Machine Learning Final Project

Table Of Contents

```
from google.colab import drive
drive.mount('/content/drive')

    Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

1. Baseline Model Development

Imports and Configuration

```
# ---------------- Imports ----------------
# Data manipulation and analysis
import pandas as pd
import numpy as np
import xgboost as xgb

# Visualization
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns

# Machine learning - Preprocessing
from sklearn.model_selection import train_test_split, cross_val_score, RandomizedSearchCV
from sklearn.metrics import mean_squared_error, r2_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer

# Machine learning - Models
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LassoCV

# Machine learning - Metrics
from sklearn.metrics import mean_squared_error

# Time series analysis
from statsmodels.tsa.seasonal import seasonal_decompose
```

------------------------ Mission 2 --------------------------------

------------------------------ Exploratory Data Analysis ------------------------------

Data Loading and Initial Analysis

Data Import

We load our training and test datasets from Google Drive to perform comprehensive exploratory data analysis (EDA).

Feature Selection

Selected features for EDA include:

- **Architectural**: `full_sq`, `life_sq`, `floor`, `max_floor`, `num_room`, `kitch_sq`
- **Construction Material and Age**: `material`, `build_year`
- **Property Classification**: `state`, `product_type`, `sub_area`

Descriptive Statistics

Descriptive statistics are computed for numerical features to summarize central tendencies, dispersion, and shape of the dataset's distribution.

Implementation

```
# Load the training data to perform EDA on the specified features

train_df = pd.read_csv('/content/drive/MyDrive/פרויקט למידת מכונה/train.csv')
#train_df = pd.read_csv('/content/drive/MyDrive/פרויקט למידת מכונה/train.csv')
test_df = pd.read_csv('/content/drive/MyDrive/פרויקט למידת מכונה/Task2_test.csv')

# Selecting the specified features for EDA
features = ['full_sq', 'life_sq', 'floor', 'max_floor', 'material',
            'build_year', 'num_room', 'kitch_sq', 'state', 'product_type', 'sub_area']

# Filtering the dataset for these features
filtered_df = train_df[features]

# Calculating descriptive statistics for numerical features
descriptive_stats = filtered_df.describe(include=[np.number])  # Including only numeric data types for these stats
# descriptive_stats
```

Categorical Feature Distribution Analysis

Overview

Analyzing the distribution of categorically significant features to assess their prevalence and variability.

Features Analyzed

- **Categorical (Numeric Representation)**: `material`, `state`
- **Direct Categorical**: `product_type`, `sub_area`

```
# Calculating frequency distributions for categorical features
# For 'material' and 'state', although numeric, they represent categories
material_freq = filtered_df['material'].value_counts(dropna=False)
state_freq = filtered_df['state'].value_counts(dropna=False)
product_type_freq = filtered_df['product_type'].value_counts(dropna=False)
sub_area_freq = filtered_df['sub_area'].value_counts(dropna=False)

# material_freq, state_freq, product_type_freq, sub_area_freq
```

Visualization Summary

Setup and Style

- Aesthetic set to 'whitegrid' for clarity.
- Multiple subplots arranged in a figure of size (18, 12).

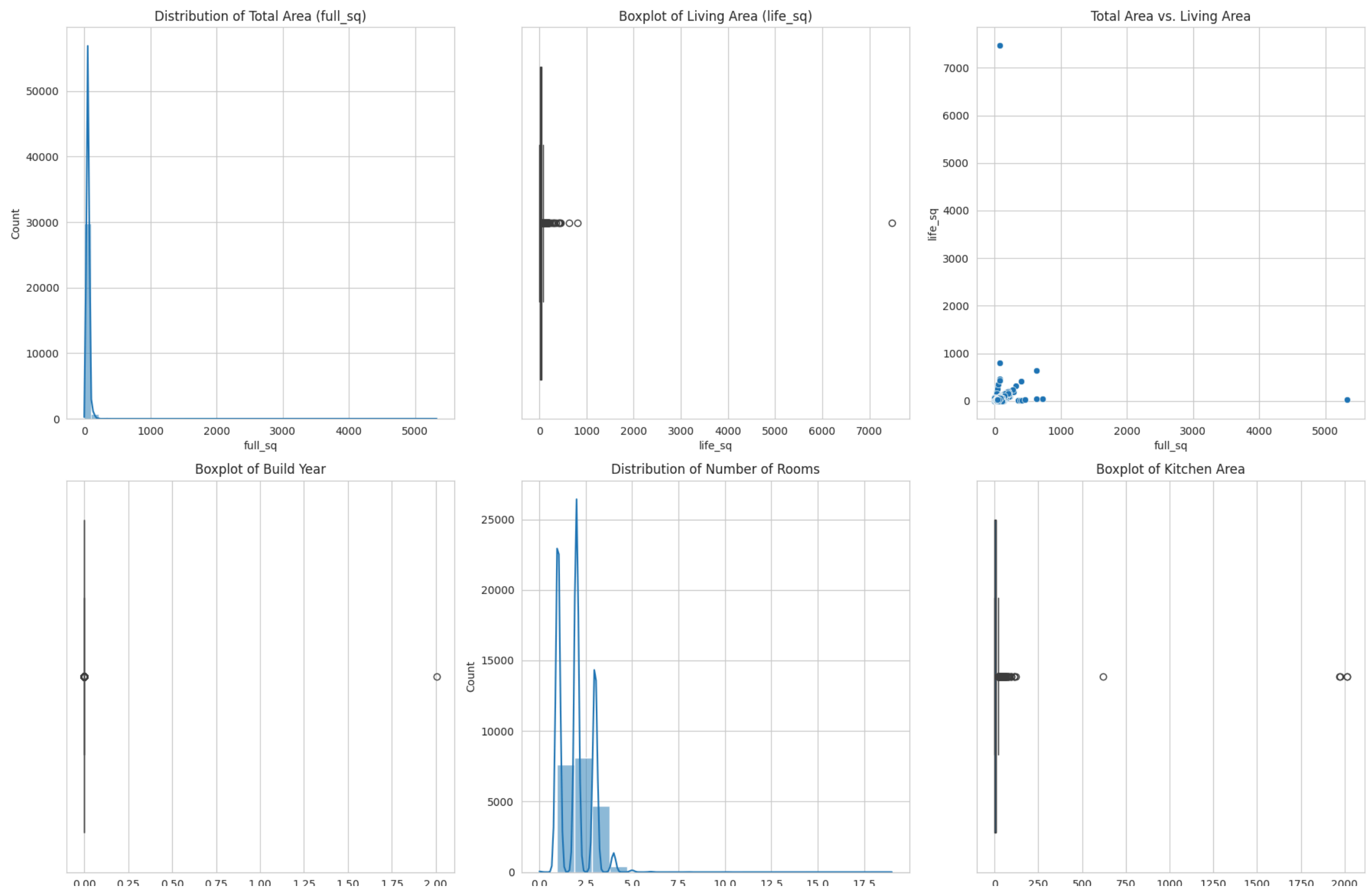
Plots Overview

- **Histograms and KDE**: Used for `full_sq` and `num_room` to assess distribution and density.
- **Boxplots**: Applied to `life_sq`, `build_year`, and `kitch_sq` to highlight median, quartiles, and outliers.
- **Scatter Plot**: Explores the relationship between `full_sq` and `life_sq`.

Purpose

These visualizations aid in understanding the data's distribution, detecting outliers, and identifying relationships between features, crucial for informed modeling and analysis decisions.

```
# Setting the aesthetic style of the plots
sns.set_style("whitegrid")

# Creating a figure to hold the subplots
plt.figure(figsize=(18, 12))

# Histogram for 'full_sq'
plt.subplot(2, 3, 1)
sns.histplot(data=filtered_df, x='full_sq', bins=50, kde=True)
plt.title('Distribution of Total Area (full_sq)')

# Boxplot for 'life_sq'
plt.subplot(2, 3, 2)
sns.boxplot(data=filtered_df, x='life_sq')
plt.title('Boxplot of Living Area (life_sq)')

# Scatterplot of 'full_sq' vs. 'life_sq'
plt.subplot(2, 3, 3)
sns.scatterplot(data=filtered_df, x='full_sq', y='life_sq')
plt.title('Total Area vs. Living Area')

# Boxplot for 'build_year'
plt.subplot(2, 3, 4)
sns.boxplot(data=filtered_df, x='build_year')
plt.title('Boxplot of Build Year')

# Histogram for 'num_room'
plt.subplot(2, 3, 5)
sns.histplot(data=filtered_df, x='num_room', bins=20, kde=True)
plt.title('Distribution of Number of Rooms')

# Boxplot for 'kitch_sq'
plt.subplot(2, 3, 6)
sns.boxplot(data=filtered_df, x='kitch_sq')
plt.title('Boxplot of Kitchen Area')

# Adjusting layout for better readability
plt.tight_layout()
plt.show()
```



------------------------------ Missing Values ------------------------------

Handling Missing Values

Missing Value Analysis

- **Detection**: Calculated the count and percentage of missing values for each feature in `filtered_df`.
- **Summary Table**: Created a DataFrame `missing_df` to display missing values and their percentages, focusing on features with missing data.

Imputation Strategy

- **Method**: Applied median imputation using `SimpleImputer` for non-predictive, numerical features.
- **Features Imputed**: `life_sq`, `floor`, `kitch_sq`, `build_year`, `state`, `max_floor`, `material`, `num_room`.
- **Verification**: After imputation, rechecked for missing values to ensure all were addressed.

Purpose

This process ensures the dataset is complete, addressing potential biases or errors in analysis due to missing data. Median imputation maintains the central tendency without being skewed by outliers.

```
# Identifying missing values in each feature
missing_values = filtered_df.isnull().sum()

# Calculating the percentage of missing values for each feature
missing_percentage = (filtered_df.isnull().sum() / len(filtered_df)) * 100

# Combining both the count and percentage of missing values into a DataFrame
missing_df = pd.DataFrame({'Missing Values': missing_values, 'Percentage': missing_percentage})

# Sorting the features by percentage of missing values in descending order
missing_df = missing_df[missing_df['Missing Values'] > 0].sort_values(by='Percentage', ascending=False)

missing_df
```

	Missing Values	Percentage
build_year	13605	44.649011
state	13559	44.498047
max_floor	9572	31.413475
material	9572	31.413475
num_room	9572	31.413475
kitch_sq	9572	31.413475
life_sq	6383	20.947786
floor	167	0.548062

Next steps: View recommended plots

```
# For simplicity, directly imputing median/mode for non-predictive features
simple_imputer_median = SimpleImputer(strategy='median')

# Imputing 'life_sq', 'floor', 'kitch_sq' with median values
for feature in ['build_year', 'state', 'max_floor', 'material', 'num_room', 'kitch_sq', 'life_sq', 'floor']:
    filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])


# Checking the current state of missing values after imputation
filtered_df.isnull().sum()
```

```
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    <ipython-input-135-9c190c955858>:6: SettingWithCopyWarning:
    A value is trying to be set on a copy of a slice from a DataFrame.
    Try using .loc[row_indexer,col_indexer] = value instead

    See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
      filtered_df[feature] = simple_imputer_median.fit_transform(filtered_df[[feature]])
    full_sq         0
    life_sq         0
    floor           0
    max_floor       0
    material        0
    build_year      0
    num_room        0
    kitch_sq        0
    state           0
    product_type    0
    sub_area        0
    dtype: int64
```

```
filtered_df
```

	full_sq	life_sq	floor	max_floor	material	build_year	num_room	kitch_sq	state	product_type	sub_area
0	43	27.0	4.0	12.0	1.0	1979.0	2.0	6.0	2.0	Investment	Bibirevo
1	34	19.0	3.0	12.0	1.0	1979.0	2.0	6.0	2.0	Investment	Nagatinskij Zaton
2	43	29.0	2.0	12.0	1.0	1979.0	2.0	6.0	2.0	Investment	Tekstil'shhiki
3	89	50.0	9.0	12.0	1.0	1979.0	2.0	6.0	2.0	Investment	Mitino
4	77	77.0	4.0	12.0	1.0	1979.0	2.0	6.0	2.0	Investment	Basmannoe
...	...	...	...	...	...	...	...	...	...	...	...
30466	44	27.0	7.0	9.0	1.0	1975.0	2.0	6.0	3.0	Investment	Otradnoe
30467	86	59.0	3.0	9.0	2.0	1935.0	4.0	10.0	3.0	Investment	Tverskoe
30468	45	30.0	10.0	20.0	1.0	1979.0	1.0	1.0	1.0	OwnerOccupier	Poselenie Vnukovskoe
30469	64	32.0	5.0	15.0	1.0	2003.0	2.0	11.0	2.0	Investment	Obruchevskoe
30470	43	28.0	1.0	9.0	1.0	1968.0	2.0	6.0	2.0	Investment	Novogireevo

30471 rows × 11 columns

Next steps: View recommended plots

Classifying features aids in applying appropriate data processing techniques such as scaling for numeric features and encoding for categorical features, ensuring accurate model training.

```
numeric_features = ['full_sq', 'life_sq', 'floor', 'max_floor', 'material',
                    'build_year', 'num_room', 'kitch_sq', 'state']
categorical_features = ['product_type', 'sub_area']
```

------------------------- Outliers -------------------------

Outlier Detection and Visualization

Overview

We employ the Interquartile Range (IQR) method to identify and visualize outliers in key numeric features, enhancing data reliability for modeling.

Process

- **IQR Calculation**: Determine the IQR for `full_sq`, `life_sq`, and `build_year` to identify the range within which most data points lie.
- **Boundaries for Outliers**:
  - **Lower Bound**: Calculated as `Q1 - 1.5 * IQR`.
  - **Upper Bound**: Calculated as `Q3 + 1.5 * IQR`.
- **Visualization**: Outliers are visualized using box plots for each feature, providing a graphical representation of the data spread and outlier presence.

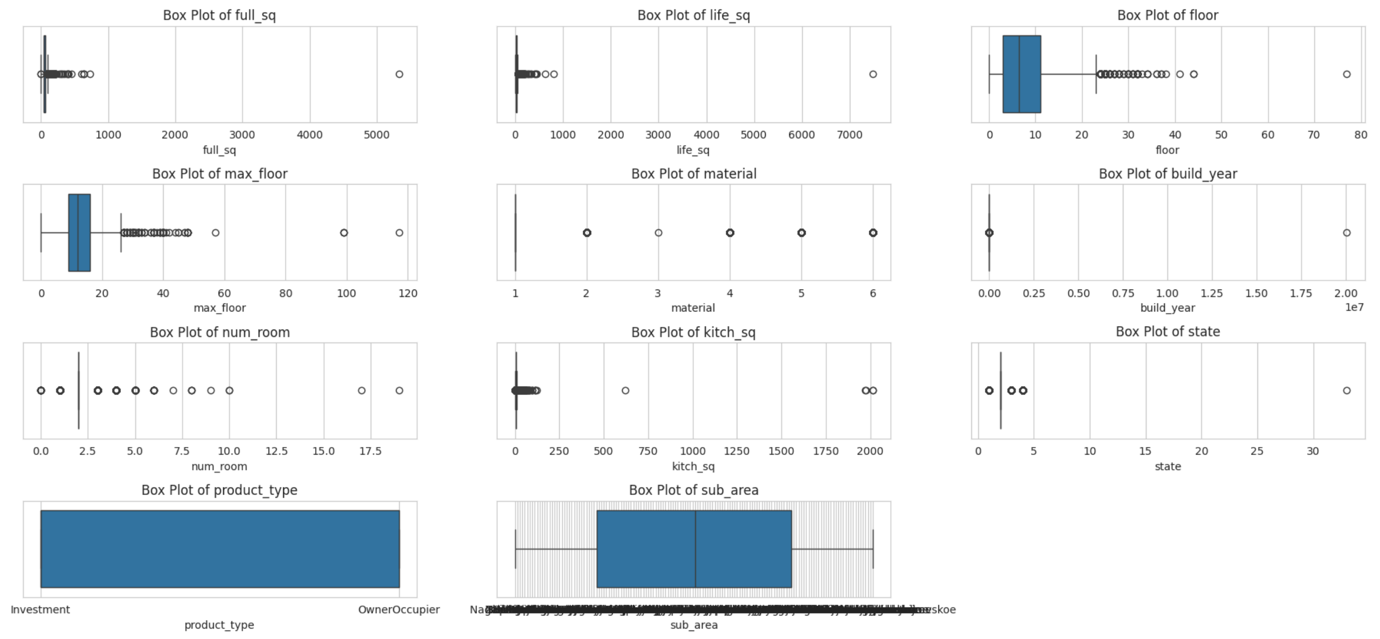
Visualization Setup

- **Figure Configuration**: Set up a large figure to accommodate multiple box plots for a clear and comparative view of feature distributions.
- **Box Plot Creation**: Each feature's distribution is displayed in a subplot, with titles indicating the specific feature being analyzed.

Insights

This method helps in pinpointing extreme values that may skew analysis, allowing for decisions on whether to remove or adjust these data points before further analysis.

```
# Detecting outliers in a few key features using the IQR method
# and visualize them using box plots for 'full_sq', 'life_sq', and 'build_year'
# Calculating descriptive statistics for numeric features only

# Calculating IQR for 'full_sq', 'life_sq', and 'build_year'
Q1 = filtered_df[numeric_features].quantile(0.05)
Q3 = filtered_df[numeric_features].quantile(0.95)
IQR = Q3 - Q1

# Determining the bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Visualizing outliers with box plots
plt.figure(figsize=(18, 10))


for i in range(len(features)):
  plt.subplot(5, 3, int(i)+1)
  sns.boxplot(x=filtered_df[f'{features[i]}'])
  plt.title(f'Box Plot of {features[i]}')


plt.tight_layout()
plt.show()

# Outputting bounds for potential outlier detection
lower_bound, upper_bound
# filtered_df
```



```
(full_sq        -56.0
 life_sq        -53.5
 floor          -23.0
 max_floor      -28.0
 material        -5.0
 build_year    1874.0
 num_room        -2.0
 kitch_sq       -14.0
 state           -2.0
 dtype: float64,
 full_sq        176.0
 life_sq        134.5
 floor           41.0
 max_floor       52.0
 material        11.0
 build_year    2098.0
 num_room         6.0
 kitch_sq        26.0
 state            6.0
 dtype: float64)
```

Double-click (or enter) to edit

Cleaning the dataset by removing outliers from key features, enhancing model accuracy and robustness.

```
# Remove outliers based on IQR for all features except 'build_year'
for feature in features:
    if feature != 'product_type' and feature != 'sub_area' and feature != 'timestamp' :
        lower = lower_bound[feature]
        upper = upper_bound[feature]
        filtered_df = filtered_df[(filtered_df[feature] >= lower) &
                                                  (filtered_df[feature] <= upper)]

# Display the shape of the DataFrame after outlier removal

filtered_df.shape
```

```
    (29343, 11)
```

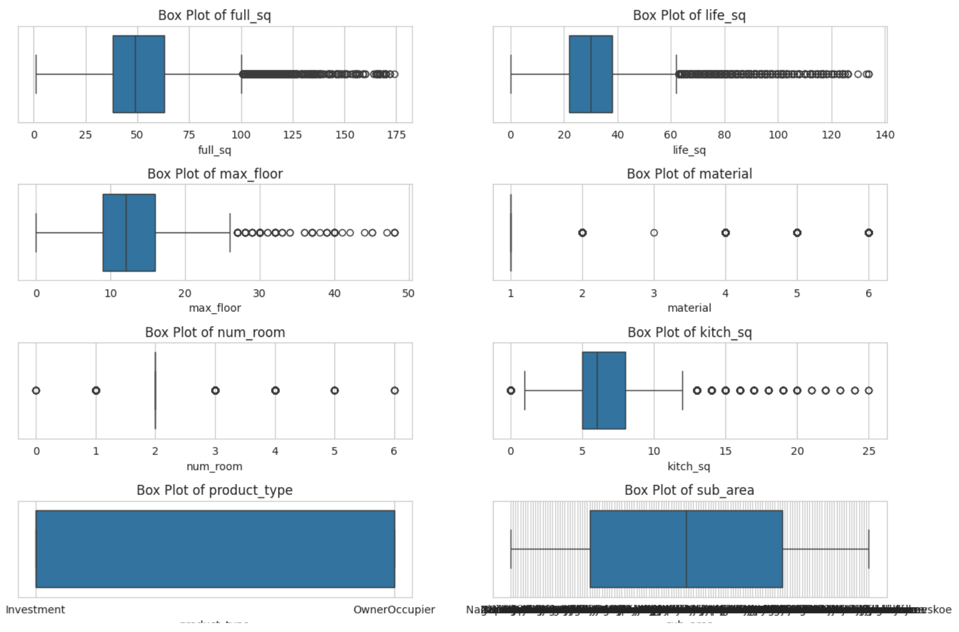
Checking if the range of Build Year is accurate

```
# Calculating the minimum and maximum of the 'build_year' after excluding obvious outliers
build_year_min = train_df[features]['build_year'].min()
build_year_max = train_df[features]['build_year'].max()

build_year_min, build_year_max
```

```
    (0.0, 20052009.0)
```

Visualizing the plots after removing outliers

```
plt.figure(figsize=(18, 10))

for i in range(len(features)):
  plt.subplot(5, 3, int(i)+1)
  sns.boxplot(x=filtered_df[f'{features[i]}'])
  plt.title(f'Box Plot of {features[i]}')


plt.tight_layout()
plt.show()
```



------------------------------ Data Transformation ------------------------------

```
# First, filter the DataFrame to exclude outliers for 'full_sq', 'life_sq', 'max_floor', 'material', 'num_room', 'kitch_sq', and 'state'
# We've already determined the bounds earlier (not shown here due to execution limitations)

# Let's define our numerical and categorical features again
numerical_features = ['full_sq', 'life_sq', 'floor', 'max_floor', 'num_room', 'kitch_sq', 'build_year']  # Assuming these are all numeric
categorical_features = ['product_type', 'state', 'material', 'sub_area']  # We'll one-hot encode this

# Now we apply the preprocessor to the filtered DataFrame
# filtered_df['product_type'] = filtered_df['product_type'].map({'Investment': 0, 'OwnerOccupier': 1})
```

All numeric features stands for a size or abstract material, it can't be negative

```
filtered_df = filtered_df[(filtered_df[numerical_features] >= 0).all(axis=1)]
filtered_df.shape
```

```
    (29343, 11)
```

-------------------------------- Basic Feature Engineering --------------------------------

We enhance the dataset by creating additional features that could provide more insights into property values and their characteristics.

New Features Created

- **Date and Price Initialization**:
  - `timestamp` : Converted to datetime for temporal analysis.
  - `price_doc` : Directly included to facilitate price-based calculations.
- **Calculated Features**:
  - `price_per_sqm` : Calculates the price per square meter, providing a standard measure of value.
  - `living_to_total_area_ratio` : Represents the proportion of living space, useful for assessing space utilization.
  - `area_per_room` : Offers an average area per room, indicating room spaciousness.
  - `rooms_per_sqm` : Gives an idea of room density in relation to property size.
  - `transaction_year` : Extracted from the timestamp for year-based analysis.
  - `age_of_building` : Determines the building's age from the transaction year and the construction year, useful for valuation adjustments.

Purpose

These engineered features are designed to deepen the analysis by introducing metrics that directly relate to property evaluation, market trends, and investment potential.

```
# Adding neccesary features
filtered_df['timestamp'] = pd.to_datetime(train_df['timestamp'])
filtered_df['price_doc'] = train_df['price_doc']

# Feature Engineering
filtered_df['price_per_sqm'] = filtered_df['price_doc']/filtered_df['full_sq']
filtered_df['living_to_total_area_ratio'] = filtered_df['life_sq'] / filtered_df['full_sq']
filtered_df['area_per_room'] = filtered_df['full_sq'] / filtered_df['num_room']

filtered_df['transaction_year'] = filtered_df['timestamp'].dt.year
filtered_df['rooms_per_sqm'] = filtered_df['num_room'] / filtered_df['full_sq']
filtered_df['age_of_building'] = filtered_df['transaction_year'] - filtered_df['build_year']
```

-------------------------------- 4 --------------------------------

Trend and Seasonal Analysis of Sale Prices

Objective

To explore the temporal dynamics in property sale prices, analyzing both long-term trends and seasonal variations.

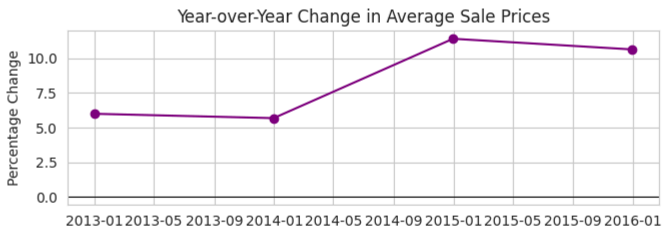
Analysis Techniques

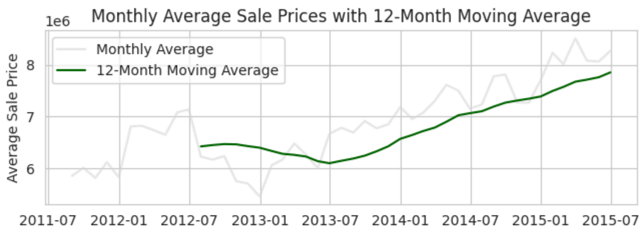
- **Yearly and Monthly Average Prices**: Computed to provide insights into broader trends and finer monthly fluctuations.
- **Yearly Price Changes**: The percentage change year-over-year is calculated to identify significant shifts in market dynamics.
- **Seasonal Decomposition**: Applied to monthly prices to dissect the underlying trend, seasonality, and residual components.

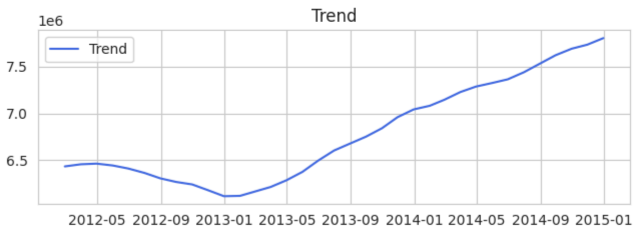
Visualization

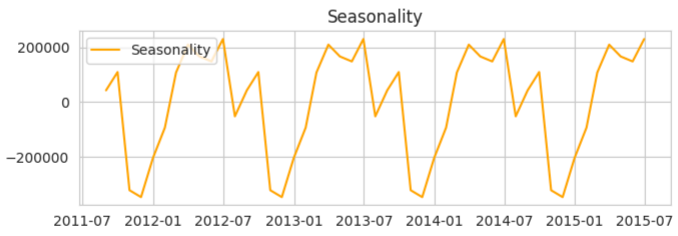
- **Composite Plotting**:
  - **Year-over-Year Change**: Visualized using a line graph to highlight annual variations.
  - **Moving Average**: A 12-month rolling average plot provides a smoothed perspective on monthly price changes.
  - **Decomposition Outputs**: Trend and seasonal plots clarify the persistent pattern and cyclic behavior of prices.

```
# Calculate yearly and monthly average sale prices
yearly_prices = filtered_df.resample('A', on='timestamp')['price_doc'].mean()
monthly_prices = filtered_df.resample('M', on='timestamp')['price_doc'].mean()

# Calculate the yearly price change to quantify trends
yearly_price_change = yearly_prices.pct_change().dropna()

# Perform seasonal decomposition
decomposition = seasonal_decompose(monthly_prices, model='additive', period=12)

# Creating a combined figure for all plots
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(7, 10))

# Year-over-Year Change Plot
axes[0].plot(yearly_price_change.index, yearly_price_change * 100, marker='o', linestyle='-', color='purple')
axes[0].set_title('Year-over-Year Change in Average Sale Prices')
axes[0].set_ylabel('Percentage Change')
axes[0].grid(True)
axes[0].axhline(0, color='black', linewidth=0.8)  # Add a line at 0% change for reference

# Moving Average Plot
axes[1].plot(monthly_prices.index, monthly_prices, linestyle='-', color='lightgray', alpha=0.5, label='Monthly Average')
axes[1].plot(monthly_prices.rolling(window=12).mean(), color='darkgreen', label='12-Month Moving Average')
axes[1].set_title('Monthly Average Sale Prices with 12-Month Moving Average')
axes[1].set_ylabel('Average Sale Price')
axes[1].legend()
axes[1].grid(True)

# Seasonal Decomposition: Trend
axes[2].plot(decomposition.trend, label='Trend', color='royalblue')
axes[2].set_title('Trend')
axes[2].legend(loc='upper left')

# Seasonal Decomposition: Seasonality
axes[3].plot(decomposition.seasonal, label='Seasonality', color='orange')
axes[3].set_title('Seasonality')
axes[3].legend(loc='upper left')

# Adjust layout
plt.tight_layout()

plt.show()
```









a. ---------------------------- Baseline Model ----------------------------

Preparing Data for Model Training

Objective

To ready the dataset for machine learning by transforming and encoding necessary features to ensure optimal input format for modeling.

Feature Transformation

- **Date Handling**: Convert the `timestamp` column to a datetime format, then extract `year` and `month` for more granular time-based analysis.
- **Feature Encoding**:
  - **One-Hot Encoding**: Categorical features are encoded to create binary columns for each category, facilitating their use in predictive modeling.

Implementation Steps

1. **Drop Unneeded Columns**: Remove the `price_per_sqm` feature from the dataset to focus on predicting `price_doc`.
2. **Datetime Conversion**: Transform `timestamp` to datetime object and extract `year` and `month` as separate features.
3. **Feature Encoding**:
   - Initialize and fit `OneHotEncoder` to the categorical features.
   - Create a DataFrame from the encoded data and merge it back with the numeric features.

```
X = filtered_df.drop(columns = ['price_doc', 'price_per_sqm'] , axis=1)  # Features
y = filtered_df['price_doc']  # Target variable


# Convert 'timestamp' column from string to datetime
X['timestamp'] = pd.to_datetime(X['timestamp'])

# Extract year and month as separate columns
X['timestamp_year'] = X['timestamp'].dt.year
X['timestamp_month'] = X['timestamp'].dt.month

# Drop the original 'timestamp' column if it's no longer needed
X = X.drop(columns=['timestamp'])

# Update it to include the newly created year and month columns
numerical_features.append('timestamp_year')
categorical_features.append('timestamp_month')
```

```
# Initializing the OneHotEncoder
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')

# Fit and transform the categorical features
encoded_categorical = encoder.fit_transform(X[categorical_features])

# Create a DataFrame with the encoded features
encoded_df = pd.DataFrame(encoded_categorical, columns=encoder.get_feature_names_out(categorical_features))

# Reset index to ensure concatenation works correctly if indices are not aligned
X = X.reset_index(drop=True)
encoded_df = encoded_df.reset_index(drop=True)

# Drop the original categorical columns from X
X = X.drop(columns=categorical_features)

# Concatenate the encoded categorical features with the rest of the dataset (numeric features already in X)
X = pd.concat([X, encoded_df], axis=1)
# Now, X_final is ready with one-hot encoded categorical features and numeric features, ready for model training
```

```
    /usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be r
      warnings.warn(
```

```
#sanity check
X.isna().sum()[1].sum()
```

```
    0
```

```
encoded_df
```

	product_type_Investment	product_type_OwnerOccupier	state_1.0	state_2.0	state_3.0	state_4.0	material_1.0	material_2.0	material_3.0	material_4.0	...
0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...
1	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...
2	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...
3	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...
4	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...
...	...	...	...	...	...	...	...	...	...	...	...
29338	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	...
29339	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	...
29340	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...
29341	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...
29342	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	...

29343 rows × 170 columns

```
# Step 1: Split into training + validation (85%) and test (15%)
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

# Step 2: Split the remaining data into training (70% of 85% ≈ 59.5% of the total) and validation (15% of the total)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=(15/85), random_state=42)

# Print the sizes of each dataset to verify the splits
print(f"Training set size: {len(X_train)} ({len(X_train)/len(X)*100:.2f}%)")
print(f"Validation set size: {len(X_val)} ({len(X_val)/len(X)*100:.2f}%)")
print(f"Test set size: {len(X_test)} ({len(X_test)/len(X)*100:.2f}%)")
```

```
    Training set size: 20539 (70.00%)
    Validation set size: 4402 (15.00%)
    Test set size: 4402 (15.00%)
```

Model Training and Evaluation

Data Preprocessing

- **Handling Infinite Values**: Replace `inf` and `-inf` in `X_train` and `X_test` with `NaN` to prevent errors during model training.
- **Imputation**: Missing values are imputed using the median value strategy for consistency and to maintain data integrity.

Model Fitting

- **Random Forest Model**: A RandomForestRegressor with 100 trees is fitted to the training data. This model choice is due to its robustness against overfitting and its effectiveness in handling both linear and non-linear relationships.

Prediction and Evaluation

- **Prediction**: The model is used to predict the test set outcomes.
- **Evaluation**: The mean squared error (MSE) is calculated to quantify the model's prediction accuracy. MSE: 7482393445412.973

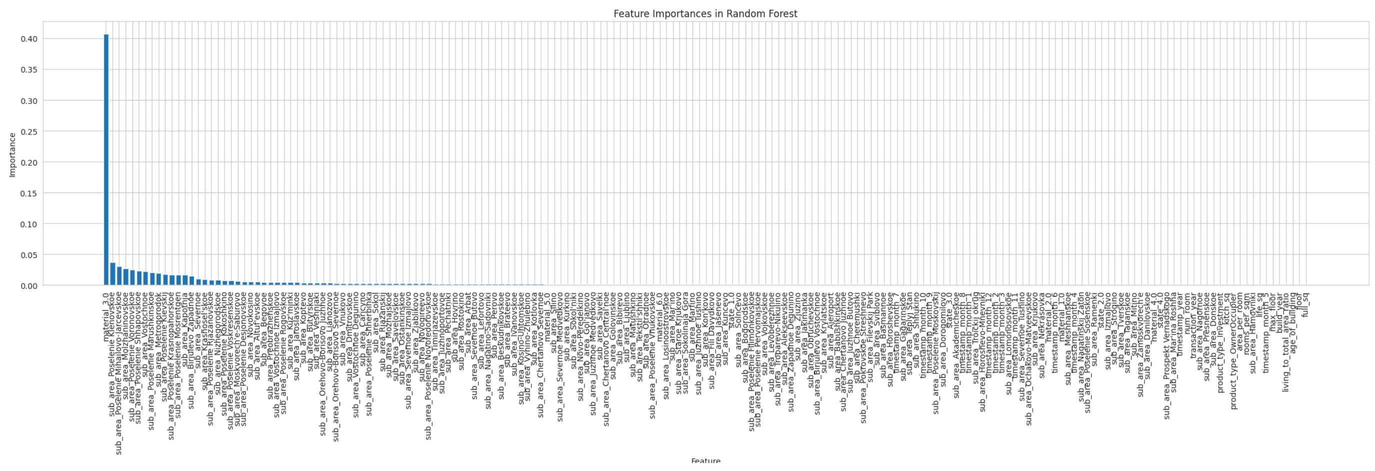
Feature Importance Analysis

- **Importance Calculation**: Feature importances derived from the model provide insights into which variables are most influential in predicting the target variable.
- **Sorting**: Importances are sorted to highlight the most significant features.

```
# Check for inf and -inf values, then replace them with NaN
X_train.replace([np.inf, -np.inf], np.nan, inplace=True)
X_test.replace([np.inf, -np.inf], np.nan, inplace=True)

# Optional: Impute NaN values, here using median but you could use mean or mode as well
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='median')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Now fit the model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predicting and evaluating the model
y_pred = model.predict(X_test)
original_mse = mean_squared_error(y_test, y_pred)
print(f'MSE: {original_mse}')

# Analyzing feature importances
importances = model.feature_importances_
sorted_indices = np.argsort(importances)[::-1]
```

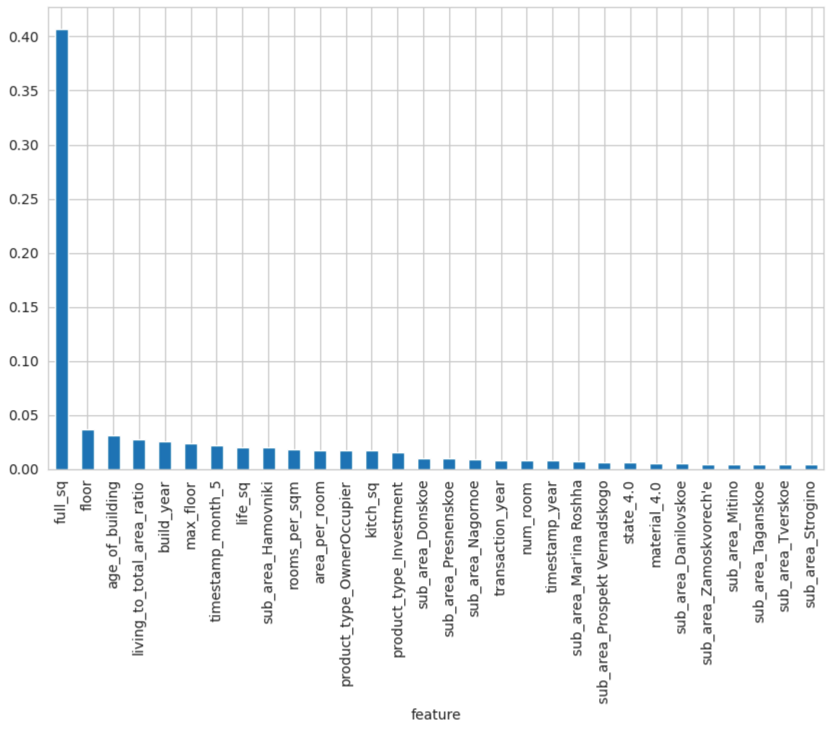
```
    MSE: 7425597700275.702
```

```
sorted_indices = np.argsort(importances)
plt.figure(figsize=(30, 6))
plt.title('Feature Importances in Random Forest')
plt.bar(range(X.shape[1]), importances[sorted_indices][::-1], align='center')
plt.xticks(range(X.shape[1]), [X.columns[i] for i in sorted_indices], rotation=90)
plt.xlabel('Feature')
plt.ylabel('Importance')
plt.show()
```



```
all_fi = pd.DataFrame({'feature': X.columns, 'importance': importances}).sort_values(by='importance', ascending=False)
all_fi[:30].plot(kind='bar', x='feature', y='importance', figsize=(10, 6), legend=False)
```

```
    <Axes: xlabel='feature'>
```



Data Imputation

- **Strategy**: Median imputation applied to handle missing values in both training (`X_train`) and test datasets (`X_test`), ensuring data completeness.

Prediction

- **Execution**: The trained model is used to make predictions on the test set.
- **Evaluation**: The Mean Squared Error (MSE) is calculated to assess the accuracy of the predictions.

Feature Importance Analysis

- **Extraction**: Feature importances are derived from the model to identify which features significantly influence the target variable.
- **Ordering**: Features are ranked according to their importance, aiding in understanding the predictive power of each feature.

```
imputer = SimpleImputer(missing_values=np.nan, strategy='median')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Predicting and evaluating the model
y_pred = model.predict(X_test)
original_mse = mean_squared_error(y_test, y_pred)
print(f'MSE: {original_mse}')

# Analyzing feature importances
importances = model.feature_importances_
sorted_indices = np.argsort(importances)[::-1]
```

```
    MSE: 7425597700275.702
```

Taking only Features above 0.01 importance

```
# Decide on a cutoff for feature importance (e.g., using a threshold or the top N features)
threshold = 0.01
selected_features_indices = sorted_indices[importances[sorted_indices] > threshold]

# Select data with chosen features
X_train_selected = X_train[:, selected_features_indices]
X_test_selected = X_test[:, selected_features_indices]
```

Perform cv after filtering only important features

```
# Perform 5-Fold Cross-Validation
cv_scores = cross_val_score(model, X_train_selected, y_train, cv=5, scoring='neg_mean_squared_error')

# Calculate MSE
cv_mse = -cv_scores.mean()
```

CV_mse: 10340855811818.316

```
cv_mse
```

```
    10402028725793.09
```

Hyperparameter Tuning with XGBoost

Introduction

In this notebook, we perform hyperparameter tuning for an XGBoost regression model using Randomized Search Cross Validation. The goal is to find the optimal combination of hyperparameters that minimizes the mean squared error (MSE) on a given dataset.

Setup

We begin by setting up the XGBoost regressor model and defining the hyperparameters to be tuned.

Evaluation

After the search, we retrieve the best estimator and evaluate its performance on the test data.

The model's performance on the test data is as follows:

Mean Squared Error (MSE): 6384006762126.02

Best Parameters: {'subsample': 1.0, 'n_estimators': 400, 'max_depth': 6, 'learning_rate': 0.1, 'gamma': 0, 'colsample_bytree': 0.3}

```
# Setup the model
xg_reg = xgb.XGBRegressor(objective ='reg:squarederror', seed=42)

# Hyperparameter tuning setup
params = {
    "colsample_bytree": [0.3, 0.7],
    "gamma": [0, 0.1, 0.2],
    "learning_rate": [0.01, 0.1, 0.2],
    "max_depth": [4, 6, 8],
    "n_estimators": [100, 200, 300, 400, 500],
    "subsample": [0.6, 0.8, 1.0]
}

# Randomized search on hyper parameters
random_search = RandomizedSearchCV(xg_reg, param_distributions=params, n_iter=5, scoring='neg_mean_squared_error', n_jobs=-1, cv=5, verbose=3, random_state=42)

# Training the model
random_search.fit(X_train, y_train)

# Best estimator
best_xgb = random_search.best_estimator_

# Predicting and Evaluating
y_pred = best_xgb.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"MSE: {mse:.2f}")
print("Best Parameters:", random_search.best_params_)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752: UserWarning: A worker stopped while some jobs were given to the executor. Th
  warnings.warn(
MSE: 6384006762126.02
Best Parameters: {'subsample': 1.0, 'n_estimators': 400, 'max_depth': 6, 'learning_rate': 0.1, 'gamma': 0, 'colsample_bytree': 0.3}
```

b. Baseline Results

- Discussion of Model Performance

```
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print(f"MSE: {mse:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R^2 Score: {r2:.2f}")
```

```
MSE: 6384006762126.02
RMSE: 2526659.21
R^2 Score: 0.67
```

Discussion

1. **MSE** (Mean Squared Error) MSE is a risk metric that provides the sum of the squared differences between actual and predicted values. It gives a sense of how wrong the model predictions are (errors are squared, thus exaggerating the impact of large errors). An MSE of 6,384,006,762,126.02 is exceptionally high, typically indicating either very large errors in some predictions or high variance in the data values themselves.
2. **RMSE** (Root Mean Squared Error) The RMSE of 2,526,659.21 indicates that typical predictions of the price_doc variable, which represents property prices, deviate from actual values by about 2.57 million units. Considering the average property price (price_doc) in the dataset is 7.5 million, an RMSE of this magnitude (approximately 34% of the mean) suggests a substantial error relative to the values being predicted. This level of error highlights potential limitations in the model's predictive accuracy and suggests a need for further model tuning and feature engineering to improve outcomes, especially in high-stakes financial contexts like real estate.
3. **R^2** Score of 0.67 for predicting property prices (price_doc) indicates that the model explains 67% of the variance in property prices. This suggests moderate predictive accuracy, but with 33% of the variance unexplained, there is significant scope for model improvement to enhance reliability and financial decision-making effectiveness in real estate valuation.

Analysis Summary: Sberbank Russian Housing Dataset Exploration

Methodology

The notebook follows a structured approach to explore the Sberbank Russian Housing Market dataset, primarily focused on understanding the distribution and characteristics of the target variable (`price_doc`) and other features like `floor` and `max_floor`. Here's how the analysis is structured:

Initial Setup

Libraries such as Pandas, Matplotlib, Seaborn, and Sklearn are used for data manipulation, plotting, and modeling. The data is loaded from a CSV file into a Pandas dataframe for analysis.

Exploration of Target Variable (price_doc)

Scatter and distribution plots are used to identify outliers and visualize the distribution. A log transformation is applied to normalize the distribution.

Feature Exploration (floor and max_floor)

Count plots and price trend analyses provide insights into floor-related trends and pricing anomalies. Box plots for the `max_floor` feature further explore pricing dynamics across building heights.

Results

Target Variable (price_doc)

The analysis shows no significant outliers, and the log transformation helps normalize the price distribution. This transformation is critical for models assuming normally distributed inputs.

Floor Analysis

Most buildings have 1 to 17 floors. Prices generally increase with floor level, with specific floors showing price spikes possibly due to unique features.

Max Floor Analysis

Count plots show prevalent building heights, and box plots across max floors reveal varied price ranges, suggesting higher floors might command higher prices.

Conclusions

The notebook sets a foundational understanding of the dataset, indicating the importance of `floor` and `max_floor` as predictors. Suggestions for further analysis include more rigorous outlier handling, extended feature engineering, and additional visualizations such as geographical heatmaps or time-series analysis.

Critique and Improvements

Critique of the Notebook's Approach

1. Limited Preliminary Data Analysis: The approach skips comprehensive preliminary exploration, which might overlook important variables. A thorough initial analysis could set a stronger foundation for detailed insights.

2. Basic Statistical Analyses: Relies mostly on basic statistics and visual explorations, which might not be sufficient to uncover complex patterns in economic data.

3. Assumption of Normality: Log transformation of `price_doc` is used without testing if this assumption benefits the model specifically, potentially affecting predictions.

4. Outlier Management: Outliers are not thoroughly managed, which can significantly skew regression models and affect generalizability.

5. Feature Depth: The exploration is surface-level with respect to potential interactions and multicollinearity between features, missing deeper insights.

Suggested Improvements

1. Comprehensive Data Cleaning and Exploration: Initial cleaning should address missing data and outliers across all features. Tools like `missingno` and z-score methods are recommended.

2. Extended Feature Exploration: Investigate additional features, especially those relating to location and amenities, using PCA and geographical data visualization.

3. Advanced Statistical Testing: Apply tests to confirm assumptions and understand feature relationships, such as the Shapiro-Wilk test for normality.

4. Robust Modeling Approach: Use ensemble methods or stacking different models to enhance predictive performance and robustness.

5. Interaction Terms and Polynomial Features: Creating interaction terms or polynomial features could help model non-linear relationships more effectively using `PolynomialFeatures`.

6. Cross-Validation and Hyperparameter Tuning: Implement cross-validation and systematic parameter tuning with `GridSearchCV` to optimize model settings.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
     Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# ---------------- Imports ----------------
# Data manipulation and analysis
import pandas as pd
import numpy as np
import xgboost as xgb

# Visualization
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns

# Machine learning - Preprocessing
from sklearn.model_selection import train_test_split, cross_val_score, RandomizedSearchCV
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import mean_squared_log_error
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder
from sklearn.impute import SimpleImputer

# Machine learning - Models
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LassoCV
from fancyimpute import KNN

# Machine learning - Metrics
from sklearn.metrics import mean_squared_error

# Time series analysis
from statsmodels.tsa.seasonal import seasonal_decompose


# Upload csv
# df = pd.read_csv('/Users/ronimuradov/Desktop/Docs/סמסטר ה׳/למידת מכונה/פרויקט/sberbank-russian-housing-market/train.csv')
df = pd.read_csv('/content/drive/MyDrive/פרויקט למידת מכונה/train.csv')
test_df = pd.read_csv('/content/drive/MyDrive/פרויקט למידת מכונה/Manipulated_Test.csv')


print(test_df.shape)
print(df.shape)
```

```
     (7662, 563)
     (30471, 292)
```

Data Exploration and Cleaning

The code identifies and quantifies missing values in a DataFrame, calculates their percentages, organizes this information into a sorted DataFrame focusing on features with significant missing data, and displays the results.

```
# Identifying missing values in each feature
missing_values = df.isnull().sum()

# Calculating the percentage of missing values for each feature
missing_percentage = (df.isnull().sum() / len(df)) * 100

# Combining both the count and percentage of missing values into a DataFrame
missing_df = pd.DataFrame({'Missing Values': missing_values, 'Percentage': missing_percentage})

# Sorting the features by percentage of missing values in descending order
missing_df = missing_df[missing_df['Missing Values'] > 0].sort_values(by='Percentage', ascending=False)

missing_df
```

	Missing Values	Percentage
hospital_beds_raion	14441	47.392603
build_year	13605	44.649011
state	13559	44.498047
cafe_sum_500_max_price_avg	13281	43.585704
cafe_sum_500_min_price_avg	13281	43.585704
cafe_avg_price_500	13281	43.585704
max_floor	9572	31.413475
material	9572	31.413475
num_room	9572	31.413475
kitch_sq	9572	31.413475
preschool_quota	6688	21.948738
school_quota	6685	21.938893
cafe_avg_price_1000	6524	21.410521
cafe_sum_1000_max_price_avg	6524	21.410521
cafe_sum_1000_min_price_avg	6524	21.410521
life_sq	6383	20.947786
raion_build_count_with_builddate_info	4991	16.379508
build_count_after_1995	4991	16.379508
build_count_1946-1970	4991	16.379508
build_count_1921-1945	4991	16.379508
build_count_before_1920	4991	16.379508
build_count_1971-1995	4991	16.379508
build_count_mix	4991	16.379508
build_count_monolith	4991	16.379508
raion_build_count_with_material_info	4991	16.379508
build_count_slag	4991	16.379508
build_count_wood	4991	16.379508
build_count_frame	4991	16.379508
build_count_brick	4991	16.379508
build_count_block	4991	16.379508
build_count_panel	4991	16.379508
build_count_foam	4991	16.379508
cafe_avg_price_1500	4199	13.780316
cafe_sum_1500_min_price_avg	4199	13.780316
cafe_sum_1500_max_price_avg	4199	13.780316
cafe_avg_price_2000	1725	5.661120
cafe_sum_2000_min_price_avg	1725	5.661120
cafe_sum_2000_max_price_avg	1725	5.661120
cafe_avg_price_3000	991	3.252273
cafe_sum_3000_max_price_avg	991	3.252273
cafe_sum_3000_min_price_avg	991	3.252273
cafe_sum_5000_max_price_avg	297	0.974697
cafe_sum_5000_min_price_avg	297	0.974697
cafe_avg_price_5000	297	0.974697
prom_part_5000	178	0.584162
floor	167	0.548062
ID_railroad_station_walk	25	0.082045
metro_min_walk	25	0.082045
metro_km_walk	25	0.082045

railroad_station_walk_km	25	0.082045
railroad_station_walk_min	25	0.082045

Next steps: Generate code with missing_df View recommended plots

The script categorizes DataFrame columns into categorical and numeric types by evaluating data types and the uniqueness of values, then identifies and excludes numerically encoded categorical features from the numeric list, displaying the refined lists of categorical and numeric features.

```
# Split cols to categorical and numerical
# Modified to include both string and numeric columns
columns_of_interest = df.select_dtypes(include=['object', 'int64', 'float64']).columns

categorial_features = []

for col in columns_of_interest:
    if df[col].nunique() < 10:
        categorial_features.append(col)

print("categorial_features", categorial_features)

all_numeric_features = df.select_dtypes(include=['int64', 'float64']).columns.tolist()
numeric_features = [feature for feature in all_numeric_features if feature not in categorial_features]

# Exclude categorical features that are numerically encoded from the numeric_features list (not to be in categorial numeric)
numeric_features = [feature for feature in all_numeric_features if feature not in categorial_features]

print("Final numeric features, excluding categorical ones:", numeric_features)
```

```
categorial_features ['material', 'state', 'product_type', 'school_education_centers_top_20_raion', 'healthcare_centers_raion', 'uni
Final numeric features, excluding categorical ones: ['id', 'full_sq', 'life_sq', 'floor', 'max_floor', 'build_year', 'num_room', 'k
```

The code filters outliers from numeric columns in a DataFrame using modified quantile ranges and IQR calculation, removes these outliers, resets indices, and prepares to eliminate negative values, culminating in a display of the cleaned data's statistics.

```
df_numeric = df[numeric_features]

# Outliers detection
features = df_numeric.columns  # Work directly with numeric features

Q1 = df_numeric.quantile(0.02)
Q3 = df_numeric.quantile(0.98)
IQR = Q3 - Q1

# Determining the bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# delete outliers from the data
df = df[~((df_numeric < (Q1 - 1.5 * IQR)) | (df_numeric > (Q3 + 1.5 * IQR))).any(axis=1)].reset_index(drop=True)

#delete all negative values in numeric features
df.describe()
```

	id	full_sq	life_sq	floor	max_floor	materi
count	29889.000000	29889.000000	23651.000000	29724.000000	20424.000000	20424.0000
mean	15177.491652	53.562682	33.670754	7.719890	12.634743	1.8212
std	8788.019919	20.404758	18.278701	5.302509	6.589900	1.4779
min	1.000000	0.000000	0.000000	0.000000	0.000000	1.0000
25%	7557.000000	38.000000	20.000000	3.000000	9.000000	1.0000
50%	15145.000000	49.000000	30.000000	7.000000	12.000000	1.0000
75%	22786.000000	63.000000	43.000000	11.000000	17.000000	2.0000
max	30472.000000	226.000000	191.000000	44.000000	48.000000	6.0000

8 rows × 276 columns

The script identifies and lists columns in a DataFrame based on data types, separately cataloging integer and float columns, and then displays these categorized lists.

```
# Select columns that are integers
int_features = df.select_dtypes(include=['int64']).columns.tolist()

# Select columns that are floats
float_features = df.select_dtypes(include=['float64']).columns.tolist()

# Print the lists
print("Integer features:", int_features)
print("Float features:", float_features)
```

```
Integer features: ['id', 'full_sq', 'raion_popul', 'children_preschool', 'preschool_education_centers_raion', 'children_school', 's
Float features: ['life_sq', 'floor', 'max_floor', 'material', 'build_year', 'num_room', 'kitch_sq', 'state', 'area_m', 'green_zone_
```

The code first prints the initial data types of selected float columns, checks if all non-null values in these columns are integers, and if so, converts them to integer type while preserving NaN values. It then verifies and displays the updated data types to confirm the conversions.

```
# Print initial data types for verification
print("Initial data types (selected):")
print(df[float_features].dtypes)

for col in float_features:
    # Check if all non-NaN entries in the column are integers
    if df[col].dropna().apply(lambda x: x % 1 == 0).all():
        print(f"Converting {col} to integers.")

        # Convert to Int64 dtype while preserving NaNs
        df[col] = df[col].astype('Int64')

# Verify changes by checking data types again
print("\nUpdated data types (selected):")
conversion_results = df[float_features].dtypes
print(conversion_results)
```

```
Initial data types (selected):
life_sq                        float64
floor                          float64
max_floor                      float64
material                       float64
build_year                     float64
                                 ...
green_part_5000                float64
prom_part_5000                 float64
cafe_sum_5000_min_price_avg    float64
cafe_sum_5000_max_price_avg    float64
cafe_avg_price_5000            float64
Length: 119, dtype: object
Converting life_sq to integers.
Converting floor to integers.
Converting max_floor to integers.
Converting material to integers.
Converting build_year to integers.
Converting num_room to integers.
Converting kitch_sq to integers.
Converting state to integers.
Converting preschool_quota to integers.
Converting school_quota to integers.
Converting hospital_beds_raion to integers.
Converting raion_build_count_with_material_info to integers.
Converting build_count_block to integers.
Converting build_count_wood to integers.
Converting build_count_frame to integers.
Converting build_count_brick to integers.
Converting build_count_monolith to integers.
Converting build_count_panel to integers.
Converting build_count_foam to integers.
Converting build_count_slag to integers.
Converting build_count_mix to integers.
Converting raion_build_count_with_builddate_info to integers.
Converting build_count_before_1920 to integers.
Converting build_count_1921-1945 to integers.
Converting build_count_1946-1970 to integers.
Converting build_count_1971-1995 to integers.
Converting build_count_after_1995 to integers.
Converting ID_railroad_station_walk to integers.

Updated data types (selected):
life_sq                          Int64
floor                            Int64
```

```
max_floor                       Int64
material                        Int64
build_year                      Int64
                              ...
green_part_5000               float64
prom_part_5000                float64
cafe_sum_5000_min_price_avg   float64
cafe_sum_5000_max_price_avg   float64
cafe_avg_price_5000           float64
Length: 119, dtype: object
```

The script selects and lists columns from a DataFrame based on their data types, identifying those that are integers and those that are floats, and then prints these lists to provide a clear overview of the data types present in the dataset.

```
# Select columns that are integers
int_features = df.select_dtypes(include=['int64']).columns.tolist()

# Select columns that are floats
float_features = df.select_dtypes(include=['float64']).columns.tolist()

# Print the lists
print("Integer features:", int_features)
print("Float features:", float_features)
```

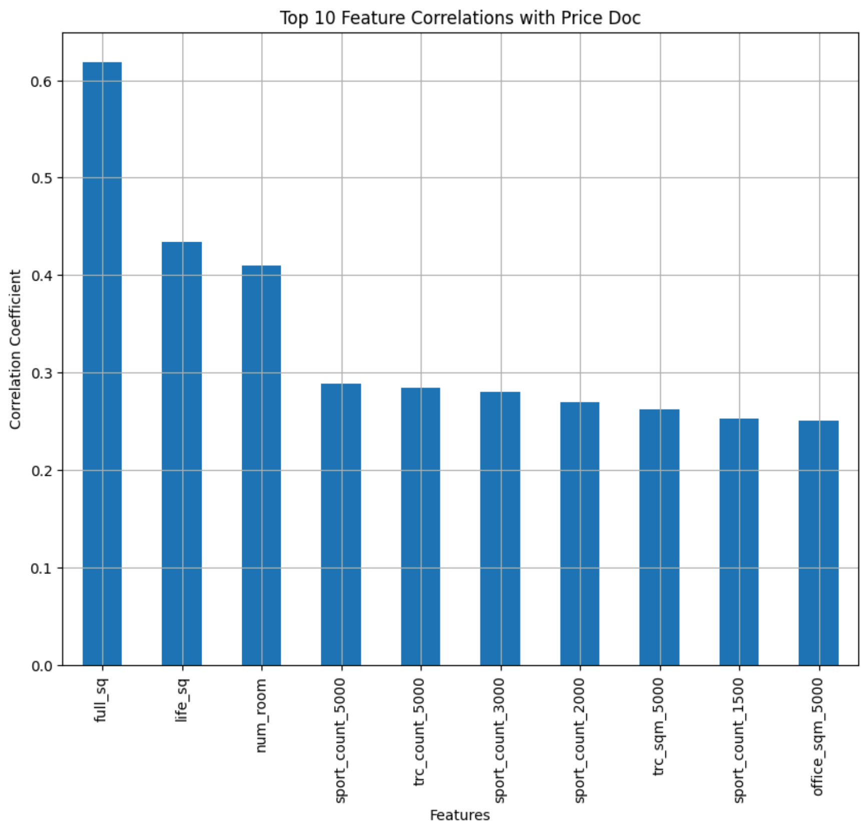
```
Integer features: ['id', 'full_sq', 'life_sq', 'floor', 'max_floor', 'material', 'build_year', 'num_room', 'kitch_sq', 'state', 'ra
Float features: ['area_m', 'green_zone_part', 'indust_part', 'metro_min_avto', 'metro_km_avto', 'metro_min_walk', 'metro_km_walk',
```

The code imputes missing values in integer columns using the median and in float columns using the mean, effectively handling missing data by replacing it with central tendency measures appropriate to each data type.

```
# Impute missing values for integers with median
df[int_features] = df[int_features].fillna(df[int_features].median())

# Impute missing values for floats with mean
df[float_features] = df[float_features].fillna(df[float_features].mean())
```

The code calculates the correlation matrix for numeric columns, isolates correlations with 'price_doc', and visualizes the top 10 strongest correlations in a bar chart to highlight key predictors.

```
numeric_cols = df.select_dtypes(include=['int64', 'float64'])

# Calculate the correlation matrix
corr_matrix = numeric_cols.corr()

# Isolate the correlation coefficients for 'price_doc' and sort them
price_doc_corr = corr_matrix['price_doc'].sort_values(ascending=False)

# Remove 'price_doc' correlation with itself for clarity in the plot
price_doc_corr = price_doc_corr[price_doc_corr.index != 'price_doc']

# Plotting the correlation with 'price_doc', showing only the top 10 features
plt.figure(figsize=(10, 8))
price_doc_corr.head(10).plot(kind='bar')  # Correctly applying head(10) to select top 10 features
plt.title('Top 10 Feature Correlations with Price Doc')
plt.xlabel('Features')
plt.ylabel('Correlation Coefficient')
plt.grid(True)
plt.show()
```



3. **Model Enhancement**

a. **Feature Engineering**: The notebook employs a variety of feature engineering techniques to enhance the dataset's complexity and informativeness for modeling. This includes polynomial feature generation, correlation analysis, interaction features across multiple domains, and conversion of 'timestamp' into separate 'year' and 'month' components. It also uses OneHotEncoder for categorical data including 'sub_area' and selects important features based on their correlation with 'price_doc'. These methods uncover non-linear relationships, prioritize impactful predictors, and transform categorical data into a model-friendly format.

b. **Enhanced Model Development**: The development process incorporates preprocessing steps such as outlier removal, missing value imputation, data type conversions, and the normalization of interaction terms using MinMaxScaler. Additionally, it includes thorough treatment of categorical data via one-hot encoding, correlation analysis to identify and visualize key features, and the strategic selection of significant features verified against the test dataset. These efforts aim to construct a robust model that effectively captures underlying patterns, enhancing predictive accuracy and ensuring the model is relevant and optimized for performance during training.

The code generates polynomial features by squaring the values of specific columns (full_sq, life_sq, num_room) in the DataFrame to capture non-linear relationships.

```
# Polynomial Features
df['full_sq ** 2'] = df['full_sq'] ** 2
df['life_sq ** 2'] = df['life_sq'] ** 2
df['num_room ** 2'] = df['num_room'] ** 2
```

```
<ipython-input-185-c866c34ee94f>:2: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fra
  df['full_sq ** 2'] = df['full_sq'] ** 2
<ipython-input-185-c866c34ee94f>:3: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fra
  df['life_sq ** 2'] = df['life_sq'] ** 2
<ipython-input-185-c866c34ee94f>:4: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fra
  df['num_room ** 2'] = df['num_room'] ** 2
```

The code creates interaction features by multiplying relevant variables and derives additional features from date information.

```
# Interaction Terms
# Transport-related Interactions
df['railroad_station_avto_km_x_railroad_station_avto_min'] = df['railroad_station_avto_km'] * df['railroad_station_avto_min']

# Distance-related Interactions
df['mkad_km_x_ttk_km'] = df['mkad_km'] * df['ttk_km']
df['kremlin_km_x_big_road1_km'] = df['kremlin_km'] * df['big_road1_km']

# Healthcare-related Interactions
df['public_healthcare_km_x_hospice_morgue_km'] = df['public_healthcare_km'] * df['hospice_morgue_km']
df['university_km_x_public_healthcare_km'] = df['university_km'] * df['public_healthcare_km']

# Cultural-related Interactions
df['theater_km_x_museum_km'] = df['theater_km'] * df['museum_km']
df['church_synagogue_km_x_mosque_km'] = df['church_synagogue_km'] * df['mosque_km']

# Commercial-related Interactions
df['shopping_centers_km_x_office_km'] = df['shopping_centers_km'] * df['office_km']
df['market_shop_km_x_office_count_500'] = df['market_shop_km'] * df['office_count_500']

# Recreational-related Interactions
df['fitness_km_x_swim_pool_km'] = df['fitness_km'] * df['swim_pool_km']
df['ice_rink_km_x_stadium_km'] = df['ice_rink_km'] * df['stadium_km']

# Feature Engineering
df['living_to_total_area_ratio'] = df['life_sq'] / df['full_sq']
df['area_per_room'] = df['full_sq'] / df['num_room']

df['timestamp'] = pd.to_datetime(df['timestamp'])

df['transaction_year'] = df['timestamp'].dt.year
df['rooms_per_sqm'] = df['num_room'] / df['full_sq']
df['age_of_building'] = df['transaction_year'] - df['build_year']
```

```
<ipython-input-186-fe01ade359e2>:3: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fra
  df['railroad_station_avto_km_x_railroad_station_avto_min'] = df['railroad_station_avto_km'] * df['railroad_station_avto_min']
<ipython-input-186-fe01ade359e2>:6: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fra
  df['mkad_km_x_ttk_km'] = df['mkad_km'] * df['ttk_km']
<ipython-input-186-fe01ade359e2>:7: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fra
  df['kremlin_km_x_big_road1_km'] = df['kremlin_km'] * df['big_road1_km']
<ipython-input-186-fe01ade359e2>:10: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['public_healthcare_km_x_hospice_morgue_km'] = df['public_healthcare_km'] * df['hospice_morgue_km']
<ipython-input-186-fe01ade359e2>:11: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['university_km_x_public_healthcare_km'] = df['university_km'] * df['public_healthcare_km']
<ipython-input-186-fe01ade359e2>:14: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['theater_km_x_museum_km'] = df['theater_km'] * df['museum_km']
<ipython-input-186-fe01ade359e2>:15: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['church_synagogue_km_x_mosque_km'] = df['church_synagogue_km'] * df['mosque_km']
<ipython-input-186-fe01ade359e2>:18: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['shopping_centers_km_x_office_km'] = df['shopping_centers_km'] * df['office_km']
<ipython-input-186-fe01ade359e2>:19: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['market_shop_km_x_office_count_500'] = df['market_shop_km'] * df['office_count_500']
<ipython-input-186-fe01ade359e2>:22: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['fitness_km_x_swim_pool_km'] = df['fitness_km'] * df['swim_pool_km']
<ipython-input-186-fe01ade359e2>:23: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['ice_rink_km_x_stadium_km'] = df['ice_rink_km'] * df['stadium_km']
<ipython-input-186-fe01ade359e2>:26: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['living_to_total_area_ratio'] = df['life_sq'] / df['full_sq']
<ipython-input-186-fe01ade359e2>:27: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['area_per_room'] = df['full_sq'] / df['num_room']
<ipython-input-186-fe01ade359e2>:31: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['transaction_year'] = df['timestamp'].dt.year
<ipython-input-186-fe01ade359e2>:32: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['rooms_per_sqm'] = df['num_room'] / df['full_sq']
<ipython-input-186-fe01ade359e2>:33: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `fr
  df['age_of_building'] = df['transaction_year'] - df['build_year']
```

The code selects a list of interaction terms, scales them using MinMaxScaler, and replaces the original columns in the DataFrame with the scaled versions, ensuring they fall within the range [0, 1].

```
# List of interaction terms
interaction_columns = [
    'railroad_station_avto_km_x_railroad_station_avto_min',
    'mkad_km_x_ttk_km',
    'kremlin_km_x_big_road1_km',
    'public_healthcare_km_x_hospice_morgue_km',
    'university_km_x_public_healthcare_km',
    'theater_km_x_museum_km',
    'church_synagogue_km_x_mosque_km',
    'shopping_centers_km_x_office_km',
    'market_shop_km_x_office_count_500',
    'fitness_km_x_swim_pool_km',
    'ice_rink_km_x_stadium_km'
]

# Selecting the interaction terms
df_interactions = df[interaction_columns]

# Initialize MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the interaction terms
df_interactions_scaled = scaler.fit_transform(df_interactions)

# Replace the original interaction columns in the DataFrame
df[interaction_columns] = df_interactions_scaled

# Now df contains the scaled version of your interaction terms within the range [0, 1]
df[interaction_columns].head()
```

	railroad_station_avto_km_x_railroad_station_avto_min	mkad_km_x_ttk_km	kremlin_ki
0	0.052589	0.016613	
1	0.023944	0.031577	
2	0.003053	0.017553	
3	0.028261	0.041877	
4	0.004834	0.021403	

The code converts the 'timestamp' column to datetime format, extracts 'year' and 'month' as separate columns, and then drops the original 'timestamp' column from the DataFrame.

```
# timestamp to datetime
# df['timestamp'] = pd.to_datetime(df['timestamp'])
# add the year and month columns
df['year'] = df['timestamp'].dt.year
df['month'] = df['timestamp'].dt.month
# drop timestamp
df.drop('timestamp', axis=1, inplace=True)
```

```
<ipython-input-188-33c0d80992e8>:4: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `frai
  df['year'] = df['timestamp'].dt.year
<ipython-input-188-33c0d80992e8>:5: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `frai
  df['month'] = df['timestamp'].dt.month
```

The code appends the 'sub_area' column to the list of categorical features, encodes these features using OneHotEncoder, creates a DataFrame with the encoded features, and concatenates them with the existing DataFrame, dropping the original categorical columns. The resulting DataFrame is now ready for model training with both one-hot encoded categorical features and numeric features.

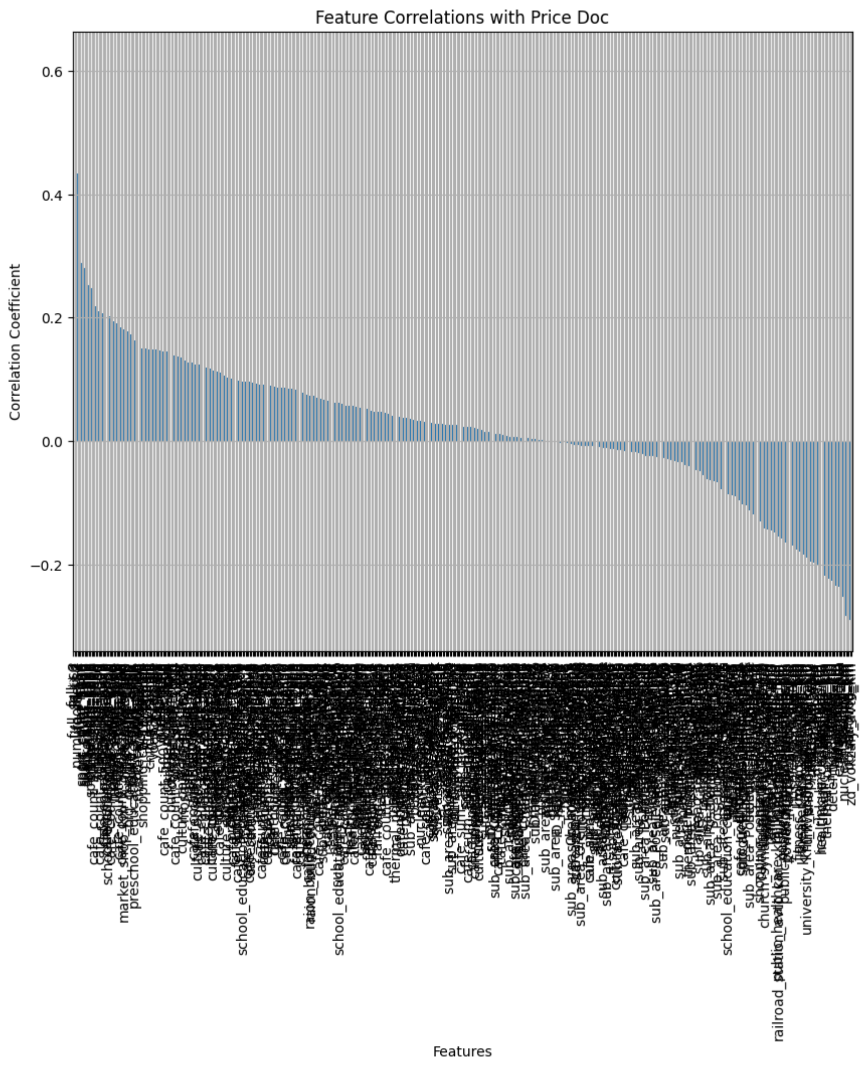
```
categorial_features.append('sub_area')
# Initializing the OneHotEncoder
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')

# Fit and transform the categorical features
encoded_categorical = encoder.fit_transform(df[categorial_features])

# Create a DataFrame with the encoded features
encoded_df = pd.DataFrame(encoded_categorical, columns=encoder.get_feature_names_out(categorial_features))

# Reset index to ensure concatenation works correctly if indices are not aligned
df = df.reset_index(drop=True)
encoded_df = encoded_df.reset_index(drop=True)

# Drop the original categorical columns from X
df = df.drop(columns=categorial_features)

# Concatenate the encoded categorical features with the rest of the dataset (numeric features already in X)
df = pd.concat([df, encoded_df], axis=1)
# Now, X_final is ready with one-hot encoded categorical features and numeric features, ready for model training
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_outp
  warnings.warn(
```

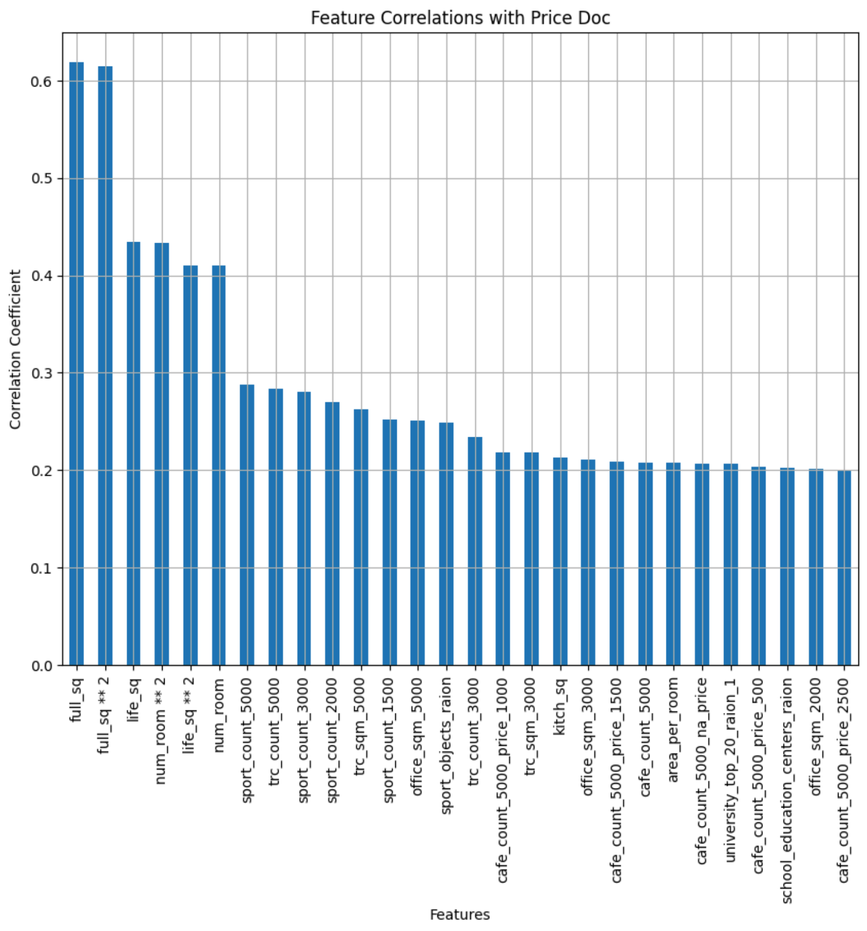
Double-click (or enter) to edit

The code calculates the correlation matrix for all features in the DataFrame, isolates the correlation coefficients for the target variable 'price_doc', and plots the feature correlations with 'price_doc'. It then filters the DataFrame to keep only features with a correlation coefficient greater than 0.2, ensuring only important features are retained for further analysis.

```
# Calculate the correlation matrix
corr_matrix = df.corr()

# Isolate the correlation coefficients for 'price_doc' and sort them
price_doc_corr = corr_matrix['price_doc'].sort_values(ascending=False)

# Remove 'price_doc' correlation with itself for clarity in the plot
price_doc_corr = price_doc_corr[price_doc_corr.index != 'price_doc']

# Plotting the correlation with 'price_doc
plt.figure(figsize=(10, 8))
price_doc_corr.plot(kind='bar')
plt.title('Feature Correlations with Price Doc')
plt.xlabel('Features')
plt.ylabel('Correlation Coefficient')
plt.grid(True)
plt.show()

# Filter the DataFrame to keep only features with a correlation coefficient > 0
important_features = price_doc_corr[price_doc_corr > 0.2].index.tolist()
df = df[important_features + ['price_doc']]
```



```
# Calculate the correlation matrix
corr_matrix = df.corr()

# Isolate the correlation coefficients for 'price_doc' and sort them
price_doc_corr = corr_matrix['price_doc'].sort_values(ascending=False)

# Remove 'price_doc' correlation with itself for clarity in the plot
price_doc_corr = price_doc_corr[price_doc_corr.index != 'price_doc']

# Plotting the correlation with 'price_doc
plt.figure(figsize=(10, 8))
price_doc_corr.plot(kind='bar')
plt.title('Feature Correlations with Price Doc')
plt.xlabel('Features')
plt.ylabel('Correlation Coefficient')
plt.grid(True)
plt.show()
```



```
df.replace([np.inf, -np.inf], np.nan, inplace=True)
```

The code filters the list of important features to include only those that exist in the test dataset. It then subsets the test DataFrame to include only these existing important features, ensuring consistency between the features used in the training and testing phases of model development.

```
# Filter the list to include only existing columns in test_df
existing_important_features = [feature for feature in important_features if feature in test_df.columns]

# Subset the DataFrame to only include the existing important features
test_df = test_df[existing_important_features]
```

The code iterates through columns in both the training (df) and test (test_df) DataFrames, removing any columns that are present in one DataFrame but not the other, except for the 'price_doc' column. This ensures that both datasets have the same columns, facilitating consistent model training and evaluation.

```
# Remove extra columns from df
for column in df.columns:
    if column not in test_df.columns and column != 'price_doc':
         df.drop(columns=[column], inplace=True)

# Remove extra columns from test_df
for column in test_df.columns:
    if column not in df.columns:
        test_df.drop(columns=[column], inplace=True)
```

```
print("Columns in X_train not in X_test:", set(df.columns) - set(test_df.columns))
print("Columns in X_test not in X_train:", set(test_df.columns) - set(df.columns))
```

```
Columns in X_train not in X_test: {'price_doc'}
Columns in X_test not in X_train: set()
```

```
print(df.shape)
print(test_df.shape)
```

```
(29889, 29)
(7662, 28)
```

```
# # Export predictions as CSV

# from google.colab import files

# df.to_csv('New_Train.csv', index=False)

# files.download('New_Train.csv')
```

```
# # Export predictions as CSV

# from google.colab import files

# test_df.to_csv('New_Test.csv', index=False)

# files.download('New_Test.csv')
```

4. **Advanced Modeling**

a. Model Implementatio

1. XGBoost model
2. LGB model - The Best

```
# import xgboost as xgb
# from sklearn.model_selection import train_test_split, GridSearchCV
# from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
# import numpy as np

# # Data splitting
# X = df.drop('price_doc', axis=1)
# y = np.log1p(df['price_doc'])
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# # Model initialization
# xg_reg = xgb.XGBRegressor(objective='reg:squarederror', seed=42)

# # Parameter grid
# params = {
#     'max_depth': [3, 5, 7],
#     'learning_rate': [0.01, 0.1, 0.2],
#     'n_estimators': [100, 200, 300],
#     'colsample_bytree': [0.3, 0.7]
# }

# # Hyperparameter tuning
# grid = GridSearchCV(xg_reg, params, verbose=1, cv=3, n_jobs=-1)
# grid.fit(X_train, y_train)

# # Best parameters and model
# print("Best parameters found: ", grid.best_params_)
# best_xgb_model = grid.best_estimator_

# # Evaluation
# y_pred_xgb = best_xgb_model.predict(X_test)
# y_pred = np.expm1(y_pred_xgb)
# Regression Metrics
def regression_metrics(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)
    rmse = np.sqrt(mse)
    return mse, mae, r2, rmse


# # Calculate regression metrics
# mse, mae, r2, rmse = regression_metrics(y_test, y_pred_xgb)

# # Print the metrics
# print("Mean Squared Error:", mse)
# print("Mean Absolute Error:", mae)
# print("R-squared:", r2)
# print("Root Mean Squared Error:", rmse)
```

XGBoost Results:

Mean Squared Error: 0.20025134487084675

Mean Absolute Error: 0.2702928064332375

R-squared: 0.4112069725958637

Root Mean Squared Error: 0.4474945193752062

LightGBM, a gradient boosting is renowned for its speed, efficiency, and high accuracy. It works by constructing decision trees in a leaf-wise manner, prioritizing the leaf nodes that lead to larger reductions in loss. This approach results in faster training times and lower memory usage compared to other gradient boosting implementations, making it well-suited for handling large datasets and achieving state-of-the-art performance in various regression and classification tasks.

The provided code snippet demonstrates the development and tuning of a LightGBM regression model for predicting house prices. It involves:

Data Preparation: Separating the target variable and transforming it for normalization. Model Initialization: Initializing LightGBM as a regression model. Hyperparameter Tuning: Employing GridSearchCV to find the best combination of hyperparameters for optimal model performance. Model Training: Fitting the GridSearchCV object to the training data to identify the best model. Model Evaluation: Evaluating the best model's performance on the test set by making predictions and transforming them back to the original scale.

```
# import lightgbm as lgb
# from sklearn.model_selection import GridSearchCV

# X = df.drop('price_doc', axis=1)
# y = np.log1p(df['price_doc'])
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# # Model initialization with LightGBM
# lgb_reg = lgb.LGBMRegressor(objective='regression', random_state=42)

# # Parameter grid definition for hyperparameter tuning
# params = {
#     "colsample_bytree": [0.25, 0.3, 0.35],
#     "learning_rate": [0.03, 0.05, 0.07],
#     "max_depth": [4, 5, 6],
#     "n_estimators": [325, 350, 375],
#     "subsample": [0.1, 0.2, 0.3]
# }

# # Setting up GridSearchCV for hyperparameter tuning
# grid = GridSearchCV(lgb_reg, params, scoring='neg_mean_squared_error', n_jobs=-1, cv=5, verbose=3)

# # Fitting the model to the training data
# grid.fit(X_train, y_train)

# # Outputting the best parameters and the best model from GridSearch
# print("Best parameters found: ", grid.best_params_)
# best_lgb_model = grid.best_estimator_

# # Making predictions on the test set using the best model
# y_pred_lgb = best_lgb_model.predict(X_test)
# # Transform predictions back to the original price scale
# y_pred = np.expm1(y_pred_lgb)
# # Calculate regression metrics
# mse, mae, r2, rmse = regression_metrics(y_test, y_pred_lgb)

# # Print the metrics
# print("Mean Squared Error:", mse)
# print("Mean Absolute Error:", mae)
# print("R-squared:", r2)
# print("Root Mean Squared Error:", rmse)
```

LGB Results:

Mean Squared Error: 0.19959599724730162

Mean Absolute Error: 0.2704067489087203

R-squared: 0.4131338715713392

Root Mean Squared Error: 0.44676167835581154

Double-click (or enter) to edit

b. Evaluation and Comparison

Comparison of LGB and XGBoost Results:

- **LightGBM (LGB) Results:**
  - Mean Squared Error: 0.19959599724730162
  - Mean Absolute Error: 0.2704067489087203
  - R-squared: 0.4131338715713392
  - Root Mean Squared Error: 0.44676167835581154
- **XGBoost Results:**
  - Mean Squared Error: 0.20025134487084675
  - Mean Absolute Error: 0.2702928064332375
  - R-squared: 0.4112069725958637
  - Root Mean Squared Error: 0.4474945193752062

Reasons Why LGB Might Perform Better:

- **Efficiency in Training:** LightGBM is known for its faster training speed.
- **Handling of Large Datasets:** LightGBM efficiently handles large-scale datasets.
- **Optimized Tree Growth:** LightGBM uses a leaf-wise tree growth strategy.
- **Regularization Techniques:** LightGBM provides various regularization parameters.
- **Handling of Imbalanced Data:** LightGBM offers support for handling imbalanced datasets.
- **Hyperparameter Tuning:** LightGBM provides a wide range of hyperparameters for optimization.

```
# Train df & Test df

import lightgbm as lgb
from sklearn.model_selection import GridSearchCV

# Assuming 'df' and 'test_df' are already defined and preprocessed appropriately
X_train = df.drop('price_doc', axis=1)
y_train = np.log1p(df['price_doc'])  # Transforming the target variable for normalization
X_test = test_df

# Model initialization with LightGBM
lgb_reg = lgb.LGBMRegressor(objective='regression', random_state=42)

# Parameter grid definition for hyperparameter tuning
params = {
    "colsample_bytree": [0.25, 0.3, 0.35],
    "learning_rate": [0.03, 0.05, 0.07],
    "max_depth": [4, 5, 6],
    "n_estimators": [325, 350, 375],
    "subsample": [0.1, 0.2, 0.3]
}

# Setting up GridSearchCV for hyperparameter tuning
grid = GridSearchCV(lgb_reg, params, scoring='neg_mean_squared_error', n_jobs=-1, cv=5, verbose=3)

# Fitting the model to the training data
grid.fit(X_train, y_train)

# Outputting the best parameters and the best model from GridSearch
print("Best parameters found: ", grid.best_params_)
best_lgb_model = grid.best_estimator_

# Making predictions on the test set using the best model
y_pred_lgb = best_lgb_model.predict(X_test)
# Transform predictions back to the original price scale
y_pred_test = np.expm1(y_pred_lgb)

# Best parameters found:  {'colsample_bytree': 0.3, 'learning_rate': 0.1, 'max_depth': 4, 'n_estimators': 300, 'subsample': 0.6}
# Best parameters found:  {'colsample_bytree': 0.3, 'learning_rate': 0.1, 'max_depth': 4, 'n_estimators': 300, 'subsample': 0.4}
# Best parameters found:  {'colsample_bytree': 0.3, 'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 350, 'subsample': 0.2}
```

```
     [LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```
# Export predictions as CSV

from google.colab import files

y_pred_df = pd.DataFrame(y_pred_test, columns=['price_doc'], index=(X_test.index + 30474).rename('id'))
y_pred_df.to_csv('predicted_price_doc_LGB4.csv', index=True)

files.download('predicted_price_doc_LGB4.csv')
```

```
y_pred_df.shape
```

```
     (7662, 1)
```

Kaggle Competition Winning Techniques

1. State-of-the-Art Techniques

a. Technique Exploration:

The winners of this Kaggle competition utilized a range of state-of-the-art techniques that clearly differentiated their approach from competitors, ultimately leading to their success. Here’s an exploration of the advanced techniques they used, along with insights into how these were applied and their impact on model performance and ranking:

Ensemble Methods:

Ensemble methods combine multiple machine learning models to produce a single optimal predictive model. They often leverage diverse model architectures or configurations to reduce variance, bias, or improve predictions.

Application:

The winners used an ensemble of different models tailored for specific segments of the data (Investment and OwnerOccupier product types). Each segment had models fine-tuned for its unique characteristics, which were then combined to enhance the overall prediction accuracy.

Cross-Validation Techniques:

Cross-validation is a technique used to assess how the results of a statistical analysis will generalize to an independent data set. It is mainly used to flag problems like overfitting or selection bias and to give insights on how the model will perform on an independent dataset.

Application:

The team used a 5-fold cross-validation strategy with random shuffling, particularly effective for the OwnerOccupier product type. This approach helped them in refining model parameters and selecting features that genuinely contributed to model performance.

Feature Engineering:

Feature engineering is the process of using domain knowledge to select, modify, or create new features from raw data, which improves the performance of machine learning models.

Application:

Critical to their success, the team removed unreliable features (like `full_sq` which had data inaccuracies) and engineered new ones that better captured the underlying patterns in the data, such as ratios of `full_sq` to categorical group means.

Probing Techniques:
Probing techniques involve making strategic submissions to gain insights about the characteristics of the test set or to uncover hidden parameters.

Application:

The team used probing submissions to understand and correct for discrepancies in the mean predictions between their models and the public test set. This allowed them to adjust their models dynamically, aligning them more closely with the expected test distribution.

Data Cleaning:
Data cleaning involves removing or correcting inaccurate, corrupt, or irrelevant records from a dataset, which can significantly influence the performance of machine learning models.

Application:

The competitors undertook extensive data cleaning to remove outliers and incorrect data points, particularly in features that were prone to errors such as apartment sizes and build years.

b. Application and Performance:

The application of these advanced techniques had a significant impact on both the model performance and the team's ranking in the competition:

- Impact on Model Performance: The combination of ensemble modeling and robust cross-validation significantly enhanced the reliability and accuracy of their predictions. Feature engineering, particularly the innovative use of normalized space features, allowed the models to better capture essential influences on property prices without being misled by data errors. Probing and data cleaning ensured that the models were not only theoretically sound but also practically effective, closely aligned with the realities of the test dataset.

- Impact on Ranking: These techniques helped the team initially secure a leading position with their private leaderboard score. Although they were third in the public leaderboard, the reliability and robustness of their approach shone through in the final evaluation, securing them the first place. Their strategic use of probing to refine model estimates and the ensemble approach that combined strengths across different model types were particularly effective in a competition setup where both public and private leaderboard rankings could differ significantly due to overfitting on public data.