House Prices - Advanced Regression Techniques

February 11, 2026

1 House Prices - Advanced Regression Techniques

1.0.1 ESI 6612: Statistical Data Intelligence

1.0.2 Professor: Dr. Walter Silva

1.1 ### Justin Stutler

1.2 Citation

During the duration of the project, I encountered the 'feature engineering' course on kaggle which lead to the 'feature engineering for housing price prediction' course. I use some code from this project which I credit throughout by labeling above each code block used with the following (feature-engineering-for-house-prices).
The feature-engineering-for-house-prices course can be found here:
https://www.kaggle.com/code/ryanholbrook/feature-engineering-for-house-prices

APA Style

Holbrook, R. (n.d.). Feature engineering for house prices [Jupyter Notebook]. Kaggle. https://www.kaggle.com/code/ryanholbrook/feature-engineering-for-house-prices

2 Preparation

2.1 Installs

Run installs for a new notebook

[1]:
```
# %pip install pandas
# %pip install numpy
# %pip install scikit-learn
# %pip install xgboost
# %pip install lightgbm
# %pip install catboost
# %pip install optuna
# %pip install matplotlib
# %pip install seaborn
# %pip install scipy
```

2.2 Imports

Always run imports

[2]:
```
# installs and imports
import os
import math
import warnings
from pathlib import Path

# core data manipulation
import pandas as pd
import numpy as np

# visualization
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import display

# statistics
import scipy.stats
from scipy.stats import skew

# scikit-learn: preprocessing and transformation
from sklearn.base import BaseEstimator, TransformerMixin, clone
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (
    OneHotEncoder,
    OrdinalEncoder,
    RobustScaler,
    StandardScaler,
    PowerTransformer,
    QuantileTransformer,
    FunctionTransformer
)
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.decomposition import PCA

# scikit-learn: model selection and evaluation
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.feature_selection import mutual_info_regression

# models
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor, VotingRegressor
from xgboost import XGBRegressor
```

```
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor

# hyperparameter optimization
import optuna

# configuration
#pd.set_option('display.max_rows', None)
#pd.set_option('display.max_columns', None)
warnings.filterwarnings('ignore')
sns.set_style("whitegrid")
```

3 Establish Baseline Score

3.0.1 Method to prepare data

[3]:
```
def load_data():
    # train and test splits
    train = pd.read_csv("train.csv")
    test = pd.read_csv("test.csv")

    # set input(x) and output(y) variables
    y = train['SalePrice']
    X = train.drop('SalePrice', axis=1)

    # Save ID for submission
    test_ids = test['Id']

    return X, y, test, test_ids
```

3.0.2 Method to Evaluate a Model (RMSE)

[4]:
```
def score_model(model, X, y):
    kf = KFold(n_splits=7, shuffle=True, random_state=42)

    # Calculate scores (results come back negative, so we flip them)
    scores = cross_val_score(model, X, y,␣
 ↪scoring='neg_root_mean_squared_error', cv=kf, n_jobs=-1)

    # Return average score and the standard deviation (how much the score␣
 ↪wobbles)
    return -scores.mean(), scores.std()
```

3.0.3 Method to get Baseline Data

[5]:
```
def get_baseline_data():
    # load datasets
    train = pd.read_csv("train.csv")
    test = pd.read_csv("test.csv")

    # normalize saleprice: log transform
    #y = np.log1p(train['SalePrice'])
    y = train['SalePrice']

    # drop target and id columns
    X = train.drop(['SalePrice', 'Id'], axis=1)

    # save id for submission later
    test_ids = test['Id']
    test = test.drop(['Id'], axis=1)

    return X, y, test, test_ids
```

3.0.4 Subset Baseline Fearures and Impute Missing Values

[6]:
```
from sklearn.compose import make_column_selector
# load data
X, y, test, test_ids = get_baseline_data()

# subset
baseline_features = ['Neighborhood', 'LotArea', '1stFlrSF']
X_baseline = X[baseline_features]

# numeric pipeline: impute missing values
num_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', RobustScaler()) # robustscaler is good for all models
])

# categorical pipeline: impute -> one-hot encode
# one-hot encoding works for linear models, sums, and trees
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

# combined preprocessor
pipe_preprocessor = ColumnTransformer([
    ('num', num_pipe, make_column_selector(dtype_include=np.number)),
    ('cat', cat_pipe, make_column_selector(dtype_include=object))
```

```
])
```

3.1 Baseline Model

The Baseline Model is the example model discussed in class which utilizes linear regression with the input parameters of Neighborhood, LotArea, 1stFlrSF to predict the SalePrice of the house.

[7]:
```
# build model pipeline
model_pipeline = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor), # process data
    ('model', LinearRegression()) # use linear regression
])

# fit the pipeline
model_pipeline.fit(X_baseline, y)

# predict
baseline_preds = model_pipeline.predict(test)

# name model
model_name = 'Baseline Model'
print(f'{model_name} created!')
```

```
Baseline Model created!
```

Evaluate the model performance

3.2 Baseline Score

[8]:
```
# get rmse, sd
baseline_rmse, baseline_sd = score_model(model_pipeline, X_baseline, y)
# print results
print(f"Average RMSE: ${baseline_rmse:,.2f}")
print(f"Standard Deviation: ${baseline_sd:,.2f}")
```

```
Average RMSE: $46,149.45
Standard Deviation: $4,899.83
```

3.2.1 Keep List of Scores

Create an array of tuples to store the model name, rmse, and standard deviation

[9]:
```
rmse_scores = []
rmse_scores.append((model_name, baseline_rmse, baseline_sd))
```

3.2.2 Method to show scores

[10]:
```
def show_scores(scores):
    score_dict = {}
    # iteration
    for model_name, rmse, sd in scores:
        score_dict[model_name] = (rmse, sd)

    # Print Header
    print(f"{'Model':<40} | {'RMSE':<10} | {'SD':<10}")
    print("-" * 68)

    # Print the final unique scores
    for model_name, (rmse, sd) in score_dict.items():
        print(f'{model_name:<40} | {rmse:.6f}   | {sd:.6f}')
```

[11]:
```
show_scores(rmse_scores)
```

```
Model                                    | RMSE       | SD
--------------------------------------------------------------------
Baseline Model                           | 46149.451257   | 4899.832441
```

During the first class, the professor set a goal: create a linear regression model that performs better than the baseline model.
I added '2ndFlrSF' to the features which improved the score.

4 Model 1 - Linear Regression

[12]:
```
# load data
X, y, test, test_ids = load_data()
model1_features = ['Neighborhood', 'LotArea', '1stFlrSF', '2ndFlrSF']

# model
model_pipeline = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor), # process
    ('model', LinearRegression()) # use linear regression
])

# model1 features only
X_m1 = X[model1_features]

# fit the pipeline
model_pipeline.fit(X_m1, y)

# predict
model1_preds = model_pipeline.predict(test)

# name model
```

```
model_name = 'Model 1.0'
print(f'{model_name} created!')
```

```
Model 1.0 created!
```

[13]:
```
# find rmse, sd
model1_rmse, model1_sd = score_model(model_pipeline, X_m1, y)
# result
print(f"Average RMSE: ${model1_rmse:,.2f}")
print(f"Standard Deviation: ${model1_sd:,.2f}")
```

```
Average RMSE: $39,863.05
Standard Deviation: $7,310.36
```

[14]:
```
# add scores to rmse_scores list
rmse_scores.append((model_name, model1_rmse, model1_sd))
show_scores(rmse_scores)
```

```
Model                                     | RMSE        | SD
--------------------------------------------------------------------
Baseline Model                            | 46149.451257    | 4899.832441
Model 1.0                                 | 39863.045099    | 7310.359640
```

Our Model 1.0 has a smaller RMSE than the baseline model, so the score improved!

4.0.1 Feature Engineering Course

I created a few features that did not help the model's score, so I completed the feature engineering course early on during the project in efforts to learn more about the feature creation process. After finishing the feature engineering course, kaggle recommended a feature engineering for housing prices course. This course provided a walkthrough on how to utilize various methods to create an XBG model that scores ~ 0.12 for this competition. With my score being ~ 0.19 at the time using linear regression, I wanted to understand how they received such a lower score. I realized I needed a better strategy than just randomly choosing features and seeing what the difference was.

5 Strategy

Adding random features will only improve the score so much. Instead, we will implement a strategy: Explore Data, Process Data, Assess Model Performance throughout 1. Encode Data Types for categorical and numeric features 2. Clean Data 3. Impute Data / Handle Missing Values 4. Outliers 5. Assess skew of output variable SalePrice 6. Log Transform SalePrice 7. Assess feature importance by MI 8. Feature Engineering 9. Drop highly correlated variables to reduce multicolinearity and noise 10. Assess skew of all numeric variables 11. Transform variables to improve skew(closer to 0) 12. Test various transforms to find optimal transform for each feature 13. Target encode neighborhood into 3 levels 14. create models: lasso, random forest, svm, xgb, lgb, cat 15. Tune Models 16. create an ensemble model

6 Implementing the Strategy

7 Data Exploration and Processing

Here we read the data from the train.csv and test.csv files provided by the kaggle competition found here: https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/data train and test splits are defined and their shapes are printed to confirm the data loaded

[15]:
```
train = pd.read_csv("train.csv", index_col="Id")
test = pd.read_csv("test.csv", index_col="Id")
data = pd.concat([train, test])
print("train shape is {}".format(train.shape))
print("test shape is {}".format(test.shape))
```

```
train shape is (1460, 80)
test shape is (1459, 79)
```

7.0.1 check for duplicates

[16]:
```
train.duplicated().sum()
```

[16]:
```
np.int64(0)
```

[17]:
```
test.duplicated().sum()
```

[17]:
```
np.int64(0)
```

no duplicates were found in train or test

Merge Datasets

[18]:
```
data = pd.concat([train, test])
data.reset_index(drop=True, inplace=True)
data.shape
```

[18]:
```
(2919, 80)
```

7.0.2 Missing Values

[19]:
```
def missing_values(data):
    return data.isnull().sum().sort_values(ascending=False).head(40)
missing_values(data)
```

[19]:
```
PoolQC          2909
MiscFeature     2814
Alley           2721
Fence           2348
MasVnrType      1766
SalePrice       1459
```

```
FireplaceQu     1420
LotFrontage      486
GarageYrBlt      159
GarageCond       159
GarageQual       159
GarageFinish     159
GarageType       157
BsmtExposure      82
BsmtCond          82
BsmtQual          81
BsmtFinType2      80
BsmtFinType1      79
MasVnrArea        23
MSZoning           4
Utilities          2
Functional         2
BsmtFullBath       2
BsmtHalfBath       2
BsmtFinSF2         1
BsmtFinSF1         1
TotalBsmtSF        1
BsmtUnfSF          1
SaleType           1
KitchenQual        1
GarageCars         1
GarageArea         1
Exterior2nd        1
Exterior1st        1
Electrical         1
Condition1         0
HouseStyle         0
BldgType           0
Condition2         0
LandContour        0
dtype: int64
```

7.0.3 Clean Data

(feature-engineering-for-house-prices) The feature engineering for housing prices included this following code block:

[20]:
```
def clean(df):
    # df["Exterior2nd"] = df["Exterior2nd"].replace({"Brk Cmn": "BrkComm"})
    # Some values of GarageYrBlt are corrupt, so we'll replace them
    # with the year the house was built
    df["GarageYrBlt"] = df["GarageYrBlt"].where(df.GarageYrBlt <= 2010, df.
 ↪YearBuilt)
```

```
    # Names beginning with numbers are awkward to work with
    # df.rename(columns={
    #     "1stFlrSF": "FirstFlrSF",
    #     "2ndFlrSF": "SecondFlrSF",
    #     "3SsnPorch": "Threeseasonporch",
    # }, inplace=True,
    # )
    return df
```

GarageYrBlt contains many missing values. We will impute these missing values as the YearBuilt

7.0.4 Set the feature data type

(feature-engineering-for-house-prices) The feature engineering for housing prices included this following code block:

Categorical variables are set as nominative (unordered) or ordinal (ordered). Ordinal variables have their levels specified.

[21]:
```
# The numeric features are already encoded correctly (`float` for
# continuous, `int` for discrete), but the categoricals we'll need to
# do ourselves. Note in particular, that the `MSSubClass` feature is
# read as an `int` type, but is actually a (nominative) categorical.

# The nominative (unordered) categorical features
features_nom = ["MSSubClass", "MSZoning", "Street", "Alley", "LandContour",␣
 ↪"LotConfig", "Neighborhood",
                "Condition1", "Condition2", "BldgType", "HouseStyle",␣
 ↪"RoofStyle", "RoofMatl",
                "Exterior1st", "Exterior2nd", "MasVnrType", "Foundation",␣
 ↪"Heating", "CentralAir",
                "GarageType", "MiscFeature", "SaleType", "SaleCondition"]

# The ordinal (ordered) categorical features

# Pandas calls the categories "levels"
five_levels = ["Po", "Fa", "TA", "Gd", "Ex"]
ten_levels = list(range(10))

ordered_levels = {
    "OverallQual": ten_levels,
    "OverallCond": ten_levels,
    "ExterQual": five_levels,
    "ExterCond": five_levels,
    "BsmtQual": five_levels,
    "BsmtCond": five_levels,
    "HeatingQC": five_levels,
    "KitchenQual": five_levels,
```

```
    "FireplaceQu": five_levels,
    "GarageQual": five_levels,
    "GarageCond": five_levels,
    "PoolQC": five_levels,
    "LotShape": ["Reg", "IR1", "IR2", "IR3"],
    "LandSlope": ["Sev", "Mod", "Gtl"],
    "BsmtExposure": ["No", "Mn", "Av", "Gd"],
    "BsmtFinType1": ["Unf", "LwQ", "Rec", "BLQ", "ALQ", "GLQ"],
    "BsmtFinType2": ["Unf", "LwQ", "Rec", "BLQ", "ALQ", "GLQ"],
    "Functional": ["Sal", "Sev", "Maj1", "Maj2", "Mod", "Min2", "Min1", "Typ"],
    "GarageFinish": ["Unf", "RFn", "Fin"],
    "PavedDrive": ["N", "P", "Y"],
    "Utilities": ["NoSeWa", "NoSewr", "AllPub"],
    "CentralAir": ["N", "Y"],
    "Electrical": ["Mix", "FuseP", "FuseF", "FuseA", "SBrkr"],
    "Fence": ["MnWw", "GdWo", "MnPrv", "GdPrv"],
}

# Add a None level for missing values
ordered_levels = {key: ["None"] + value for key, value in ordered_levels.
 ↪items()}

def set_feature_types(df):
    # Nominal categories
    for name in features_nom:
        if name in df.columns:
            df[name] = df[name].astype("category")
            # Add a None category for missing values
            if "None" not in df[name].cat.categories:
                df[name] = df[name].cat.add_categories("None")
    # Ordinal categories
    for name, levels in ordered_levels.items():
        if name in df.columns:
            df[name] = df[name].astype(CategoricalDtype(levels, ordered=True))
    return df
```

7.0.5 Encode

(feature-engineering-for-house-prices) The feature engineering for housing prices included this following code block:

Performs one-hot encoding, ordinal encoding, imputation, and column alignment.

[22]:
```
def encode(train, test):
    # identify object columns to encode
    features_to_encode = list(train.select_dtypes(include=['object']).columns)

    # one-hot encode train
```

```
    train = pd.get_dummies(train, columns=features_to_encode, dummy_na=False,␣
↪drop_first=True)

    # one-hot encode test using only existing columns
    test_features = [f for f in features_to_encode if f in test.columns]
    test = pd.get_dummies(test, columns=test_features, dummy_na=False,␣
↪drop_first=True)

    # convert categories to codes
    for colname in train.select_dtypes(["category"]):
        train[colname] = train[colname].cat.codes
    for colname in test.select_dtypes(["category"]):
        test[colname] = test[colname].cat.codes

    # align test columns to match train
    train_cols = train.columns
    test = test.reindex(columns=train_cols, fill_value=0)

    # fill missing values
    train.fillna(0, inplace=True)
    test.fillna(0, inplace=True)

    return train, test
```

7.0.6 Imputations

As an example: PoolQC has many missing values likely due to many houses not having pools, so we will perform imputations to fill in missing values.

[23]:
```
data['PoolQC'].fillna('None')
```

[23]:
```
0       None
1       None
2       None
3       None
4       None
        …
2914    None
2915    None
2916    None
2917    None
2918    None
Name: PoolQC, Length: 2919, dtype: object
```

7.0.7 Handle Missing Values

(feature-engineering-for-house-prices) The feature engineering for housing prices included this following code block:

[24]:
```
def impute(df):
    # Imputations
    # Some Houses simply do not have some features like a pool
    # Here we replace these features with a standardized 'None', so one-hot␣
 ↪encoding is easier later
    # features to fill with 'None'
    cols_fill_none = [
        'PoolQC', 'Alley', 'MiscFeature', 'Fence', 'FireplaceQu',
        'GarageType', 'GarageFinish', 'GarageQual', 'GarageCond',
        'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
        'MasVnrType'
    ]

    for col in cols_fill_none:
        df[col] = df[col].fillna('None')

    # features to fill with '0'
    cols_fill_zero = [
        'MasVnrArea', 'GarageArea', 'GarageCars', 'GarageYrBlt', #␣
 ↪GarageYrBlt=0 is a common placeholder
        'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF',
        'BsmtFullBath', 'BsmtHalfBath'
    ]

    for col in cols_fill_zero:
        df[col] = df[col].fillna(0)

    # features to fill with 'mode'
    cols_fill_mode = [
        'MSZoning', 'Utilities', 'Functional', 'Exterior1st',
        'Exterior2nd', 'KitchenQual', 'SaleType', 'Electrical'
    ]

    # .mode()[0] gets the most frequent value
    for col in cols_fill_mode:
        df[col] = df[col].fillna(df[col].mode()[0])

    # Group by Neighborhood, get the median LotFrontage, and use transform to␣
 ↪align it
    median_lot_frontage = df.groupby('Neighborhood',␣
 ↪observed=True)['LotFrontage'].transform('median')

    # Fill the missing LotFrontage values with the median of their neighborhood
    df['LotFrontage'] = df['LotFrontage'].fillna(median_lot_frontage)

    return df
```

[25]:
```
data = impute(data)
missing_values(data)
```

[25]:
```
SalePrice       1459
MSSubClass         0
LotFrontage        0
MSZoning           0
Street             0
Alley              0
LotShape           0
LotArea            0
Utilities          0
LotConfig          0
LandSlope          0
Neighborhood       0
Condition1         0
Condition2         0
BldgType           0
LandContour        0
HouseStyle         0
OverallQual        0
YearBuilt          0
OverallCond        0
RoofStyle          0
RoofMatl           0
Exterior1st        0
YearRemodAdd       0
MasVnrType         0
MasVnrArea         0
ExterQual          0
ExterCond          0
Foundation         0
BsmtQual           0
BsmtCond           0
Exterior2nd        0
BsmtFinType1       0
BsmtFinSF1         0
BsmtFinType2       0
BsmtFinSF2         0
BsmtUnfSF          0
TotalBsmtSF        0
Heating            0
HeatingQC          0
dtype: int64
```

Encoding the feature types allows us to perform imputations to address missing values. Now there are no missing values.

8 Assessing Skew

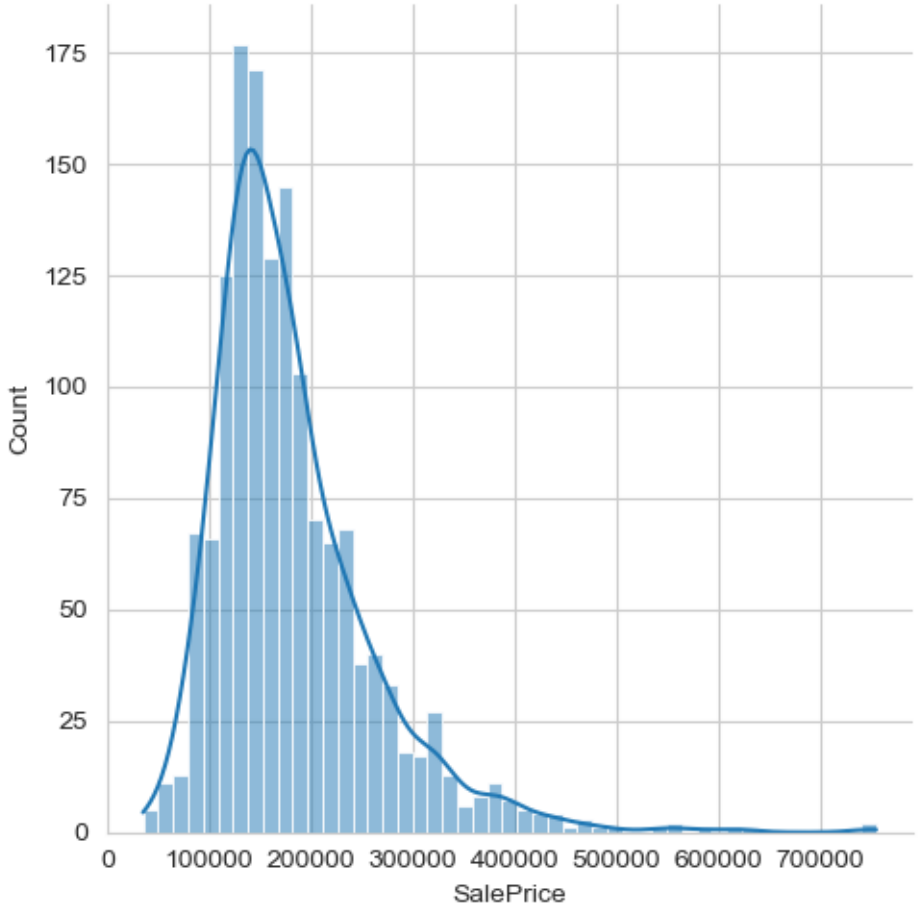
8.0.1 Assessing SalePrice Skew

Now we will begin exploring variables

We plot a histogram to visualize the ditribution of SalePrice
The skew is then calculalted to numerically show the skew of SalePrice

[26]:
```
# plot histogram
sns.displot(train.get('SalePrice'), kde=True)
# calculate skew
saleprice_skew = train['SalePrice'].skew()
# print skew
print(f"Skew: {saleprice_skew}")
```

```
Skew: 1.8828757597682129
```



Skew > 1 : Right Skew Skew < 0 : Left Skew Skew ~ 0 : Zero Skew

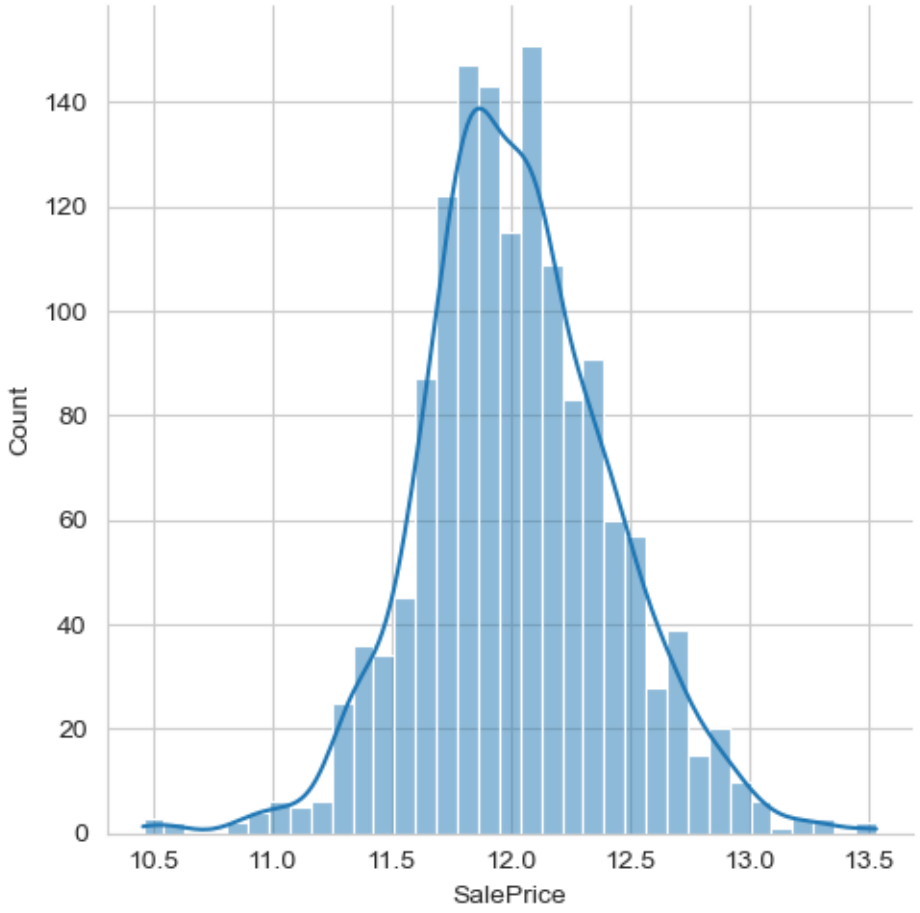
We normalize SalPrice in efforts to improve the skew (make the skew get closer to 0)

We want as close to a normal distribution as possible (Skew ~ 0)

[27]:
```
# log transform SalePrice
saleprice_log = np.log(train['SalePrice'])
# calculate new skew
saleprice_norm_skew = saleprice_log.skew()
# print new skew
print(f"Skew: {saleprice_norm_skew}")
# historgram
sns.displot(saleprice_log, kde=True)
```

```
Skew: 0.12133506220520406
```

[27]:
```
<seaborn.axisgrid.FacetGrid at 0x1b0df1bec10>
```



Results

[28]:
```
# calculate improvement
improvement = saleprice_skew - saleprice_norm_skew
```

```
# create table
results_table = pd.DataFrame({
    'Metric': ['Original SalePrice', 'Log-Transformed SalePrice',␣
 ↪'Improvement'],
    'Skewness': [saleprice_skew, saleprice_norm_skew, improvement]
})
# print results
print(results_table)
```

```
                     Metric  Skewness
0         Original SalePrice  1.882876
1  Log-Transformed SalePrice  0.121335
2                Improvement  1.761541
```

The Skew was improved and the shape became more normal

8.0.2 New Load Data Method

We will normalize SalePrice

[29]:
```
def load_data():
    # Load datasets
    train = pd.read_csv("train.csv")
    test = pd.read_csv("test.csv")

    # Normalize SalePrice: Log Transform
    y = np.log1p(train['SalePrice'])

    # Drop Target and Id columns
    X = train.drop(['SalePrice', 'Id'], axis=1)

    # Save ID for submission later
    test_ids = test['Id']
    test = test.drop(['Id'], axis=1)

    return X, y, test, test_ids
```

8.0.3 Update Score Methods (RMSLE)

Since we are log transforming SalePrice, we will update the score method to use RMSLE instead of RMSE.

[30]:
```
def score_model(model, X, y, cv=5):
    kf = KFold(n_splits=cv, shuffle=True, random_state=42)

    # 'neg_root_mean_squared_error' returns negative values, so we flip the sign
    scores = cross_val_score(model, X, y,␣
 ↪scoring='neg_root_mean_squared_error', cv=kf, n_jobs=-1)
```

```
    # Return average RMSE (which is RMSLE) and standard deviation
    return -scores.mean(), scores.std()
```

[31]:
```
def show_scores(scores):
    score_dict = {}
    # iteration
    for model_name, rmse, sd in scores:
        score_dict[model_name] = (rmse, sd)

    # Print Header
    print(f"{'Model':<40} | {'RMSLE':<10} | {'SD':<10}")
    print("-" * 68)

    # Print the final unique scores
    for model_name, (rmse, sd) in score_dict.items():
        print(f'{model_name:<40} | {rmse:.6f}   | {sd:.6f}')
```

[32]:
```
rmsle_scores = []
```

8.0.4 Preprocess Data Method

[33]:
```
def preprocess(train, test):
    # set feature types for both
    train = set_feature_types(train)
    test = set_feature_types(test)

    # clean both datasets
    train = clean(train)
    test = clean(test)

    # encode features ensuring alignment
    train, test = encode(train, test)

    # impute
    train = impute(train)
    test = impute(test)

    return train, test
```

8.0.5 Assess Impact of normalizing SalePrice and preprocessing data

[34]:
```
def get_baseline_data():
    # load datasets
    train = pd.read_csv("train.csv")
    test = pd.read_csv("test.csv")

    # normalize saleprice: log transform
```

```
    y = np.log1p(train['SalePrice'])
    #y = train['SalePrice']

    # drop target and id columns
    X = train.drop(['SalePrice', 'Id'], axis=1)

    # save id for submission later
    test_ids = test['Id']
    test = test.drop(['Id'], axis=1)

    return X, y, test, test_ids
```

[35]:
```
from pandas.api.types import CategoricalDtype
# load data
X, y, test, test_ids = get_baseline_data()

# baseline pipeline
baseline_model = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor), # process data
    ('model', LinearRegression())
])

# score baseline
rmse_base, sd_base = score_model(baseline_model, X_baseline, y)

# save baseline score
rmsle_scores.append(('Baseline - normalized target', rmse_base, sd_base))

X_m1 = X[model1_features]

# model 1 pipeline
model1 = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor), # process data
    ('model', LinearRegression())
])

# score model 1
rmse_mod1, sd_mod1 = score_model(model1, X_m1, y)

# save model 1 score
rmsle_scores.append(('Model 1 - normalized target', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
```

```
Model 1 - normalized target              | 0.195823   | 0.019063
```

8.0.6 Process Data Impact

[36]:
```
X, y, test, testIds = load_data()
X, test = preprocess(X, test)

# model 1 pipeline
model1 = Pipeline(steps=[
    ('model', LinearRegression())
])
X_m1 = X[model1_features]

# score model 1
rmse_mod1, sd_mod1 = score_model(model1, X_m1, y)

# save model 1 score
rmsle_scores.append(('Model 1 - Preprocess', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
```

[37]:
```
X, y, test, testIds = load_data()
X, test = preprocess(X, test)

# model 1 pipeline
model1 = Pipeline(steps=[
    ('model', LinearRegression())
])

# score model 1
rmse_mod1, sd_mod1 = score_model(model1, X, y)

# save model 1 score
rmsle_scores.append(('Model 1 - All Features', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
```

```
Model 1 - Preprocess                    | 0.265751   | 0.035764
Model 1 - All Features                  | 0.163591   | 0.041842
```

9 Feature Engineering

I created many features and assessed them by ranking their mutual information.

[38]:
```python
def engineer_features(df):
    # creates all engineered features by default without filtering
    new_features_created = []
    qual_map = {'None': 0, 'Po': 1, 'Fa': 2, 'TA': 3, 'Gd': 4, 'Ex': 5}

    # date/time features
    df["Age"] = df['YrSold'] - df['YearBuilt']
    df["Remodel_Age"] = df['YrSold'] - df['YearRemodAdd']
    df["Remodel_Score"] = df["YearRemodAdd"] - df["YearBuilt"]

    # garage age logic
    df["Garage_Age"] = df['YrSold'] - df['GarageYrBlt']

    new_features_created.extend(["Age", "Remodel_Age", "Garage_Age",␣
 ↪"Remodel_Score"])

    # base calculation features (dependencies for later)
    # we create these first so they are available for interaction terms below
    df["Total_SF"] = df["TotalBsmtSF"] + df["GrLivArea"]

    # summing up quality scores (filling nans with 0 just in case)
    df['Total_Qual'] = (df['OverallQual'] +
                        df['ExterQual'].map(qual_map).fillna(0) +
                        df['BsmtQual'].map(qual_map).fillna(0) +
                        df['KitchenQual'].map(qual_map).fillna(0))

    new_features_created.extend(["Total_SF", "Total_Qual"])

    # general area and bath features
    df["Bathrooms"] = (df["FullBath"] + 0.5 * df["HalfBath"] +
                       df["BsmtFullBath"] + 0.5 * df["BsmtHalfBath"])

    df["Porch_SF"] = (df["OpenPorchSF"] + df["3SsnPorch"] + df["EnclosedPorch"]␣
 ↪+
                      df["ScreenPorch"] + df["WoodDeckSF"])

    df["Live_Area"] = df["GrLivArea"] / (df["LotArea"] + 1e-6)

    df['OverallQual_sq'] = df['OverallQual']**2
    df['GrLivArea_sq'] = df['GrLivArea']**2
```

```
    new_features_created.extend([
        "Bathrooms", "Porch_SF", "Live_Area", "OverallQual_sq", "GrLivArea_sq"
    ])

    # interaction features
    # these rely on total_qual and total_sf existing
    df['Total_Qual_sq_x_Total_SF'] = (df['Total_Qual']**2) * df['Total_SF']
    df['Total_Qual_x_GrLivArea'] = df['Total_Qual'] * df['GrLivArea']
    df['OverallQual_x_GrLivArea'] = df['OverallQual'] * df['GrLivArea']
    df['Garage_Score'] = df['GarageCars'] * df['GarageQual'].map(qual_map).
↪fillna(0)
    df['ExterQual_x_Total_SF'] = df['ExterQual'].map(qual_map).fillna(0) *␣
↪df['Total_SF']

    new_features_created.extend([
        "Total_Qual_sq_x_Total_SF", "Total_Qual_x_GrLivArea",
        "OverallQual_x_GrLivArea", "Garage_Score", "ExterQual_x_Total_SF"
    ])

    # location based interaction features
    # only create these if target encoding (neighborhood_median_price) was done␣
↪previously
    if 'Neighborhood_Median_Price' in df.columns:
        df['Interaction_NMP_x_Total_SF'] = df['Neighborhood_Median_Price'] *␣
↪df['Total_SF']
        df['Interaction_NMP_x_Total_Qual'] = df['Neighborhood_Median_Price'] *␣
↪df['Total_Qual']
        df['Ultimate_Interaction'] = df['Total_Qual_sq_x_Total_SF'] *␣
↪df['Neighborhood_Median_Price']

        new_features_created.extend([
            "Interaction_NMP_x_Total_SF",
            "Interaction_NMP_x_Total_Qual",
            "Ultimate_Interaction"
        ])

    # binary variables
    binary_map = {
        'Has_Pool': 'PoolArea',
        'Has_2nd_Floor': '2ndFlrSF',
        'Has_Garage': 'GarageArea',
        'Has_Basement': 'TotalBsmtSF',
        'Has_Fireplace': 'Fireplaces',
        'Has_ScreenPorch': 'ScreenPorch',
        'Has_3SeasonPorch': '3SsnPorch',
```

```
        'Has_Misc': 'MiscVal'
    }

    for new_name, old_col in binary_map.items():
        if old_col in df.columns:
            df[new_name] = df[old_col].apply(lambda x: 1 if x > 0 else 0)
            new_features_created.append(new_name)

    return df
```

9.0.1 Target Encode Neighborhood

[39]:
```
# target encoder for neighborhood
class TargetEncoder(BaseEstimator, TransformerMixin):
    # replaces 'neighborhood' with the median saleprice of that neighborhood.
    def __init__(self, col_name='Neighborhood'):
        self.col_name = col_name
        self.target_mapping_ = {}
        self.global_median_ = 0

    def fit(self, X, y):
        if y is None:
            raise ValueError("Target y is required for Target Encoding")

        temp_df = X.copy()
        temp_df['target'] = y
        # learn median price per neighborhood
        self.target_mapping_ = temp_df.groupby(self.col_name)['target'].
 ↪median().to_dict()
        self.global_median_ = y.median()
        return self

    def transform(self, X):
        X_out = X.copy()
        # map values; fill unseen neighborhoods with global median
        X_out[self.col_name] = X_out[self.col_name].map(self.target_mapping_).
 ↪fillna(self.global_median_)
        return X_out
```

9.0.2 drop columns

We created one-hot encoded variables, so now we have to drop the original feature associated

[40]:
```
def drop_cols(df):
    component_map = {
        # engineered features : [features used to create]
        "Age": ["YearBuilt", "YrSold"],
```

```
        "Remodel_Age": ["YearRemodAdd", "YrSold"],
        "Garage_Age": ["GarageYrBlt", "YrSold"],
        "Remodel_Score": ["YearRemodAdd", "YearBuilt"],
        "Total_SF": ["TotalBsmtSF", "GrLivArea"],
        "Bathrooms": ["FullBath", "HalfBath", "BsmtFullBath", "BsmtHalfBath"],
        "Porch_SF": ["OpenPorchSF", "3SsnPorch", "EnclosedPorch",␣
↪"ScreenPorch", "WoodDeckSF"],
        # one hot encoded : [associated feature]
        "Has_Pool": ["PoolArea"],
        "Has_2nd_Floor": ["2ndFlrSF"],
        "Has_Garage": ["GarageArea"],
        "Has_Basement": ["TotalBsmtSF"],
        "Has_Fireplace": ["Fireplaces"],
        "Has_ScreenPorch": ["ScreenPorch"],
        "Has_3SeasonPorch": ["3SsnPorch"],
        "Has_Misc": ["MiscVal"]
    }

    # extract all columns to drop from the map's values
    cols_to_drop = []
    for components in component_map.values():
        cols_to_drop.extend(components)

    # use set() to remove duplicates (ex. YrSold)
    cols_to_drop = list(set(cols_to_drop))

    # drop all selected columns
    df = df.drop(columns=cols_to_drop, errors='ignore')

    return df
```

10 Mutual Information

10.0.1 Feature Utility Score

Here we calculate a feature utility score which uses mutual informaiton to access how useful a feature may be.
(feature-engineering-for-house-prices) The feature engineering for housing prices included this following code block:

[41]:
```
def make_mi_scores(X, y):
    X = X.copy()
    for colname in X.select_dtypes(["object", "category"]):
        X[colname], _ = X[colname].factorize()
    # All discrete features should now have integer dtypes
    discrete_features = [pd.api.types.is_integer_dtype(t) for t in X.dtypes]
```

```
    mi_scores = mutual_info_regression(X, y,␣
 ↪discrete_features=discrete_features, random_state=0)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
    return mi_scores


def plot_mi_scores(scores):
    scores = scores.sort_values(ascending=True)
    width = np.arange(len(scores))
    ticks = list(scores.index)
    plt.barh(width, scores)
    plt.yticks(width, ticks)
    plt.title("Mutual Information Scores")
```

10.0.2 Assess Mutual Information of standard feature set

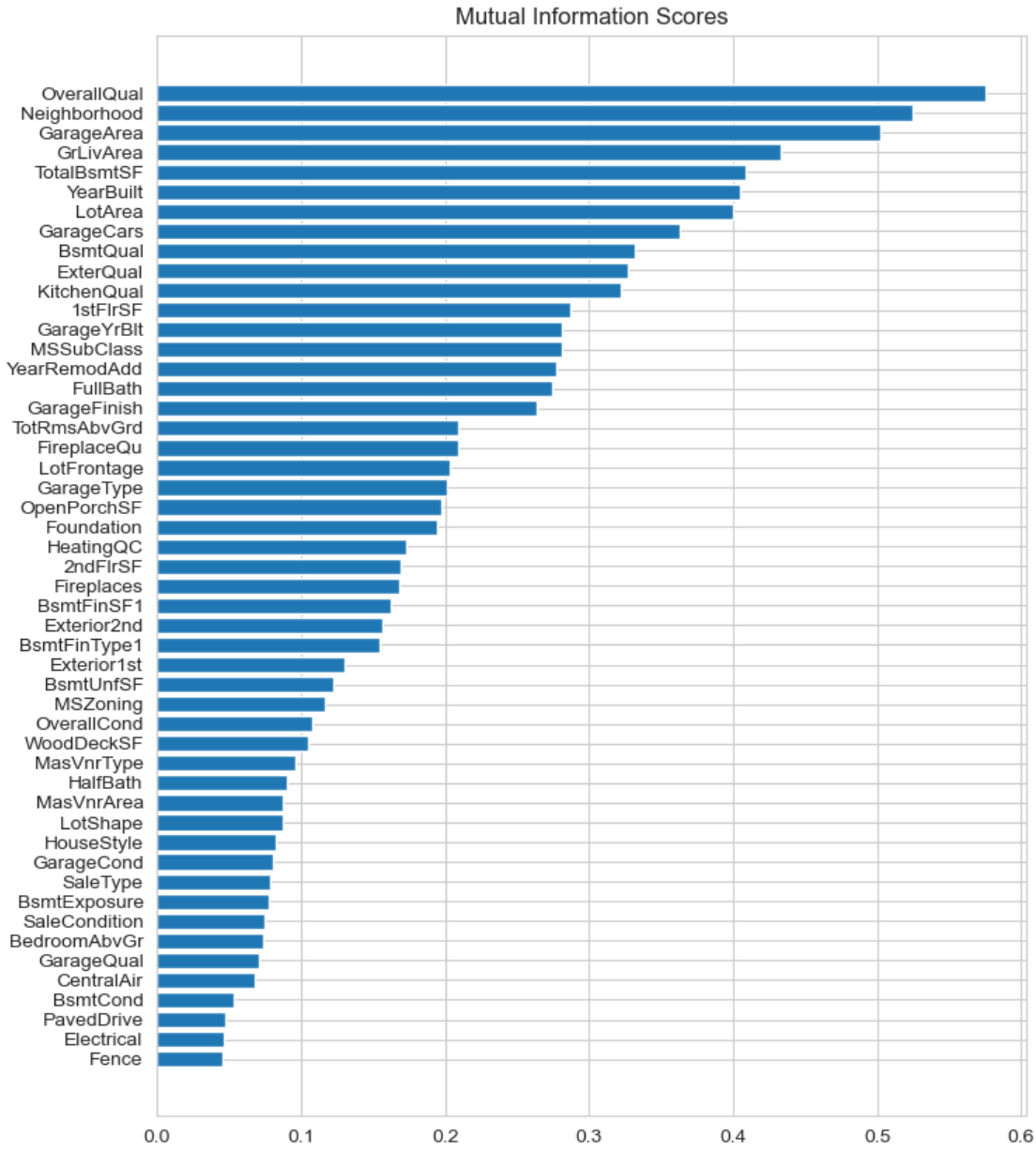
[42]:
```
# load data
X, y, test, test_ids = load_data()
X, test = preprocess(X, test)

# calculate scores
mi_scores = make_mi_scores(X, y)

# print the top 50 scores
print(mi_scores.head(50))

# plot the scores for visualization
plt.figure(figsize=(8, 10))
plot_mi_scores(mi_scores.head(50)) # plot the top 50 features
plt.show()
```

```
  File "C:\Users\justi\anaconda3\Lib\site-
packages\joblib\externals\loky\backend\context.py", line 257, in
_count_physical_cores
    cpu_info = subprocess.run(
        "wmic CPU Get NumberOfCores /Format:csv".split(),
        capture_output=True,
        text=True,
    )
  File "C:\Users\justi\anaconda3\Lib\subprocess.py", line 554, in run
    with Popen(*popenargs, **kwargs) as process:
         ~~~~~^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\justi\anaconda3\Lib\subprocess.py", line 1039, in __init__
    self._execute_child(args, executable, preexec_fn, close_fds,
    ~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                        pass_fds, cwd, env,
                        ^^^^^^^^^^^^^^^^^^
```

```
    …<5 lines>…
                         gid, gids, uid, umask,
                         ^^^^^^^^^^^^^^^^^^^^^^
                         start_new_session, process_group)
                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\justi\anaconda3\Lib\subprocess.py", line 1554, in
_execute_child
    hp, ht, pid, tid = _winapi.CreateProcess(executable, args,
                       ~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^
                             # no special security
                             ^^^^^^^^^^^^^^^^^^^^^
    …<4 lines>…
                             cwd,
                             ^^^^
                             startupinfo)
                             ^^^^^^^^^^^^
```

OverallQual	0.575338
Neighborhood	0.525055
GarageArea	0.502075
GrLivArea	0.433316
TotalBsmtSF	0.408730
YearBuilt	0.404795
LotArea	0.399774
GarageCars	0.363229
BsmtQual	0.331366
ExterQual	0.326718
KitchenQual	0.321579
1stFlrSF	0.287102
GarageYrBlt	0.281337
MSSubClass	0.280966
YearRemodAdd	0.276804
FullBath	0.274439
GarageFinish	0.263928
TotRmsAbvGrd	0.208606
FireplaceQu	0.208567
LotFrontage	0.202979
GarageType	0.201094
OpenPorchSF	0.197021
Foundation	0.194043
HeatingQC	0.173456
2ndFlrSF	0.169081
Fireplaces	0.168542
BsmtFinSF1	0.162328
Exterior2nd	0.156561
BsmtFinType1	0.154815
Exterior1st	0.130336
BsmtUnfSF	0.122686

```
MSZoning         0.116164
OverallCond      0.107972
WoodDeckSF       0.104439
MasVnrType       0.096344
HalfBath         0.090613
MasVnrArea       0.087095
LotShape         0.086910
HouseStyle       0.082152
GarageCond       0.080666
SaleType         0.078203
BsmtExposure     0.077071
SaleCondition    0.074363
BedroomAbvGr     0.073342
GarageQual       0.070876
CentralAir       0.067654
BsmtCond         0.052778
PavedDrive       0.047649
Electrical       0.046609
Fence            0.045646
Name: MI Scores, dtype: float64
```



10.0.3 Update preprocess() to include Feature Engineering

[43]:
```
def preprocess(train, y, test):
    # set feature types for both
    train = set_feature_types(train)
    test = set_feature_types(test)

    # clean both datasets
    train = clean(train)
```

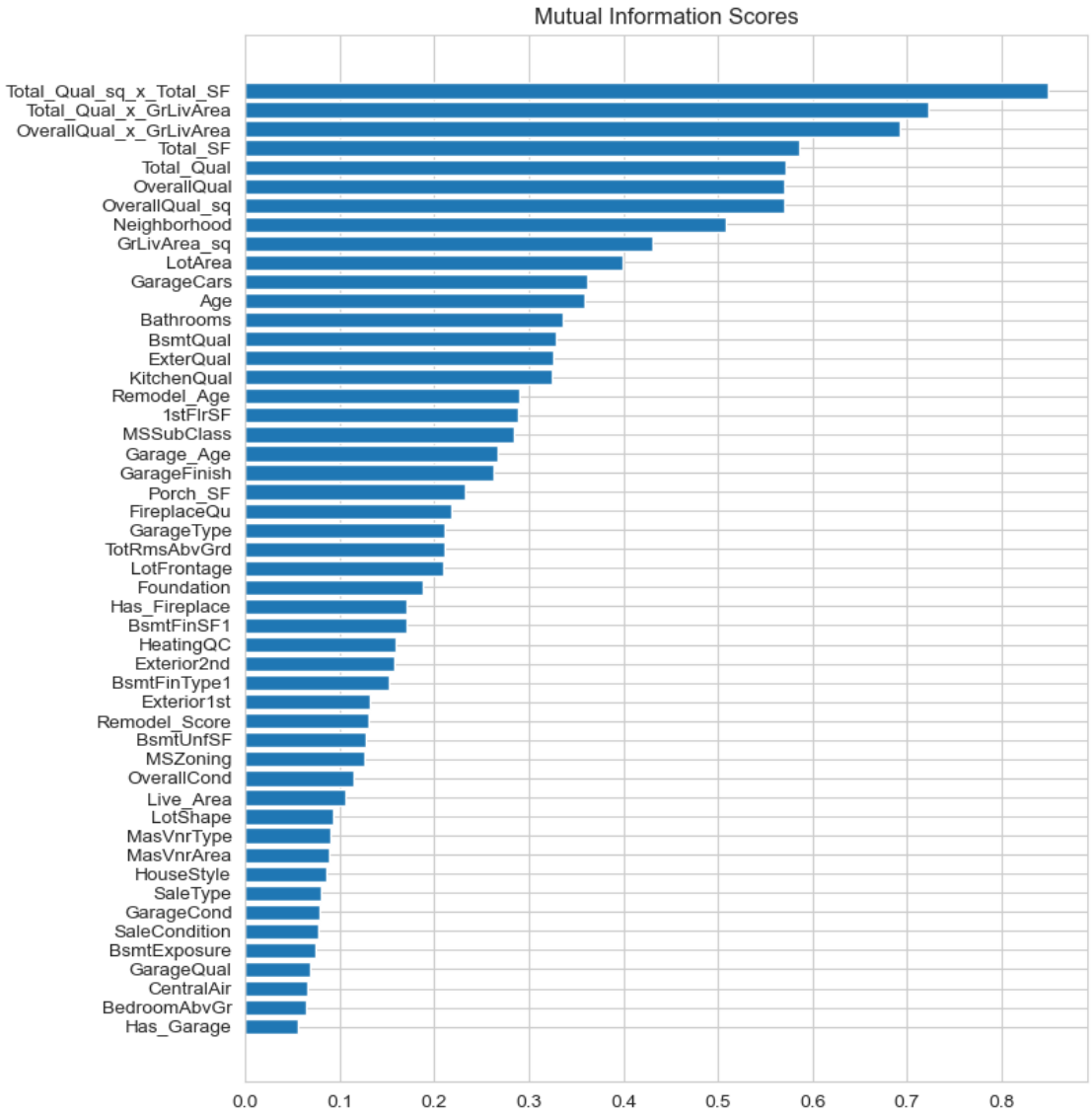
```
    test = clean(test)

    # target encode Neighborhood
    # fit on train to prevent data leakage, then transform both
    target_enc = TargetEncoder(col_name='Neighborhood')
    target_enc.fit(train, y)
    train = target_enc.transform(train)
    test = target_enc.transform(test)

    # encode features ensuring alignment
    train, test = encode(train, test)

    # feature engineering
    train = engineer_features(train)
    test = engineer_features(test)

    # impute
    train = impute(train)
    test = impute(test)

    # drop cols
    train = drop_cols(train)
    test = drop_cols(test)

    return train, test
```

10.0.4 Assess Mutual Information including Engineered Features

[44]:
```
# load data
X, y, test, test_ids = load_data()
X, test = preprocess(X, y, test)

# calculate scores
mi_scores = make_mi_scores(X, y)

# print the top 50 scores
print(mi_scores.head(50))

# plot the scores for visualization
plt.figure(figsize=(8, 10))
plot_mi_scores(mi_scores.head(50))
plt.show()
```

```
Total_Qual_sq_x_Total_SF    0.848265
Total_Qual_x_GrLivArea      0.721711
OverallQual_x_GrLivArea     0.691826
Total_SF                    0.585263
Total_Qual                  0.571127
```

```
OverallQual_sq              0.569937
OverallQual                 0.569937
Neighborhood                0.507754
GrLivArea_sq                0.430905
LotArea                     0.399599
GarageCars                  0.361837
Age                         0.358489
Bathrooms                   0.335247
BsmtQual                    0.328123
ExterQual                   0.325156
KitchenQual                 0.324785
Remodel_Age                 0.289970
1stFlrSF                    0.288264
MSSubClass                  0.283976
Garage_Age                  0.266466
GarageFinish                0.262287
Porch_SF                    0.232874
FireplaceQu                 0.217635
GarageType                  0.211126
TotRmsAbvGrd                0.210946
LotFrontage                 0.210096
Foundation                  0.188307
Has_Fireplace               0.171133
BsmtFinSF1                  0.170674
HeatingQC                   0.159251
Exterior2nd                 0.157451
BsmtFinType1                0.152403
Exterior1st                 0.131757
Remodel_Score               0.130123
BsmtUnfSF                   0.127316
MSZoning                    0.125688
OverallCond                 0.114176
Live_Area                   0.106617
LotShape                    0.093307
MasVnrType                  0.090621
MasVnrArea                  0.088670
HouseStyle                  0.085677
SaleType                    0.079575
GarageCond                  0.078924
SaleCondition               0.077732
BsmtExposure                0.074369
GarageQual                  0.068217
CentralAir                  0.065781
BedroomAbvGr                0.064591
Has_Garage                  0.055570
Name: MI Scores, dtype: float64
```



Feature Engineering allows us to create features that have high mutual information with SalePrice which means these features will contain useful information for our models. One problem introduced with feature engineering is multicolinearity. To address multicolinearity, we will remove variables that are highly correlated.

10.0.5 Model with Feature Engineering

[45]:
```
# all mi scored variables from best to worst
mi_features = mi_scores.index.tolist()

print(mi_features)
```

```
['Total_Qual_sq_x_Total_SF', 'Total_Qual_x_GrLivArea',
```

```
'OverallQual_x_GrLivArea', 'Total_SF', 'Total_Qual', 'OverallQual_sq',
'OverallQual', 'Neighborhood', 'GrLivArea_sq', 'LotArea', 'GarageCars', 'Age',
'Bathrooms', 'BsmtQual', 'ExterQual', 'KitchenQual', 'Remodel_Age', '1stFlrSF',
'MSSubClass', 'Garage_Age', 'GarageFinish', 'Porch_SF', 'FireplaceQu',
'GarageType', 'TotRmsAbvGrd', 'LotFrontage', 'Foundation', 'Has_Fireplace',
'BsmtFinSF1', 'HeatingQC', 'Exterior2nd', 'BsmtFinType1', 'Exterior1st',
'Remodel_Score', 'BsmtUnfSF', 'MSZoning', 'OverallCond', 'Live_Area',
'LotShape', 'MasVnrType', 'MasVnrArea', 'HouseStyle', 'SaleType', 'GarageCond',
'SaleCondition', 'BsmtExposure', 'GarageQual', 'CentralAir', 'BedroomAbvGr',
'Has_Garage', 'BsmtCond', 'PavedDrive', 'Electrical', 'Fence', 'BldgType',
'BsmtFinType2', 'LandContour', 'Has_Basement', 'Alley', 'KitchenAbvGr',
'ExterCond', 'Has_ScreenPorch', 'Has_2nd_Floor', 'LotConfig', 'Heating',
'Condition1', 'Has_3SeasonPorch', 'RoofStyle', 'Functional', 'LowQualFinSF',
'RoofMatl', 'LandSlope', 'BsmtFinSF2', 'Condition2', 'Street', 'Utilities',
'MoSold', 'MiscFeature', 'PoolQC', 'ExterQual_x_Total_SF', 'Garage_Score',
'Has_Pool', 'Has_Misc']
```

[46]:
```
# load data
X, y, test, test_ids = load_data()
X, test = preprocess(X, y, test)

# model 1 pipeline
model1 = Pipeline(steps=[
    ('model', LinearRegression())
])

# score model
rmse_mod1, sd_mod1 = score_model(model1, X, y)

# save score
rmsle_scores.append(('Model 1 - Feature Engineering', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
```

11 Multicolinearity

Deal with Multicolinearity by dropping some highly correlated variables. I played with the threshold. If threshold = 0.99, there is no change in score, as no variables are dropped. If threshold = 0.9, the score gets worse as it drops variables with high correlation that also contain useful information

to the model. This strategy did not lead to score improvements.

[47]:
```
# define list of binary features created during engineering
ohe_features = [
    "Has_Pool",
    "Has_2nd_Floor",
    "Has_Garage",
    "Has_Basement",
    "Has_Fireplace",
    "Has_ScreenPorch",
    "Has_3SeasonPorch",
    "Has_Misc"
]

def drop_highly_correlated(X, y, ohe_features, threshold=0.9, protect_ohe=True,␣
 ↪keep_both_if_ohe=False):
    # drops highly correlated features with specific logic for ohe variables

    # calculate target correlations
    temp_df = X.copy()
    temp_df['target'] = y
    target_corr = temp_df.corr()['target'].abs()

    # create correlation matrix
    corr_matrix = X.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).
 ↪astype(bool))

    to_drop = set()

    for col_name in upper.columns:
        # find correlated pairs
        correlated_pair_names = upper.index[upper[col_name] > threshold].
 ↪tolist()

        for row_name in correlated_pair_names:
            if row_name in to_drop or col_name in to_drop:
                continue

            # check if features are in our ohe list
            is_col_ohe = col_name in ohe_features
            is_row_ohe = row_name in ohe_features
            pair_has_ohe = is_col_ohe or is_row_ohe

            # logic 3: keep both if ohe is involved
            if keep_both_if_ohe and pair_has_ohe:
                continue
```

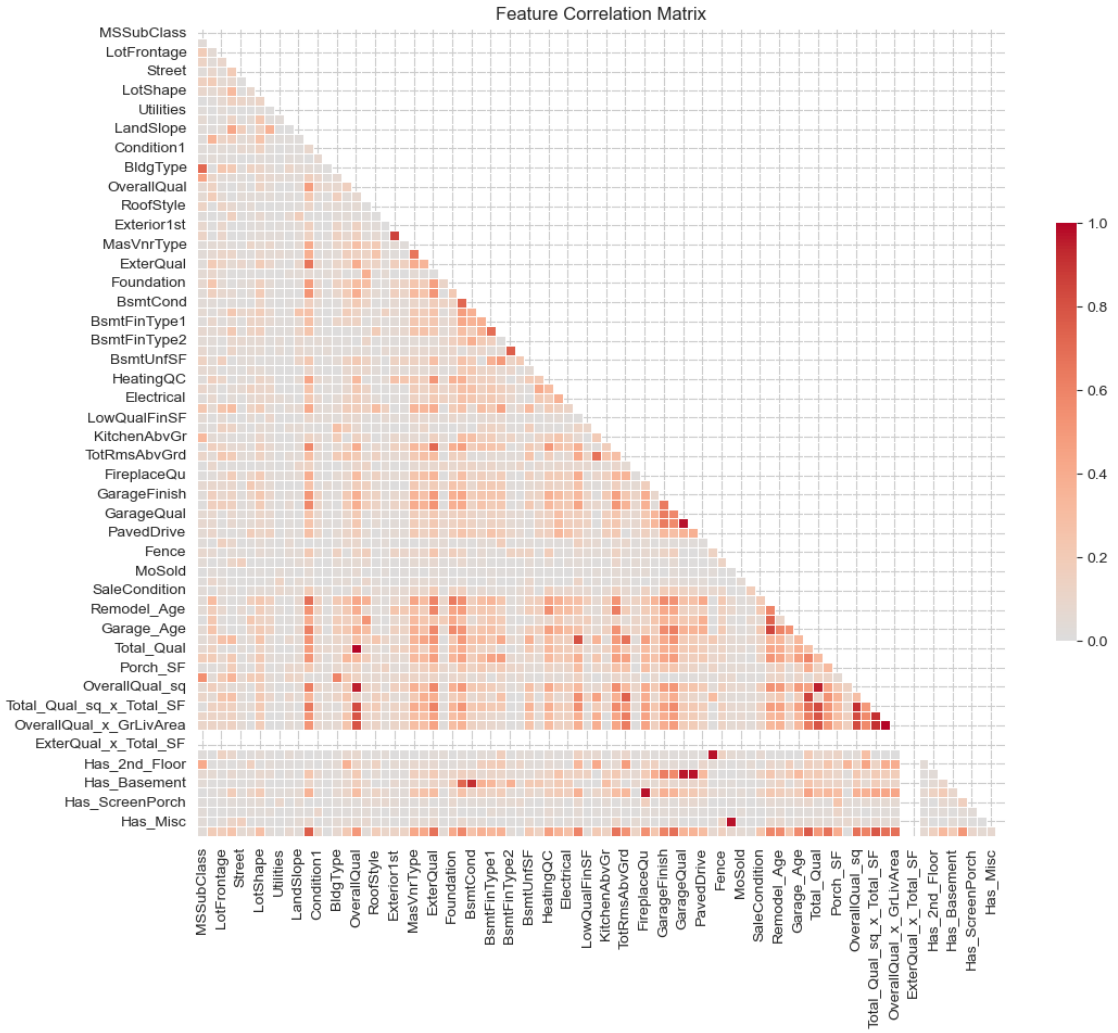
```
            # logic 2: protect ohe (drop the continuous variable instead)
            if protect_ohe and pair_has_ohe:
                # if col is ohe and row is continuous -> drop row
                if is_col_ohe and not is_row_ohe:
                    to_drop.add(row_name)
                    continue
                # if row is ohe and col is continuous -> drop col
                elif is_row_ohe and not is_col_ohe:
                    to_drop.add(col_name)
                    continue

            # logic 1 (default): drop variable with less correlation to target
            corr_col = target_corr.get(col_name, 0)
            corr_row = target_corr.get(row_name, 0)

            if corr_col >= corr_row:
                to_drop.add(row_name)
            else:
                to_drop.add(col_name)

    print(f"Dropping {len(to_drop)} highly correlated features")
    print(f"Dropping: {to_drop}")

    return X.drop(to_drop, axis=1)
```

11.1 Plot Correlation Matrix

[48]:
```
def analyze_correlations(X, y, threshold=0.9):
    # plots a correlation heatmap and returns a table of highly correlated pairs
    # recommending which feature to drop based on correlation with the target

    # combine x and y to calculate all correlations at once
    data = X.copy()
    target_col = 'SalePrice'
    data[target_col] = y

    # filter for numeric columns only
    numeric_data = data.select_dtypes(include=[np.number])

    # calculate absolute correlation matrix
    corr_matrix = numeric_data.corr().abs()

    # plot correlation heatmap
    plt.figure(figsize=(12,10))
    # mask the upper triangle to make it easier to read
    mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
```

```
    sns.heatmap(corr_matrix, mask=mask, cmap='coolwarm', vmax=1, vmin=0,␣
↪center=0,
                square=True, linewidths=.5, cbar_kws={"shrink": .5})
    plt.title('Feature Correlation Matrix')
    plt.show()

    # identify highly correlated pairs and create decision table
    # get the upper triangle of the correlation matrix
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).
↪astype(bool))

    # calculate correlation of all features with the target
    target_corrs = numeric_data.corrwith(numeric_data[target_col]).abs()

    record_list = []

    # iterate through columns to find correlations above threshold
    for col in upper.columns:
        if col == target_col: continue # skip target column

        # find features in this column with correlation > threshold
        high_corr_rows = upper.index[upper[col] > threshold].tolist()

        for row in high_corr_rows:
            if row == target_col: continue # skip target column

            # get the pair correlation value
            pair_corr = upper.loc[row, col]

            # get each feature's correlation with saleprice
            feat1_target_corr = target_corrs[row]
            feat2_target_corr = target_corrs[col]

            # decision logic: drop the feature with lower correlation to target
            if feat1_target_corr < feat2_target_corr:
                drop = row
                keep = col
            else:
                drop = col
                keep = row

            record_list.append({
                'Feature 1': row,
                'Feature 2': col,
                'Pair Correlation': pair_corr,
                'F1-Target Corr': feat1_target_corr,
                'F2-Target Corr': feat2_target_corr,
```

```
                'Drop': drop
            })

    # create dataframe and sort by pair correlation
    results_df = pd.DataFrame(record_list)

    if not results_df.empty:
        results_df = results_df.sort_values(by='Pair Correlation',␣
 ↪ascending=False).reset_index(drop=True)
        print(f"Found {len(results_df)} pairs with correlation > {threshold}")
    else:
        print("No highly correlated pairs found.")

    return results_df

# execution
X, y, test, test_ids = load_data()
X, test = preprocess(X, y, test)

analysis_table = analyze_correlations(X, y, threshold=0.9)
display(analysis_table)
```



Found 14 pairs with correlation > 0.9

	Feature 1	Feature 2	Pair Correlation	\
0	OverallQual	Total_Qual	1.000000	
1	Total_Qual_x_GrLivArea	OverallQual_x_GrLivArea	1.000000	
2	GarageQual	GarageCond	0.975987	
3	MiscFeature	Has_Misc	0.974487	
4	FireplaceQu	Has_Fireplace	0.971868	
5	PoolQC	Has_Pool	0.970875	
6	GarageCond	Has_Garage	0.968804	
7	GarageQual	Has_Garage	0.966539	
8	Total_Qual	OverallQual_sq	0.947351	
9	OverallQual	OverallQual_sq	0.947351	
10	OverallQual_sq	Total_Qual_sq_x_Total_SF	0.916164	
11	BsmtCond	Has_Basement	0.911712	
12	Total_Qual_sq_x_Total_SF	Total_Qual_x_GrLivArea	0.908882	

```
13  Total_Qual_sq_x_Total_SF   OverallQual_x_GrLivArea          0.908882
```

	F1-Target Corr	F2-Target Corr	Drop
0	0.503215	0.503215	Total_Qual
1	0.685099	0.685099	OverallQual_x_GrLivArea
2	0.357347	0.352098	GarageCond
3	0.073822	0.073430	Has_Misc
4	0.542706	0.510026	Has_Fireplace
5	0.076337	0.069835	Has_Pool
6	0.352098	0.322998	Has_Garage
7	0.357347	0.322998	Has_Garage
8	0.503215	0.673233	Total_Qual
9	0.503215	0.673233	OverallQual
10	0.673233	0.775809	OverallQual_sq
11	0.264531	0.199634	Has_Basement
12	0.775809	0.685099	Total_Qual_x_GrLivArea
13	0.775809	0.685099	OverallQual_x_GrLivArea

11.1.1 preprocess_drop is preprocess with Dropping Highly correlated variables

[49]:
```python
def preprocess_drop(train, y, test, t=0.9, protect_ohe=True,␣
↪keep_both_if_ohe=False):
    # set feature types for both
    train = set_feature_types(train)
    test = set_feature_types(test)

    # clean both datasets
    train = clean(train)
    test = clean(test)

    # target encode Neighborhood
    target_enc = TargetEncoder(col_name='Neighborhood')
    target_enc.fit(train, y)
    train = target_enc.transform(train)
    test = target_enc.transform(test)

    # encode features ensuring alignment
    train, test = encode(train, test)

    # feature engineering
    train = engineer_features(train)
    test = engineer_features(test)

    # impute
    train = impute(train)
    test = impute(test)
```

```
    # drop cols
    train = drop_cols(train)
    test = drop_cols(test)

    # toggles
    train = drop_highly_correlated(
        train,
        y,
        ohe_features,
        threshold=t,
        protect_ohe=protect_ohe,
        keep_both_if_ohe=keep_both_if_ohe
    )

    # Align test columns
    test = test[train.columns]

    return train, test
```

11.1.2 Drop Highly Correlated (control threshold t)

[50]:
```
t=0.95
# load data
X, y, test, test_ids = load_data()

# Keep Both = True
X, test = preprocess_drop(X, y, test, t=t, protect_ohe=False,␣
 ↪keep_both_if_ohe=False)

# score
rmse_mod1, sd_mod1 = score_model(model1, X, y)

# save score
rmsle_scores.append(('Model 1 - drop standard', rmse_mod1, sd_mod1))

# show results
# show_scores(rmsle_scores)
```

```
Dropping 7 highly correlated features
Dropping: {'OverallQual', 'Has_Fireplace', 'Total_Qual_x_GrLivArea', 'Has_Pool',
'GarageCond', 'Has_Garage', 'Has_Misc'}
```

[51]:
```
# load data
X, y, test, test_ids = load_data()

# Protect OHE = True
```

```
X, test = preprocess_drop(X, y, test, t=t, protect_ohe=True,␣
 ↪keep_both_if_ohe=False)

model1 = Pipeline(steps=[
    ('model', LinearRegression())
])

# score
rmse_mod1, sd_mod1 = score_model(model1, X, y)

# save score
rmsle_scores.append(('Model 1 - drop keep ohe', rmse_mod1, sd_mod1))

# show results
# show_scores(rmsle_scores)
```

```
Dropping 7 highly correlated features
Dropping: {'OverallQual', 'MiscFeature', 'Total_Qual_x_GrLivArea', 'GarageCond',
'PoolQC', 'GarageQual', 'FireplaceQu'}
```

[52]:
```
# load data
X, y, test, test_ids = load_data()

# Protect OHE = False
X, test = preprocess_drop(X, y, test, t=t, protect_ohe=False,␣
 ↪keep_both_if_ohe=True)

# score
rmse_mod1, sd_mod1 = score_model(model1, X, y)

# save score
rmsle_scores.append(('Model 1 - drop keep both', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Dropping 3 highly correlated features
Dropping: {'OverallQual', 'Total_Qual_x_GrLivArea', 'GarageCond'}
Model                                  | RMSLE      | SD
-----------------------------------------------------------------------
Baseline - normalized target           | 0.230609   | 0.010861
Model 1 - normalized target            | 0.195823   | 0.019063
Model 1 - Preprocess                   | 0.265751   | 0.035764
Model 1 - All Features                 | 0.163591   | 0.041842
Model 1 - Feature Engineering          | 0.137945   | 0.027634
Model 1 - drop standard                | 0.138487   | 0.029015
Model 1 - drop keep ohe                | 0.138833   | 0.029472
Model 1 - drop keep both               | 0.138080   | 0.028071
```

12 Skew

12.0.1 Method to create a skew report

This method generates the histograms before and after log transforming variables with skew > threshold (0.5). Then it creates a table with columns for each variable, the skew before, the skew after, and if the log transform was beneficial. A list of the features that benefit from a log transform is saved.

[53]:
```
def create_skew_report(df_before, df_after, feature_list):
    report_data = []

    for col in feature_list:
        # calculate skew stats
        s_before = skew(df_before[col].dropna())
        s_after = skew(df_after[col].dropna())

        # determine if transformed
        # we check if the column name is in our transformed list
        is_transformed = not df_before[col].equals(df_after[col])

        beneficial = "N/A"
        if is_transformed:
            # check if absolute skew decreased
            if abs(s_after) < abs(s_before):
                beneficial = "YES"
            else:
                beneficial = "NO"

        if is_transformed:
            report_data.append({
                'Feature': col,
                'Skew_Before': s_before,
                'Skew_After': s_after,
                'Beneficial': beneficial
            })

    return pd.DataFrame(report_data).set_index('Feature').
↪sort_values(by='Skew_Before', ascending=False)
```

12.0.2 Assess Skew of All Numeric Variables

Now that we have engineered features, we must assess the skew of all the numeric variables and perform log transforms on all variables that benefit (skew gets closer to 0).

[54]:
```
def visualize_skew_strategy(df, numeric_cols, fig_width=20, row_height=4):
    # analyzes numeric features, outputs a detailed skewness table,
    # and plots a zoomed-in 'original' (red) vs 'best' (green) comparison.
    print(f"Analyzing {len(numeric_cols)} Numeric Features...")
```

```
    X_num = df[numeric_cols].copy()

    # define pipelines
    # original (scale only)
    pipe_scale = Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', RobustScaler())
    ])

    # quantile
    pipe_quant = Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('power', QuantileTransformer(output_distribution='normal',␣
↪random_state=42)),
        ('scaler', RobustScaler())
    ])

    # log
    pipe_log = Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('power', FunctionTransformer(lambda x: np.log1p(np.maximum(x, 0)),␣
↪validate=False)),
        ('scaler', RobustScaler())
    ])

    # box-cox
    pipe_boxcox = Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('add_one', FunctionTransformer(
            lambda df: df.apply(lambda x: x - x.min() + 1 if x.min() <= 0 else␣
↪x),
            validate=False
        )),
        ('power', PowerTransformer(method='box-cox')),
        ('scaler', RobustScaler())
    ])

    # fit and transform
    pipe_scale.set_output(transform="pandas")
    pipe_quant.set_output(transform="pandas")
    pipe_log.set_output(transform="pandas")
    pipe_boxcox.set_output(transform="pandas")

    print("Applying transformations...")
    import warnings
    with warnings.catch_warnings():
```

```
        warnings.simplefilter("ignore")
        X_scaled = pipe_scale.fit_transform(X_num)
        X_quant = pipe_quant.fit_transform(X_num)
        X_log = pipe_log.fit_transform(X_num)
        X_boxcox = pipe_boxcox.fit_transform(X_num)

    # calculate skews
    skew_raw = X_scaled.apply(lambda x: skew(x.dropna()))
    skew_quant = X_quant.apply(lambda x: skew(x.dropna()))
    skew_log = X_log.apply(lambda x: skew(x.dropna()))
    skew_boxcox = X_boxcox.apply(lambda x: skew(x.dropna()))

    # determine best strategy
    strategy_df = pd.DataFrame({
        'Scale Only': skew_raw.abs(),
        'Quantile': skew_quant.abs(),
        'Log': skew_log.abs(),
        'Box-Cox': skew_boxcox.abs()
    })

    best_strategy_col = strategy_df.idxmin(axis=1)

    # create full report dataframe
    report = pd.DataFrame({
        'Original_Skew': skew_raw,
        'Log_Skew': skew_log,
        'BoxCox_Skew': skew_boxcox,
        'Quantile_Skew': skew_quant,
        'Best_Method': best_strategy_col,
    })

    # get signed skew of best method for display
    report['Best_Signed_Skew'] = report.apply(
        lambda row: row['Original_Skew'] if row['Best_Method'] == 'Scale Only'␣
↪else
                    (row['Quantile_Skew'] if row['Best_Method'] == 'Quantile'␣
↪else
                    (row['Log_Skew'] if row['Best_Method'] == 'Log' else␣
↪row['BoxCox_Skew'])), axis=1
    )

    report['Is_Good'] = report['Best_Signed_Skew'].abs() < 0.5
    report['Sort_Key'] = report['Original_Skew'].abs()
    report = report.sort_values('Sort_Key', ascending=False).
↪drop(columns=['Sort_Key'])

    # display full table
```

```
    print(f"\nSkewness Optimization Report ({len(report)} features):")

    # define the columns exactly as requested in the detailed view
    display_cols = ['Original_Skew', 'Log_Skew', 'BoxCox_Skew',␣
↪'Quantile_Skew', 'Best_Method', 'Best_Signed_Skew', 'Is_Good']

    styled_report = (report[display_cols].style
        .format("{:.2f}", subset=['Original_Skew', 'Log_Skew', 'BoxCox_Skew',␣
↪'Quantile_Skew', 'Best_Signed_Skew'])
        .background_gradient(cmap='coolwarm', subset=['Original_Skew'],␣
↪vmin=-3, vmax=3)
        .apply(lambda x: ['background-color: #d4edda' if v else '' for v in x],␣
↪subset=['Is_Good'], axis=1)
    )
    display(styled_report)

    # generate "red vs green" zoomed plots
    n_features = len(numeric_cols)
    cols = 3
    rows = math.ceil(n_features / cols)

    print(f"\nGenerating {n_features} comparison plots (Original vs Best)...")

    fig, axes = plt.subplots(rows, cols, figsize=(fig_width, row_height * rows))
    fig.subplots_adjust(top=0.95, hspace=0.4)
    axes = np.array(axes).flatten()

    # helper to calculate zoomed limits (1st to 99th percentile)
    def get_zoomed_limits(series1, series2):
        s1 = series1.dropna()
        s2 = series2.dropna()
        combined = np.concatenate([s1, s2])
        low = np.percentile(combined, 1)
        high = np.percentile(combined, 99)
        padding = (high - low) * 0.1 # add 10% padding
        return low - padding, high + padding

    for i, ax in enumerate(axes):
        if i >= n_features:
            ax.set_visible(False)
            continue

        col_name = report.index[i]
        best_strat = report.loc[col_name, 'Best_Method']

        # prepare data for plotting
        if best_strat == 'Scale Only':
```

```
            data_best = X_scaled[col_name]
            method_color = 'red'
            label_best = "Original (Best)"
        else:
            if best_strat == 'Log': data_best = X_log[col_name]
            elif best_strat == 'Box-Cox': data_best = X_boxcox[col_name]
            elif best_strat == 'Quantile': data_best = X_quant[col_name]
            method_color = 'green'
            label_best = f"Best: {best_strat}"

        # plot original (red) - always visible
        sns.kdeplot(X_scaled[col_name], ax=ax, color='red', fill=True, alpha=0.
↪3, label='Original')

        # plot best (green) - only if different from original
        if best_strat != 'Scale Only':
            sns.kdeplot(data_best, ax=ax, color='green', fill=True, alpha=0.3,␣
↪label=label_best)

            # zoom: calculate limits based on percentiles of both
            xlim_min, xlim_max = get_zoomed_limits(X_scaled[col_name],␣
↪data_best)
            ax.set_xlim(xlim_min, xlim_max)
        else:
            # if original is best, just zoom on original
            xlim_min, xlim_max = get_zoomed_limits(X_scaled[col_name],␣
↪X_scaled[col_name])
            ax.set_xlim(xlim_min, xlim_max)

        # title
        s_orig = report.loc[col_name, 'Original_Skew']
        s_best = report.loc[col_name, 'Best_Signed_Skew']
        ax.set_title(f"{col_name}\nOrig: {s_orig:.2f} → {best_strat}: {s_best:.
↪2f}", fontsize=11, fontweight='bold')
        ax.set_yticks([])
        ax.set_xlabel('')

        # legend only on first plot
        if i == 0:
            ax.legend(loc='upper right')

    plt.tight_layout()
    plt.show()

    return report
```

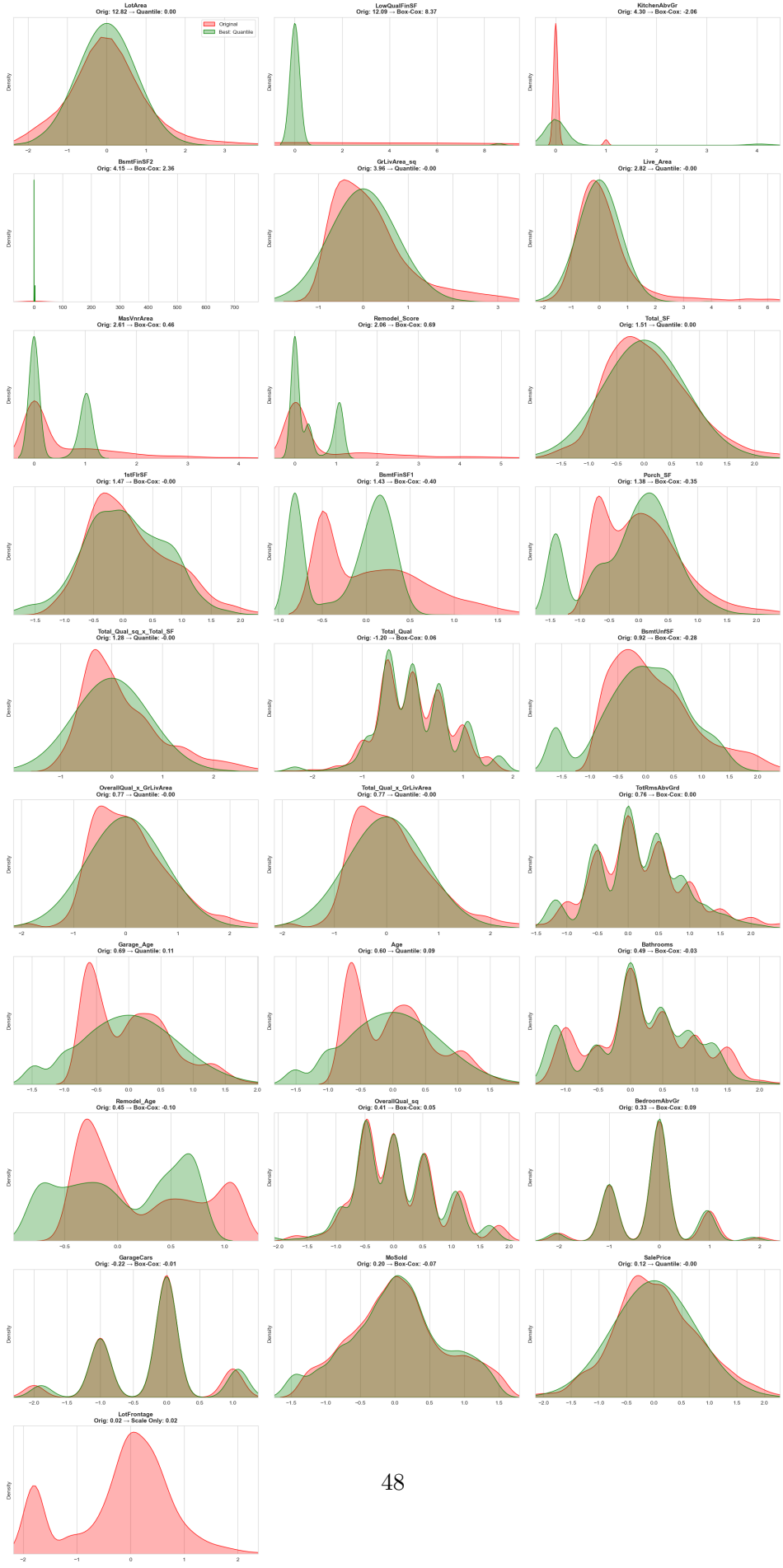
[55]:
```
# define map of features to drop
component_map = {
    "Age": ["YearBuilt", "YrSold"],
    "Remodel_Age": ["YearRemodAdd", "YrSold"],
    "Garage_Age": ["GarageYrBlt", "YrSold"],
    "Remodel_Score": ["YearRemodAdd", "YearBuilt"],
    "Total_SF": ["TotalBsmtSF", "GrLivArea"],
    "Bathrooms": ["FullBath", "HalfBath", "BsmtFullBath", "BsmtHalfBath"],
    "Porch_SF": ["OpenPorchSF", "3SsnPorch", "EnclosedPorch", "ScreenPorch",␣
 ↪"WoodDeckSF"],
    # one hot encoded features
    "Has_Pool": ["PoolArea"],
    "Has_2nd_Floor": ["2ndFlrSF"],
    "Has_Garage": ["GarageArea"],
    "Has_Basement": ["TotalBsmtSF"],
    "Has_Fireplace": ["Fireplaces"],
    "Has_ScreenPorch": ["ScreenPorch"],
    "Has_3SeasonPorch": ["3SsnPorch"],
    "Has_Misc": ["MiscVal"]
}

# extract columns to drop
cols_to_drop = []
for components in component_map.values():
    cols_to_drop.extend(components)

# remove duplicates
final_drop_list = list(set(cols_to_drop))

# toggles
exclude_dropped = True
exclude_onehot = True

# load data
X, y, test, test_ids = load_data()
X, test = preprocess(X, y, test)
data = pd.concat([X, test])
data['SalePrice'] = y

# define base numeric features
defined_categorical_cols = set(features_nom + list(ordered_levels.keys()))
all_numeric = [
    col for col in data.columns
    if pd.api.types.is_numeric_dtype(data[col])
    and col not in defined_categorical_cols
    and col not in ['Id']
]
```

```
# apply filters
current_features = all_numeric.copy()

if exclude_dropped:
    # filter out dropped columns
    current_features = [c for c in current_features if c not in final_drop_list]
    print(f"Filter Active: Excluded {len(set(all_numeric) - set(current_features))} dropped features.")

if exclude_onehot:
    # filter out binary/constant columns
    before_count = len(current_features)
    current_features = [c for c in current_features if data[c].dropna().nunique() > 2]
    print(f"Filter Active: Excluded {before_count - len(current_features)} one-hot/binary features.")

# run analysis
print(f"\nRunning skew analysis on {len(current_features)} features...")
skew_report = visualize_skew_strategy(data, current_features)
```

```
Filter Active: Excluded 0 dropped features.
Filter Active: Excluded 10 one-hot/binary features.

Running skew analysis on 28 features…
Analyzing 28 Numeric Features…
Applying transformations…

Skewness Optimization Report (28 features):
```

```
<pandas.io.formats.style.Styler at 0x1b0eb0f96a0>
```

```
Generating 28 comparison plots (Original vs Best)…
```


LotArea
Orig: 12.82 → Quantile: 0.00
Original
Best: Quantile
LowQualFinSF
Orig: 12.09 → Box-Cox: 8.37
KitchenAbvGr
Orig: 4.30 → Box-Cox: -2.06
BsmtFinSF2
Orig: 4.15 → Box-Cox: 2.36
GrLivArea_sq
Orig: 3.96 → Quantile: -0.00
Live_Area
Orig: 2.82 → Quantile: -0.00
MasVnrArea
Orig: 2.61 → Box-Cox: 0.46
Remodel_Score
Orig: 2.06 → Box-Cox: 0.69
Total_SF
Orig: 1.51 → Quantile: 0.00
1stFlrSF
Orig: 1.47 → Box-Cox: -0.00
BsmtFinSF1
Orig: 1.43 → Box-Cox: -0.40
Porch_SF
Orig: 1.38 → Box-Cox: -0.35
Total_Qual_sq_x_Total_SF
Orig: 1.28 → Quantile: -0.00
Total_Qual
Orig: -1.20 → Box-Cox: 0.06
BsmtUnfSF
Orig: 0.92 → Box-Cox: -0.28
OverallQual_x_GrLivArea
Orig: 0.77 → Quantile: -0.00
Total_Qual_x_GrLivArea
Orig: 0.77 → Quantile: -0.00
TotRmsAbvGrd
Orig: 0.76 → Box-Cox: 0.00
Garage_Age
Orig: 0.69 → Quantile: 0.11
Age
Orig: 0.60 → Quantile: 0.09
Bathrooms
Orig: 0.49 → Box-Cox: -0.03
Remodel_Age
Orig: 0.45 → Box-Cox: -0.10
OverallQual_sq
Orig: 0.41 → Box-Cox: 0.05
BedroomAbvGr
Orig: 0.33 → Box-Cox: 0.09
GarageCars
Orig: -0.22 → Box-Cox: -0.01
MoSold
Orig: 0.20 → Box-Cox: -0.07
SalePrice
Orig: 0.12 → Quantile: -0.00
LotFrontage
Orig: 0.02 → Scale Only: 0.02
Density

12.0.3 Apply Optimal Transformations

[56]:
```
# optimal transformations for each numeric feature
transformation_map = {
    'LotFrontage': 'Quantile', 'ExterCond': 'Quantile', 'BsmtQual': 'Quantile',
    'BsmtUnfSF': 'Scale Only', 'ExterQual': 'Scale Only', 'OverallQual_sq':␣
 ↪'Quantile',
    'BsmtCond': 'Quantile', 'OverallCond': 'Scale Only', 'BsmtFinSF1':␣
 ↪'Quantile',
    'Age': 'Scale Only', 'HeatingQC': 'Scale Only', 'YearRemodAdd': 'Scale␣
 ↪Only',
    'Remodel_Age': 'Scale Only', 'TotalBsmtSF': 'Scale Only', 'MasVnrArea':␣
 ↪'Quantile',
    'KitchenQual': 'Scale Only', 'Total_Qual': 'Scale Only', '2ndFlrSF':␣
 ↪'Quantile',
    'Garage_Score': 'Scale Only', 'WoodDeckSF': 'Quantile', 'GarageArea':␣
 ↪'Scale Only',
    'FireplaceQu': 'Scale Only', 'OpenPorchSF': 'Quantile', 'GarageYrBlt':␣
 ↪'Quantile',
    'TotRmsAbvGrd': 'Quantile', 'YearBuilt': 'Quantile', 'GrLivArea':␣
 ↪'Quantile',
    'GrLivArea_sq': 'Quantile', 'Total_Qual_sq_x_TotalSF': 'Quantile',
    'Total_Qual_x_GrLivArea': 'Quantile', '1stFlrSF': 'Quantile',
    'Total_Qual_x_TotalSF': 'Quantile', 'LotArea': 'Quantile',
    'TotalSF_sq': 'Quantile', 'TotalSF': 'Quantile'
}

# apply tranasformations
def optimal_transform(df, trans_map=transformation_map):
    df_out = df.copy()
    cols_to_process = [c for c in trans_map.keys() if c in df_out.columns]

    for col in cols_to_process:
        method = trans_map[col]
        col_data = df_out[[col]].values

        if method == 'Quantile':
            qt = QuantileTransformer(output_distribution='normal',␣
 ↪random_state=42)
            df_out[col] = qt.fit_transform(col_data).flatten()
        elif method == 'Log':
            df_out[col] = np.log1p(np.maximum(df_out[col], 0))
        elif method == 'Box-Cox':
            min_val = np.min(col_data)
```

```
            if min_val <= 0: col_data = col_data - min_val + 1.0
            pt = PowerTransformer(method='box-cox')
            df_out[col] = pt.fit_transform(col_data).flatten()

    return df_out
```

12.0.4 Update Preprocess to Apply Optimal Tranformations

[57]:
```
def preprocess(train, y, test):
    # set feature types for both
    train = set_feature_types(train)
    test = set_feature_types(test)

    # clean both datasets
    train = clean(train)
    test = clean(test)

    # target encode Neighborhood
    # We fit ONLY on train to prevent data leakage, then transform both
    target_enc = TargetEncoder(col_name='Neighborhood')
    target_enc.fit(train, y)
    train = target_enc.transform(train)
    test = target_enc.transform(test)

    # encode features ensuring alignment
    train, test = encode(train, test)

    # feature engineering
    train = engineer_features(train)
    test = engineer_features(test)

    # impute
    train = impute(train)
    test = impute(test)

    # Apply Optimized Transformations
    train = optimal_transform(train, transformation_map)
    test = optimal_transform(test, transformation_map)

    # drop cols
    train = drop_cols(train)
    test = drop_cols(test)

    return train, test
```

12.0.5 Test Transformations on Model

[58]:
```
# load data
X, y, test, test_ids = load_data()
X, test = preprocess(X, y, test)

# model 1 pipeline
model1 = Pipeline(steps=[
    ('model', LinearRegression())
])

# subset and score model 1
X_mod1 = X[mi_features]
rmse_mod1, sd_mod1 = score_model(model1, X_mod1, y)

# save model 1 score
rmsle_scores.append(('Model 1 - MI FE Transform', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Model                                       | RMSLE      | SD
------------------------------------------------------------------------
Baseline - normalized target                | 0.230609   | 0.010861
Model 1 - normalized target                 | 0.195823   | 0.019063
Model 1 - Preprocess                        | 0.265751   | 0.035764
Model 1 - All Features                      | 0.163591   | 0.041842
Model 1 - Feature Engineering               | 0.137945   | 0.027634
Model 1 - drop standard                     | 0.138487   | 0.029015
Model 1 - drop keep ohe                     | 0.138833   | 0.029472
Model 1 - drop keep both                    | 0.138080   | 0.028071
Model 1 - MI FE Transform                   | 0.145121   | 0.035136
```

12.0.6 Test best transformations on Model 1

[59]:
```
# load data
X, y, test, test_ids = load_data()
X, test = preprocess_drop(X, y, test)

# model 1 pipeline
model1 = Pipeline(steps=[
    ('model', LinearRegression())
])

# score
rmse_mod1, sd_mod1 = score_model(model1, X, y)

# save model 1 score
```

```
rmsle_scores.append(('Model 1 - Optimal Transforms', rmse_mod1, sd_mod1))

# show results
show_scores(rmsle_scores)
```

```
Dropping 11 highly correlated features
Dropping: {'OverallQual', 'Total_Qual', 'MiscFeature', 'Total_Qual_x_GrLivArea',
'BsmtCond', 'OverallQual_sq', 'GarageCond', 'PoolQC', 'OverallQual_x_GrLivArea',
'GarageQual', 'FireplaceQu'}
Model                                    | RMSLE      | SD
-------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
```

13 Implementing Models

I score each model with the baseline features, then with the optimized data to evaluate how much of an improvement was made.

optimized data

13.0.1 Lasso

[60]:
```
# define lasso pipeline
lasso_pipeline = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor),
    ('scaler', RobustScaler()), # scales values
    ('Lasso', Lasso(alpha=0.0005, random_state=42))
])

# get rmse, sd
rmse_lasso, sd_lasso = score_model(lasso_pipeline, X_baseline, y)

# print score
print(f"Lasso Score: {rmse_lasso:.5f} (SD: {sd_lasso:.5f})")

# add score to rmsle_scores
rmsle_scores.append(('Lasso - Baseline', rmse_lasso, sd_lasso))

# show results
```

```
show_scores(rmsle_scores)
```

```
Lasso Score: 0.23114 (SD: 0.01146)
Model                                          | RMSLE      | SD
-------------------------------------------------------------------
Baseline - normalized target                   | 0.230609   | 0.010861
Model 1 - normalized target                    | 0.195823   | 0.019063
Model 1 - Preprocess                           | 0.265751   | 0.035764
Model 1 - All Features                         | 0.163591   | 0.041842
Model 1 - Feature Engineering                  | 0.137945   | 0.027634
Model 1 - drop standard                        | 0.138487   | 0.029015
Model 1 - drop keep ohe                        | 0.138833   | 0.029472
Model 1 - drop keep both                       | 0.138080   | 0.028071
Model 1 - MI FE Transform                      | 0.145121   | 0.035136
Model 1 - Optimal Transforms                   | 0.143010   | 0.027290
Lasso - Baseline                               | 0.231138   | 0.011462
```

[61]:
```
# load data
X, y, test, test_ids = load_data()
X, test = preprocess(X, y, test)

# define lasso pipeline
lasso_pipeline = Pipeline(steps=[
    ('scaler', RobustScaler()), # scales values
    ('Lasso', Lasso(alpha=0.0005, random_state=42))
])

# get rmse, sd
rmse_lasso, sd_lasso = score_model(lasso_pipeline, X, y)

# print score
print(f"Lasso Score: {rmse_lasso:.5f} (SD: {sd_lasso:.5f})")

# add score to rmsle_scores
rmsle_scores.append(('Lasso', rmse_lasso, sd_lasso))

# show results
show_scores(rmsle_scores)
```

```
Lasso Score: 0.14039 (SD: 0.02862)
Model                                          | RMSLE      | SD
-------------------------------------------------------------------
Baseline - normalized target                   | 0.230609   | 0.010861
Model 1 - normalized target                    | 0.195823   | 0.019063
Model 1 - Preprocess                           | 0.265751   | 0.035764
Model 1 - All Features                         | 0.163591   | 0.041842
Model 1 - Feature Engineering                  | 0.137945   | 0.027634
Model 1 - drop standard                        | 0.138487   | 0.029015
```

```
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
```

14 Model 2 : Random Forest

Establish Baseline Score using Baseline Features

[62]:
```
# pipeline
model_rf = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor),
    ('model', RandomForestRegressor(n_estimators=100, random_state=42))
])

# score
rmse_rf, sd_rf = score_model(model_rf, X_baseline, y)

# save rf score
rmsle_scores.append(('Model 2 - Random Forest Baseline', rmse_rf, sd_rf))

# show results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
```

We use all features and do not drop highly correlated variables as tree models deal with multicollinearity well and instead benefit from increased information.

[63]:
```
# pipeline
model_rf = Pipeline(steps=[
    ('model', RandomForestRegressor(n_estimators=100, random_state=42))
```

```
])

# score
rmse_rf, sd_rf = score_model(model_rf, X, y)

# save rf score
rmsle_scores.append(('Model 2 - Random Forest', rmse_rf, sd_rf))

# show results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
Model 2 - Random Forest                  | 0.137574   | 0.016046
```

15 Model 3 - SVM

[64]:
```
# pipeline
model_svm = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor),
    ('scaler', StandardScaler()), # SVM requires scaling
    ('model', SVR(C=1.0, epsilon=0.2))
])

# use all features
rmse_svm, sd_svm = score_model(model_svm, X_baseline, y)

# add score
rmsle_scores.append(('Model 3: SVM - Baseline', rmse_svm, sd_svm))

# results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
```

```
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
Model 2 - Random Forest                  | 0.137574   | 0.016046
Model 3: SVM - Baseline                  | 0.221384   | 0.009869
```

[65]:
```python
# pipeline
model_svm = Pipeline(steps=[
    ('scaler', StandardScaler()), # SVM requires scaling
    ('model', SVR(C=1.0, epsilon=0.2))
])

# use all features
rmse_svm, sd_svm = score_model(model_svm, X, y)

# add score
rmsle_scores.append(('Model 3: SVM', rmse_svm, sd_svm))

# results
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
Model 2 - Random Forest                  | 0.137574   | 0.016046
Model 3: SVM - Baseline                  | 0.221384   | 0.009869
```

```
Model 3: SVM                                | 0.181070   | 0.018990
```

[66]:
```
# pipeline
model_svm = Pipeline(steps=[
    ('scaler', StandardScaler()), # SVM requires scaling
    ('pca', PCA(n_components=0.95)),     # PCA: Keep 95% of the variance
    ('model', SVR(C=1.0, epsilon=0.2))
])

# use all features
rmse_svm, sd_svm = score_model(model_svm, X, y)

# add score
rmsle_scores.append(('Model 3: SVM - PCA', rmse_svm, sd_svm))

# results
show_scores(rmsle_scores)
```

```
Model                                       | RMSLE      | SD
--------------------------------------------------------------------------
Baseline - normalized target                | 0.230609   | 0.010861
Model 1 - normalized target                 | 0.195823   | 0.019063
Model 1 - Preprocess                        | 0.265751   | 0.035764
Model 1 - All Features                      | 0.163591   | 0.041842
Model 1 - Feature Engineering               | 0.137945   | 0.027634
Model 1 - drop standard                     | 0.138487   | 0.029015
Model 1 - drop keep ohe                     | 0.138833   | 0.029472
Model 1 - drop keep both                    | 0.138080   | 0.028071
Model 1 - MI FE Transform                   | 0.145121   | 0.035136
Model 1 - Optimal Transforms                | 0.143010   | 0.027290
Lasso - Baseline                            | 0.231138   | 0.011462
Lasso                                       | 0.140390   | 0.028622
Model 2 - Random Forest Baseline            | 0.223581   | 0.008681
Model 2 - Random Forest                     | 0.137574   | 0.016046
Model 3: SVM - Baseline                     | 0.221384   | 0.009869
Model 3: SVM                                | 0.181070   | 0.018990
Model 3: SVM - PCA                          | 0.180902   | 0.019918
```

16 Model 4 XGBRegressor

[67]:
```
# pipeline
model_xgb = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor),
    ('model', XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=-1))
])

# score
```

```
rmse_xgb, sd_xgb = score_model(model_xgb, X_baseline, y)

# save score
rmsle_scores.append(('XGBoost - Baseline', rmse_xgb, sd_xgb))
```

[68]:
```
# pipeline
model_xgb = Pipeline(steps=[
    ('model', XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=-1))
])

# score
rmse_xgb, sd_xgb = score_model(model_xgb, X, y)

# save score
rmsle_scores.append(('XGBoost', rmse_xgb, sd_xgb))
```

17 Model 5: LGMBRegressor

[69]:
```
# pipeline
model_lgbm = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor),
    ('model', LGBMRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=-1))
])

# score model
rmse_lgbm, sd_lgbm = score_model(model_lgbm, X_baseline, y)

# save score
rmsle_scores.append(('LGBM - Baseline', rmse_lgbm, sd_lgbm))
```

[70]:
```
# pipeline
model_lgbm = Pipeline(steps=[
    ('model', LGBMRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=-1))
])

# score model
rmse_lgbm, sd_lgbm = score_model(model_lgbm, X, y)

# save score
rmsle_scores.append(('LGBM', rmse_lgbm, sd_lgbm))
```

18 Model 6: CatBoost

[71]:
```
# pipeline
model_cat = Pipeline(steps=[
    ('preprocessor', pipe_preprocessor),
    # verbose=0 silences the training output
    ('model', CatBoostRegressor(n_estimators=1000, learning_rate=0.05,␣
 ↪verbose=0))
])

# score model
rmse_cat, sd_cat = score_model(model_cat, X_baseline, y)

# save score
rmsle_scores.append(('Cat - Baseline', rmse_cat, sd_cat))
```

[72]:
```
# pipeline
model_cat = Pipeline(steps=[
    # verbose=0 silences the training output
    ('model', CatBoostRegressor(n_estimators=1000, learning_rate=0.05,␣
 ↪verbose=0))
])

# score model
rmse_cat, sd_cat = score_model(model_cat, X, y)

# save score
rmsle_scores.append(('Cat', rmse_cat, sd_cat))
```

[73]:
```
show_scores(rmsle_scores)
```

```
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
Model 2 - Random Forest                  | 0.137574   | 0.016046
Model 3: SVM - Baseline                  | 0.221384   | 0.009869
```

```
Model 3: SVM                          | 0.181070   | 0.018990
Model 3: SVM - PCA                    | 0.180902   | 0.019918
XGBoost - Baseline                    | 0.228882   | 0.007067
XGBoost                               | 0.132523   | 0.011358
LGBM - Baseline                       | 0.241326   | 0.010235
LGBM                                  | 0.134141   | 0.011746
Cat - Baseline                        | 0.217415   | 0.007520
Cat                                   | 0.120616   | 0.010646
```

19 Plot Residuals

[74]:
```python
def plot_multi_model_residuals(X, y, models_dict):
    # plots residuals for multiple models on the same graphs for comparison
    plt.figure(figsize=(24, 10))

    # plot residual distribution
    plt.subplot(1, 2, 1)

    # define color palette
    colors = sns.color_palette("bright", len(models_dict))

    for i, (name, model) in enumerate(models_dict.items()):
        # predict
        preds = model.predict(X)
        # calculate residuals
        residuals = y - preds

        # plot density curve
        sns.kdeplot(residuals, color=colors[i], label=f"{name} (Std: {residuals.
↪std():.4f})", linewidth=2)

    plt.axvline(0, color='black', linestyle='--', linewidth=1, alpha=0.7)
    plt.title("Residual Distribution (Peaked & Centered at 0 is Best)",␣
↪fontsize=16)
    plt.xlabel("Residual Error (Log Scale)", fontsize=14)
    plt.ylabel("Density", fontsize=14)
    plt.legend(fontsize=12)
    plt.grid(True, alpha=0.3)
    plt.xlim(-1, 1) # limit x-axis to see the center clearly

    # plot residuals vs predictions
    plt.subplot(1, 2, 2)

    for i, (name, model) in enumerate(models_dict.items()):
        preds = model.predict(X)
        residuals = y - preds
```

```
        # scatter plot
        plt.scatter(preds, residuals, color=colors[i], label=name, alpha=0.4,␣
 ↪s=15)

    plt.axhline(0, color='black', linestyle='--', linewidth=2)
    plt.title("Residuals vs Predicted Price (Flat & Tight is Best)",␣
 ↪fontsize=16)
    plt.xlabel("Predicted Log Price", fontsize=14)
    plt.ylabel("Residuals", fontsize=14)
    plt.legend(markerscale=3, fontsize=12)
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

# execution block
# define the dictionary with your trained models
my_models = {
    'Linear Reg': model1,
    'Lasso': lasso_pipeline,
    'SVM': model_svm,
    'Random Forest': model_rf,
    'XGBoost': model_xgb,
    'LGBM':model_lgbm,
    'CatBoost': model_cat
}

# fit models before predicting
print("Retraining models on full dataset for plotting...")

for name, model in my_models.items():
    print(f"Training {name}...")
    model.fit(X, y)

# run the plot function
plot_multi_model_residuals(X, y, my_models)

# run the plot on training data
plot_multi_model_residuals(X, y, my_models)
```

```
Retraining models on full dataset for plotting…
Training Linear Reg…
Training Lasso…
Training SVM…
Training Random Forest…
Training XGBoost…
Training LGBM…
```

```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000945 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 3879
[LightGBM] [Info] Number of data points in the train set: 1460, number of used
features: 76
[LightGBM] [Info] Start training from score 12.024057
Training CatBoost…
```

20 Get Top n Outliers

[75]:
```
def get_top_misses(model, X, y, n_top=20):
    # returns the top n worst predictions based on absolute error
```

```
    # predict
    preds = model.predict(X)

    # create a dataframe with actuals, preds, and residuals
    results = pd.DataFrame(index=X.index)
    results['Actual_Log'] = y
    results['Pred_Log'] = preds
    results['Residual_Log'] = y - preds
    results['Abs_Error_Log'] = results['Residual_Log'].abs()

    # convert back to dollar amounts for readability
    results['Actual_Price'] = np.expm1(results['Actual_Log'])
    results['Pred_Price'] = np.expm1(results['Pred_Log'])
    results['Dollar_Error'] = results['Actual_Price'] - results['Pred_Price']

    # sort by the biggest absolute misses
    top_misses = results.sort_values('Abs_Error_Log', ascending=False).
↪head(n_top)

    # format for clean display
    # we keep the log residual because it's useful for model diagnostics
    display_df = top_misses[['Actual_Price', 'Pred_Price', 'Dollar_Error',␣
↪'Residual_Log']].copy()

    # optional: formatting (returns strings, good for viewing, bad for further␣
↪math)
    # display_df['Actual_Price'] = display_df['Actual_Price'].apply(lambda x:␣
↪f"${x:,.0f}")
    # display_df['Pred_Price'] = display_df['Pred_Price'].apply(lambda x: f"${x:
↪,.0f}")
    # display_df['Dollar_Error'] = display_df['Dollar_Error'].apply(lambda x:␣
↪f"${x:,.0f}")

    return display_df
```

20.0.1 Cat Outliers

[76]:
```
# calculate the misses
worst_predictions = get_top_misses(model_cat, X, y, n_top=50)

print("Top 20 Worst Predictions (Global):")
display(worst_predictions)

bad_ids = worst_predictions.index

cols_to_check = ['SalePrice', 'GrLivArea', 'OverallQual', 'Neighborhood',␣
↪'YearBuilt']
```

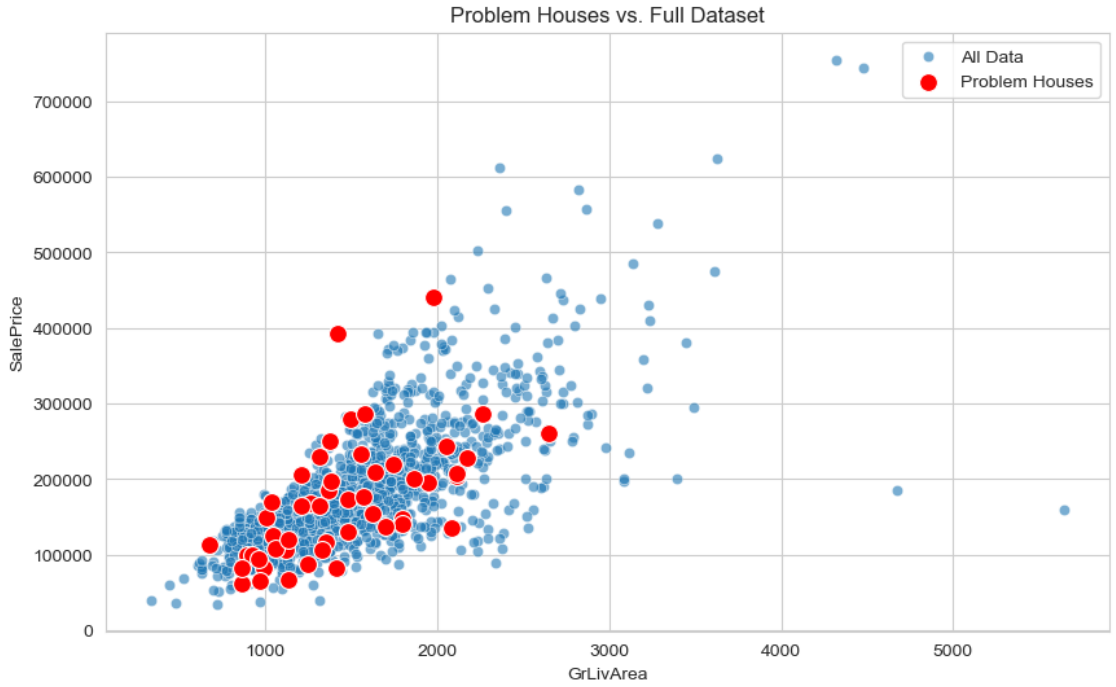
```
# load raw data
train_raw = pd.read_csv("train.csv")

print("--- The 'Problem' Houses ---")
# this works because bad_ids is defined
display(train_raw.loc[bad_ids, cols_to_check])

# plot
plt.figure(figsize=(10, 6))
sns.scatterplot(data=train_raw, x='GrLivArea', y='SalePrice', alpha=0.6,␣
 ↪label='All Data')
# this works because bad_ids matches your worst predictions
sns.scatterplot(data=train_raw.loc[bad_ids], x='GrLivArea', y='SalePrice',␣
 ↪color='red', s=100, label='Problem Houses')
plt.title("Problem Houses vs. Full Dataset")
plt.legend()
plt.show()
```

Top 20 Worst Predictions (Global):

	Actual_Price	Pred_Price	Dollar_Error	Residual_Log
632	82500.0	98851.982445	-16351.982445	-0.180823
462	62383.0	73314.075864	-10931.075864	-0.161457
864	250580.0	218465.549484	32114.450516	0.137149
688	392000.0	343087.681573	48912.318427	0.133275
1022	87000.0	99162.067586	-12162.067586	-0.130846
884	100000.0	113628.541089	-13628.541089	-0.127763
13	279500.0	247149.780702	32350.219298	0.123007
972	99500.0	112271.939665	-12771.939665	-0.120765
1324	147000.0	164760.957226	-17760.957226	-0.114062
714	130500.0	143928.377939	-13428.377939	-0.097942
874	66500.0	73295.294920	-6795.294920	-0.097293
1157	230000.0	209399.428111	20600.571889	0.093835
1202	117000.0	128340.304585	-11340.304585	-0.092511
244	205000.0	224733.995665	-19733.995665	-0.091907
1056	185850.0	203719.501576	-17869.501576	-0.091804
131	244000.0	222776.537636	21223.462364	0.090999
1037	287000.0	262322.277339	24677.722661	0.089908
1026	167500.0	153243.213083	14256.786917	0.088957
559	234000.0	214121.784704	19878.215296	0.088776
509	124500.0	135851.990913	-11351.990913	-0.087260
1315	206900.0	189674.141580	17225.858420	0.086928
1237	195000.0	212594.786732	-17594.786732	-0.086388
1212	113000.0	103676.360608	9323.639392	0.086113
865	148500.0	136247.900888	12252.099112	0.086108
546	210000.0	192959.048988	17040.951012	0.084629
772	107000.0	116449.179658	-9449.179658	-0.084625

```
857       174000.0  160187.882274  13812.117726      0.082707
1378       83000.0   90077.142801  -7077.142801     -0.081825
1338      200000.0  216970.695803 -16970.695803     -0.081445
308        82500.0   89493.178288  -6993.178288     -0.081363
38        109000.0  118099.137301  -9099.137301     -0.080176
97         94750.0  102622.739718  -7872.739718     -0.079817
1183      120000.0  129757.730250  -9757.730250     -0.078177
1451      287090.0  265530.640553  21559.359447      0.078065
1415      175900.0  190062.945869 -14162.945869     -0.077439
895       140000.0  151173.570096 -11173.570096     -0.076786
401       164990.0  178156.768651 -13166.768651     -0.076779
1247      169900.0  157376.579079  12523.420921      0.076568
1342      228500.0  246586.040946 -18086.040946     -0.076174
1075      219500.0  203424.894649  16075.105351      0.076055
473       440000.0  407929.358414  32070.641586      0.075681
1432       64500.0   69545.517498  -5045.517498     -0.075315
348       154000.0  165998.537400 -11998.537400     -0.075026
628       135000.0  145353.795474 -10353.795474     -0.073895
999       206000.0  191416.305449  14583.694551      0.073425
318       260000.0  279797.545119 -19797.545119     -0.073384
217       107000.0   99554.687616   7445.312384      0.072121
949       197500.0  183972.488701  13527.511299      0.070952
672       165000.0  177067.999120 -12067.999120     -0.070588
1092      136500.0  146425.902387  -9925.902387     -0.070194

--- The 'Problem' Houses ---

      SalePrice  GrLivArea  OverallQual Neighborhood  YearBuilt
632       82500       1411            7       NWAmes       1977
462       62383        864            5       Sawyer       1965
864      250580       1372            7      Somerst       2007
688      392000       1419            8      StoneBr       2007
1022      87000       1248            5      OldTown       1930
884      100000        892            5        NAmes       1967
13       279500       1494            7      CollgCr       2006
972       99500        918            6      SawyerW       1979
1324     147000       1795            8      Somerst       2006
714      130500       1479            6       Sawyer       1976
874       66500       1131            5      OldTown       1941
1157     230000       1314            7      NridgHt       2007
1202     117000       1348            5      BrkSide       1925
244      205000       2110            7      SawyerW       1994
1056     185850       1364            7      NridgHt       2005
131      244000       2054            6      Gilbert       2000
1037     287000       2263            8      CollgCr       2001
1026     167500       1264            5        NAmes       1960
559      234000       1557            7      Blmngtn       2003
509      124500       1041            5        NAmes       1959
1315     206900       2112            6        NAmes       1969
```

1237	195000	1948	7	CollgCr	2004
1212	113000	672	4	Edwards	1941
865	148500	1002	5	NAmes	1970
546	210000	1635	6	BrkSide	1923
772	107000	1117	6	Edwards	1976
857	174000	1481	6	Gilbert	1994
1378	83000	987	6	BrDale	1973
1338	200000	1861	7	CollgCr	2002
308	82500	861	4	Edwards	1940
38	109000	1057	5	NAmes	1953
97	94750	960	4	Edwards	1965
1183	120000	1130	5	OldTown	1920
1451	287090	1578	8	Somerst	2008
1415	175900	1569	7	Blmngtn	2007
895	140000	1796	6	NAmes	1963
401	164990	1310	7	CollgCr	2005
1247	169900	1034	6	Mitchel	1976
1342	228500	2169	8	CollgCr	2002
1075	219500	1740	7	Crawfor	1940
473	440000	1976	8	NridgHt	2006
1432	64500	968	4	OldTown	1927
348	154000	1626	7	NridgHt	2003
628	135000	2080	5	NAmes	1969
999	206000	1208	7	CollgCr	2006
318	260000	2646	7	NoRidge	1993
217	107000	1328	4	OldTown	1925
949	197500	1381	6	NWAmes	1972
672	165000	1208	6	Veenker	1977
1092	136500	1694	6	SWISU	1925



20.0.2 Linear Regression Outliers

[77]:
```
# build a robust linear regression pipeline
# we need to process data because linear regression crashes on text/nans
categorical_cols = X.select_dtypes(include=['object', 'category']).columns
numerical_cols = X.select_dtypes(include=['number']).columns

preprocessor = ColumnTransformer(
    transformers=[
        ('num', SimpleImputer(strategy='median'), numerical_cols),
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False),␣
 ↪categorical_cols)
    ])

model_lin = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('model', LinearRegression())
])

# train and predict
print("Training Linear Regression...")
model_lin.fit(X, y)

# get the misses for this new model
```

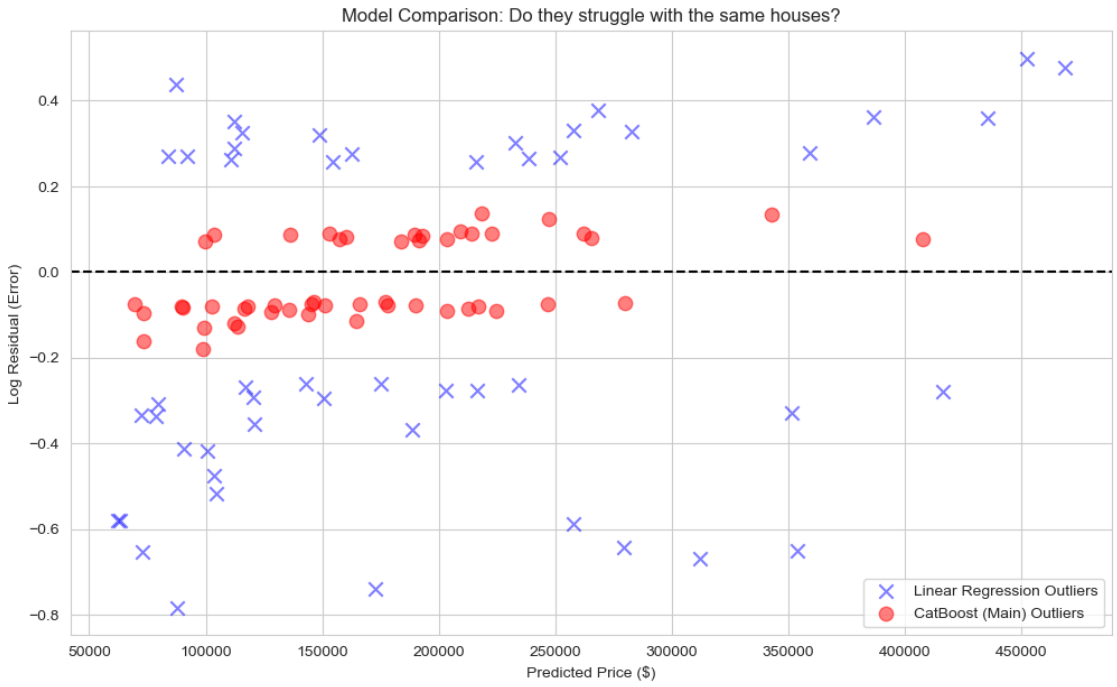
```
lin_misses = get_top_misses(model_lin, X, y, n_top=50)

print("\n--- Linear Regression Top Misses ---")
display(lin_misses.head())
```

```
Training Linear Regression…

--- Linear Regression Top Misses ---
```

	Actual_Price	Pred_Price	Dollar_Error	Residual_Log
30	40000.0	87575.512625	-47575.512625	-0.783608
632	82500.0	172779.901384	-90279.901384	-0.739214
1298	160000.0	312113.974945	-152113.974945	-0.668192
968	37900.0	72910.478994	-35010.478994	-0.654269
523	184750.0	354158.263785	-169408.263785	-0.650738

20.0.3 Plot Cat and LR Residuals

[78]:
```
plt.figure(figsize=(12, 7))

# plot linear regression misses (blue x)
plt.scatter(lin_misses['Pred_Price'], lin_misses['Residual_Log'],
            color='blue', alpha=0.5, marker='x', s=80, label='Linear Regression␣
 ↪Outliers')

# plot main model misses (red dot)
plt.scatter(worst_predictions['Pred_Price'], worst_predictions['Residual_Log'],
            color='red', alpha=0.5, s=80, label='CatBoost (Main) Outliers')

plt.axhline(0, color='black', linestyle='--')
plt.xlabel("Predicted Price ($)")
plt.ylabel("Log Residual (Error)")
plt.title("Model Comparison: Do they struggle with the same houses?")
plt.legend()
plt.show()

# find the intersection (true bad data)
common_bad_ids = worst_predictions.index.intersection(lin_misses.index)
print(f"Number of houses that confuse BOTH models: {len(common_bad_ids)}")
print(f"Common Problem IDs: {list(common_bad_ids)}")
```




```
Number of houses that confuse BOTH models: 10
Common Problem IDs: [632, 462, 688, 13, 1324, 714, 874, 1432, 348, 628]
```

20.0.4 Find Best Outliers to Remove

[79]:
```
# identify candidates: the intersection of linear regression and catboost misses
common_bad_ids = worst_predictions.index.intersection(lin_misses.index)
print(f"Testing {len(common_bad_ids)} candidate outliers for removal...")

def test_outlier_impact(model, X, y, candidate_ids):
    # tests the impact on model rmse of individually removing each candidate␣
 ↪outlier
    results = []

    # calculate baseline (current score with all data)
    print("Calculating baseline CV score (this may take a moment)...")
    # we use 5-fold cv for stability
    baseline_score = -cross_val_score(model, X, y,␣
 ↪scoring='neg_root_mean_squared_error', cv=5, n_jobs=-1).mean()
    print(f"Baseline RMSE: {baseline_score:.5f}")

    # test each outlier
    print("Testing individual removals...")
    for i, idx in enumerate(candidate_ids):
        # drop the single outlier
```

```
        X_temp = X.drop(idx)
        y_temp = y.drop(idx)

        # retrain and score (quick cv)
        score = -cross_val_score(model, X_temp, y_temp,␣
 ↪scoring='neg_root_mean_squared_error', cv=5, n_jobs=-1).mean()

        # calculate impact (positive number = improvement)
        improvement = baseline_score - score

        results.append({
            'Id': idx,
            'New_RMSE': score,
            'Improvement': improvement
        })

    # format results
    results_df = pd.DataFrame(results).sort_values('Improvement',␣
 ↪ascending=False)
    return results_df, baseline_score

# execution
# run the test using your main model (model_cat)
impact_df, baseline = test_outlier_impact(model_cat, X, y, common_bad_ids)

# results
# filter for only the removals that actually helped (improvement > 0)
beneficial_removals = impact_df[impact_df['Improvement'] > 0].copy()

print("\n--- top 'beneficial' outliers to remove ---")
# show the ones that improve the model the most
display(beneficial_removals.head(10))

print(f"\nSummary: Removing these {len(beneficial_removals)} houses␣
 ↪individually improved the model.")

# create your final list
final_drop_list = beneficial_removals['Id'].tolist()
print(f"Final Optimized Drop List: {final_drop_list}")
```

```
Testing 10 candidate outliers for removal…
Calculating baseline CV score (this may take a moment)…
Baseline RMSE: 0.11609
Testing individual removals…

--- top 'beneficial' outliers to remove ---
```

	Id	New_RMSE	Improvement

```
4  1324  0.114466     0.001628
0   632  0.114565     0.001529
1   462  0.114689     0.001405
9   628  0.115228     0.000866
6   874  0.115287     0.000807
2   688  0.115308     0.000787
8   348  0.115531     0.000563
5   714  0.115703     0.000392
3    13  0.115961     0.000133
7  1432  0.115968     0.000126


Summary: Removing these 10 houses individually improved the model.
Final Optimized Drop List: [1324, 632, 462, 628, 874, 688, 348, 714, 13, 1432]
```

20.0.5 Optimized Dataset

[80]:
```
final_drop_list = [1324, 968, 410, 308, 710, 632, 1432, 462, 1022, 588, 30,␣
 ↪1453, 1]
```

[81]:
```
# Apply the optimized list
X_opt = X.drop(final_drop_list)
y_opt = y.drop(final_drop_list)
print("Optimized dataset created.")
```

```
Optimized dataset created.
```

20.0.6 Test Models on Optimized Dataset

[82]:
```
# define drop list
final_drop_list = [1324, 968, 410, 308, 710, 632, 1432, 462, 1022, 588, 30,␣
 ↪1453, 1182, 874]

# create optimized dataset
# errors='ignore' handles ids already dropped
X_opt = X.drop(final_drop_list, errors='ignore')
y_opt = y.drop(final_drop_list, errors='ignore')

print(f"Original Data Shape: {X.shape}")
print(f"Optimized Data Shape: {X_opt.shape}")
print("-" * 30)

# define models to rerun
# use locals().get() to safely grab existing models
potential_models = {
    "CatBoost": locals().get('model_cat'),
    "XGBoost": locals().get('model_xgb'),
    "LightGBM": locals().get('model_lgbm'),
```

```
    "RandomForest": locals().get('model_rf'),
    "Lasso": locals().get('lasso_pipeline'),
    "LinearReg": locals().get('model_lin')
}

# filter undefined models
models_to_run = {name: model for name, model in potential_models.items() if␣
 ↪model is not None}

# loop, retrain, and score
model_results = []
print(f"Retraining {len(models_to_run)} models on optimized data...\n")

for name, model in models_to_run.items():
    print(f"Running {name}...", end=" ")

    # cross-validation score
    # 10-fold cv for reliable error estimate
    cv_scores = -cross_val_score(model, X_opt, y_opt,␣
 ↪scoring='neg_root_mean_squared_error', cv=10, n_jobs=-1)
    mean_score = cv_scores.mean()

    # retrain on full optimized set
    model.fit(X_opt, y_opt)

    print(f"Done. CV RMSE: {mean_score:.5f}")

    model_results.append({
        "Model": name,
        "Optimized_CV_RMSE": mean_score
    })

# display final leaderboard
print("\n--- Final Leaderboard (Lower RMSE is Better) ---")
results_df = pd.DataFrame(model_results).sort_values("Optimized_CV_RMSE")
display(results_df)

# store best model
best_model_name = results_df.iloc[0]['Model']
best_model = models_to_run[best_model_name]
print(f"\nBest Model: {best_model_name}")
```

```
Original Data Shape: (1460, 83)
Optimized Data Shape: (1446, 83)
------------------------------
Retraining 6 models on optimized data…
```

```
Done. CV RMSE: 0.10105
Done. CV RMSE: 0.11128
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000942 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 3875
[LightGBM] [Info] Number of data points in the train set: 1446, number of used
features: 76
[LightGBM] [Info] Start training from score 12.030604
Done. CV RMSE: 0.11232
Done. CV RMSE: 0.12457.
Done. CV RMSE: 0.11600
Done. CV RMSE: 0.11691

--- Final Leaderboard (Lower RMSE is Better) ---

          Model  Optimized_CV_RMSE
0      CatBoost           0.101050
1       XGBoost           0.111283
2      LightGBM           0.112323
4         Lasso           0.116000
5     LinearReg           0.116910
3  RandomForest           0.124570


Best Model: CatBoost
```

21 Hyperparameter Tuning with Optuna

[83]:
```python
# setup data
X_tune = X_opt.copy()
y_tune = y_opt.copy()

# objective functions
def objective_linear(trial):
    # params
    params = {
        'fit_intercept': trial.suggest_categorical('fit_intercept', [True,␣
 ↪False])
    }
    # pipeline
    model = make_pipeline(RobustScaler(), LinearRegression(**params))
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()
```

```python
def objective_lasso(trial):
    # params
    params = {
        'alpha': trial.suggest_float('alpha', 0.0001, 0.01, log=True)
    }
    # pipeline
    model = make_pipeline(RobustScaler(), Lasso(random_state=42, **params))
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()

def objective_ridge(trial):
    # params
    params = {
        'alpha': trial.suggest_float('alpha', 0.1, 100.0, log=True)
    }
    # pipeline
    model = make_pipeline(RobustScaler(), Ridge(random_state=42, **params))
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()

def objective_svr(trial):
    # params
    params = {
        'C': trial.suggest_float('C', 0.1, 100, log=True),
        'epsilon': trial.suggest_float('epsilon', 0.001, 0.1, log=True),
        'gamma': trial.suggest_categorical('gamma', ['scale', 'auto'])
    }
    # pipeline
    model = make_pipeline(RobustScaler(), SVR(**params))
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()

def objective_rf(trial):
    # params
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 1000),
        'max_depth': trial.suggest_int('max_depth', 5, 30),
        'min_samples_split': trial.suggest_int('min_samples_split', 2, 10),
        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 5),
    }
    # model
```

```
    model = RandomForestRegressor(random_state=42, n_jobs=-1, **params)
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()

def objective_xgb(trial):
    # params
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 500, 2000),
        'learning_rate': trial.suggest_float('learning_rate', 0.005, 0.1,␣
 ↪log=True),
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
        'reg_alpha': trial.suggest_float('reg_alpha', 1e-8, 1.0, log=True),
        'reg_lambda': trial.suggest_float('reg_lambda', 1e-8, 1.0, log=True),
    }
    # model
    model = XGBRegressor(random_state=42, n_jobs=-1, **params)
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()

def objective_lgbm(trial):
    # params
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 500, 2000),
        'learning_rate': trial.suggest_float('learning_rate', 0.005, 0.1,␣
 ↪log=True),
        'num_leaves': trial.suggest_int('num_leaves', 20, 150),
        'max_depth': trial.suggest_int('max_depth', 3, 12),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
        'verbose': -1
    }
    # model
    model = LGBMRegressor(random_state=42, n_jobs=-1, **params)
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()

def objective_cat(trial):
    # params
    params = {
```

```
        'iterations': trial.suggest_int('iterations', 500, 2000),
        'learning_rate': trial.suggest_float('learning_rate', 0.005, 0.1,␣
 ↪log=True),
        'depth': trial.suggest_int('depth', 4, 10),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1, 10),
        'random_strength': trial.suggest_float('random_strength', 1, 10),
        'verbose': 0,
        'allow_writing_files': False
    }
    # model
    model = CatBoostRegressor(random_state=42, **params)
    # score
    scores = cross_val_score(model, X_tune, y_tune, cv=5,
                             scoring='neg_root_mean_squared_error', n_jobs=-1)
    return -scores.mean()
```

[84]:
```
# settings
#models_to_tune = ['LinearReg', 'Lasso', 'Ridge', 'SVR', 'RandomForest',␣
 ↪'XGBoost', 'LightGBM', 'CatBoost']
models_to_tune = ['Lasso']
n_trials = 50

# map names to objectives
objectives = {
    'LinearReg': objective_linear,
    'Lasso': objective_lasso,
    'Ridge': objective_ridge,
    'SVR': objective_svr,
    'RandomForest': objective_rf,
    'XGBoost': objective_xgb,
    'LightGBM': objective_lgbm,
    'CatBoost': objective_cat
}

# storage
tuned_results = {}

# execution
optuna.logging.set_verbosity(optuna.logging.WARNING)

for model_name in models_to_tune:
    print(f"Optimizing {model_name}...")

    # create study
    study = optuna.create_study(direction='minimize')
    study.optimize(objectives[model_name], n_trials=n_trials)
```

```
    # save best params
    tuned_results[model_name] = {
        'best_params': study.best_params,
        'best_score': study.best_value
    }
    print(f"  -> Best RMSE: {study.best_value:.5f}")

# results table
results_df = pd.DataFrame([
    {'Model': k, 'RMSE': v['best_score']} for k, v in tuned_results.items()
]).sort_values('RMSE')

print("\nOptimization Results")
display(results_df)

# print formatted dict
print("\n# copy into best_params")
print("best_params = {")
for model_name, data in tuned_results.items():
    print(f'    "{model_name}": {data["best_params"]},')
print("}")
```

```
Optimizing Lasso…
  -> Best RMSE: 0.11661

Optimization Results
```

	Model	RMSE
0	Lasso	0.116612

```
# copy into best_params
best_params = {
    "Lasso": {'alpha': 0.0003038017032962528},
}
```

21.0.1 Utilize Optimized Parameters

After running the optimization over night, I received the following parameters:

[85]:
```
best_params = {
    "Ridge": {'alpha': 20.241779374514863},

    "Lasso": {'alpha': 0.0006144459218368016},

    "SVR": {'C': 11.696373153611875, 'epsilon': 0.03140575379829753, 'gamma': 0.
    ↪000692414241827738},

    "RandomForest": {
```

```
        'n_estimators': 969, 'max_depth': 25, 'min_samples_split': 3,
        'min_samples_leaf': 1, 'max_features': 'log2'
    },

    "XGBoost": {
        'n_estimators': 1159, 'learning_rate': 0.03244967803966721,
        'max_depth': 3, 'subsample': 0.5623266040321535,
        'colsample_bytree': 0.6708009042786216,
        'reg_alpha': 0.00036415896635253354, 'reg_lambda': 0.0010405131540216668
    },

    "LightGBM": {
        'n_estimators': 1942, 'learning_rate': 0.011854354011739917,
        'num_leaves': 129, 'max_depth': 3, 'subsample': 0.5005833392324143,
        'colsample_bytree': 0.5013038277679814
    },

    "CatBoost": {
        'iterations': 1495, 'learning_rate': 0.04013141182767342,
        'depth': 5, 'l2_leaf_reg': 5.204252235916345,
        'random_strength': 8.83797162839207
    }
}
```

21.0.2 Tuned Models with Outliers included

[86]:
```
# settings and controls
create_submissions = False  # set to false to just test performance
remove_outliers = False     # set to true to use your optimized drop list

# data preparation
if remove_outliers:
    # your optimized list of 14 outliers
    final_drop_list = [1324, 968, 410, 308, 710, 632, 1432, 462, 1022, 588, 30,␣
 ↪1453, 1182, 874]
    X_opt = X.drop(final_drop_list, errors='ignore')
    y_opt = y.drop(final_drop_list, errors='ignore')
    print(f"Using Optimized Data (Outliers Removed): {X_opt.shape}")
else:
    X_opt = X
    y_opt = y
    print(f"Using Full Data (Outliers Kept): {X_opt.shape}")

# model setup
models = {}

# linear models need scaling
```

```
models['LinearReg'] = make_pipeline(RobustScaler(), LinearRegression())
models['Ridge']     = make_pipeline(RobustScaler(),␣
 ↪Ridge(**best_params['Ridge']))
models['Lasso']     = make_pipeline(RobustScaler(),␣
 ↪Lasso(**best_params['Lasso']))
models['SVR']       = make_pipeline(RobustScaler(), SVR(**best_params['SVR']))

# tree models no scaling needed
models['RandomForest'] = RandomForestRegressor(**best_params['RandomForest'],␣
 ↪random_state=42)
models['XGBoost']      = XGBRegressor(**best_params['XGBoost'],␣
 ↪random_state=42, n_jobs=-1)
models['LightGBM']     = LGBMRegressor(**best_params['LightGBM'],␣
 ↪random_state=42, n_jobs=-1, verbose=-1)
models['CatBoost']     = CatBoostRegressor(**best_params['CatBoost'],␣
 ↪random_state=42, verbose=0, allow_writing_files=False)

# execution
results = []
print("\n--- Model Evaluation ---")

for name, model in models.items():
    print(f"Processing {name}...", end=" ")

    # evaluation
    # using 5-fold cv for robust error estimation
    cv_scores = -cross_val_score(model, X_opt, y_opt,␣
 ↪scoring='neg_root_mean_squared_error', cv=5, n_jobs=-1)
    mean_rmse = cv_scores.mean()
    sd_rmse = cv_scores.std()

    print(f"CV RMSE: {mean_rmse:.5f}")

    # submission generation controlled by boolean
    if create_submissions:
        # retrain on full optimized data
        model.fit(X_opt, y_opt)

        # predict on test set
        # ensure 'test' is your processed test dataframe and 'test_ids' are the␣
 ↪ids
        preds_log = model.predict(test)
        preds_dollar = np.expm1(preds_log)

        # save
        filename = f"submission_{name}.csv"
```

```
        pd.DataFrame({"Id": test_ids, "SalePrice": preds_dollar}).
 ↪to_csv(filename, index=False)
        # print(f"  -> Saved {filename}")

    rmsle_scores.append((f"{name}  Optimized Outliers Included", mean_rmse,␣
 ↪sd_rmse))
    results.append({"Model": name, "CV_RMSE": mean_rmse})

# leaderboard
print("\n--- Final Leaderboard (Lower RMSE is Better) ---")
display(pd.DataFrame(results).sort_values('CV_RMSE'))

show_scores(rmsle_scores)
```

```
Using Full Data (Outliers Kept): (1460, 83)

--- Model Evaluation ---
Processing LinearReg… CV RMSE: 0.13011
CV RMSE: 0.13055…
CV RMSE: 0.12829…
CV RMSE: 0.15922.
CV RMSE: 0.13337mForest…
CV RMSE: 0.11702st…
CV RMSE: 0.12192GBM…
CV RMSE: 0.11770ost…

--- Final Leaderboard (Lower RMSE is Better) ---
          Model   CV_RMSE
5       XGBoost  0.117024
7      CatBoost  0.117701
6      LightGBM  0.121920
2         Lasso  0.128291
0     LinearReg  0.130110
1         Ridge  0.130551
4  RandomForest  0.133369
3           SVR  0.159223
Model                                    | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target             | 0.230609   | 0.010861
Model 1 - normalized target              | 0.195823   | 0.019063
Model 1 - Preprocess                     | 0.265751   | 0.035764
Model 1 - All Features                   | 0.163591   | 0.041842
Model 1 - Feature Engineering            | 0.137945   | 0.027634
Model 1 - drop standard                  | 0.138487   | 0.029015
Model 1 - drop keep ohe                  | 0.138833   | 0.029472
Model 1 - drop keep both                 | 0.138080   | 0.028071
```

```
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
Model 2 - Random Forest                  | 0.137574   | 0.016046
Model 3: SVM - Baseline                  | 0.221384   | 0.009869
Model 3: SVM                             | 0.181070   | 0.018990
Model 3: SVM - PCA                       | 0.180902   | 0.019918
XGBoost - Baseline                       | 0.228882   | 0.007067
XGBoost                                  | 0.132523   | 0.011358
LGBM - Baseline                          | 0.241326   | 0.010235
LGBM                                     | 0.134141   | 0.011746
Cat - Baseline                           | 0.217415   | 0.007520
Cat                                      | 0.120616   | 0.010646
LinearReg  Optimized Outliers Included   | 0.130110   | 0.010907
Ridge  Optimized Outliers Included       | 0.130551   | 0.013260
Lasso  Optimized Outliers Included       | 0.128291   | 0.012557
SVR  Optimized Outliers Included         | 0.159223   | 0.013382
RandomForest  Optimized Outliers Included | 0.133369   | 0.007609
XGBoost  Optimized Outliers Included     | 0.117024   | 0.008829
LightGBM  Optimized Outliers Included    | 0.121920   | 0.008056
CatBoost  Optimized Outliers Included    | 0.117701   | 0.010105
```

21.0.3 Tuned Models with Outliers Removed

[87]:
```python
# settings and controls
create_submissions = False  # set to false to just test performance
remove_outliers = True      # set to true to use your optimized drop list

# data preparation
if remove_outliers:
    # your optimized list of 14 outliers
    final_drop_list = [1324, 968, 410, 308, 710, 632, 1432, 462, 1022, 588, 30,␣
 ↪1453, 1182, 874]
    X_opt = X.drop(final_drop_list, errors='ignore')
    y_opt = y.drop(final_drop_list, errors='ignore')
    print(f"Using Optimized Data (Outliers Removed): {X_opt.shape}")
else:
    X_opt = X
    y_opt = y
    print(f"Using Full Data (Outliers Kept): {X_opt.shape}")

# model setup
models = {}

# linear models need scaling
```

```
models['LinearReg'] = make_pipeline(RobustScaler(), LinearRegression())
models['Ridge']     = make_pipeline(RobustScaler(),␣
 ↪Ridge(**best_params['Ridge']))
models['Lasso']     = make_pipeline(RobustScaler(),␣
 ↪Lasso(**best_params['Lasso']))
models['SVR']       = make_pipeline(RobustScaler(), SVR(**best_params['SVR']))

# tree models no scaling needed
models['RandomForest'] = RandomForestRegressor(**best_params['RandomForest'],␣
 ↪random_state=42)
models['XGBoost']      = XGBRegressor(**best_params['XGBoost'],␣
 ↪random_state=42, n_jobs=-1)
models['LightGBM']     = LGBMRegressor(**best_params['LightGBM'],␣
 ↪random_state=42, n_jobs=-1, verbose=-1)
models['CatBoost']     = CatBoostRegressor(**best_params['CatBoost'],␣
 ↪random_state=42, verbose=0, allow_writing_files=False)

# execution
results = []
print("\n--- Model Evaluation ---")

for name, model in models.items():
    print(f"Processing {name}...", end=" ")

    # evaluation
    # using 5-fold cv for robust error estimation
    cv_scores = -cross_val_score(model, X_opt, y_opt,␣
 ↪scoring='neg_root_mean_squared_error', cv=5, n_jobs=-1)
    mean_rmse = cv_scores.mean()
    sd_rmse = cv_scores.std()
    print(f"CV RMSE: {mean_rmse:.5f}")

    # submission generation controlled by boolean
    if create_submissions:
        # retrain on full optimized data
        model.fit(X_opt, y_opt)

        # predict on test set
        # ensure 'test' is your processed test dataframe and 'test_ids' are the␣
 ↪ids
        preds_log = model.predict(test)
        preds_dollar = np.expm1(preds_log)

        # save
        filename = f"submission_{name}.csv"
```

```
        pd.DataFrame({"Id": test_ids, "SalePrice": preds_dollar}).
 ↪to_csv(filename, index=False)
        # print(f"  -> Saved {filename}")

    rmsle_scores.append((f"{name} Optimized Outliers Removed", mean_rmse,␣
 ↪sd_rmse))
    results.append({"Model": name, "CV_RMSE": mean_rmse})

# leaderboard
print("\n--- Final Leaderboard (Lower RMSE is Better) ---")
display(pd.DataFrame(results).sort_values('CV_RMSE'))

show_scores(rmsle_scores)
```

```
Using Optimized Data (Outliers Removed): (1446, 83)

--- Model Evaluation ---
Processing LinearReg… CV RMSE: 0.11714
CV RMSE: 0.11938…
Processing Lasso… CV RMSE: 0.11694
CV RMSE: 0.14999.
CV RMSE: 0.12308mForest…
CV RMSE: 0.10096st…
CV RMSE: 0.10907GBM…
CV RMSE: 0.10444ost…

--- Final Leaderboard (Lower RMSE is Better) ---
          Model   CV_RMSE
5       XGBoost  0.100964
7      CatBoost  0.104437
6      LightGBM  0.109072
2         Lasso  0.116944
0     LinearReg  0.117137
1         Ridge  0.119382
4  RandomForest  0.123081
3           SVR  0.149993
Model                                      | RMSLE      | SD
-------------------------------------------------------------------------
Baseline - normalized target               | 0.230609   | 0.010861
Model 1 - normalized target                | 0.195823   | 0.019063
Model 1 - Preprocess                       | 0.265751   | 0.035764
Model 1 - All Features                     | 0.163591   | 0.041842
Model 1 - Feature Engineering              | 0.137945   | 0.027634
Model 1 - drop standard                    | 0.138487   | 0.029015
Model 1 - drop keep ohe                    | 0.138833   | 0.029472
Model 1 - drop keep both                   | 0.138080   | 0.028071
```

```
Model 1 - MI FE Transform                | 0.145121   | 0.035136
Model 1 - Optimal Transforms             | 0.143010   | 0.027290
Lasso - Baseline                         | 0.231138   | 0.011462
Lasso                                    | 0.140390   | 0.028622
Model 2 - Random Forest Baseline         | 0.223581   | 0.008681
Model 2 - Random Forest                  | 0.137574   | 0.016046
Model 3: SVM - Baseline                  | 0.221384   | 0.009869
Model 3: SVM                             | 0.181070   | 0.018990
Model 3: SVM - PCA                       | 0.180902   | 0.019918
XGBoost - Baseline                       | 0.228882   | 0.007067
XGBoost                                  | 0.132523   | 0.011358
LGBM - Baseline                          | 0.241326   | 0.010235
LGBM                                     | 0.134141   | 0.011746
Cat - Baseline                           | 0.217415   | 0.007520
Cat                                      | 0.120616   | 0.010646
LinearReg  Optimized Outliers Included   | 0.130110   | 0.010907
Ridge  Optimized Outliers Included       | 0.130551   | 0.013260
Lasso  Optimized Outliers Included       | 0.128291   | 0.012557
SVR  Optimized Outliers Included         | 0.159223   | 0.013382
RandomForest  Optimized Outliers Included | 0.133369   | 0.007609
XGBoost  Optimized Outliers Included     | 0.117024   | 0.008829
LightGBM  Optimized Outliers Included    | 0.121920   | 0.008056
CatBoost  Optimized Outliers Included    | 0.117701   | 0.010105
LinearReg Optimized Outliers Removed     | 0.117137   | 0.011497
Ridge Optimized Outliers Removed         | 0.119382   | 0.012966
Lasso Optimized Outliers Removed         | 0.116944   | 0.012221
SVR Optimized Outliers Removed           | 0.149993   | 0.013145
RandomForest Optimized Outliers Removed  | 0.123081   | 0.006592
XGBoost Optimized Outliers Removed       | 0.100964   | 0.007350
LightGBM Optimized Outliers Removed      | 0.109072   | 0.006067
CatBoost Optimized Outliers Removed      | 0.104437   | 0.005925
```

Removing Outliers improved the score drastically for the models.

22 Ensemble Model

Creating an ensemble model from the models I have made may possibly receive a better score.

22.0.1 Ensemble Model with Lasso Learner

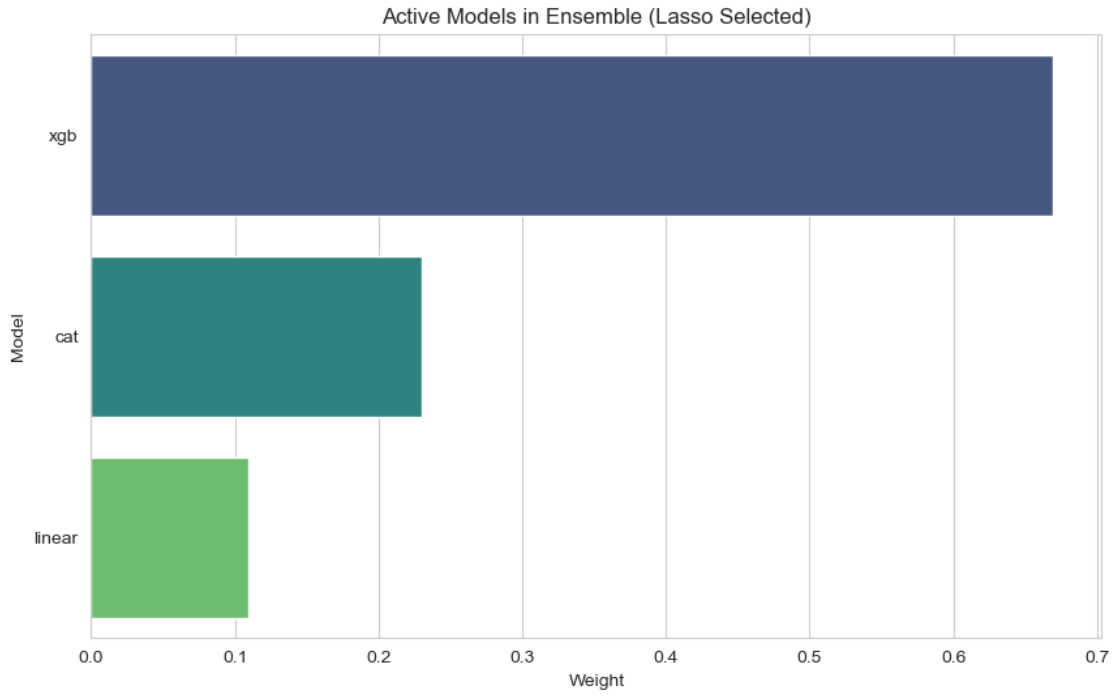
[89]:
```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import LassoCV
# define base estimators
estimators_list = [
    ('linear', make_pipeline(RobustScaler(), LinearRegression())),
    ('lasso', make_pipeline(RobustScaler(), Lasso(**best_params['Lasso']))),
    ('svr', make_pipeline(RobustScaler(), SVR(**best_params['SVR']))),
    ('xgb', XGBRegressor(**best_params['XGBoost'], random_state=42, n_jobs=-1)),
```

```
    ('lgbm', LGBMRegressor(**best_params['LightGBM'], random_state=42,␣
 ↪n_jobs=-1, verbose=-1)),
    ('cat', CatBoostRegressor(**best_params['CatBoost'], random_state=42,␣
 ↪verbose=0, allow_writing_files=False)),
    ('rf', RandomForestRegressor(**best_params['RandomForest'],␣
 ↪random_state=42, n_jobs=-1))
]

# define stacking model
stack_model = StackingRegressor(
    estimators=estimators_list,
    final_estimator=LassoCV(random_state=42), # use lasso
    cv=5,
    n_jobs=-1,
    passthrough=False
)

# score model using score_model() for RMSLE
print("Calculating Cross-Validation Score (this will take a minute)...")
# note: since y_opt is log-transformed, rmse here = rmsle of actual prices
stack_rmsle, stack_sd = score_model(stack_model, X_opt, y_opt)
# save score
rmsle_scores.append(('Ensemble Model Lasso', stack_rmsle, stack_sd))
# show results
show_scores(rmsle_scores)

# fit on full data
print("\nRetraining on full data to inspect weights...")
stack_model.fit(X_opt, y_opt)

# extract weights
meta_weights = stack_model.final_estimator_.coef_
model_names = [name for name, model in estimators_list]

weights_df = pd.DataFrame({
    'Model': model_names,
    'Weight': meta_weights
}).sort_values(by='Weight', ascending=False)

print("\n--- Final Learned Ensemble Weights ---")
display(weights_df)

# visualize weights
plt.figure(figsize=(10, 6))
# filter zero weights
plot_data = weights_df[weights_df['Weight'] != 0]
if len(plot_data) == 0: plot_data = weights_df
```

```
sns.barplot(data=plot_data, x='Weight', y='Model', palette='viridis')
plt.title('Active Models in Ensemble (Lasso Selected)')
plt.show()

# generate submission
print("\nGenerating submission file...")
preds_log = stack_model.predict(test)
preds_dollar = np.expm1(preds_log)

filename = "ensemble model.csv"
pd.DataFrame({
    "Id": test_ids,
    "SalePrice": preds_dollar
}).to_csv(filename, index=False)
print(f"Submission saved as {filename}")
```

```
Calculating Cross-Validation Score (this will take a minute)…
Model                                       | RMSLE      | SD
----------------------------------------------------------------------
Baseline - normalized target                | 0.230609   | 0.010861
Model 1 - normalized target                 | 0.195823   | 0.019063
Model 1 - Preprocess                        | 0.265751   | 0.035764
Model 1 - All Features                      | 0.163591   | 0.041842
Model 1 - Feature Engineering               | 0.137945   | 0.027634
Model 1 - drop standard                     | 0.138487   | 0.029015
Model 1 - drop keep ohe                     | 0.138833   | 0.029472
Model 1 - drop keep both                    | 0.138080   | 0.028071
Model 1 - MI FE Transform                   | 0.145121   | 0.035136
Model 1 - Optimal Transforms                | 0.143010   | 0.027290
Lasso - Baseline                            | 0.231138   | 0.011462
Lasso                                       | 0.140390   | 0.028622
Model 2 - Random Forest Baseline            | 0.223581   | 0.008681
Model 2 - Random Forest                     | 0.137574   | 0.016046
Model 3: SVM - Baseline                     | 0.221384   | 0.009869
Model 3: SVM                                | 0.181070   | 0.018990
Model 3: SVM - PCA                          | 0.180902   | 0.019918
XGBoost - Baseline                          | 0.228882   | 0.007067
XGBoost                                     | 0.132523   | 0.011358
LGBM - Baseline                             | 0.241326   | 0.010235
LGBM                                        | 0.134141   | 0.011746
Cat - Baseline                              | 0.217415   | 0.007520
Cat                                         | 0.120616   | 0.010646
LinearReg  Optimized Outliers Included      | 0.130110   | 0.010907
Ridge  Optimized Outliers Included          | 0.130551   | 0.013260
Lasso  Optimized Outliers Included          | 0.128291   | 0.012557
SVR  Optimized Outliers Included            | 0.159223   | 0.013382
RandomForest  Optimized Outliers Included | 0.133369   | 0.007609
```

```
XGBoost  Optimized Outliers Included     | 0.117024   | 0.008829
LightGBM  Optimized Outliers Included    | 0.121920   | 0.008056
CatBoost  Optimized Outliers Included    | 0.117701   | 0.010105
LinearReg Optimized Outliers Removed     | 0.117137   | 0.011497
Ridge Optimized Outliers Removed         | 0.119382   | 0.012966
Lasso Optimized Outliers Removed         | 0.116944   | 0.012221
SVR Optimized Outliers Removed           | 0.149993   | 0.013145
RandomForest Optimized Outliers Removed  | 0.123081   | 0.006592
XGBoost Optimized Outliers Removed       | 0.100964   | 0.007350
LightGBM Optimized Outliers Removed      | 0.109072   | 0.006067
CatBoost Optimized Outliers Removed      | 0.104437   | 0.005925
Ensemble Model Lasso                     | 0.103121   | 0.006336

Retraining on full data to inspect weights…

--- Final Learned Ensemble Weights ---

    Model    Weight
3     xgb  0.669697
5     cat  0.230218
0  linear  0.109120
2     svr -0.000000
1   lasso  0.000000
4    lgbm  0.000000
6      rf  0.000000
```



```
Generating submission file…
Submission saved as ensemble model.csv
```

[ ]: