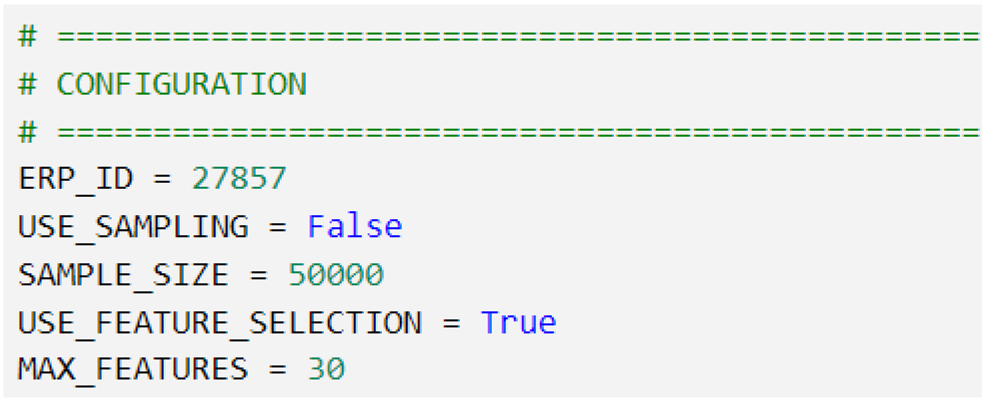
Task 1 Report

Karam Hussain 27857

1. Configuration

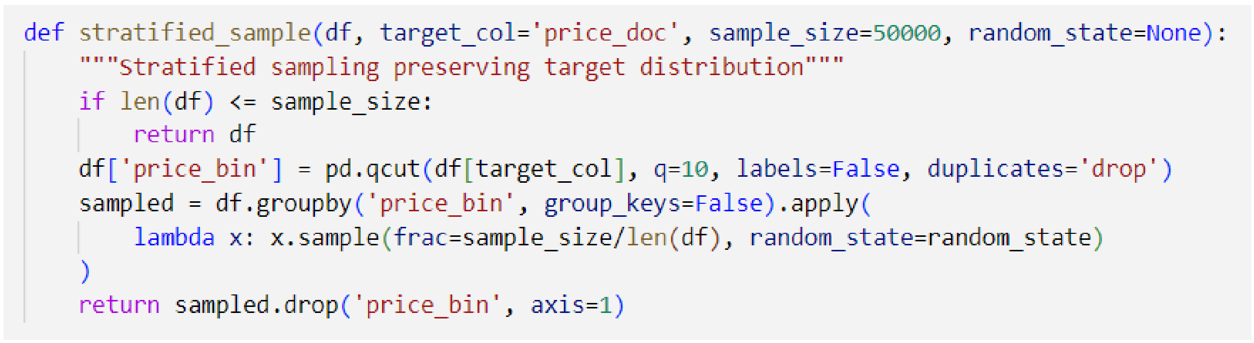


```
# ================================================
# CONFIGURATION
# ================================================
ERP_ID = 27857
USE_SAMPLING = False
SAMPLE_SIZE = 50000
USE_FEATURE_SELECTION = True
MAX_FEATURES = 30
```

My own ERP_ID , option of sampling and its size due to a very large dataset and option of feature selection as well.
For task 1, we have set max features to 30 numeric.

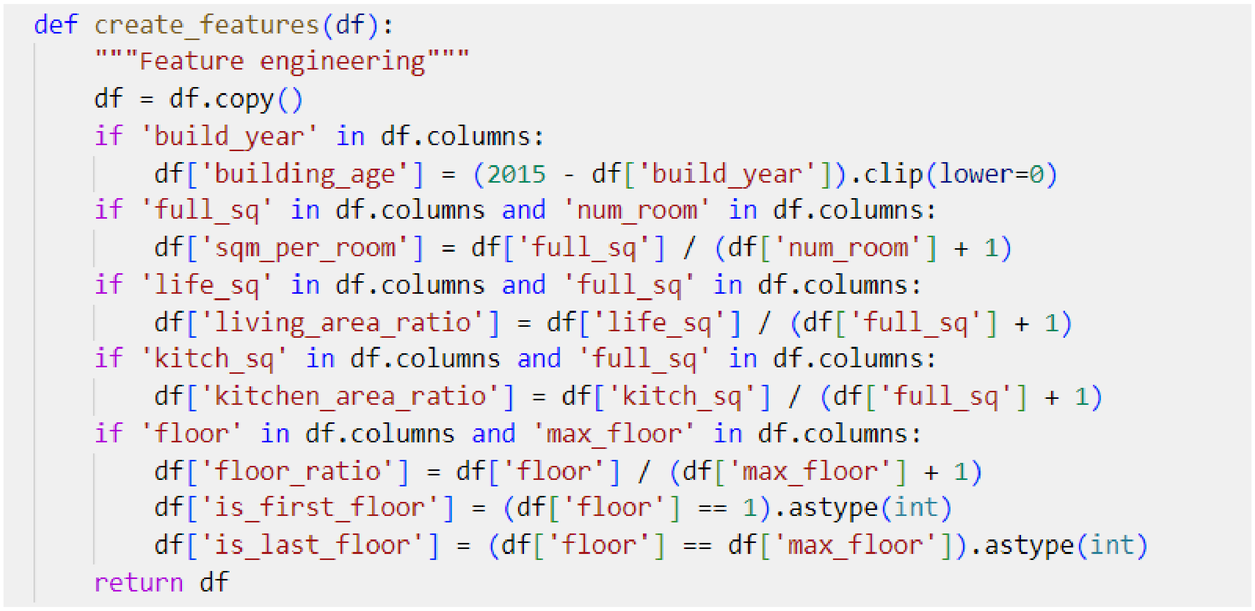
2. Helper functions

a. For stratification (not used though, as sampling not turned on)



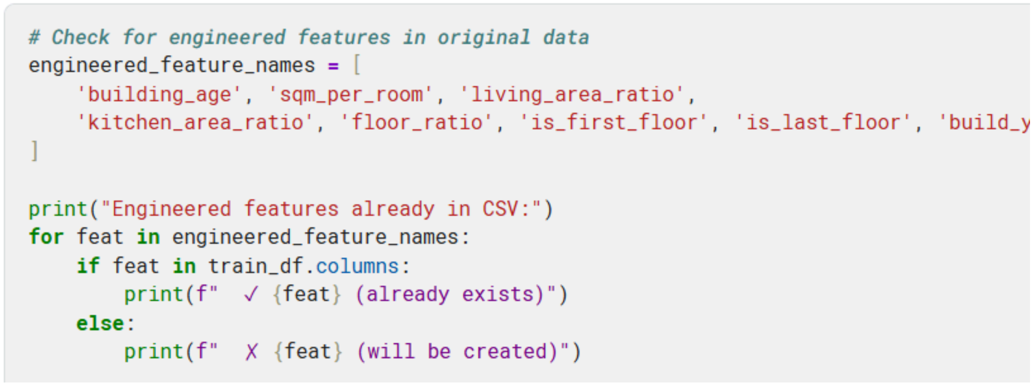
```
def stratified_sample(df, target_col='price_doc', sample_size=50000, random_state=None):
    """Stratified sampling preserving target distribution"""
    if len(df) <= sample_size:
        return df
    df['price_bin'] = pd.qcut(df[target_col], q=10, labels=False, duplicates='drop')
    sampled = df.groupby('price_bin', group_keys=False).apply(
        lambda x: x.sample(frac=sample_size/len(df), random_state=random_state)
    )
    return sampled.drop('price_bin', axis=1)
```

b. For creating features



```python
def create_features(df):
    """Feature engineering"""
    df = df.copy()
    if 'build_year' in df.columns:
        df['building_age'] = (2015 - df['build_year']).clip(lower=0)
    if 'full_sq' in df.columns and 'num_room' in df.columns:
        df['sqm_per_room'] = df['full_sq'] / (df['num_room'] + 1)
    if 'life_sq' in df.columns and 'full_sq' in df.columns:
        df['living_area_ratio'] = df['life_sq'] / (df['full_sq'] + 1)
    if 'kitch_sq' in df.columns and 'full_sq' in df.columns:
        df['kitchen_area_ratio'] = df['kitch_sq'] / (df['full_sq'] + 1)
    if 'floor' in df.columns and 'max_floor' in df.columns:
        df['floor_ratio'] = df['floor'] / (df['max_floor'] + 1)
        df['is_first_floor'] = (df['floor'] == 1).astype(int)
        df['is_last_floor'] = (df['floor'] == df['max_floor']).astype(int)
    return df
```

We will check, which features are present in order to fulfill the conditions which will in turn create new features out of them



```python
# Check for engineered features in original data
engineered_feature_names = [
    'building_age', 'sqm_per_room', 'living_area_ratio',
    'kitchen_area_ratio', 'floor_ratio', 'is_first_floor', 'is_last_floor', 'build_y
]

print("Engineered features already in CSV:")
for feat in engineered_feature_names:
    if feat in train_df.columns:
        print(f"  ✓ {feat} (already exists)")
    else:
        print(f"  ✗ {feat} (will be created)")
```

```
Engineered features already in CSV:
  ✗ building_age (will be created)
  ✗ sqm_per_room (will be created)
  ✓ living_area_ratio (already exists)
  ✗ kitchen_area_ratio (will be created)
  ✗ floor_ratio (will be created)
  ✗ is_first_floor (will be created)
  ✗ is_last_floor (will be created)
  ✗ build_year (will be created)
  ✓ full_sq (already exists)
  ✗ num_room (will be created)
  ✓ life_sq (already exists)
  ✗ kitch_sq (will be created)
  ✓ floor (already exists)
  ✗ max_floor (will be created)
```

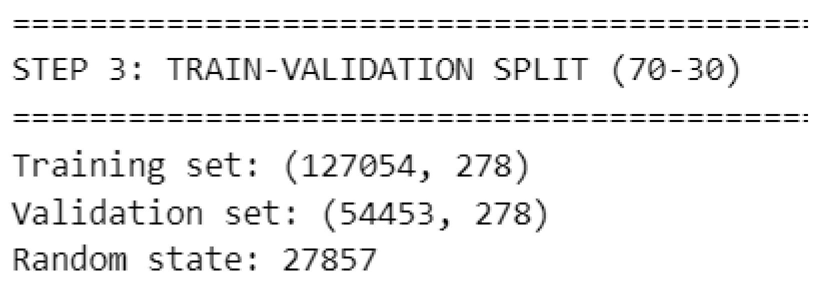
As only the features for making the living_area_ratio are present which are life_sq and full_sq hence only that feature will be created, due to its condition being met.

c. Feature Selection

```
# Method 1: Correlation
correlations = numeric_X.corrwith(y).abs()
top_corr = correlations.nlargest(n_features).index.tolist()

# Method 2: F-statistic
selector = SelectKBest(score_func=f_regression, k=n_features)
selector.fit(X_imputed, y)
top_fstat = numeric_X.columns[selector.get_support()].tolist()

# Method 3: Random Forest
print("  Training Random Forest...")
rf = RandomForestRegressor(n_estimators=50, max_depth=8, random_state=random_state, n_jobs=1)
rf.fit(X_imputed, y)
importances = pd.Series(rf.feature_importances_, index=numeric_X.columns)
top_rf = importances.nlargest(n_features).index.tolist()

# Consensus
from collections import Counter
all_features = top_corr + top_fstat + top_rf
feature_votes = Counter(all_features)
selected = [f for f, votes in feature_votes.most_common() if votes >= 2]
```

We will select 30 numeric features, with methods of correlation, F-stat, and Random Forest and a combined consensus to get the best 30 numeric features.

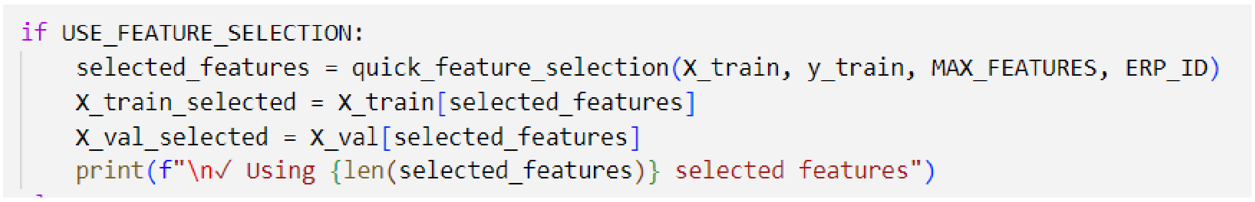
3. Train validation split



```
==========================================
STEP 3: TRAIN-VALIDATION SPLIT (70-30)
==========================================
Training set: (127054, 278)
Validation set: (54453, 278)
Random state: 27857
```

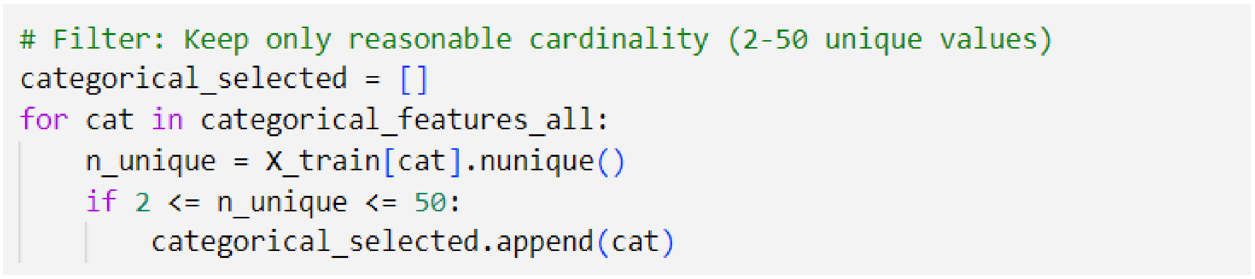
Val set 0.3
‘Id’ and ‘price_doc’ cols were removed from 279 cols to make it 277 with +1 engineered feature to make it 278.

4. Combining categorical variables to numerical variables



```python
if USE_FEATURE_SELECTION:
    selected_features = quick_feature_selection(X_train, y_train, MAX_FEATURES, ERP_ID)
    X_train_selected = X_train[selected_features]
    X_val_selected = X_val[selected_features]
    print(f"\n✓ Using {len(selected_features)} selected features")
```

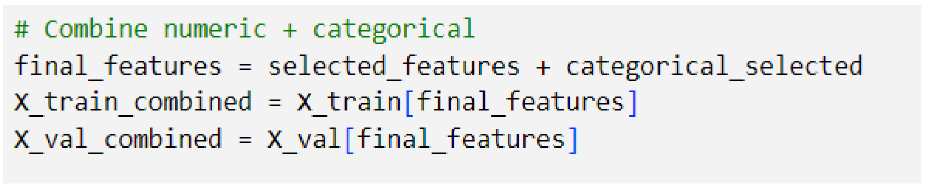
Got 30 numeric variables
Then



```python
# Filter: Keep only reasonable cardinality (2-50 unique values)
categorical_selected = []
for cat in categorical_features_all:
    n_unique = X_train[cat].nunique()
    if 2 <= n_unique <= 50:
        categorical_selected.append(cat)
```

From the already less number of cat variables around 17 of them, we will choose with 2-50 unique values, making it 14 cat variables.

Totalling 30+14 = 44 variables.



```python
# Combine numeric + categorical
final_features = selected_features + categorical_selected
X_train_combined = X_train[final_features]
X_val_combined = X_val[final_features]
```

```
✓ Found 15 categorical features in dataset
  Examples:
    • product_type: 2 categories
    • sub_area: 1924 categories
    • culture_objects_top_25: 2 categories
    • thermal_power_plant_raion: 2 categories
    • incineration_raion: 2 categories

✓ Selected 14 categorical features (2-50 categories)

✓ FINAL FEATURE SET:
  Numeric:       30
  Categorical:   14
  Total:         44
```

5. Pre-processing pipeline

```
# Numeric transformer
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),  # Median for numeric
    ('scaler', StandardScaler())                     # Standardize
])
```

For numeric, applying median as imputer and scaling them as well.

```
# Categorical transformer - WITH MODE FOR MISSING VALUES
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),  # MODE for categorical
    ('onehot', OneHotEncoder(
        handle_unknown='ignore',
        sparse_output=False,
        max_categories=20
    ))
])
```

For categorical, using mode as imputer and also doing one-hot encoding.

```
✓ Numeric features: 30
✓ Categorical features: 14

✓ Preprocessing Pipeline Created:
  [Numeric]     Median imputation → Standard scaling
  [Categorical] MODE imputation → One-hot encoding
```

6. Model Evaluation

a. Baseline linear regression

```
baseline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])
```

```
Baseline Linear
  Val RMSE: 13.2200 | Train RMSE: 13.2154
  Val R²: 0.6271 | Train R²: 0.6216
  Training: 2.27s | Inference: 0.7824s
```

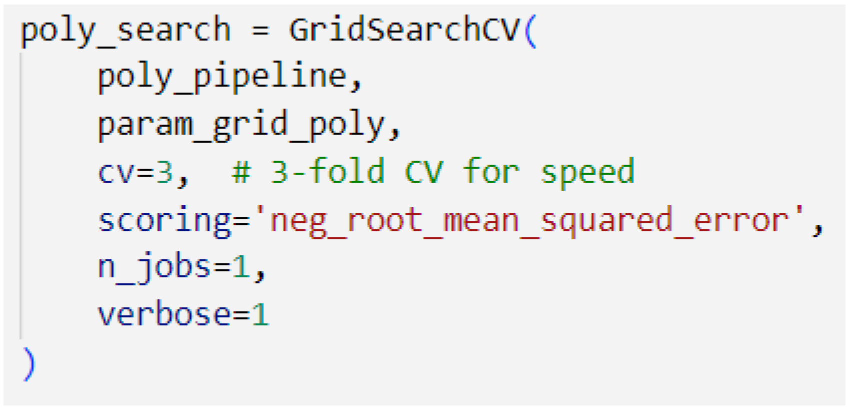
b. Polynomial Regression with Gridsearch CV

```
poly_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('poly', PolynomialFeatures(include_bias=False)),
    ('regressor', LinearRegression())
])
```

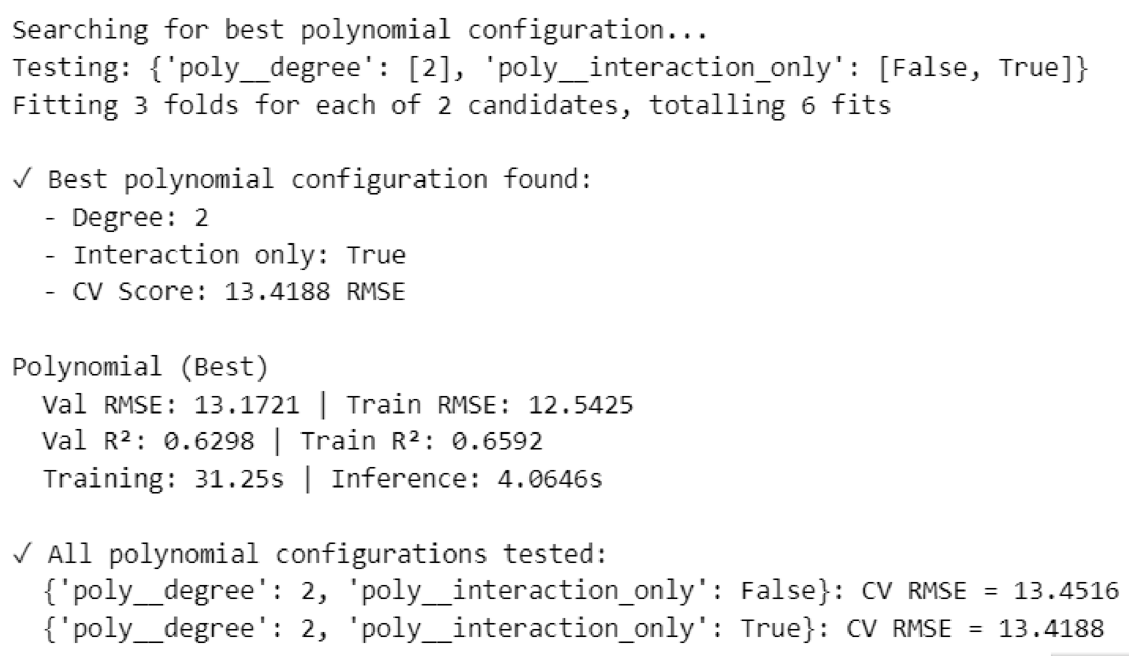
```
# Parameter grid for polynomial tuning
param_grid_poly = {
    'poly__degree': [2],  # Degree 2 and 3 as required
    'poly__interaction_only': [False, True]  # With and without interactions
}
```

At the moment I did with poly degree = 2 as my local machine and even kaggle was crashing but for poly degree = 3, did a work around for that in which I used a sample of the dataset with the selected features to reduce the load by lot.

CV = 3

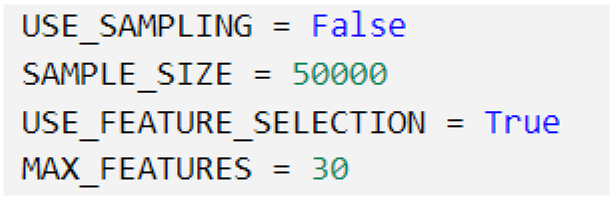


```
poly_search = GridSearchCV(
    poly_pipeline,
    param_grid_poly,
    cv=3,  # 3-fold CV for speed
    scoring='neg_root_mean_squared_error',
    n_jobs=1,
    verbose=1
)
```



```
Searching for best polynomial configuration...
Testing: {'poly__degree': [2], 'poly__interaction_only': [False, True]}
Fitting 3 folds for each of 2 candidates, totalling 6 fits

✓ Best polynomial configuration found:
  - Degree: 2
  - Interaction only: True
  - CV Score: 13.4188 RMSE

Polynomial (Best)
  Val RMSE: 13.1721 | Train RMSE: 12.5425
  Val R²: 0.6298 | Train R²: 0.6592
  Training: 31.25s | Inference: 4.0646s

✓ All polynomial configurations tested:
  {'poly__degree': 2, 'poly__interaction_only': False}: CV RMSE = 13.4516
  {'poly__degree': 2, 'poly__interaction_only': True}: CV RMSE = 13.4188
```

For poly degree = 3, used sampling with size of 50k and selected features:



```
USE_SAMPLING = False
SAMPLE_SIZE = 50000
USE_FEATURE_SELECTION = True
MAX_FEATURES = 30
```

```
Searching for best polynomial configuration...
Testing: {'poly__degree': [2, 3], 'poly__interaction_only': [False, True]}
Fitting 3 folds for each of 4 candidates, totalling 12 fits

✓ Best polynomial configuration found:
  - Degree: 2
  - Interaction only: True
  - CV Score: 13.1697 RMSE

Polynomial (Best)
  Val RMSE: 13.0829 | Train RMSE: 12.8238
  Val R²: 0.6348 | Train R²: 0.6437
  Training: 13.41s | Inference: 2.8239s

✓ All polynomial configurations tested:
  {'poly__degree': 2, 'poly__interaction_only': False}: CV RMSE = 13.1798
  {'poly__degree': 2, 'poly__interaction_only': True}: CV RMSE = 13.1697
  {'poly__degree': 3, 'poly__interaction_only': False}: CV RMSE = nan
  {'poly__degree': 3, 'poly__interaction_only': True}: CV RMSE = nan
```

Still it was unable to compute the CV RMSE probably due to a technical error or load, even though it ran full.

c. Ridge regression with GridSearch CV

```
ridge_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', Ridge())
])
```

```
param_grid_ridge = {
    'regressor__alpha': [0.01, 0.1, 1, 10, 100, 1000]
}
```

```
ridge_search = GridSearchCV(
    ridge_pipeline, param_grid_ridge, cv=3,
    scoring='neg_root_mean_squared_error', n_jobs=1
)
```

Applying Grid CV for alpha as mentioned with best selected as 1000.

```
Tuning Ridge regularization strength: [0.01, 0.1, 1, 10, 100, 1000]

✓ Best Ridge alpha: 1000
  CV Score: 13.2427 RMSE

Ridge (Best)
  Val RMSE: 13.2199 | Train RMSE: 13.2158
  Val R²: 0.6271 | Train R²: 0.6216
  Training: 1.82s | Inference: 0.7255s
```

d. Lasso regression with GridSearch CV

```python
lasso_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', Lasso(max_iter=10000))
])
```

```python
param_grid_lasso = {
    'regressor__alpha': [0.001, 0.01, 0.1, 1, 10, 100]
}
```

```python
lasso_search = GridSearchCV(
    lasso_pipeline, param_grid_lasso, cv=3,
    scoring='neg_root_mean_squared_error', n_jobs=1
)
```

Again grid CV on alpha as mentioned in which we got 0.01 alpha as best.

```
Tuning Lasso regularization strength: [0.001, 0.01, 0.1, 1, 10, 100]

✓ Best Lasso alpha: 0.01
  CV Score: 13.2427 RMSE

Lasso (Best)
  Val RMSE: 13.2191 | Train RMSE: 13.2162
  Val R²: 0.6271 | Train R²: 0.6216
  Training: 5.06s | Inference: 0.7543s
```

e. Elastic Net with Grid Search CV

```
elastic_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', ElasticNet(max_iter=10000))
])
```

```
param_grid_elastic = {
    'regressor__alpha': [0.01, 0.1, 1, 10],
    'regressor__l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9]
}
```

```
elastic_search = GridSearchCV(
    elastic_pipeline, param_grid_elastic, cv=3,
    scoring='neg_root_mean_squared_error', n_jobs=-1
)
```

Grid CV on alpha as mentioned upon which we got alpha 0.01 and L1 ratio 0.5

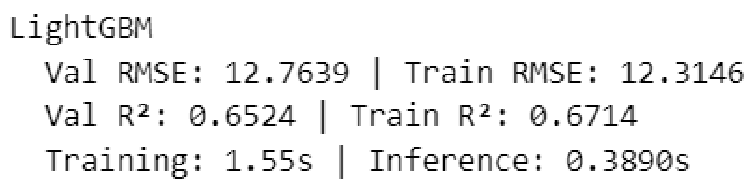
```
Tuning Elastic Net hyperparameters:
  Alpha: [0.01, 0.1, 1, 10]
  L1 ratio: [0.1, 0.3, 0.5, 0.7, 0.9]

✓ Best Elastic Net parameters:
  Alpha: 0.01
  L1 ratio: 0.5
  CV Score: 13.2425 RMSE

Elastic Net (Best)
  Val RMSE: 13.2194 | Train RMSE: 13.2159
  Val R²: 0.6271 | Train R²: 0.6216
  Training: 5.40s | Inference: 0.7456s
```

f. Gradient Boosting (LightGBM)

```
lgb_model = lgb.LGBMRegressor(
    objective='regression',
    metric='rmse',
    n_estimators=500,
    learning_rate=0.05,
    max_depth=7,
    num_leaves=31,
    min_child_samples=20,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=ERP_ID,
    verbose=-1
)
```

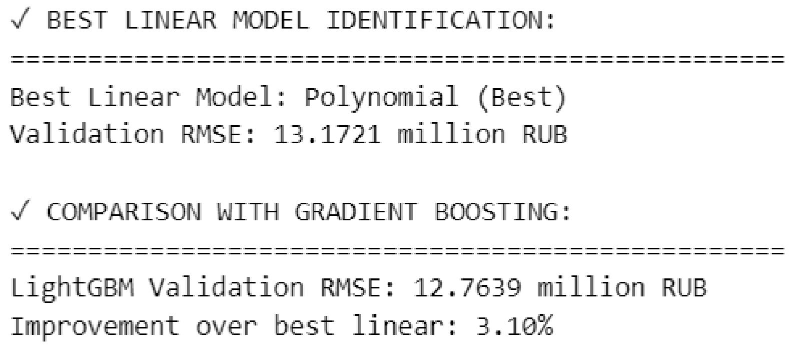
```
lgb_model.fit(
    X_train_lgb_imputed, y_train,
    eval_set=[(X_val_lgb_imputed, y_val)],
    callbacks=[lgb.early_stopping(50, verbose=False)]
)
```

Selected estimated best known hyper parameters in grid instead of tuning to save time as the model was already taking a lot of time.

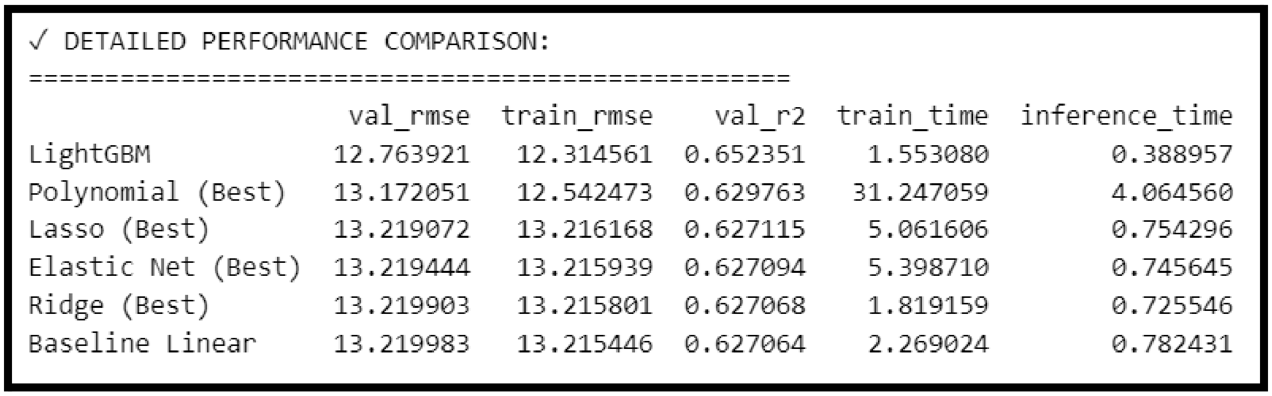


```
LightGBM
  Val RMSE: 12.7639 | Train RMSE: 12.3146
  Val R²: 0.6524 | Train R²: 0.6714
  Training: 1.55s | Inference: 0.3890s
```

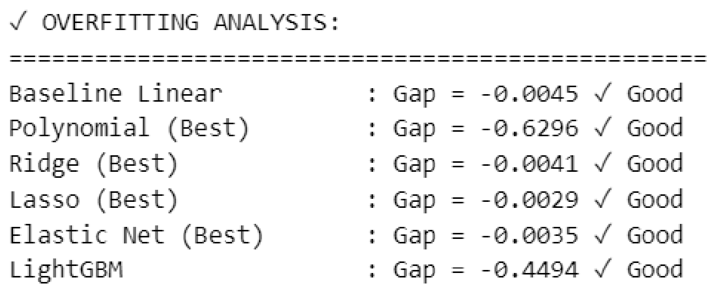
7. Result Summary



```
✓ BEST LINEAR MODEL IDENTIFICATION:
==================================================
Best Linear Model: Polynomial (Best)
Validation RMSE: 13.1721 million RUB

✓ COMPARISON WITH GRADIENT BOOSTING:
==================================================
LightGBM Validation RMSE: 12.7639 million RUB
Improvement over best linear: 3.10%
```



```
✓ DETAILED PERFORMANCE COMPARISON:
==================================================
                     val_rmse  train_rmse    val_r2  train_time  inference_time
LightGBM            12.763921   12.314561  0.652351    1.553080        0.388957
Polynomial (Best)   13.172051   12.542473  0.629763   31.247059        4.064560
Lasso (Best)        13.219072   13.216168  0.627115    5.061606        0.754296
Elastic Net (Best)  13.219444   13.215939  0.627094    5.398710        0.745645
Ridge (Best)        13.219903   13.215801  0.627068    1.819159        0.725546
Baseline Linear     13.219983   13.215446  0.627064    2.269024        0.782431
```



```
✓ OVERFITTING ANALYSIS:
==================================================
Baseline Linear          : Gap = -0.0045 ✓ Good
Polynomial (Best)        : Gap = -0.6296 ✓ Good
Ridge (Best)             : Gap = -0.0041 ✓ Good
Lasso (Best)             : Gap = -0.0029 ✓ Good
Elastic Net (Best)       : Gap = -0.0035 ✓ Good
LightGBM                 : Gap = -0.4494 ✓ Good
```

The final best **linear** model was Lasso while polynomial with degree 2 (as its relation with target variable was non linear so didnt consider it completely “linear”)took the second position after LGBM.

Analysis:
The results clearly show notable differences in performance across the tested models, especially between purely linear methods, polynomial transformation models, and gradient-boosted decision trees. Among the linear family models, **Lasso Regression with α = 0.01 emerged as the best linear model**, achieving a validation RMSE of approximately **13.219M RUB**. Its performance was very similar to Ridge, Elastic Net, and the baseline Linear Regression model, indicating that the dataset does not benefit significantly from strong regularization or sparsity constraints. Lasso performed best among linear models mainly because its mild regularization preserved important features while controlling noise. However, all linear models displayed almost identical RMSE values, which suggests that purely linear relationships are insufficient to fully explain the complexity in housing prices.

When extending beyond linearity, the **Polynomial Regression model (degree 2, interaction-only)** provided the best overall performance among the “linear-family” methods. It achieved a lower validation RMSE of **13.1721M RUB**, outperforming all linear regularized models. This slight improvement indicates that feature interactions do contribute meaningful information to the model. By expanding the feature space to include second-order interaction terms, the model was able to capture nonlinear relationships that ordinary linear models could not represent. However, this increased accuracy came at a significant computational cost: Polynomial Regression required substantially more training time compared to Ridge, Lasso, and Elastic Net. Additionally, a small gap between training and validation RMSE suggests mild overfitting, which is expected when the feature space becomes more complex.

The **LightGBM model** achieved the strongest performance overall, reducing the validation RMSE to **12.7639M RUB**, which represents a **3.10% improvement over the best polynomial model** and an even larger improvement over the best strictly linear model (Lasso). This demonstrates LightGBM's ability to capture complex nonlinear patterns, sharp thresholds, and higher-order interactions without manually engineering features. LightGBM also trained very quickly (around 1.55 seconds) and produced the fastest inference time among the non-linear models, showing both high efficiency and strong accuracy. Its superior performance highlights the limitations of linear and polynomial transformations for structured tabular data, where boosted decision trees often excel.

Hyperparameter Analysis:

The hyperparameter choices for the Lasso model reveal that the dataset requires only light regularization. The best value of **α = 0.01** suggests that the model benefits from controlling coefficient magnitude just enough to reduce noise, but stronger penalties would have removed useful predictive information. This explains why Ridge and Elastic Net, which applied heavier penalties, did not outperform Lasso. Their similar RMSE values across the board indicate that the dataset lacks sparse structure and instead requires keeping most features intact. This also reinforces why the baseline Linear Regression performed nearly identically: regularization had little effect.

For the Polynomial Regression model, the optimal configuration of **degree = 2 with interaction-only terms** shows that higher-order nonlinearities contribute modestly to predictive power. Including full polynomial features would have introduced many unnecessary squared terms and dramatically increased dimensionality, likely causing overfitting. The fact that interaction-only features outperformed full quadratic expansion suggests that relationships between pairs of variables—such as size × location or building type × district—matter more than curved (squared) effects. The model improved accuracy slightly but required dramatically more training time, indicating a trade-off between complexity and performance.

The LightGBM hyperparameters reflect a balanced configuration aimed at capturing nonlinear structure while preventing overfitting. A **learning rate of 0.05** combined with **500 estimators** allows the model to learn gradually and improve generalization. The **max_depth of 7** and **num_leaves = 31** restrict tree complexity, preventing overly deep trees that could fit noise. Parameters like **min_child_samples = 20**, **subsample = 0.8**, and **colsample_bytree = 0.8** add randomness to training, reducing tree correlation and improving robustness. Overall, these hyperparameters form a conservative, stability-focused configuration—which explains the excellent validation performance and relatively low training time.

Insights:

The results show that the dataset contains nonlinear relationships that simple linear models cannot capture. The fact that polynomial interactions improved performance indicates that feature combinations—such as size × location or building type × district—play an important role in determining housing prices. This suggests that the underlying structure of the data is moderately complex and influenced by interacting factors rather than independent linear effects.

Although Lasso was the best strictly linear model, its performance was nearly identical to Ridge, Elastic Net, and even the baseline Linear Regression. This similarity highlights that regularization does not significantly improve predictive power, implying the dataset does not suffer from severe overfitting in linear space. The polynomial model performed better but only slightly, showing that while interactions matter, they are not strong enough to drastically change outcomes without more advanced modeling.

LightGBM's noticeable improvement over all linear-based methods demonstrates that gradient boosting captures complex nonlinear patterns and hierarchical interactions far more effectively. Its low validation RMSE and fast inference time show that boosted trees can learn the underlying structure with better generalization. This reinforces a common pattern in tabular datasets: tree-based models often outperform linear models because they naturally learn thresholds and nonlinear splits that reflect real-world price behaviors.

During testing, all degree-3 polynomial configurations returned **NaN for the CV RMSE**, meaning the model could not compute valid predictions. This likely happened because expanding to degree 3 created an extremely large number of features, which led to numerical instability or overflow during cross-validation. The model could not handle the high-dimensional feature space, and as a result, training failed to converge. In contrast, degree-2 polynomial models worked reliably, capturing meaningful feature interactions without causing computation issues. This demonstrates that the dataset only supports moderate nonlinear expansions, and pushing to higher-degree polynomials exceeds computational limits without adding predictive value.