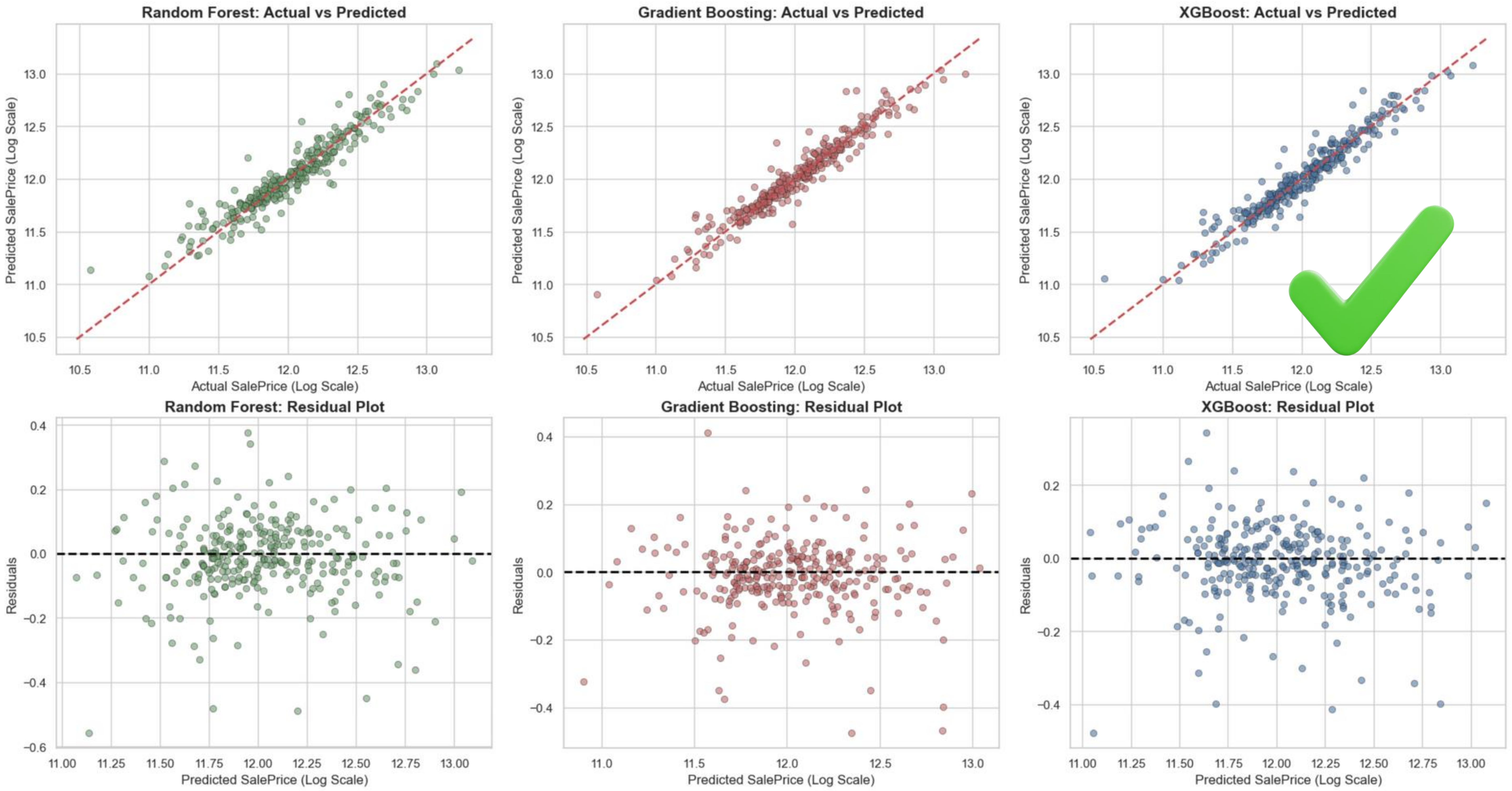
Training

Random Forest & Gradient Boosting & XGBoost



Training

Random Forest & Gradient Boosting & XGBoost



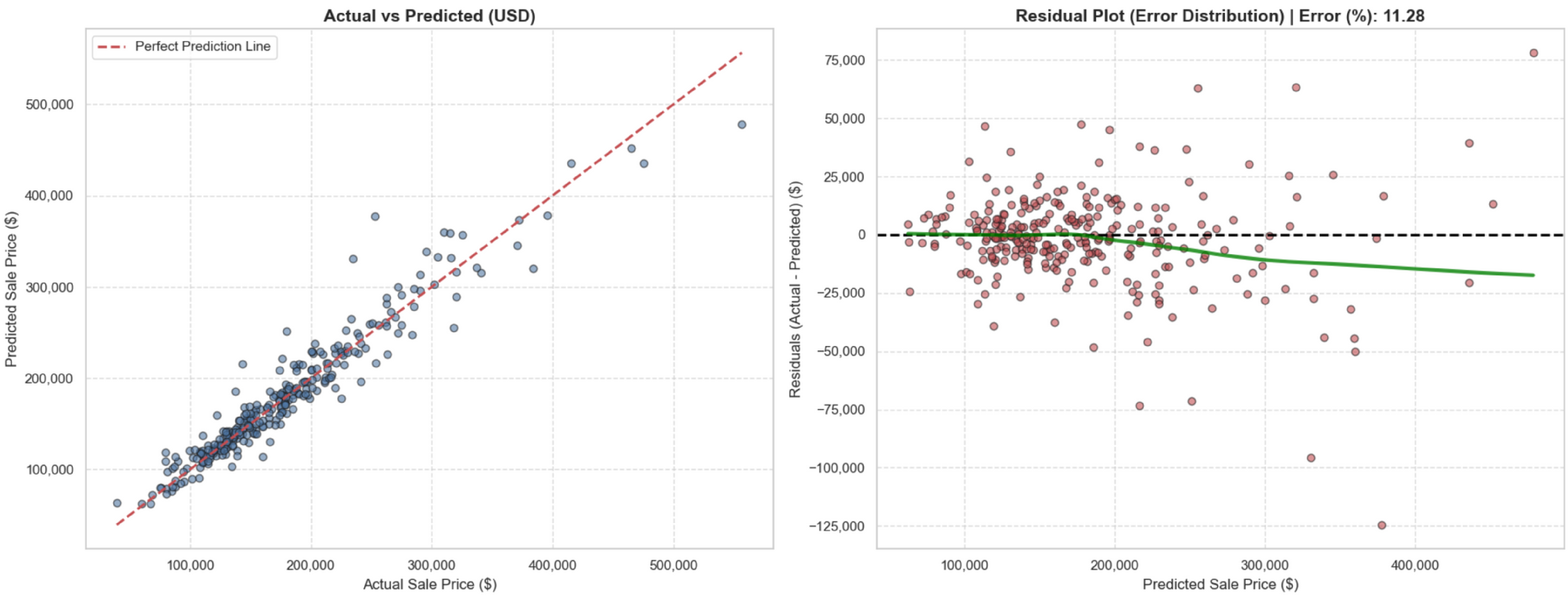
Model	RMSE (Log Scale)
XGBoost	0.10684
Gradient Boosting	0.106956
Random Forest	0.122441

Training

Random Forest & Gradient Boosting & XGBoost

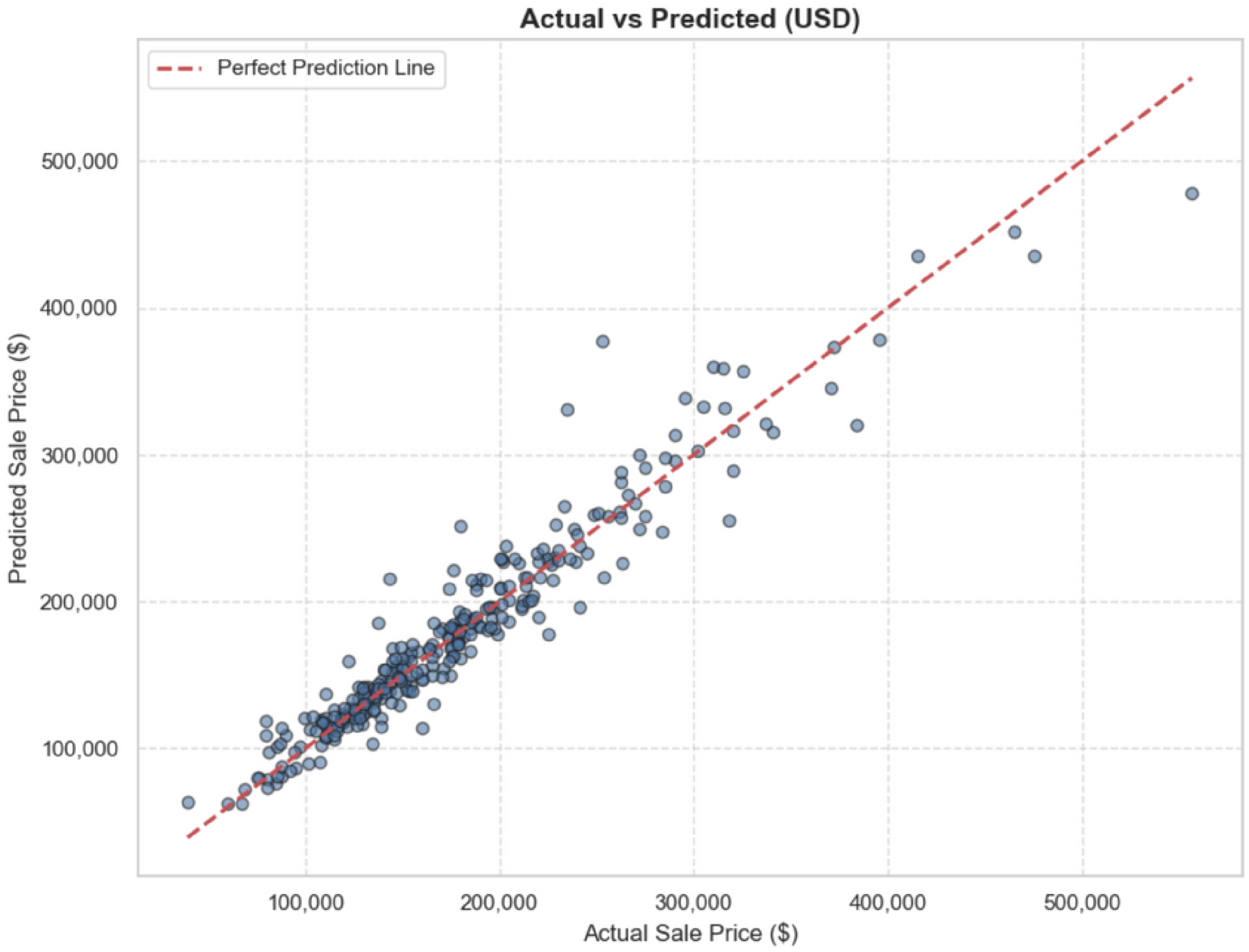
We try printing out the first 10 House price between Actual and Prediction generated by XGBoost.

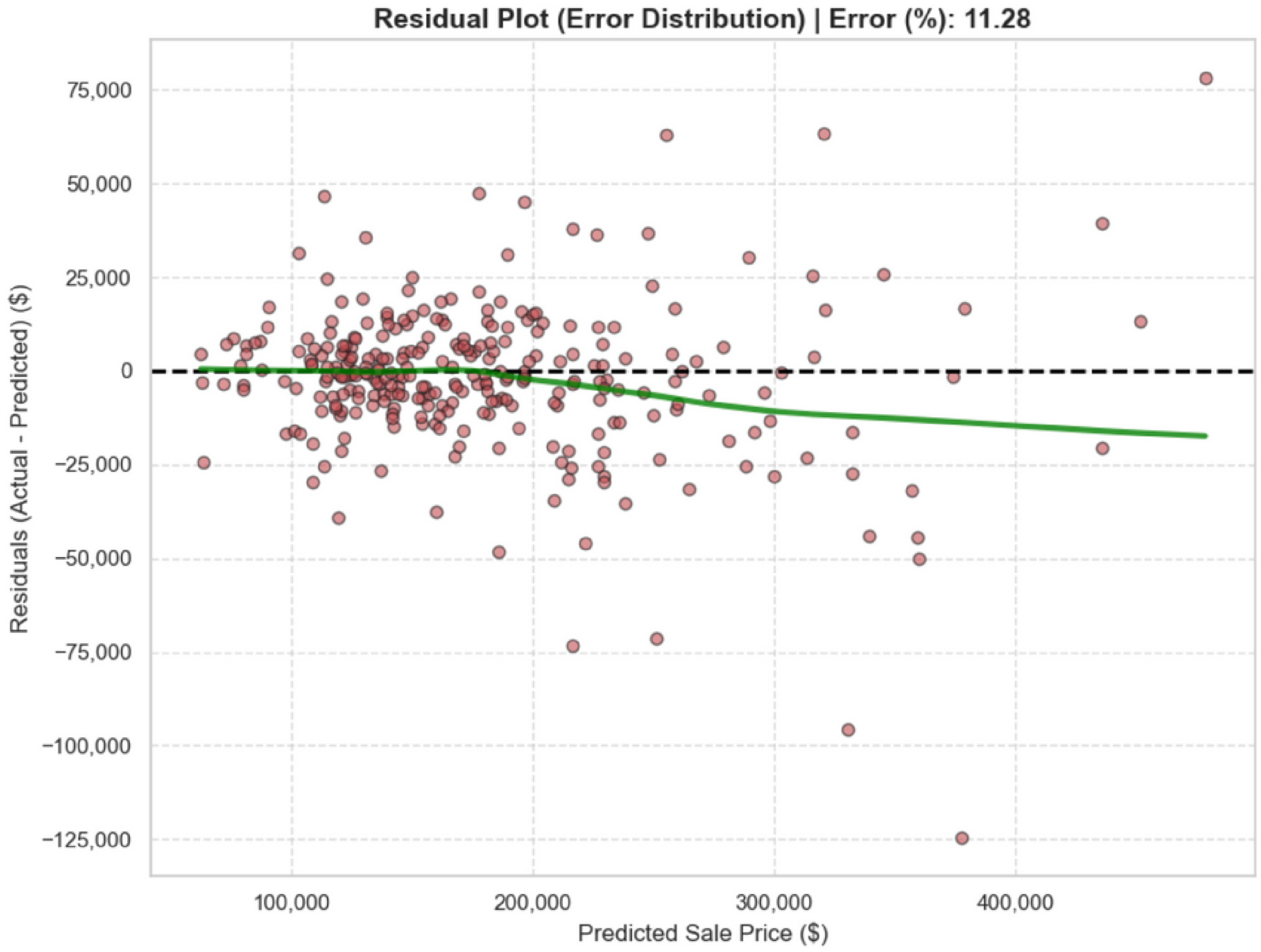
	Actual_Price ($)	Predicted_Price ($)
538	133000.0	136339.312500
754	127500.0	133970.421875
49	177000.0	180422.593750
1380	124000.0	124939.726562
141	166000.0	130506.906250
614	305000.0	332511.500000
1050	220000.0	189102.218750
793	175000.0	150086.093750
1006	227000.0	214997.859375
1323	125000.0	123034.851562



Training

Random Forest & Gradient Boosting & XGBoost



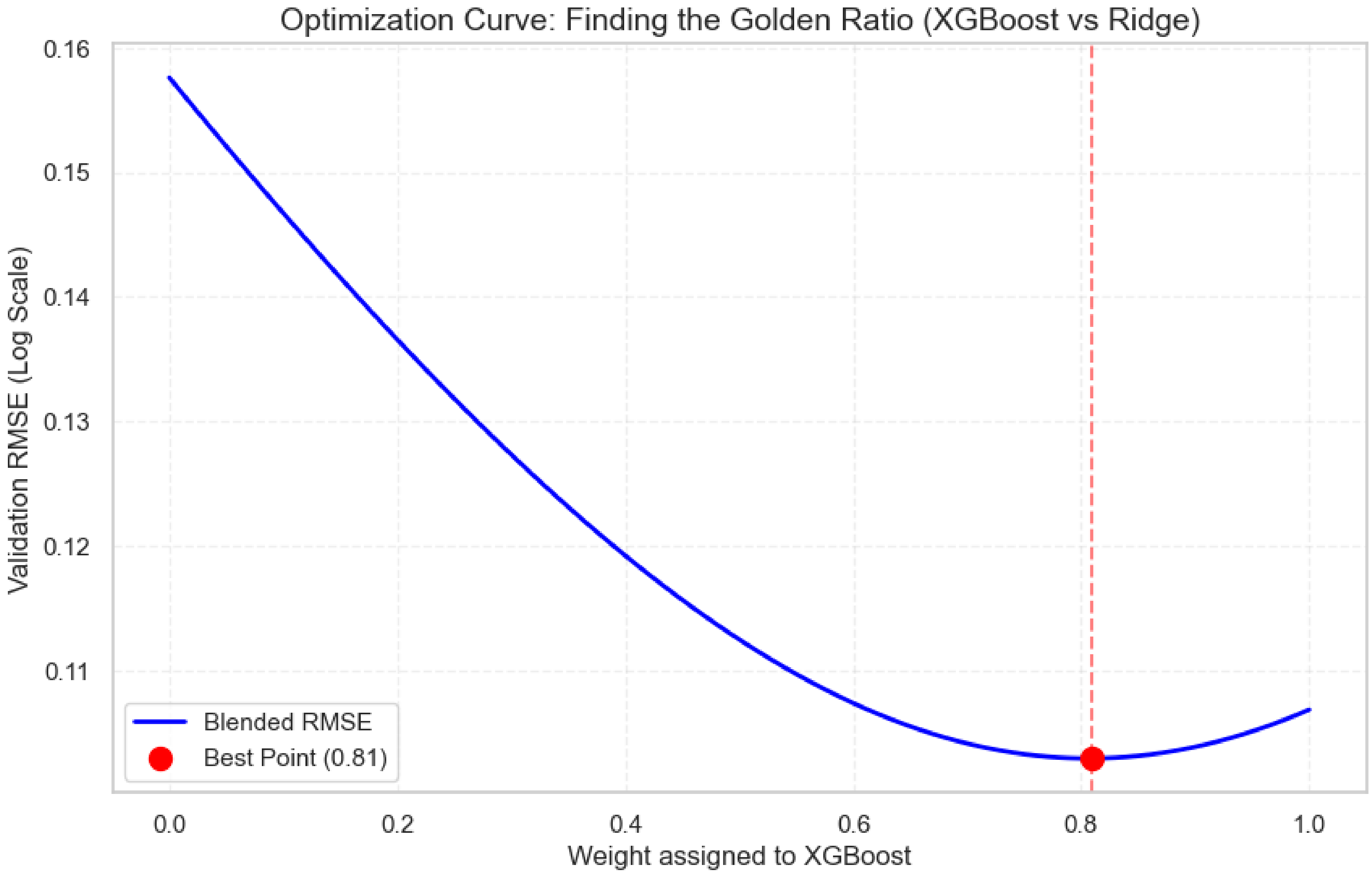


The next step to maximize our score is Blended Ensemble.
We will **combine** the stability of **Ridge** with the predictive power of **XGBoost**

Training

Blending model: XGBoost + Ridge

We try looking for the best combination of weights between Ridge and XGBoost.

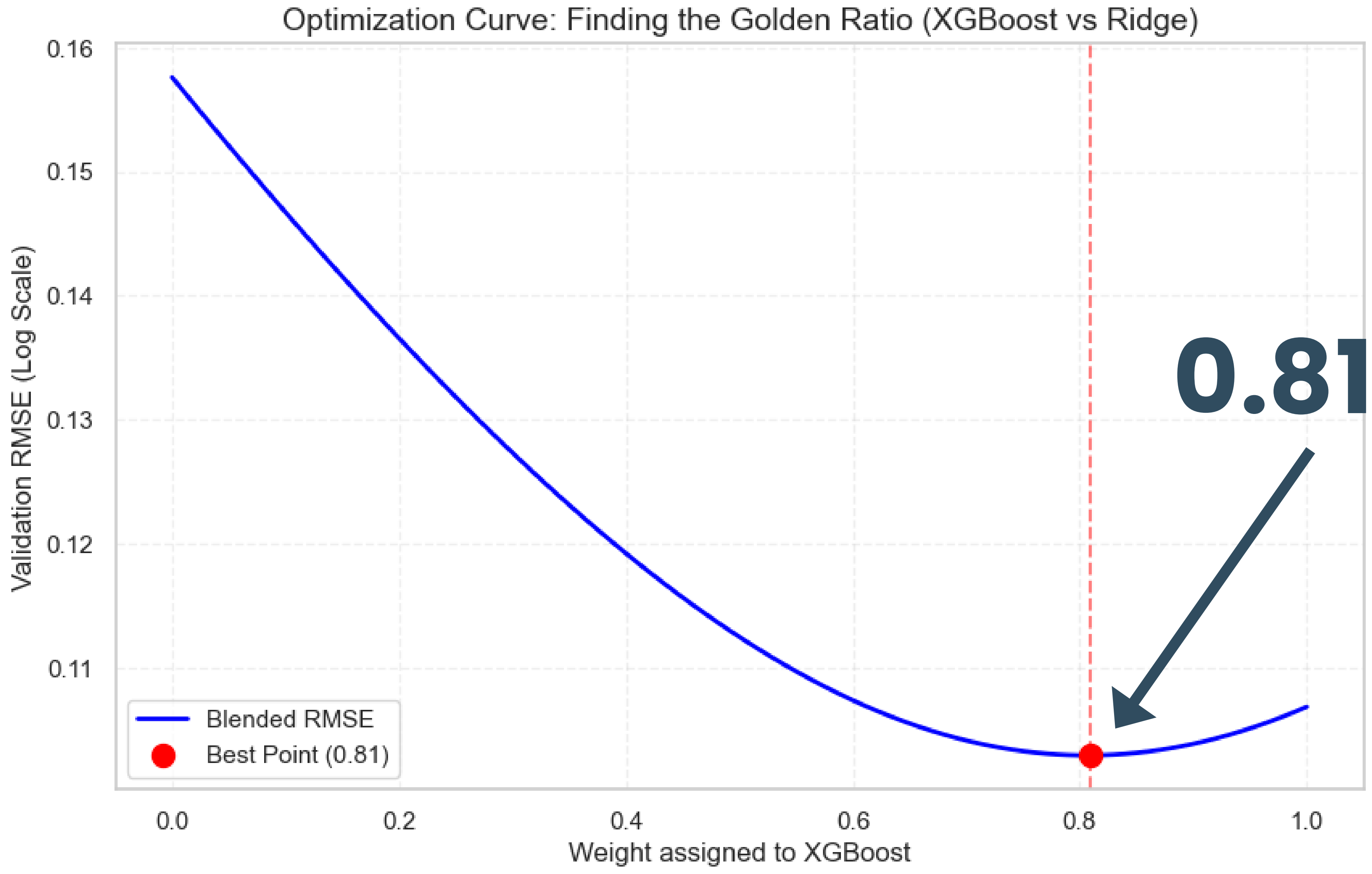



```
1. XGBoost RMSE:          0.10684
2. Ridge RMSE:            0.15761

--- Searching for Optimal Weights ---
----------------------------------------
OPTIMAL BLENDING FOUND!
Best Weight for XGBoost:          0.81 (81%)
Best Weight for Ridge:            0.19 (19%)
----------------------------------------
Best Blended RMSE: 0.10293
Improvement over XGBoost only: 0.00391
Improvement over Ridge only:   0.05469
```

Training

Blending model: XGBoost + Ridge

We try looking for the best combination of weights between Ridge and XGBoost.



0.81*XGBoost + 0.19*Ridge

Training

Blending model: XGBoost + Ridge

We try printing out the first 10 House price between Actual and Prediction generated by Blending model.

	Actual_Price	Ridge_Price	XGB_Price	Blended_Price
538	133000.0	141931.66	136339.312500	137384.58
754	127500.0	99194.86	133970.421875	126534.81
49	177000.0	178074.54	180422.593750	179974.15
1380	124000.0	138987.00	124939.726562	127494.82
141	166000.0	119258.46	130506.906250	128291.00
614	305000.0	385005.16	332511.500000	341902.42
1050	220000.0	157041.84	189102.218750	182543.90
793	175000.0	186580.89	150086.093750	156422.97
1006	227000.0	196898.16	214997.859375	211435.41
1323	125000.0	131566.28	123034.851562	124612.12

```
--- FINAL BLENDED ENSEMBLE COMPARISON (USD) ---
Blending Formula: 0.81 * XGBoost + 0.19 * Ridge
Blended RMSE (Log Scale): 0.10293
Estimated Monetary Error: 10.84%
```

Testing

Deploy transformation pipeline on Test set

Before running the defined blending model on test set, we must apply preprocessing functions on test set in order to make them aligned with train set. Once being matched, test can be undergone the final model.

```
# Apply Missing Value Processing function to the test set
test = clean_missing_values(test)
```

```
Columns with remaining missing values (filled with Median/Mode): ['MSZoning', 'Utilities', 'Exterior1st', 'Exterior2nd', 'KitchenQual', 'Functional', 'Sale
Type']
```

```
# Apply Feature Engineering fuction to the test set
test = engineer_features(test)
```

```
# Apply Log Transformation function to the test set
test, _ = add_log_transformed_features(test, exclude_cols=['Id'])
```

```
Detected 28 skewed features (skew > 0.75)
```

```
# Apply Existence Flags Creation function to the test set
test = create_existence_flags(test)
```

```
# Apply Process of Ordinal and Nominal features function to the test set
test = process_ordinal_and_nominal_features(test)
```

```
def align_test_data(test, OHE_columns_list, categorical_cols_from_train):
    # categorical_cols = test.select_dtypes(include='object').columns

    X_test_ohe = pd.get_dummies(test, columns=categorical_cols_from_train, drop_first=True)

    X_test_aligned = X_test_ohe.reindex(columns=OHE_columns_list, fill_value=0)

    X_test_aligned = X_test_aligned.fillna(0)
    return X_test_aligned

X_test_aligned = align_test_data(test, OHE_columns_list, categorical_cols)
```

Testing

Prediction on Test set

We use the blending model trained in previously cells, deploying them with the test set.

We complete predicting the price of each house in this dataset.

	Id	SalePrice
0	1461	131522.194361
1	1462	160824.707591
2	1463	185197.097681
3	1464	199443.563009
4	1465	175812.078721
5	1466	174726.568021
6	1467	181953.815145
7	1468	174900.726678
8	1469	186110.538209
9	1470	125908.049548
10	1471	197413.259260
11	1472	98690.997684
12	1473	104245.938943
13	1474	150443.646480
14	1475	116910.071165
15	1476	342000.072930
16	1477	250237.259946
17	1478	287684.264347
18	1479	276391.279324
19	1480	549653.388782

House Segmentation

In order to make more precise predictions on the “outlier” houses which can not be predicted well by the blending of Ridge and XGBoost. We will segment houses, classifying them based on their characteristics. After profiling houses, we may predict the technically accurate price of houses.

House Segmentation

In order to make more precise predictions on the “outlier” houses which can not be predicted well by the blending of Ridge and XGBoost. We will segment houses, classifying them based on their characteristics. After profiling houses, we may predict the technically accurate price of houses.

Technical procedure of House Segmentation:

Data Preprocessing

→ **Feature Engineering**

→ **Model Testing and Selection**

→ **Weight Selection of Blending Model with Profiling Data**

→ **Model Validation Check**

→ **Final Model on Test set**

House Segmentation Preprocessing on Train set

Following exactly same as previous processing pipelines on the train set in the previous Supervised Learning model.

```
# Missing Value Processing
train = clean_missing_values(train)
```

```
# Feature Engineering
train = engineer_features(train)
```

House Segmentation

Preprocessing on Train set

Following exactly same as previous processing pipelines on the train set in the previous Supervised Learning model.

```
# Missing Value Processing
train = clean_missing_values(train)
```

```
# Feature Engineering
train = engineer_features(train)
```

For Unsupervised learning, we don’t deploy nominal conversion on train set because we don’t use the method One-hot Encoding for this type of learning, just keeping using ordinal transformation.

```
def process_ordinal_features(df):
    df = df.copy()
    # 1. Standard scale (Ex -> Po)
    # Note: 'None' here is created by the previous fillna('None') step
    quality_map = {'Ex': 5, 'Gd': 4, 'Ta': 3, 'Fa': 2, 'Po': 1, 'None': 0}
    standard_qual_cols = ['ExterQual', 'ExterCond', 'BsmtQual', 'BsmtCond',
                          'HeatingQC', 'KitchenQual', 'FireplaceQu',
                          'GarageQual', 'GarageCond', 'PoolQC']

    for col in standard_qual_cols:
        if col in df.columns:
            df[col] = df[col].map(quality_map).fillna(0)

    # 2. Basement Exposure scale
    exposure_map = {'Gd': 4, 'Av': 3, 'Mn': 2, 'No': 1, 'None': 0}
    if 'BsmtExposure' in df.columns:
        df['BsmtExposure'] = df['BsmtExposure'].map(exposure_map).fillna(0)

    # 3. Basement Finish scale (GLQ -> Unf)
    bsmt_fin_map = {'GLQ': 6, 'ALQ': 5, 'BLQ': 4, 'Rec': 3, 'LwQ': 2, 'Unf': 1, 'None': 0}
    for col in ['BsmtFinType1', 'BsmtFinType2']:
        if col in df.columns:
            df[col] = df[col].map(bsmt_fin_map).fillna(0)

    # 4. Garage Finish scale
    garage_fin_map = {'Fin': 3, 'RFn': 2, 'Unf': 1, 'None': 0}
    if 'GarageFinish' in df.columns:
        df['GarageFinish'] = df['GarageFinish'].map(garage_fin_map).fillna(0)

    # 5. Functional scale
    func_map = {'Typ': 7, 'Min1': 6, 'Min2': 5, 'Mod': 4, 'Maj1': 3, 'Maj2': 2, 'Sev': 1, 'Sal': 0}
    if 'Functional' in df.columns:
        df['Functional'] = df['Functional'].fillna('Typ') # Fill NaN with Mode before mapping
        df['Functional'] = df['Functional'].map(func_map)

    # 6. Fence scale
    fence_map = {'GdPrv': 4, 'MnPrv': 3, 'GdWo': 2, 'MnWw': 1, 'None': 0}
    if 'Fence' in df.columns:
        df['Fence'] = df['Fence'].map(fence_map).fillna(0)

    # 7. LotShape (Newly added)
    # Reg (Regular) > IR1 > IR2 > IR3 (Irregular)
    lot_shape_map = {'Reg': 4, 'IR1': 3, 'IR2': 2, 'IR3': 1}
    if 'LotShape' in df.columns:
        df['LotShape'] = df['LotShape'].map(lot_shape_map).fillna(0)

    # 8. LandSlope (Newly added)
    # Gtl (Gentle) > Mod > Sev (Severe)
    slope_map = {'Gtl': 3, 'Mod': 2, 'Sev': 1}
    if 'LandSlope' in df.columns:
        df['LandSlope'] = df['LandSlope'].map(slope_map).fillna(0)

    # 9. PavedDrive (Newly added)
    # Y (Paved) > P (Partial) > N (Dirt)
    paved_map = {'Y': 3, 'P': 2, 'N': 1}
    if 'PavedDrive' in df.columns:
        df['PavedDrive'] = df['PavedDrive'].map(paved_map).fillna(0)

    # 10. CentralAir (Newly added)
    if 'CentralAir' in df.columns:
        # Only map if it's currently string type (avoid errors if run twice)
        if df['CentralAir'].dtype == 'object':
            df['CentralAir'] = df['CentralAir'].map({'Y': 1, 'N': 0}).fillna(0)

    return df

train = process_ordinal_features(train)
```

House Segmentation Preprocessing on Train set

Following exactly same as previous processing pipelines on the train set in the previous Supervised Learning model.

```python
# Missing Value Processing
train = clean_missing_values(train)
```

```python
# Feature Engineering
train = engineer_features(train)
```

For Unsupervised learning, we don't deploy nominal conversion on train set because we don't use the method One-hot Encoding for this type of learning, just keeping using ordinal transformation.

```python
def process_ordinal_features(df):
    df = df.copy()
    # 1. Standard scale (Ex -> Po)
    # Note: 'None' here is created by the previous fillna('None') step
    quality_map = {'Ex': 5, 'Gd': 4, 'Ta': 3, 'Fa': 2, 'Po': 1, 'None': 0}
    standard_qual_cols = ['ExterQual', 'ExterCond', 'BsmtQual', 'BsmtCond',
                          'HeatingQC', 'KitchenQual', 'FireplaceQu',
                          'GarageQual', 'GarageCond', 'PoolQC']

    for col in standard_qual_cols:
        if col in df.columns:
            df[col] = df[col].map(quality_map).fillna(0)

    # 2. Basement Exposure scale
    exposure_map = {'Gd': 4, 'Av': 3, 'Mn': 2, 'No': 1, 'None': 0}
    if 'BsmtExposure' in df.columns:
        df['BsmtExposure'] = df['BsmtExposure'].map(exposure_map).fillna(0)

    # 3. Basement Finish scale (GLQ -> Unf)
    bsmt_fin_map = {'GLQ': 6, 'ALQ': 5, 'BLQ': 4, 'Rec': 3, 'LwQ': 2, 'Unf': 1, 'None': 0}
    for col in ['BsmtFinType1', 'BsmtFinType2']:
        if col in df.columns:
            df[col] = df[col].map(bsmt_fin_map).fillna(0)

    # 4. Garage Finish scale
    garage_fin_map = {'Fin': 3, 'RFn': 2, 'Unf': 1, 'None': 0}
    if 'GarageFinish' in df.columns:
        df['GarageFinish'] = df['GarageFinish'].map(garage_fin_map).fillna(0)

    # 5. Functional scale
    func_map = {'Typ': 7, 'Min1': 6, 'Min2': 5, 'Mod': 4, 'Maj1': 3, 'Maj2': 2, 'Sev': 1, 'Sal': 0}
    if 'Functional' in df.columns:
        df['Functional'] = df['Functional'].fillna('Typ') # Fill NaN with Mode before mapping
        df['Functional'] = df['Functional'].map(func_map)
```

```python
    # 6. Fence scale
    fence_map = {'GdPrv': 4, 'MnPrv': 3, 'GdWo': 2, 'MnWw': 1, 'None': 0}
    if 'Fence' in df.columns:
        df['Fence'] = df['Fence'].map(fence_map).fillna(0)

    # 7. LotShape (Newly added)
    # Reg (Regular) > IR1 > IR2 > IR3 (Irregular)
    lot_shape_map = {'Reg': 4, 'IR1': 3, 'IR2': 2, 'IR3': 1}
    if 'LotShape' in df.columns:
        df['LotShape'] = df['LotShape'].map(lot_shape_map).fillna(0)

    # 8. LandSlope (Newly added)
    # Gtl (Gentle) > Mod > Sev (Severe)
    slope_map = {'Gtl': 3, 'Mod': 2, 'Sev': 1}
    if 'LandSlope' in df.columns:
        df['LandSlope'] = df['LandSlope'].map(slope_map).fillna(0)

    # 9. PavedDrive (Newly added)
    # Y (Paved) > P (Partial) > N (Dirt)
    paved_map = {'Y': 3, 'P': 2, 'N': 1}
    if 'PavedDrive' in df.columns:
        df['PavedDrive'] = df['PavedDrive'].map(paved_map).fillna(0)

    # 10. CentralAir (Newly added)
    if 'CentralAir' in df.columns:
        # Only map if it's currently string type (avoid errors if run twice)
        if df['CentralAir'].dtype == 'object':
            df['CentralAir'] = df['CentralAir'].map({'Y': 1, 'N': 0}).fillna(0)

    return df

train = process_ordinal_features(train)
```

```python
train = create_existence_flags(train)
```

House Segmentation Featuring Engineering

We select only technically appropriate features used for Clustering methods.

```
final_cluster_cols = [
    # QUALITY (Ordinal/Count)
    'OverallQual',      # Overall material and finish quality (1-10) - Most important
    'OverallCond',      # Overall condition rating (1-10)
    'ExterQual',        # Exterior material quality (Mapped 0-5)
    'KitchenQual',      # Kitchen quality (Mapped 0-5)
    'FireplaceQu',      # Fireplace quality (Mapped 0-5)
    'GarageCars',       # Size of garage in car capacity (Count)
    'FullBath',         # Full bathrooms above grade (Count)
    'TotalBath',        # Total bathrooms (Engineered)
    'TotalRooms',       # Total rooms above grade (Engineered)
    'Functional',       # Home functionality (Mapped 0-7)

    # SIZE & QUANTITATIVE (Continuous)
    'GrLivArea',        # Above grade (ground) living area square feet
    'TotalBsmtSF',      # Total square feet of basement area
    'TotalSF',          # Total square footage (Engineered)
    'LotArea',          # Lot size in square feet
    'MasVnrArea',       # Masonry veneer area in square feet

    # AGE & TIME
    'YearBuilt',        # Original construction year
    'YearRemodAdd',     # Remodel year (same as construction if no remodeling)
    'HouseAge',         # House age in years (Engineered)

    # EXISTENCE FLAGS (Binary 0/1)
    'Has_PoolArea',
    'Has_GarageArea',
    'Has_Fireplaces',
    'Has_TotalBsmtSF',
    'Has_OpenPorchSF',
    'Has_WoodDeckSF'
]
```

Featuring Engineering

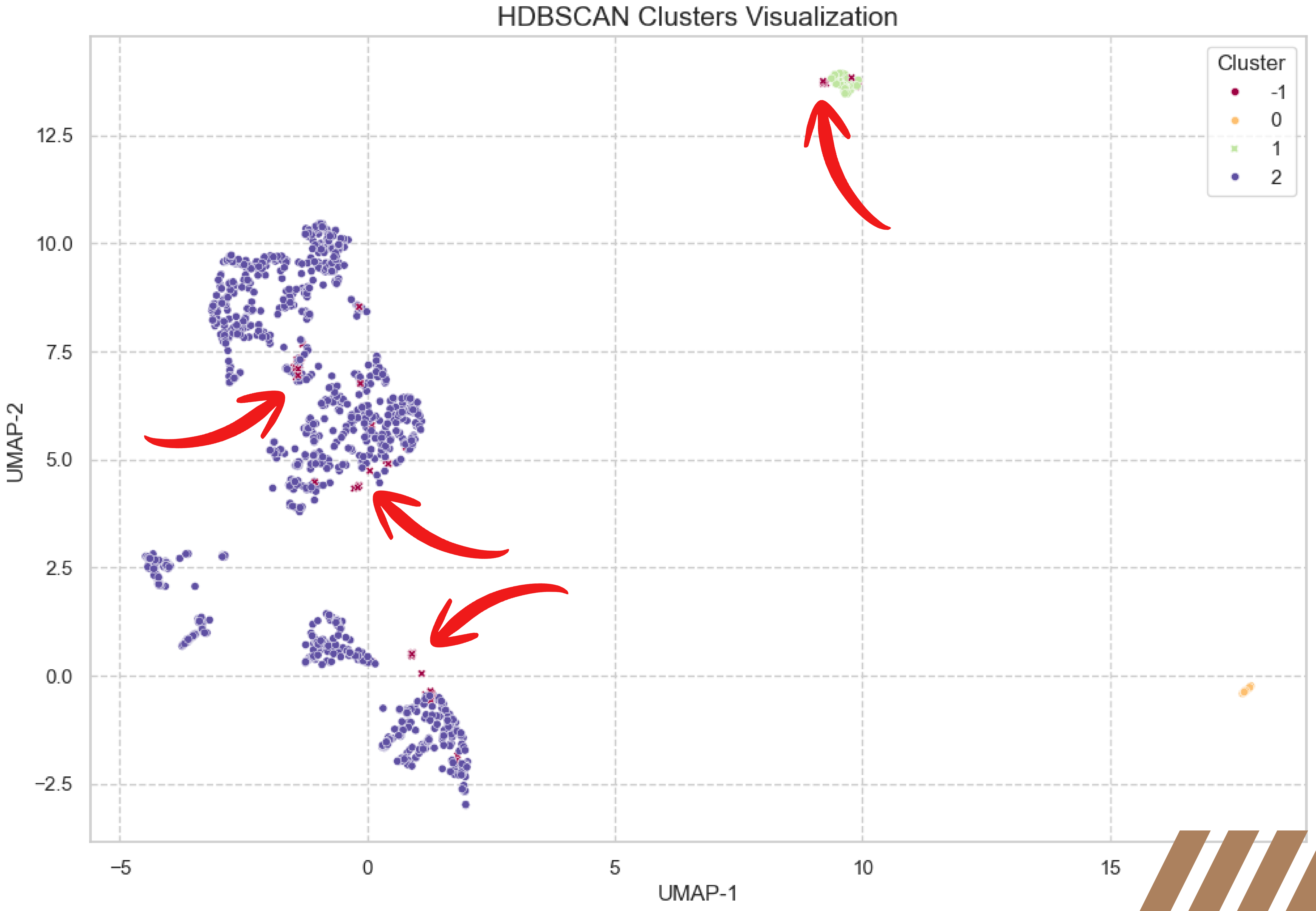
Clustering methods also requires data scaled for avoiding extreme dots' effect.

```
# --- FINAL DATA SELECTION ---
X_cluster = train[final_cluster_cols]

# --- STANDARD SCALING ---
scaler = StandardScaler()
X_cluster_scaled = scaler.fit_transform(X_cluster)
```

House Segmentation

HDBSCAN and Noise Reduction

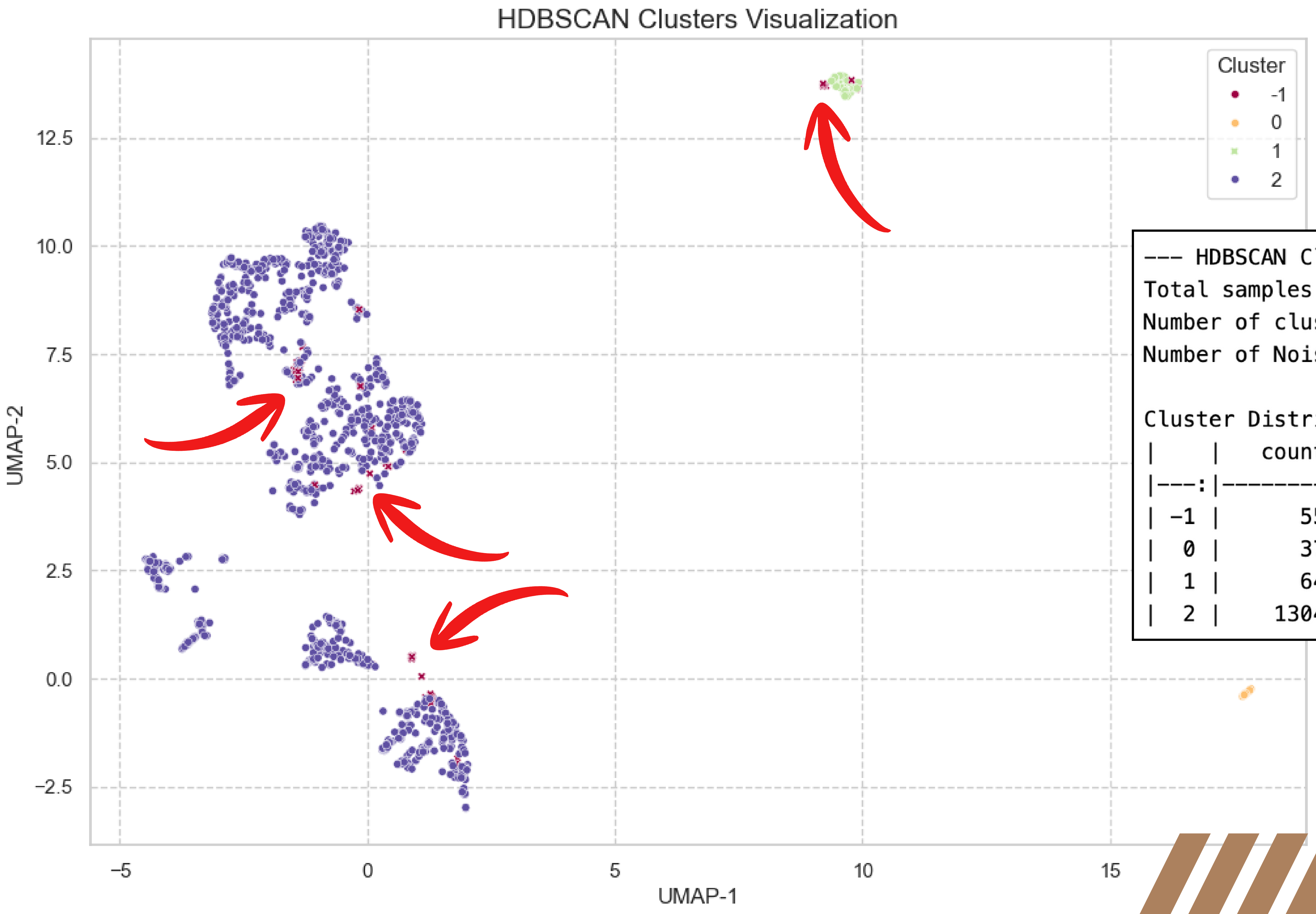
First, to detect noise and remove them out of the dataset, we run HDBSCAN, investigating what constitutes noise cloud.



House Segmentation

HDBSCAN and Noise Reduction

First, to detect noise and remove them out of the dataset, we run HDBSCAN, investigating what constitutes noise cloud.




```
--- HDBSCAN Clustering Results ---
Total samples analyzed: 1460
Number of clusters found (excluding Noise): 3
Number of Noise points (Outliers, label -1): 55

Cluster Distribution (HDBSCAN):
|    |   count |
|---:|--------:|
| -1 |      55 |
|  0 |      37 |
|  1 |      64 |
|  2 |    1304 |
```

HDBSCAN and Noise Reduction

We remove noise (data labelled as -1) and redefine the train set.

```
# --- 1. IDENTIFY AND FILTER NOISE ---
# Store original size for reporting statistics later
n_original = len(X_cluster_scaled)

# Create mask: Select points where cluster label is NOT -1 (Noise)
clean_mask = (hdb_clusters != -1)

# Apply filter: Overwrite the original variable with clean data
X_cluster_scaled = X_cluster_scaled[clean_mask]

# RECOMMENDATION: Update the labels array to match the filtered data
# hdb_clusters = hdb_clusters[clean_mask]

# --- 2. REPORT RESULTS ---
n_removed = n_original - len(X_cluster_scaled)
retention_rate = (len(X_cluster_scaled) / n_original) * 100
```

House Segmentation

HDBSCAN and Noise Reduction

We remove noise (data labelled as -1) and redefine the train set.

```
# --- 1. IDENTIFY AND FILTER NOISE ---
# Store original size for reporting statistics later
n_original = len(X_cluster_scaled)

# Create mask: Select points where cluster label is NOT -1 (Noise)
clean_mask = (hdb_clusters != -1)

# Apply filter: Overwrite the original variable with clean data
X_cluster_scaled = X_cluster_scaled[clean_mask]

# RECOMMENDATION: Update the labels array to match the filtered data
# hdb_clusters = hdb_clusters[clean_mask]

# --- 2. REPORT RESULTS ---
n_removed = n_original - len(X_cluster_scaled)
retention_rate = (len(X_cluster_scaled) / n_original) * 100
```

```
Original samples: 1460
Noise points removed: 55
Retained samples: 1405 (96.23%)
```

House Segmentation

K-Means & Clusters Detection

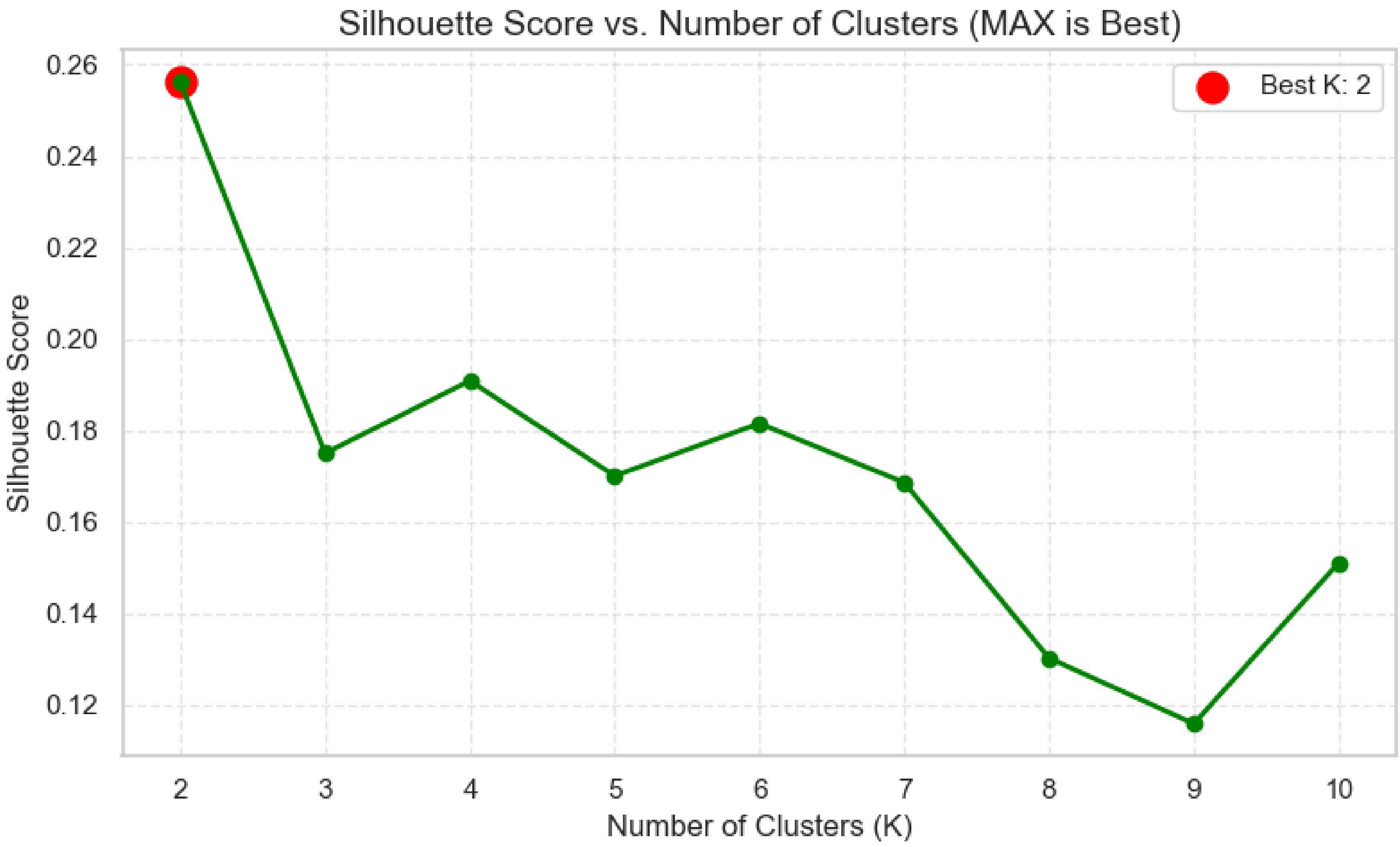
We seek out the number of clusters by deploying K-Means method.

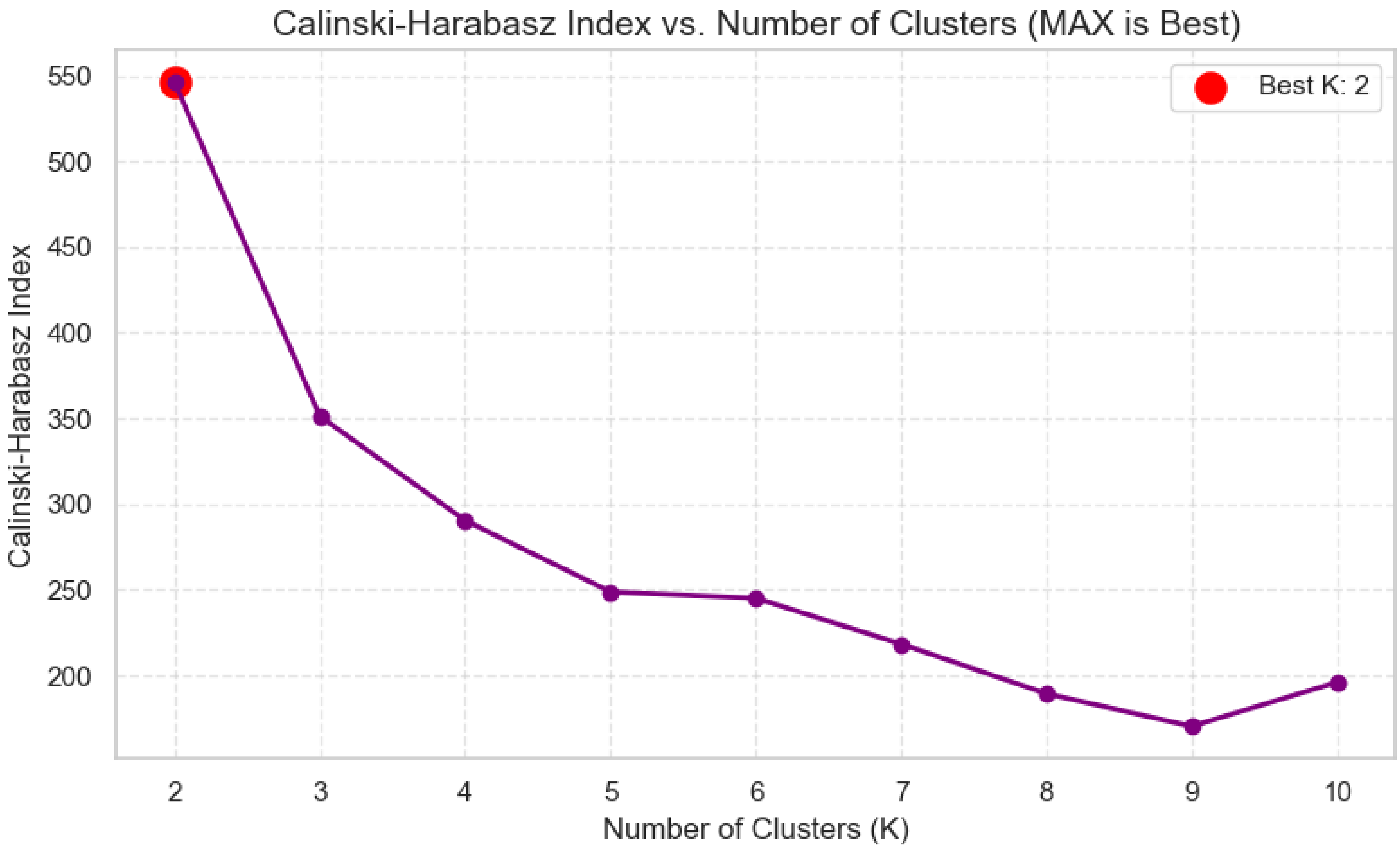
House Segmentation

K-Means & Clusters Detection

We seek out the number of clusters by deploying K-Means method.

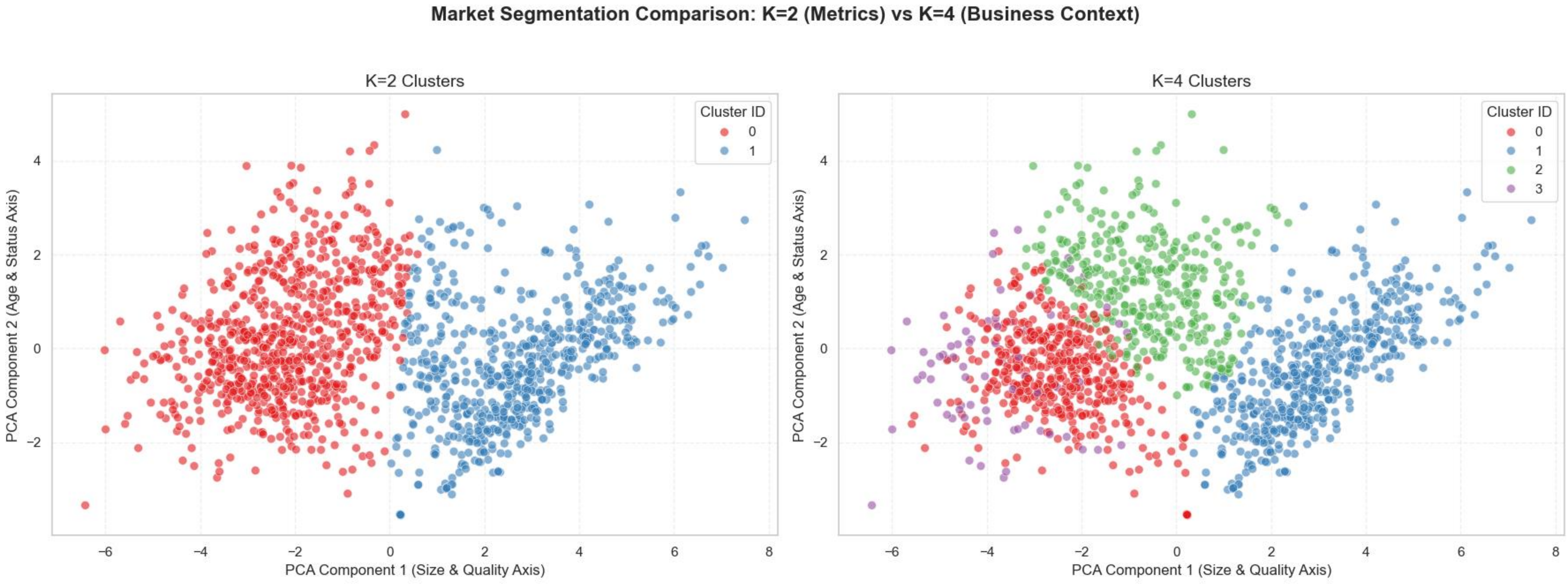
Both **Silhouette and CH-Index prove k=2** as the best and the number of clusters should be 2.





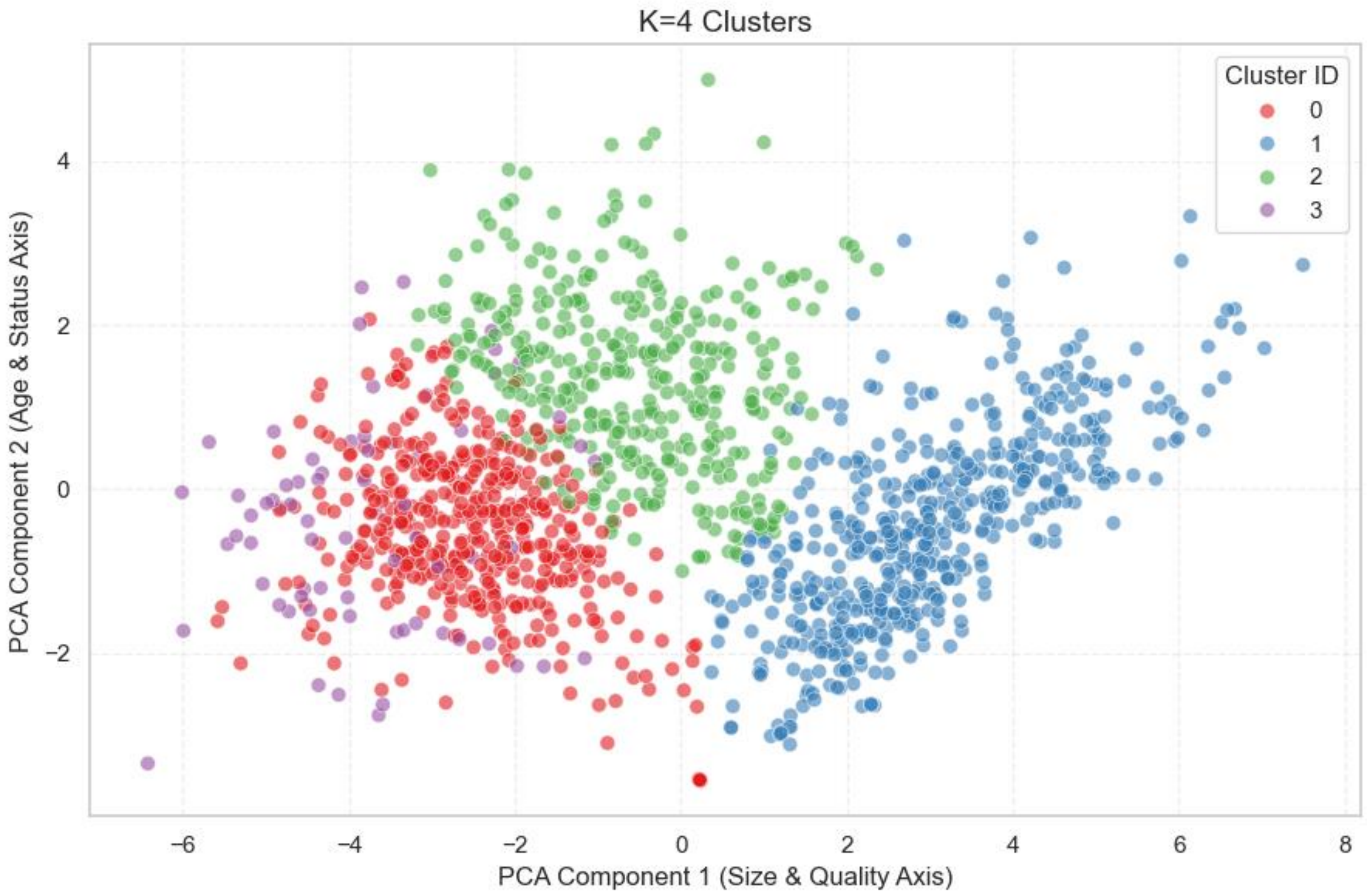
House Segmentation K-Means & Clusters Detection

Under Business landscape, we should consider k=4 for business interpretability and strategic house classification.



House Segmentation K-Means & Clusters Detection

Under Business landscape, we should consider k=4 for business interpretability and strategic house classification.



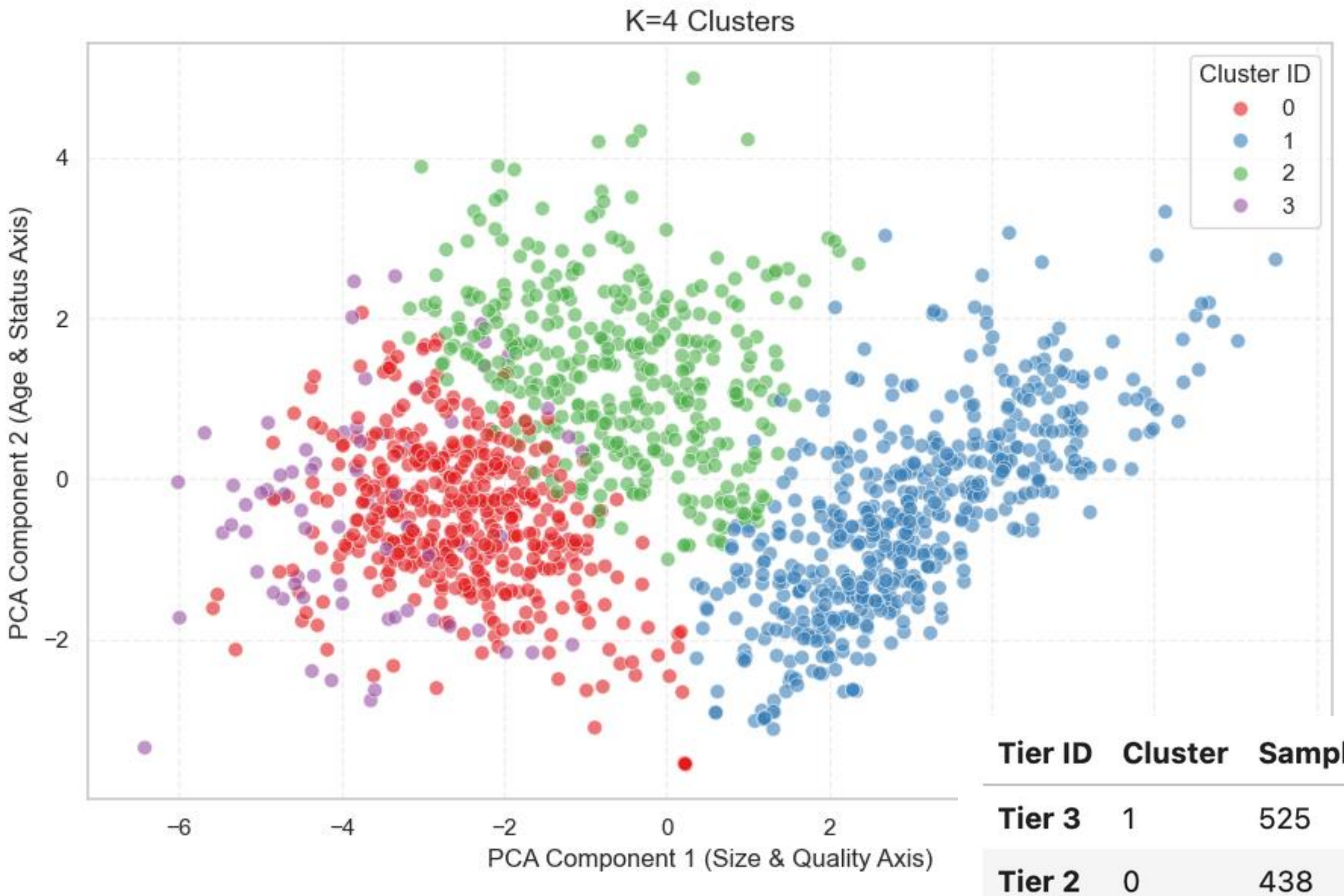
- These 4 clusters represent 4 major value segments that can be described in business terms (Starter, Middle, High-End, Luxury)
- We chose K=4 with a low Silhouette score, we accept less distinct cluster boundaries (Silhouette 0.1909) in exchange for **High Interpretability**.

→ K=4 provides the best balance between market segmentation and data structure. Although the clusters overlap, they still **represent four distinct value regions** along the PCA axes.

House Segmentation

K-Means & Clusters Detection

Under Business landscape, we should consider k=4 for business interpretability and strategic house classification.

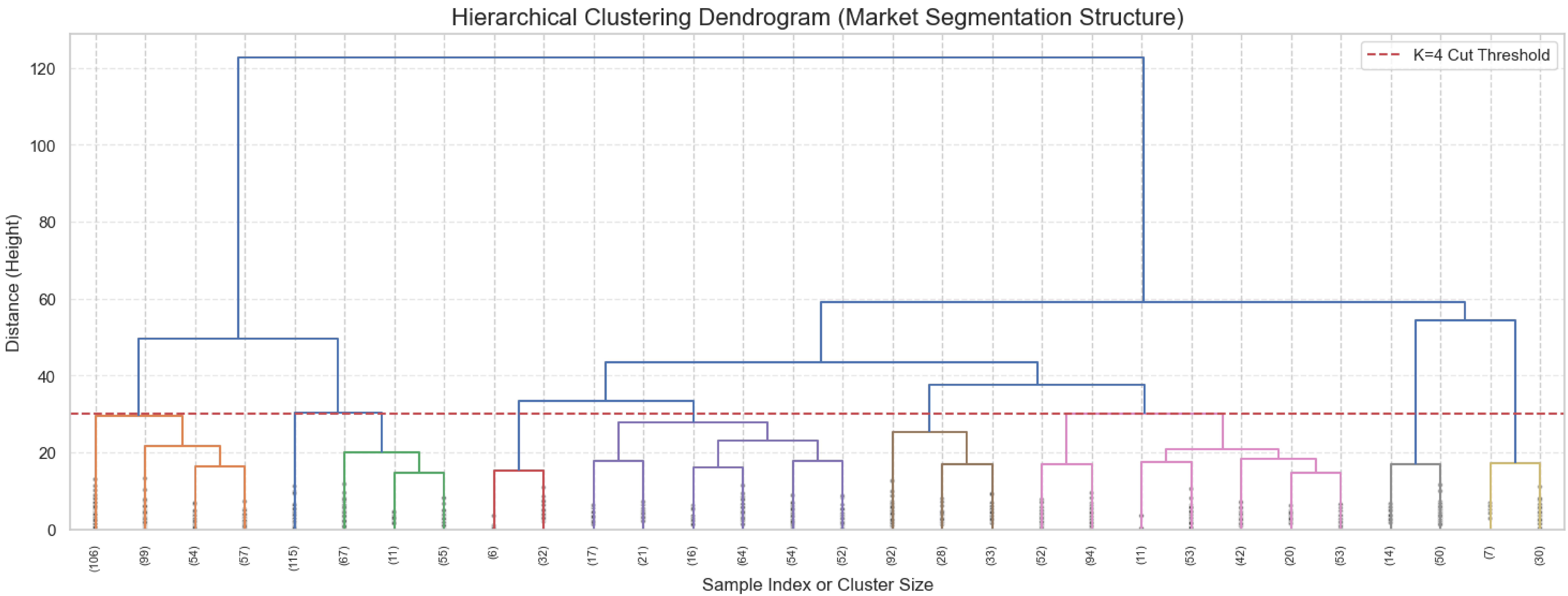


Tier ID	Cluster	Sample Count	Market Share	Structural Interpretation
Tier 3	1	525	37.4%	**Largest Segment.** (Mainstream / Upper-Middle)
Tier 2	0	438	31.2%	**Medium Segment.**
Tier 1	2	371	26.4%	**Starter Segment.**
Tier 0	3	71	5.0%	**Niche Segment.** Group of samples with premium/unique characteristics.

House Segmentation

Hierachical Clustering & Clusters Detection

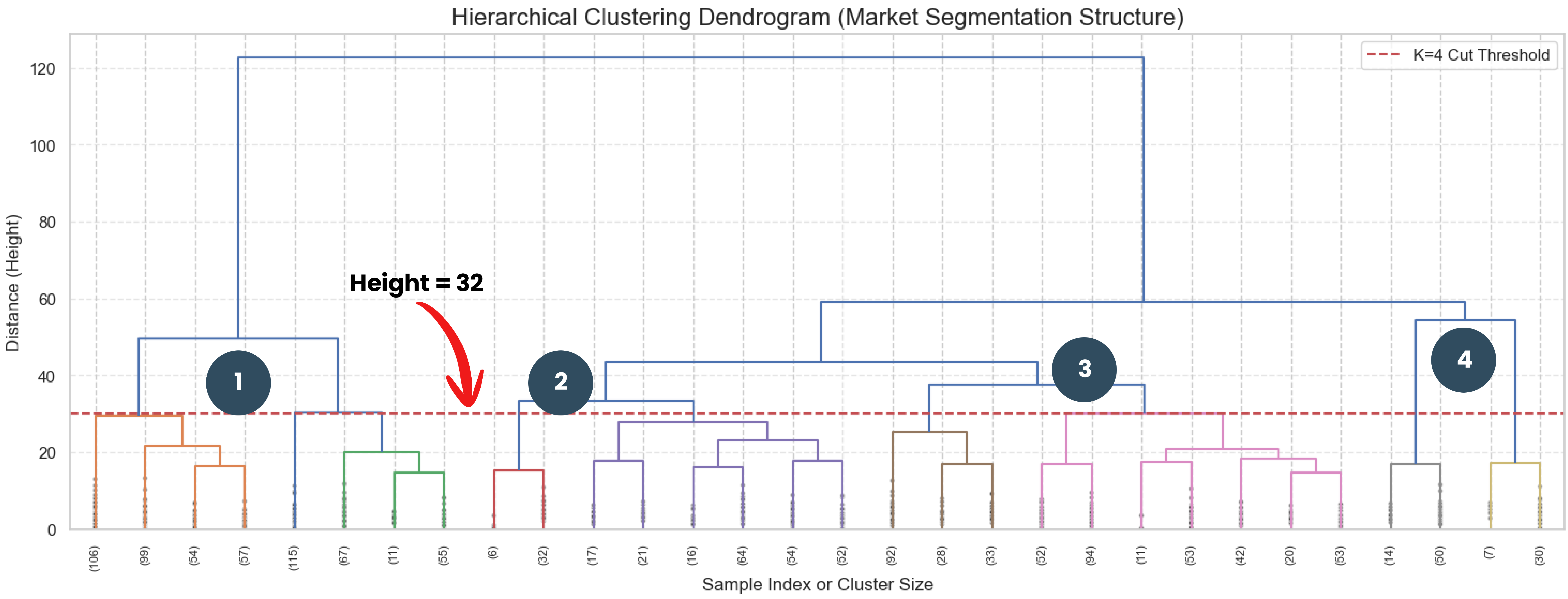
We try deploying Hierachical Clustering to assure if k=4 is optimal.



House Segmentation

Hierachical Clustering & Clusters Detection

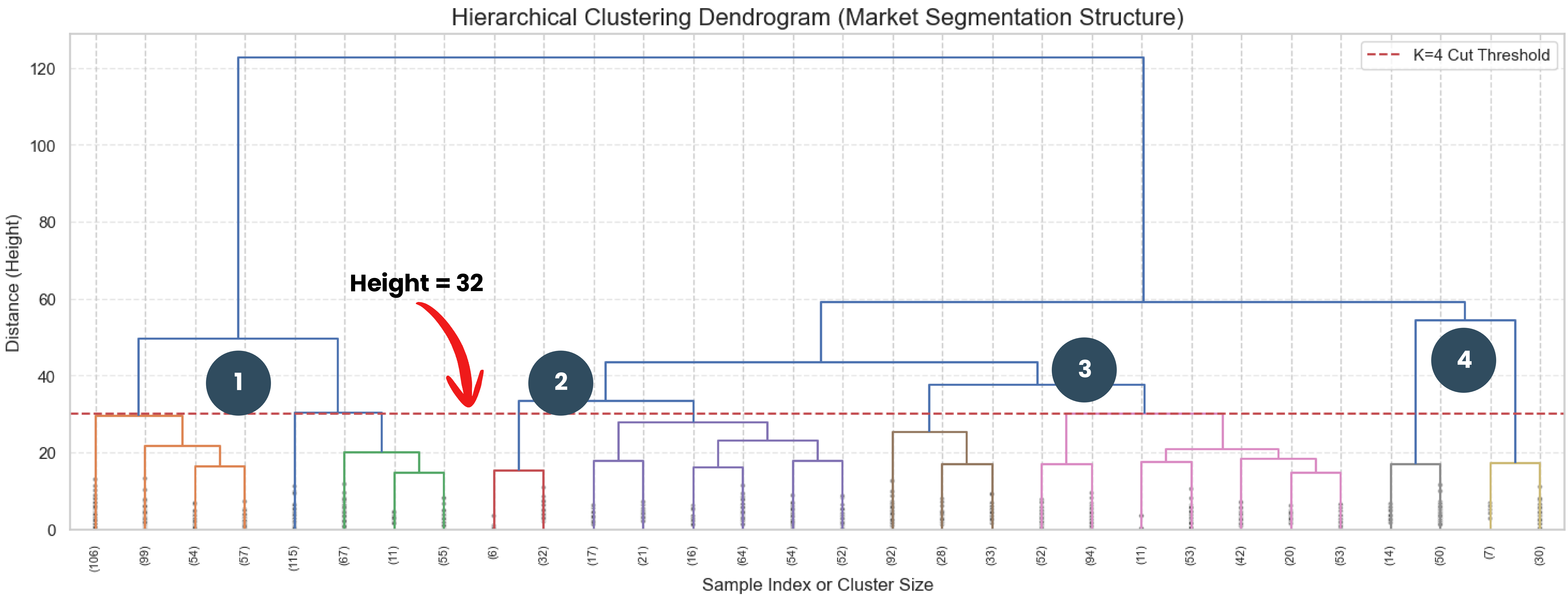
We try deploying Hierachical Clustering to assure if k=4 is optimal.



House Segmentation

Hierachical Clustering & Clusters Detection

We try deploying Hierachical Clustering to assure if k=4 is optimal.



The vertical distance from the cutoff (Height ≈ 32) to the next mergers (Height ≈ 45–50) is relatively large → 4 clusters created at this level are highly distinct (they must travel a significant distance to merge).

House Segmentation

Hierachical Clustering vs K-Means

We compare the number of data points in each cluster defined by both Hierachical and K-Means and opting out which cluster patterns should be chosen for easy and meaningful business interpretation.

Tier (By Size)	K-Means (Centroid-based)	Hierarchical (Distance-based)	Assessment
Tier 1 (Largest)	525	740	**Large Main Block**: HC consolidates most of the market into a single cluster, while K-Means divides it more finely.
Tier 2	438	564	
Tier 3	371	64	**Biggest Difference**: K-Means creates a substantial third cluster (371). HC treats this as smaller groups.
Tier 4 (Niche)	71	37	**Niche Consensus**: Both methods isolate a very small segment, but K-Means expands this segment more.
Total	1405	1405	

⟶ *Hierarchical Clustering tends to merge, creating larger clusters*

⟶ *K-Means provides more detailed segmentation, creating more balanced-sized clusters*

House Segmentation

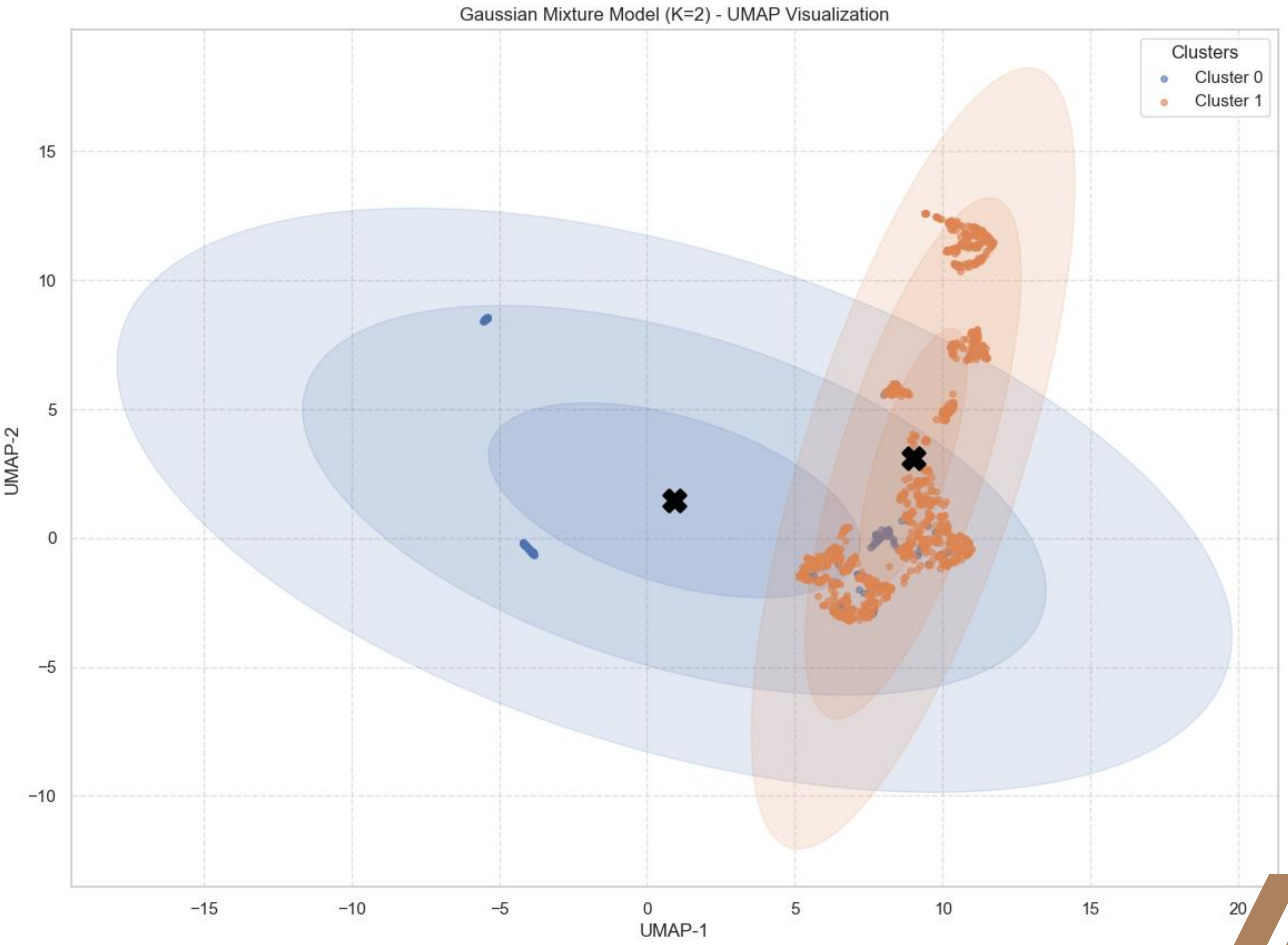
Hierachical Clustering vs K-Means

We choose K-Means clusters as optimal ones and move on to compare with other models.

Tier (By Size)	K-Means (Centroid-based)	Hierarchical (Distance-based)	Assessment
Tier 1 (Largest)	525 ✔	740	**Large Main Block**: HC consolidates most of the market into a single cluster, while K-Means divides it more finely.
Tier 2	438	564	
Tier 3	371	64	**Biggest Difference**: K-Means creates a substantial third cluster (371). HC treats this as smaller groups.
Tier 4 (Niche)	71	37	**Niche Consensus**: Both methods isolate a very small segment, but K-Means expands this segment more.
Total	1405	1405	

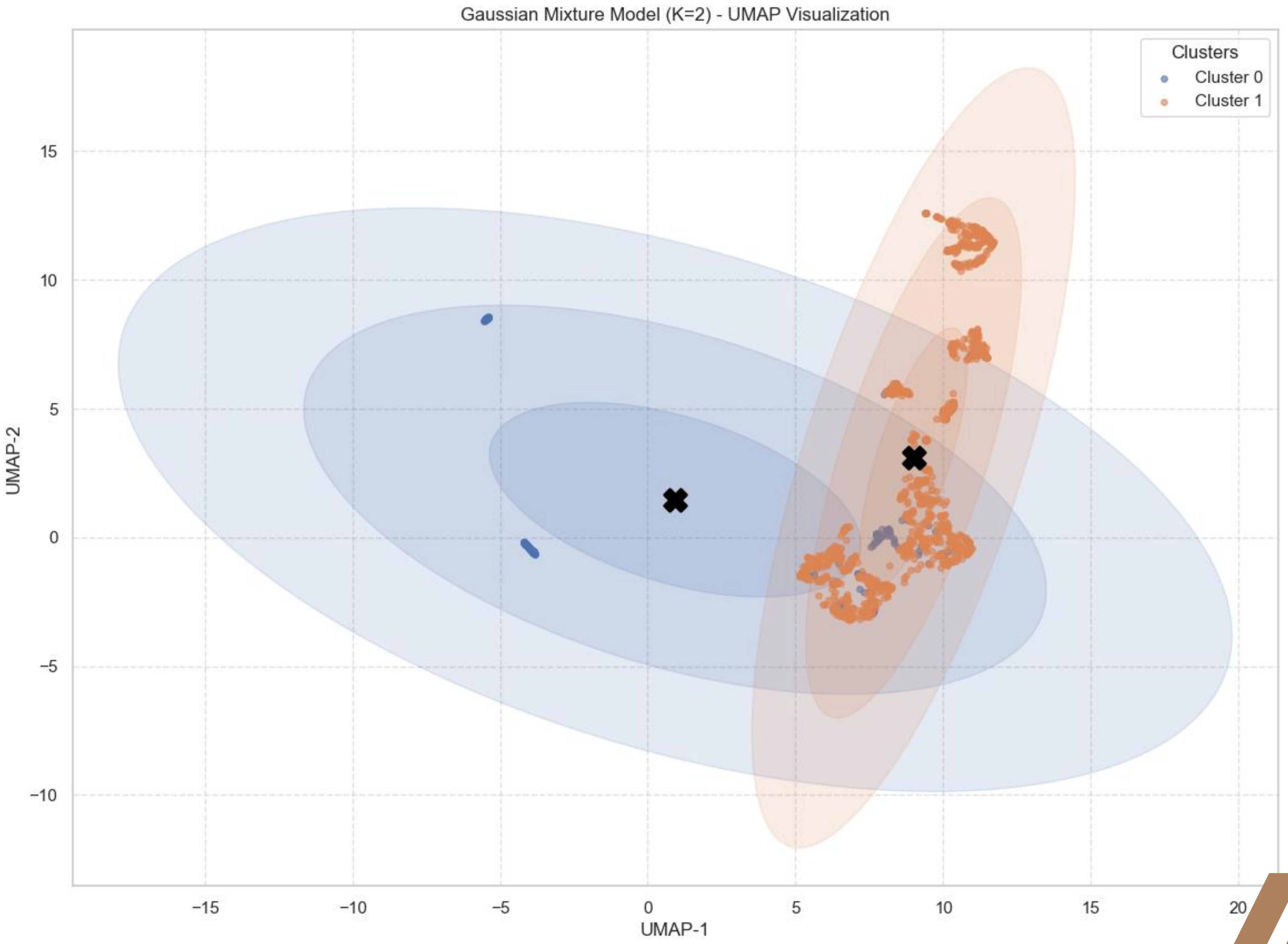
House Segmentation GMM

We also run GMM to detect clusters and compare its work to K-Means' one.



House Segmentation GMM

We also run GMM to detect clusters and compare its work to K-Means' one.



Even though GMM works very well with k=2. But following business perspective, we still set up k=4 and calculate some essential scores of these four clusters.

House Segmentation GMM

We also run GMM to detect clusters and compare its work to K-Means' one.

```
--- GMM CLUSTERING RESULTS (K=4) ---
Total samples analyzed: 1405
---------------------------------------------
AIC Score (MIN is better): -29534.11
BIC Score (MIN is better): -22717.23
---------------------------------------------
Silhouette Score (MAX is better): 0.1347
Calinski-Harabasz Index (MAX is better): 140
---------------------------------------------
Sample Distribution:
|     | count    |
|:---|:--------|
| 0   | 404      |
| 1   | 885      |
| 2   | 37       |
| 3   | 79       |
```

Metric	GMM ((K = 4))	K-Means ((K = 4)) (Estimated)	Assessment
Silhouette Score (Separation)	0.1347	≈ 0.19	**Failure:** GMM did not improve separation and performed worse than K-Means. This suggests the clusters are more spherical (K-Means) than elliptical (GMM).
Calinski-Harabasz Index (Density)	140	≈ 250	Worse than K-Means.
AIC/BIC (Model Fit)	Very Low (Good)	N/A	The model fits well statistically, but clusters poorly.

House Segmentation GMM

We also run GMM to detect clusters and compare its work to K-Means' one.

```
--- GMM CLUSTERING RESULTS (K=4) ---
Total samples analyzed: 1405
---------------------------------------------
AIC Score (MIN is better): -29534.11
BIC Score (MIN is better): -22717.23
---------------------------------------------
Silhouette Score (MAX is better): 0.1347
Calinski-Harabasz Index (MAX is better): 140
---------------------------------------------
Sample Distribution:
|     | count    |
|:---|:--------|
| 0   | 404      |
| 1   | 885      |
| 2   | 37       |
| 3   | 79       |
```

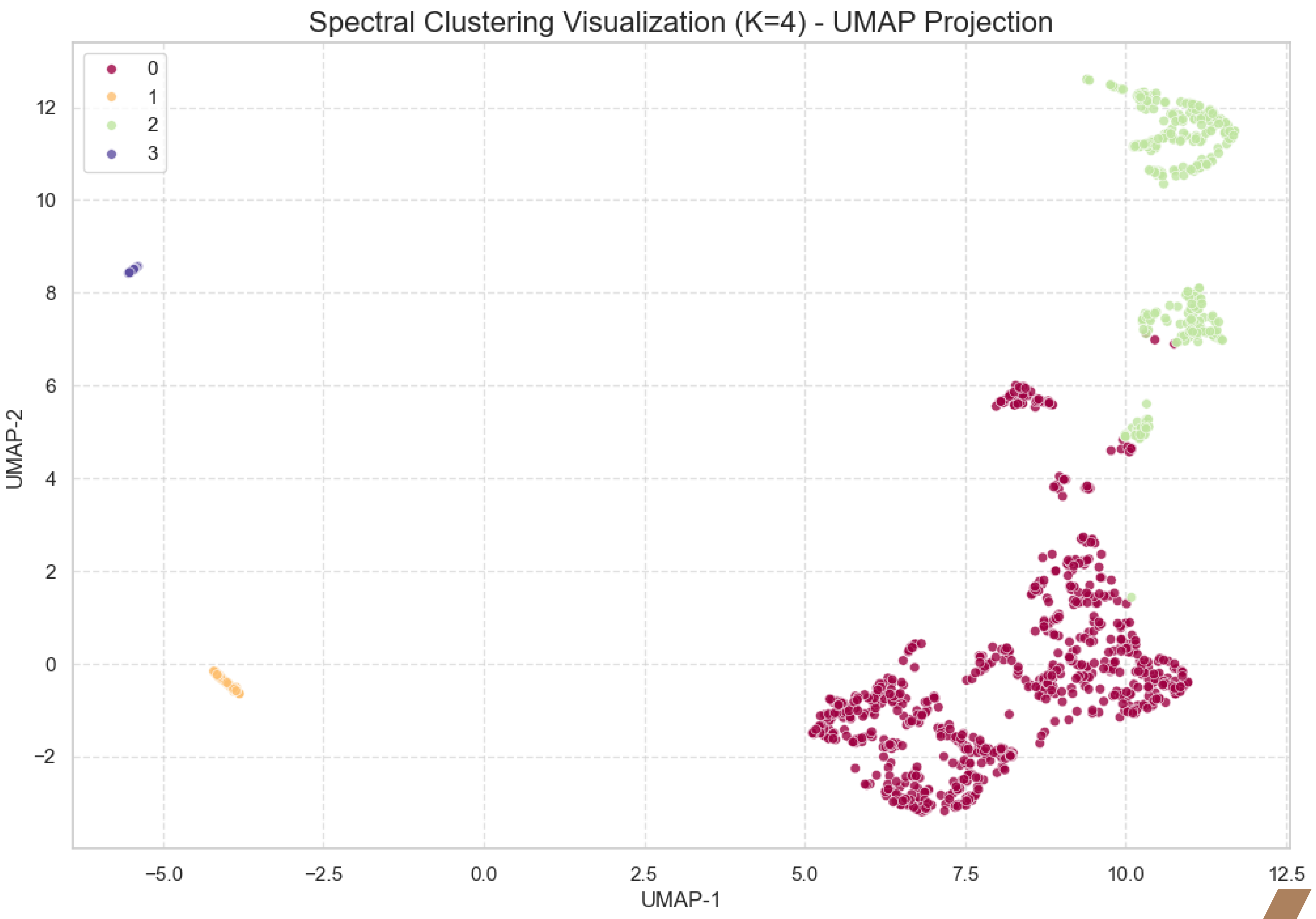
Metric	GMM ((K = 4))	K-Means ((K = 4)) (Estimated)	Assessment
Silhouette Score (Separation)	0.1347	≈ 0.19	**Failure:** GMM did not improve separation and performed worse than K-Means. This suggests the clusters are more spherical (K-Means) than elliptical (GMM).
Calinski-Harabasz Index (Density)	140	≈ 250	Worse than K-Means.
AIC/BIC (Model Fit)	Very Low (Good)	N/A	The model fits well statistically, but clusters poorly.

→ K-Means (K=4) performed better than GMM at dividing this block of 885 samples into three well-balanced and interpretable market tiers.

House Segmentation

Spectral Clustering

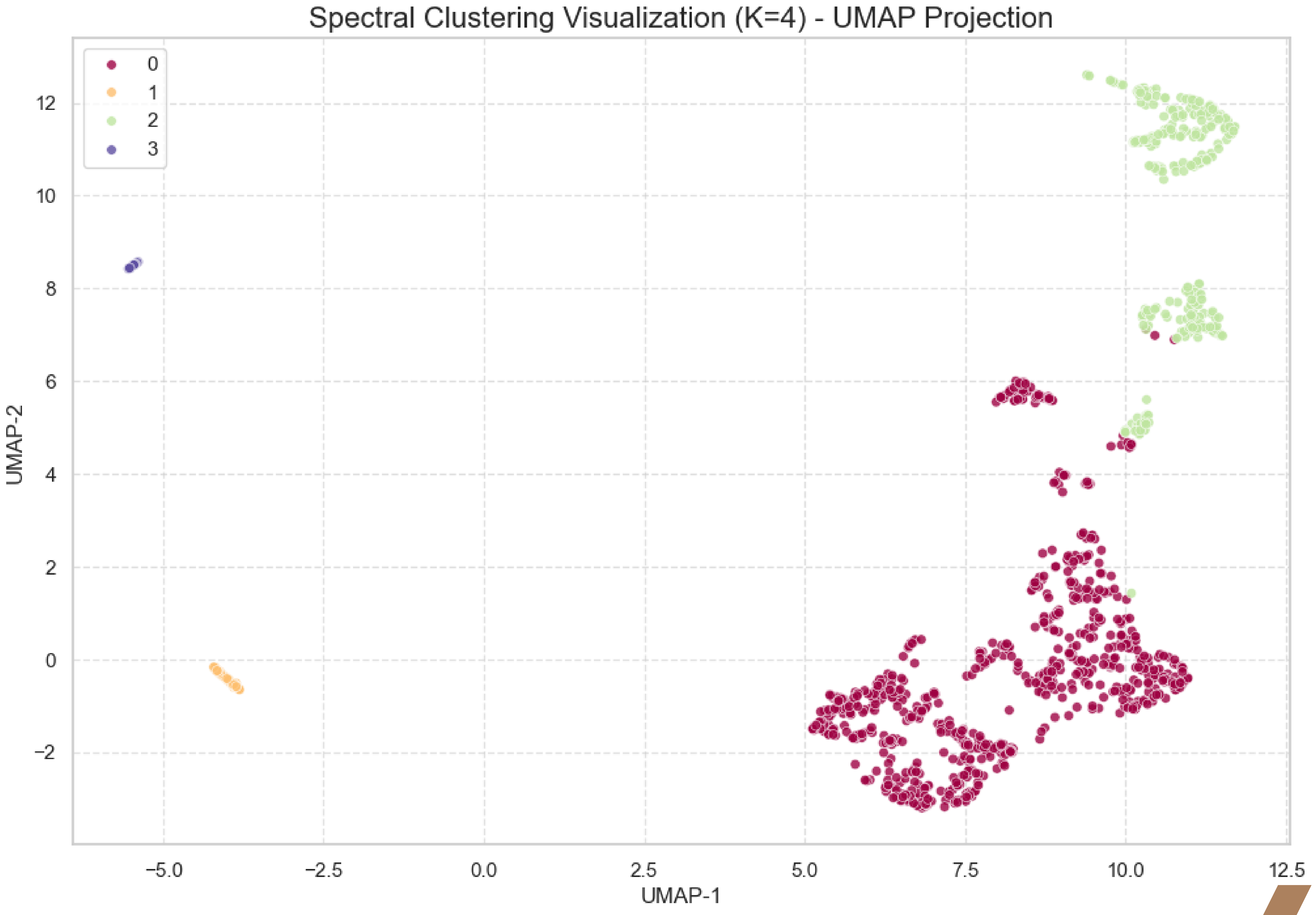
To assure if K-Means with k=4 is the best strategy, we run the last model.



House Segmentation

Spectral Clustering

To assure if K-Means with k=4 is the best strategy, we run the last model.




```
--- Running Spectral Clustering with K=4 ---

--- SPECTRAL CLUSTERING RESULTS (K=4) ---
Silhouette Score (MAX is better): 0.2103
Calinski-Harabasz Index (MAX is better): 246
---------------------------------------------
Sample Distribution:
|    | count    |
|:---|:--------|
| 0  | 914      |
| 1  | 64       |
| 2  | 390      |
| 3  | 37       |
```

House Segmentation Spectral Clustering

Compare Spectral's output to K-Means':

Metric	Value	Assessment (Compared to K-Means ≈ 0.19)
Silhouette Score	0.2103	**Best!** Higher than K-Means (0.19). This is the highest Silhouette Score we have found across all Partitioning models ((K = 3, 4, 5...)).
CH Index	246	**Very good.** Matches the high level of K-Means (250).

Structure	Count (Spectral)	Count (K-Means)	Assessment
Mainstream Block	914	525, 438, 371	Both Spectral and HDBSCAN (1304) consolidate the majority of the market into a single block (914 samples). K-Means splits this block.
Niche 2 (Premium)	64	N/A (Mixed)	**Confirmed!** Both HDBSCAN and Spectral isolate this group of 64 samples, while K-Means cannot (it gets divided into clusters 525/438).
Niche 1 (Luxury)	37	N/A (Mixed)	**Extreme Confirmation!** These 37 samples are confirmed by HDBSCAN, GMM, and Spectral.

House Segmentation Spectral Clustering

Compare Spectral's output to K-Means':

Metric	Value	Assessment (Compared to K-Means ≈ 0.19)
Silhouette Score	0.2103	**Best!** Higher than K-Means (0.19). This is the highest Silhouette Score we have found across all Partitioning models ((K = 3, 4, 5...)).
CH Index	246	**Very good.** Matches the high level of K-Means (250).

Structure	Count (Spectral)	Count (K-Means)	Assessment
Mainstream Block	914	525, 438, 371	Both Spectral and HDBSCAN (1304) consolidate the majority of the market into a single block (914 samples). K-Means splits this block.
Niche 2 (Premium)	64	N/A (Mixed)	**Confirmed!** Both HDBSCAN and Spectral isolate this group of 64 samples, while K-Means cannot (it gets divided into clusters 525/438).
Niche 1 (Luxury)	37	N/A (Mixed)	**Extreme Confirmation!** These 37 samples are confirmed by HDBSCAN, GMM, and Spectral.

Silhouette of Spectral proves it better than K-Means? ⟶ What should be the ultimate model?

House Segmentation

Ultimate model: K-Means

We ultimately select K-Means with k=4 as the most appropriate model.

Business-Driven Clustering Strategy

- Project Objective: The goal was not merely to discover data structure, but to leverage that structure to create Price Differentiation in the subsequent predictive model.
- Limitation of Spectral/HDBSCAN: Both algorithms consolidated the majority of samples (914) into a single Mainstream Block.
- Business Requirement: "We need to divide that block of 914 samples into three distinct tiers: Upper, Mid, and Starter."
- K-Means Advantage: While K-Means may not identify small niche clusters as precisely, it possesses the crucial characteristic of forcing data partition into K relatively well-balanced clusters.
- Result: K-Means (K=4) was the only model that produced four well-balanced segments (Tier 1, 2, 3, 4), providing the necessary separation for price differentiation and predictive modeling purposes.

House Segmentation

Segment Profiling

We use some representative features which can be used to appraise houses and label houses based on the mean of these following features:

Cluster	1	2	0	3
Cluster_Count	525.000000	371.000000	438.000000	71.000000
SalePrice	243421.396190	165879.991914	123622.974886	100433.802817
OverallQual	7.361905	5.838275	5.004566	4.647887
ExterQual	3.704762	0.269542	0.210046	0.112676
KitchenQual	3.979048	0.940701	0.716895	0.661972
GrLivArea	1764.158095	1623.584906	1085.438356	1105.098592
TotalSF	3046.577143	2637.566038	1903.426941	1817.985915
TotalBsmtSF	1282.876190	1019.183288	822.945205	735.873239
TotalRooms	3.849524	4.288410	3.586758	3.971831
HouseAge	7.224762	51.150943	52.073059	63.774648
TotalBath	2.756190	2.180593	1.590183	1.640845
GarageCars	2.308571	1.722372	1.433790	0.000000
Has_Fireplaces	0.687619	0.846361	0.105023	0.098592
OutdoorArea	218.045714	196.652291	117.166667	118.521127
TotalBasementBath	0.518095	0.409704	0.405251	0.316901
LuxuryIndex	685.320000	553.293801	408.789954	11.267606

House Segmentation

Segment Profiling

Cluster	1	2	0	3
Cluster_Count	525.000000	371.000000	438.000000	71.000000
SalePrice	243421.396190	165879.991914	123622.974886	100433.802817
OverallQual	7.361905	5.838275	5.004566	4.647887
ExterQual	3.704762	0.269542	0.210046	0.112676
KitchenQual	3.979048	0.940701	0.716895	0.661972
GrLivArea	1764.158095	1623.584906	1085.438356	1105.098592
TotalSF	3046.577143	2637.566038	1903.426941	1817.985915
TotalBsmtSF	1282.876190	1019.183288	822.945205	735.873239
TotalRooms	3.849524	4.288410	3.586758	3.971831
HouseAge	7.224762	51.150943	52.073059	63.774648
TotalBath	2.756190	2.180593	1.590183	1.640845
GarageCars	2.308571	1.722372	1.433790	0.000000
Has_Fireplaces	0.687619	0.846361	0.105023	0.098592
OutdoorArea	218.045714	196.652291	117.166667	118.521127
TotalBasementBath	0.518095	0.409704	0.405251	0.316901
LuxuryIndex	685.320000	553.293801	408.789954	11.267606

Below are official identifications of each Cluster:

Cluster 1 (Premium Luxury, 37.4% of market):
- Nature: New, High-Quality Homes
- Characteristics: Houses built after 2015, largest size (TotalSF=3046), and best quality (OverallQual=7.36)

Cluster 2 (Large Standard / Renovated, 26.4% of market):
- Nature: Large, Old but Well-Maintained Homes
- Characteristics: Large size (TotalSF=2638), but high age (51 years). Average quality (OverallQual=5.8), but high fireplace rate (0.84). This group typically represents old but spacious homes

Cluster 0 (Standard / Mid-Value, 31.2% of market):
- Nature: Standard, Older Homes
- Characteristics: Smaller size (TotalSF=1903), lower quality (OverallQual=5.0), high age (52 years). This is the most standard segment

Cluster 3 (Budget / Starter, 5.0% of market):
- Nature: Lowest Value and Oldest Group
- Characteristics: Lowest value ($100k), highest age (64 years), no garage (GarageCars=0). This segment is clearly separated due to lack of basic amenities

House Segmentation

Blending Model with Profiling Data

Before finding the best blending model, we apply One-hot Encoding on the Cluster column with 0,1,2,3.

If not, the model understands that House with cluster 3 is more important than those of 2,1,0.

House Segmentation

Blending Model with Profiling Data

Before finding the best blending model, we apply One-hot Encoding on the Cluster column with 0,1,2,3.

If not, the model understands that House with cluster 3 is more important than those of 2,1,0.

```python
def create_ohe_cluster_features(train, noise_mask, X_cluster_scaled_arr):

    K_FINAL = 4

    # 1. Filter original DataFrame to match data used for Clustering
    train_cleaned_df = train.loc[noise_mask].copy()

    # 2. Re-run K-Means to get cluster labels (clusters_final)
    # Note: This ensures cluster labels (0, 1, 2, 3) are correctly assigned
    kmeans_final = KMeans(n_clusters=K_FINAL, random_state=42, n_init='auto')
    clusters_final = kmeans_final.fit_predict(X_cluster_scaled_arr)

    # Assign cluster labels to cleaned DataFrame
    train_cleaned_df['Cluster_ID'] = clusters_final

    # 3. One-Hot Encoding (OHE) for Cluster_ID column
    # Reorder OHE columns by descending value for easier naming

    # Mapping K-Means labels to assigned segment names (based on previous profiling)
    cluster_mapping = {
        1: 'Premium Luxury',
        2: 'Large Standard/Renovated',
        0: 'Standard/Mid-Value',
        3: 'Budget/Starter'
    }
    train_cleaned_df['Segment'] = train_cleaned_df['Cluster_ID'].map(cluster_mapping)

    # Perform OHE on Segment column
    cluster_ohe_df = pd.get_dummies(train_cleaned_df['Segment'], prefix='Clus_Is')

    print("--- Completed OHE Feature Creation ---")
    print(f"OHE DataFrame size: {cluster_ohe_df.shape}")

    return cluster_ohe_df

OHE_features_train = create_ohe_cluster_features(train, noise_mask, X_cluster_scaled)
```

House Segmentation

Blending Model with Profiling Data

Before finding the best blending model, we apply One-hot Encoding on the Cluster column with 0,1,2,3.

If not, the model understands that House with cluster 3 is more important than those of 2,1,0.

```
def create_ohe_cluster_features(train, noise_mask, X_cluster_scaled_arr):

    K_FINAL = 4

    # 1. Filter original DataFrame to match data used for Clustering
    train_cleaned_df = train.loc[noise_mask].copy()

    # 2. Re-run K-Means to get cluster labels (clusters_final)
    # Note: This ensures cluster labels (0, 1, 2, 3) are correctly assigned
    kmeans_final = KMeans(n_clusters=K_FINAL, random_state=42, n_init='auto')
    clusters_final = kmeans_final.fit_predict(X_cluster_scaled_arr)

    # Assign cluster labels to cleaned DataFrame
    train_cleaned_df['Cluster_ID'] = clusters_final

    # 3. One-Hot Encoding (OHE) for Cluster_ID column
    # Reorder OHE columns by descending value for easier naming

    # Mapping K-Means labels to assigned segment names (based
    cluster_mapping = {
        1: 'Premium Luxury',
        2: 'Large Standard/Renovated',
        0: 'Standard/Mid-Value',
        3: 'Budget/Starter'
    }
    train_cleaned_df['Segment'] = train_cl...uster_ID

    # Perform OHE on Segment column
    cluster_ohe_df = pd.get_dummies(train_cleaned_df['Segment'...

    print("--- Completed OHE Feature Creation ---")
    print(f"OHE DataFrame size: {cluster_ohe_df.shape}")

    return cluster_ohe_df

OHE_features_train = create_ohe_cluster_features(train, noise_mask, X_cluster_scaled)
```

	Clus_Is_Budget/Starter	Clus_Is_Large Standard/Renovated	Clus_Is_Premium Luxury	Clus_Is_Standard/Mid-Value
0	False	False	True	False
1	False	True	False	False
2	False	False	True	False
3	False	True	False	False
4	False	False	True	False

House Segmentation

Blending Model with Profiling Data

Because the data now has some columns created by OHE on Cluster types.

⟶ find again the most appropriate weight between XGBoost and Ridge.

⟶ 0.81*XGBoost + 0.19*Ridge is no longer valid.

Blending Model with Profiling Data

Because the data now has some columns created by OHE on Cluster types.

⟶ find again the most appropriate weight between XGBoost and Ridge.

⟶ 0.81*XGBoost + 0.19*Ridge is no longer valid.

```
BLENDING WEIGHTS OPTIMIZATION RESULTS:
Best Blended RMSE (Log Scale): 0.11661
Best Weight for XGBoost:          0.44 (44%)
Best Weight for Ridge:            0.56 (56%)
```

New blending model: **0.44*XGBoost + 0.56*Ridge**

Blending Model with Profiling Data

Because we split internal train set into some folds for finding the best weight of the blending mode.

⟶ Deploy this mode on the whole train dataset.

```
W_XGB_FINAL = 0.44
W_RIDGE_FINAL = 0.56

print("Train on full cleaned dataset with final weights.")

# Ensure we pass only numeric data to XGBoost / Ridge (no object dtypes)
# Prefer using X_features_numeric if it was prepared earlier during validation.
if 'X_features_numeric' in globals():
        X_train_final = X_features_numeric
else:
        # Fallback: select numeric columns from X_features (drop object/categorical)
        X_train_final = X_features.select_dtypes(include=[np.number])
        dropped = list(set(X_features.columns) - set(X_train_final.columns))
        if dropped:
                print("Warning: Dropping non-numeric columns before training:", dropped)

# Convert to numpy arrays to avoid unexpected pandas dtypes issues
X_train_np = X_train_final.values
y_train_np = y_target.values

# Fit models on numeric data
xgb_model.fit(X_train_np, y_train_np)
ridge_model.fit(X_train_np, y_train_np)
```

House Segmentation

Blending Model with Profiling Data

Because we split internal train set into some folds for finding the best weight of the blending mode.

⟶ Deploy this mode on the whole train dataset.

```python
W_XGB_FINAL = 0.44
W_RIDGE_FINAL = 0.56

print("Train on full cleaned dataset with final weights.")

# Ensure we pass only numeric data to XGBoost / Ridge (no object dtypes)
# Prefer using X_features_numeric if it was prepared earlier during validation.
if 'X_features_numeric' in globals():
        X_train_final = X_features_numeric
else:
        # Fallback: select numeric columns from X_features (drop object/categorical)
        X_train_final = X_features.select_dtypes(include=[np.number])
        dropped = list(set(X_features.columns) - set(X_train_final.columns))
        if dropped:
                print("Warning: Dropping non-numeric columns before training:", dropped)

# Convert to numpy arrays to avoid unexpected pandas dtypes issues
X_train_np = X_train_final.values
y_train_np = y_target.values

# Fit models on numeric data
xgb_model.fit(X_train_np, y_train_np)
ridge_model.fit(X_train_np, y_train_np)
```

```
Train on full cleaned dataset with final weights.
▾ Ridge
Ridge(alpha=10, random_state=42)
```

House Segmentation

Blending Model with Profiling Data

To move on the final blending model on the test set.
We must deploy some preprocessing pipelines (which applied on train set formerly) on the test set in order to make them aligned each other.

```
X_features dimensions (Train): (1405, 105)
X_test_final_aligned dimensions (Test): (1459, 105)
```

House Segmentation

Blending Model with Profiling Data

Below is the prediction of House Price on Test set:

	Id	SalePrice_Cluster
0	1461	122227.531169
1	1462	161151.916189
2	1463	180919.450962
3	1464	195636.768664
4	1465	180961.389421
5	1466	173175.941552
6	1467	181158.591436
7	1468	170513.818292
8	1469	190045.731878
9	1470	115524.364037
10	1471	201416.303649
11	1472	99612.513754
12	1473	102128.166013
13	1474	145292.895644
14	1475	116931.953681
15	1476	346946.380691
16	1477	243309.616638
17	1478	287909.743910
18	1479	258107.278542
19	1480	507513.057027

House Segmentation

Blending Model with Profiling Data

We compare the current results to previous one which are generated by blending model of the un-profiled data (baseline)

	Id	SalePrice_Baseline	SalePrice_Cluster	Price_Difference
0	1461	131522.194361	122227.531169	-9294.663192
1	1462	160824.707591	161151.916189	327.208598
2	1463	185197.097681	180919.450962	-4277.646719
3	1464	199443.563009	195636.768664	-3806.794346
4	1465	175812.078721	180961.389421	5149.310700
5	1466	174726.568021	173175.941552	-1550.626469
6	1467	181953.815145	181158.591436	-795.223709
7	1468	174900.726678	170513.818292	-4386.908386
8	1469	186110.538209	190045.731878	3935.193669
9	1470	125908.049548	115524.364037	-10383.685512
10	1471	197413.259260	201416.303649	4003.044389
11	1472	98690.997684	99612.513754	921.516070
12	1473	104245.938943	102128.166013	-2117.772931
13	1474	150443.646480	145292.895644	-5150.750835
14	1475	116910.071165	116931.953681	21.882516
15	1476	342000.072930	346946.380691	4946.307762
16	1477	250237.259946	243309.616638	-6927.643308
17	1478	287684.264347	287909.743910	225.479562
18	1479	276391.279324	258107.278542	-18284.000782
19	1480	549653.388782	507513.057027	-42140.331755

House Segmentation

Blending Model with Profiling Data

	Id	SalePrice_Baseline	SalePrice_Cluster	Price_Difference
0	1461	131522.194361	122227.531169	-9294.663192
1	1462	160824.707591	161151.916189	327.208598
2	1463	185197.097681	180919.450962	-4277.646719
3	1464	199443.563009	195636.768664	-3806.794346
4	1465	175812.078721	180961.389421	5149.310700
5	1466	174726.568021	173175.941552	-1550.626469
6	1467	181953.815145	181158.591436	-795.223709
7	1468	174900.726678	170513.818292	-4386.908386
8	1469	186110.538209	190045.731878	3935.193669
9	1470	125908.049548	115524.364037	-10383.685512
10	1471	197413.259260	201416.303649	4003.044389
11	1472	98690.997684	99612.513754	921.516070
12	1473	104245.938943	102128.166013	-2117.772931
13	1474	150443.646480	145292.895644	-5150.750835
14	1475	116910.071165	116931.953681	21.882516
15	1476	342000.072930	346946.380691	4946.307762
16	1477	250237.259946	243309.616638	-6927.643308
17	1478	287684.264347	287909.743910	225.479562
18	1479	276391.279324	258107.278542	-18284.000782
19	1480	549653.388782	507513.057027	-42140.331755

Quick descriptive statistics between two models:

```
--- Price Prediction Difference Analysis ---
count       1459.000000
mean         111.003135
std        16436.974091
min       -48250.588101
25%        -4821.912808
50%         -495.518127
75%         3841.760354
max       524572.975977
Name: Price_Difference, dtype: float64
```

House Segmentation

Model Comparison

	Id	SalePrice_Baseline	SalePrice_Cluster	Price_Difference
0	1461	131522.194361	122227.531169	-9294.663192
1	1462	160824.707591	161151.916189	327.208598
2	1463	185197.097681	180919.450962	-4277.646719
3	1464	199443.563009	195636.768664	-3806.794346
4	1465	175812.078721	180961.389421	5149.310700
5	1466	174726.568021	173175.941552	-1550.626469
6	1467	181953.815145	181158.591436	-795.223709
7	1468	174900.726678	170513.818292	-4386.908386
8	1469			
9	1470			
10	1471			
11	1472			
12	1473			
13	1474			
14	1475			
15	1476			
16	1477			
17	1478			
18	1479			
19	1480	549653.388782	507513.057027	-42140.331755

Quick descriptive statistics between two models:

```
--- Price Prediction Difference Analysis ---
count      1459.000000
mean        111.003135
std       16436.974091
min      -48250.588101
25%       -4821.912808
50%        -495.518127
75%        3841.760354
max      524572.975977
Name: Price_Difference, dtype: float64
```

Metric	Value (USD)	Business Interpretation
mean	+$111	**Net Value Added**. On average, the Blending model with Clusters values properties $111 higher than the Baseline model. This is because the model learned higher price floors for the Premium/Luxury segment.
std	$16,436	**Magnitude of Adjustment**. Indicates that the Cluster model made significantly stronger price adjustments compared to the Baseline model.
max	$524,572	**Extreme Value Differentiation**. This is the strongest evidence. The Cluster model increased the price by nearly half a million dollars for a specific property (or a very small group), demonstrating that the OHE Cluster features successfully identified and correctly priced the Luxury/Niche segment (which the Baseline model failed to value properly).
min	−$48,250	**Noise Filtering**. The model reduced the price by $48,250 for some properties. This typically occurs with the Budget/Starter segment or older, problematic homes where the Cluster model learned they belong to the lowest price tier.