In [2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.impute import KNNImputer
from sklearn.neighbors import NearestNeighbors
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
```

In [3]:
```
data = pd.read_csv('data_2.csv')
data = data.dropna(subset=['SquareFootageHouse'])
data['Location'] = data['Location'].replace('Suburbann', 'Suburban')
data['HeatingType'].replace({'Oil': 'Oil Heating', 'Electric': 'Electricity
data['Bedrooms'] = data['Bedrooms'].astype(pd.Int64Dtype())
data['Bathrooms'] = data['Bathrooms'].astype(pd.Int64Dtype())
```

Mapping of certain non numeric columns but can be represented as categorical

In [4]:
```
mapping_location = {'Urban': 1, 'Suburban': 2, 'Rural': 3}
data['Location'] = data['Location'].map(mapping_location)

mapping_PoolQuality = {'Good': 2, 'Poor': 3, 'Excellent': 1}
data['PoolQuality'] = data['PoolQuality'].map(mapping_PoolQuality)


mapping_Kitchen = {'Good': 2, 'Poor': 3, 'Excellent': 1}
data['KitchensQuality'] = data['KitchensQuality'].map(mapping_Kitchen)

mapping_Bathrooms = {'Good': 2, 'Poor': 3, 'Excellent': 1}
data['BathroomsQuality'] = data['BathroomsQuality'].map(mapping_Bathrooms)

mapping_Bedrooms = {'Good': 2, 'Poor': 3, 'Excellent': 1}
data['BedroomsQuality'] = data['BedroomsQuality'].map(mapping_Bedrooms)


mapping_LivingRooms = {'Good': 2, 'Poor': 3, 'Excellent': 1}
data['LivingRoomsQuality'] = data['LivingRoomsQuality'].map(mapping_LivingR
```

In [5]:
```
data['Location'] = data['Location'].astype(pd.Int64Dtype())
```

Feature exploration

In [6]:
```
# Define the function to print value counts for a specified column
def print_column_info(column_name):
    if column_name in data.columns:
        value_counts = data[column_name].value_counts(dropna=False)

        print(value_counts)


#print_column_info('Location')
#print_column_info('PoolQuality')
#print_column_info('HasPhotovoltaics')
#print_column_info('Age')

# 0 means the house has no pool
data['PoolQuality'] = data['PoolQuality'].fillna(0)

print_column_info('HeatingType')
```

```
HeatingType
Oil Heating    414
Electricity    299
Gas            282
Name: count, dtype: int64
```

In [7]:
```
data.drop(columns=['HouseColor', 'PreviousOwnerName'], axis=1, inplace=True
data.isna().sum()
```

Out[7]:
```
Bedrooms               328
Bathrooms              165
SquareFootageHouse       0
Location               230
Age                    130
PoolQuality              0
HasPhotovoltaics       258
HeatingType              0
HasFiberglass            0
IsFurnished              0
DateSinceForSale         0
HasFireplace             0
KitchensQuality          0
BathroomsQuality         0
BedroomsQuality          0
LivingRoomsQuality       0
SquareFootageGarden      0
PreviousOwnerRating      0
HeatingCosts           452
WindowModelNames         0
Price                    0
dtype: int64
```

In [8]:
```
print(data.columns)
data.head(1)
```

```
Index(['Bedrooms', 'Bathrooms', 'SquareFootageHouse', 'Location', 'Age',
       'PoolQuality', 'HasPhotovoltaics', 'HeatingType', 'HasFiberglass',
       'IsFurnished', 'DateSinceForSale', 'HasFireplace', 'KitchensQualit
y',
       'BathroomsQuality', 'BedroomsQuality', 'LivingRoomsQuality',
       'SquareFootageGarden', 'PreviousOwnerRating', 'HeatingCosts',
       'WindowModelNames', 'Price'],
      dtype='object')
```

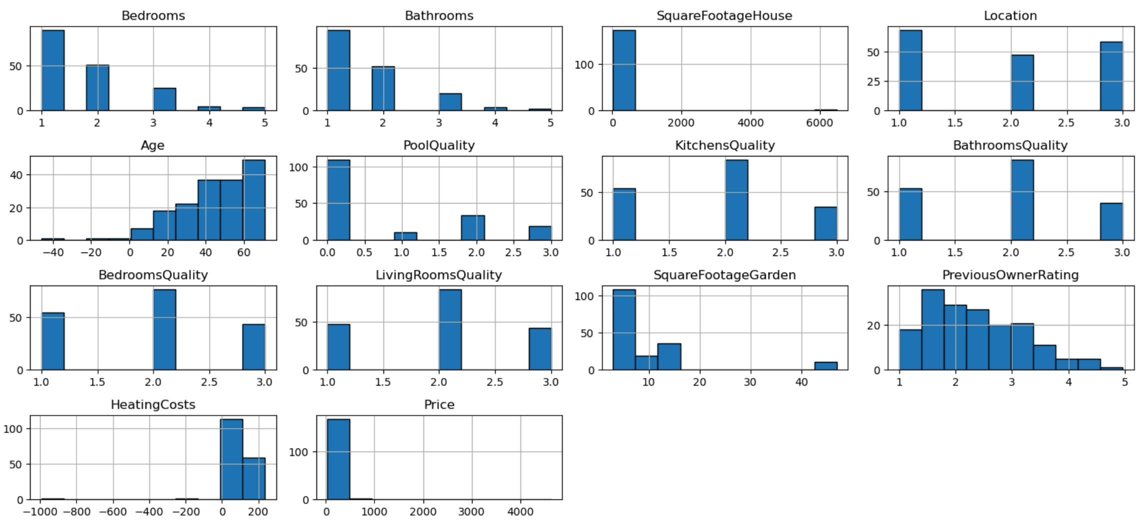
Out[8]:

	Bedrooms	Bathrooms	SquareFootageHouse	Location	Age	PoolQuality	HasPhotovoltaic
0	1	1	35.0	<NA>	69.0	0.0	NaN

1 rows × 21 columns

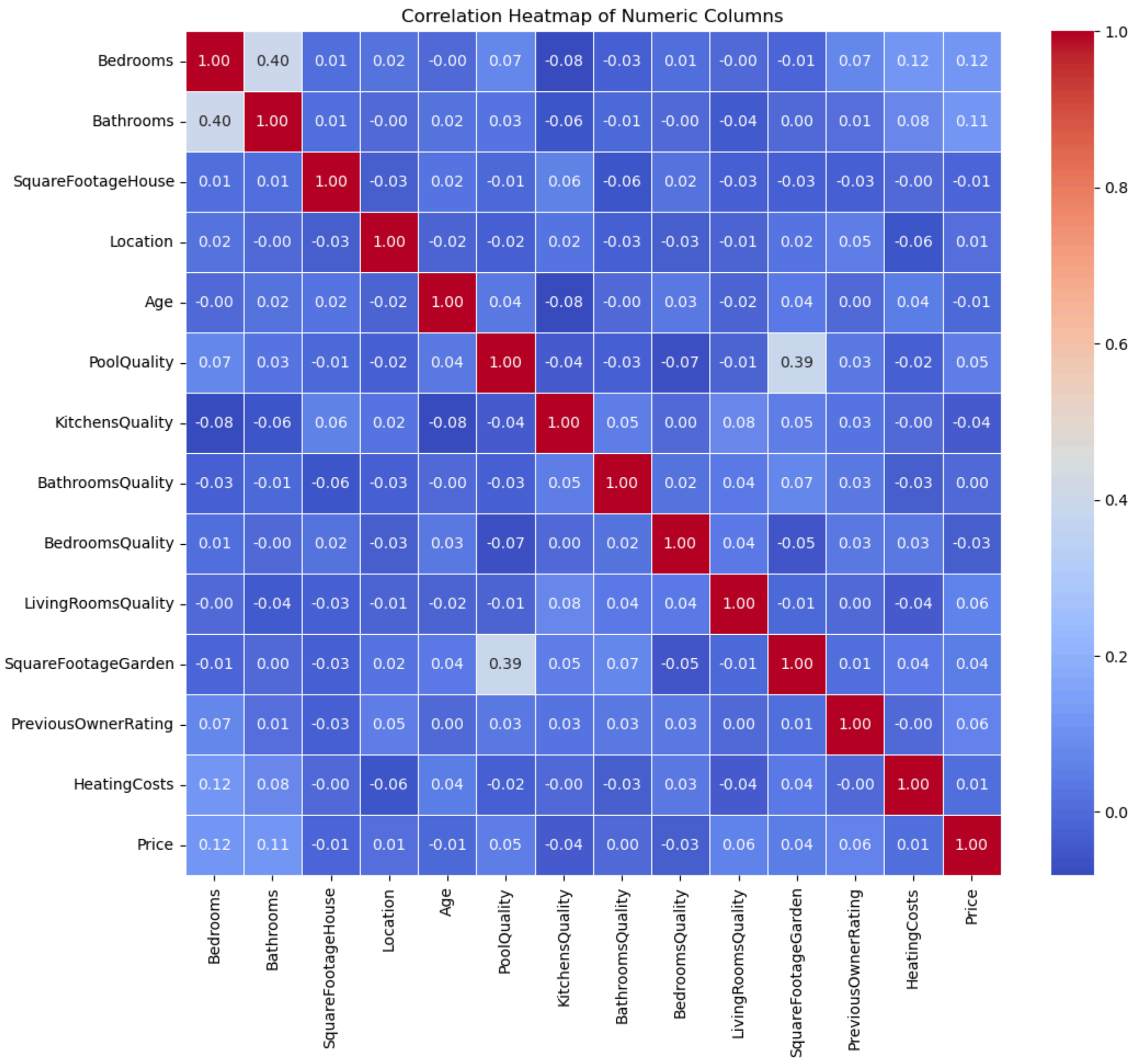
Plotting of numeric columns but ignoring NaN

In [9]:
```
# Drop rows with NaN values to avoid issues in plotting
data_cleaned = data.dropna()

# Determine the number of columns for the layout
num_columns = 4  # For example, to create a 5-column layout
num_rows = -(-len(data_cleaned.columns) // num_columns)  # Calculate rows n

# Plot histograms for each column
data_cleaned.hist(bins=10, figsize=(15, 10), layout=(num_rows, num_columns)

# Adjust layout to prevent overlap
plt.tight_layout()

# Show plot
plt.show()
```



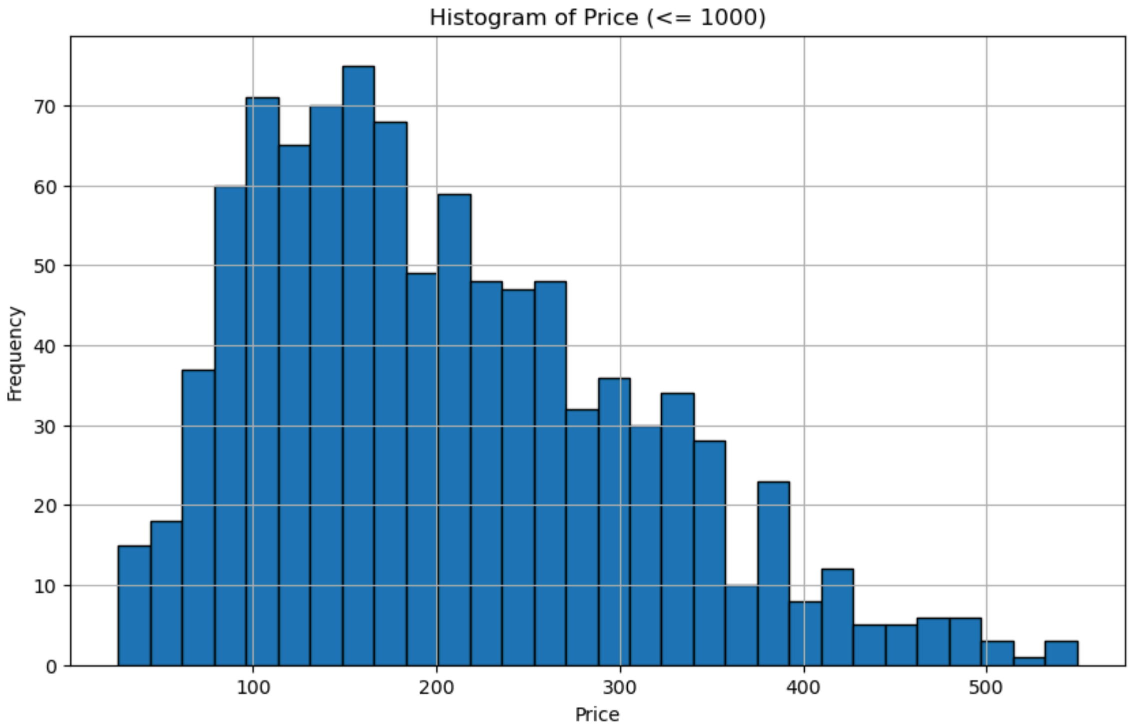
Correlation between features

In [10]:
```
# Select only numeric columns
numeric_data = data.select_dtypes(include='number')

# Calculate the correlation matrix
correlation_matrix = numeric_data.corr()

# Create a heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", lin
plt.title('Correlation Heatmap of Numeric Columns')
plt.show()
```



Exploring "Price"

In [11]:
```
# Filter the "Price" column to include only values less than or equal to 10(
filtered_price = data[data['Price'] <= 800]
dropped_price = data[data['Price'] > 800]

# Plot a histogram of the filtered "Price" column
plt.figure(figsize=(10, 6))
plt.hist(filtered_price['Price'].dropna(), bins=30, edgecolor='black')
plt.title('Histogram of Price (<= 1000)')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

dropped_price
```



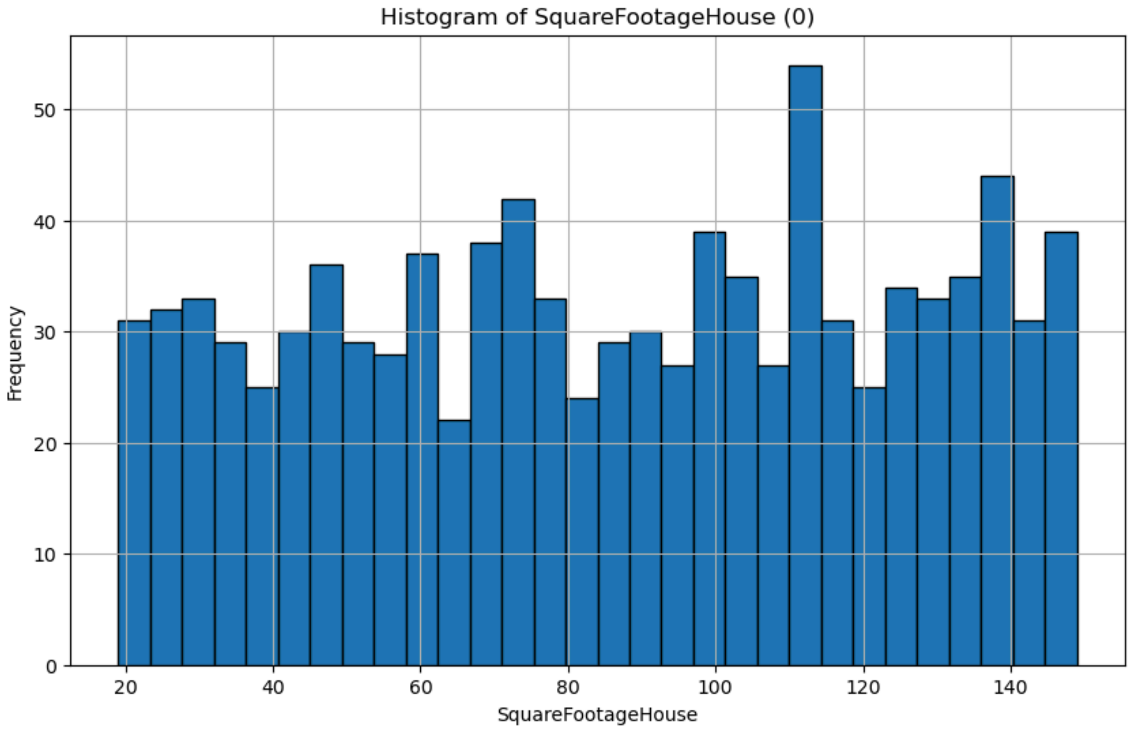
Out[11]:

	Bedrooms	Bathrooms	SquareFootageHouse	Location	Age	PoolQuality	HasPhotovolt
48	<NA>	2	59.0	1	38.0	0.0	I
144	1	1	24.0	1	49.0	0.0	F
155	3	<NA>	114.0	<NA>	22.0	2.0	F
195	<NA>	1	47.0	3	29.0	0.0	F
215	1	1	73.0	1	23.0	2.0	F
269	<NA>	2	92.0	<NA>	NaN	1.0	F
285	2	1	79.0	1	61.0	0.0	F
307	<NA>	3	134.0	2	NaN	3.0	F
356	<NA>	4	140.0	3	68.0	2.0	F
370	2	5	132.0	1	51.0	0.0	F
506	1	1	45.0	2	28.0	0.0	-
512	1	1	24.0	<NA>	-51.0	0.0	F
614	1	1	-870.0	3	48.0	2.0	I
633	4	1	142.0	1	65.0	2.0	I
662	2	3	138.0	3	44.0	0.0	F
757	3	4	142.0	2	69.0	0.0	I
772	<NA>	1	34.0	<NA>	42.0	0.0	I
803	1	2	77.0	1	57.0	0.0	-
843	4	1	147.0	1	37.0	0.0	-
931	<NA>	1	21.0	<NA>	63.0	0.0	F
939	1	1	47.0	2	17.0	0.0	F
942	1	1	39.0	2	35.0	0.0	F
987	3	1	105.0	1	70.0	3.0	F

23 rows × 21 columns

Square Footage exploration

In [12]:
```
filtered_footage = data[(data['SquareFootageHouse'] >= 0) & (data['SquareFo
dropped_footage = data[(data['SquareFootageHouse'] <= 0) | (data['SquareFoo


# Plot a histogram of the filtered "Price" column
plt.figure(figsize=(10, 6))
plt.hist(filtered_footage['SquareFootageHouse'].dropna(), bins=30, edgecolo
plt.title('Histogram of SquareFootageHouse (0)')
plt.xlabel('SquareFootageHouse')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

print(len(data) - len(filtered_footage))
dropped_footage
```




```
13
```

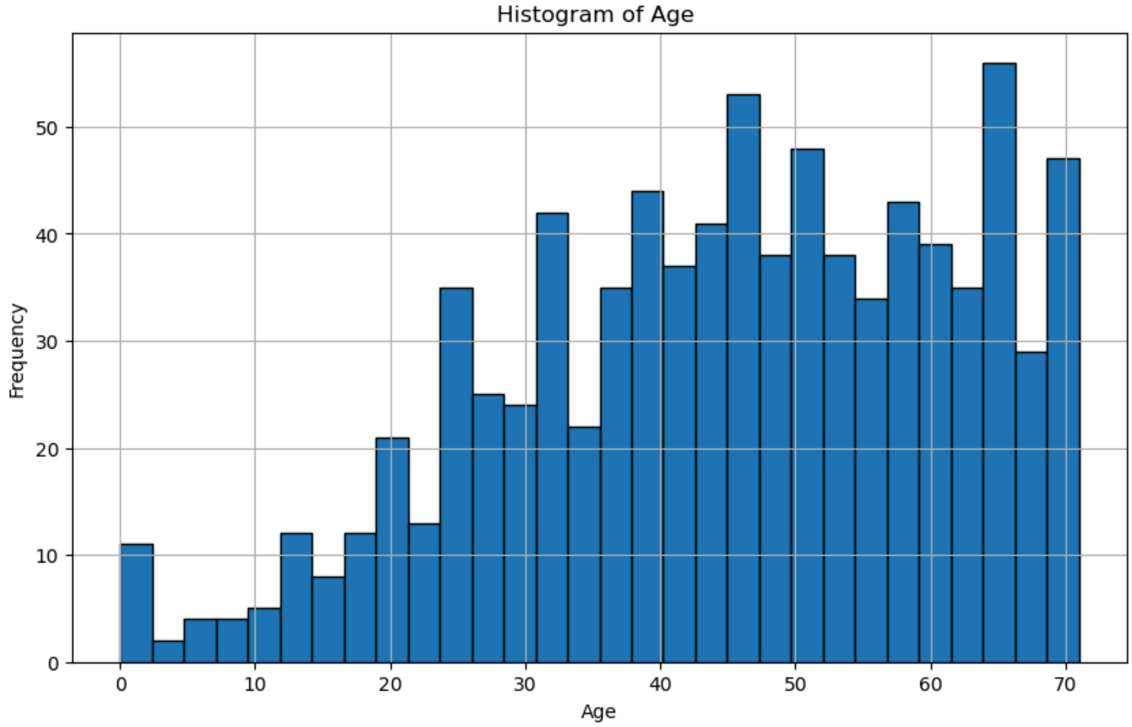
Out[12]:

	Bedrooms	Bathrooms	SquareFootageHouse	Location	Age	PoolQuality	HasPhotovolta
86	1	2	6498.0	1	33.0	2.0	Fa
119	1	1	5465.0	<NA>	58.0	0.0	N
237	2	1	8399.0	2	71.0	2.0	Fa
290	2	2	6518.0	1	71.0	0.0	Fa
330	1	2	-977.0	<NA>	21.0	0.0	Fa
387	<NA>	1	8024.0	1	38.0	0.0	Fa
447	2	<NA>	5491.0	1	35.0	2.0	Fa
492	<NA>	1	6394.0	3	NaN	0.0	Fa
498	1	1	7408.0	3	33.0	0.0	N
614	1	1	-870.0	3	48.0	2.0	N
660	2	1	-914.0	2	19.0	2.0	Fa
664	1	3	5885.0	<NA>	40.0	0.0	N
726	<NA>	3	-655.0	<NA>	58.0	0.0	T

13 rows × 21 columns

Age feature exploration

In [13]:
```
filtered_age = data[data['Age']>= 0]
dropped_age  = data[data['Age']< 0]

# Plot a histogram of the filtered "Price" column
plt.figure(figsize=(10, 6))
plt.hist(filtered_age['Age'].dropna(), bins=30, edgecolor='black')
plt.title('Histogram of Age ')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

dropped_age
```

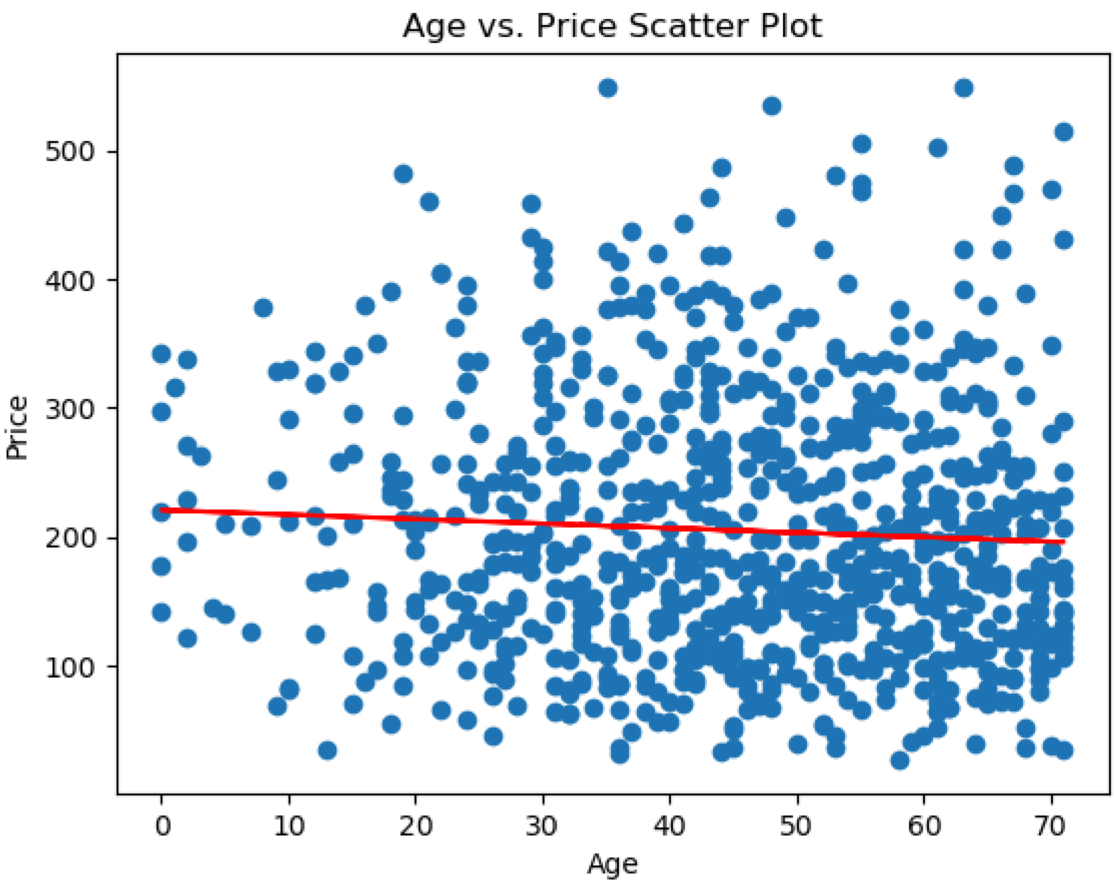


Out[13]:

	Bedrooms	Bathrooms	SquareFootageHouse	Location	Age	PoolQuality	HasPhotovolt
109	<NA>	1	85.0	1	-1.0	0.0	F
244	2	1	82.0	1	-46.0	0.0	-
417	4	4	146.0	1	-18.0	0.0	I
418	1	1	51.0	<NA>	-24.0	1.0	F
445	2	3	104.0	1	-15.0	3.0	-
460	1	<NA>	55.0	3	-77.0	0.0	I
512	1	1	24.0	<NA>	-51.0	0.0	F
816	2	1	73.0	3	-2.0	0.0	F

8 rows × 21 columns

In [14]:
```
# Apply filters simultaneously
filtered_data = data[(data['Age'] >= 0) & (data['Price'] <= 800)]

# Extract filtered age and price
filtered_age = filtered_data['Age']
filtered_price = filtered_data['Price']

# Fit a line (linear regression)
coefficients = np.polyfit(filtered_age, filtered_price, 1)
poly_function = np.poly1d(coefficients)

# Calculate correlation coefficient
correlation_matrix = np.corrcoef(filtered_age, filtered_price)
correlation = correlation_matrix[0, 1]

# Create a scatter plot
plt.scatter(filtered_age, filtered_price)

# Add the regression line
plt.plot(filtered_age, poly_function(filtered_age), color='red')

# Add labels and title
plt.xlabel('Age')
plt.ylabel('Price')
plt.title('Age vs. Price Scatter Plot')

# Show the plot
plt.show()

print("Correlation coefficient:", correlation)
```




```
Correlation coefficient: -0.05698942876856539
```

In [15]:
```
# # Get the list of column names
# columns = data.columns

# # Generate scatter plots for each pair of columns
# for i in range(len(columns)):
#     for j in range(i+1, len(columns)):
#         plt.figure(figsize=(8, 6))
#         sns.scatterplot(data=data, x=columns[i], y=columns[j])
#         plt.title(f'Scatter Plot of {columns[i]} vs {columns[j]}')
#         plt.xlabel(columns[i])
#         plt.ylabel(columns[j])
#         plt.show()
```

Let's do some Dimensionality Reduction so we have less stuff to impute and care about.

To do this, we will apply PCA to reduce the number of variables to a more manageable number.

For this we should first normalize the data using the standard normalizations since that puts out distributions with a mean of 0 and a standard deviation of 1.

However, standardization only works on numerical data (obviously) so, we need to do one-hot-enconding and convert those categorial columns into numerical. the color of the house will be split into several columns like:

Green: Yellow: Red: 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0

Before however, we have to take care immediatly of the Nan/nulls since our strategies later dont handle them well

In [16]:

```
#1.1lets print the intial state of nan
print("Initial NaN counts per column in 'data':")
print(data.isna().sum())

#columns to use knn in
columns_to_impute = ['Bedrooms', 'Bathrooms', 'Age', 'HeatingCosts']

#subset of the main dataframw with the columns we want
dataToKnn = data[columns_to_impute]

#1.2 do the knn imputation
imputer = KNNImputer(n_neighbors=3)
df_imputed_subset = pd.DataFrame(imputer.fit_transform(dataToKnn), columns=

# Verify no NaN values in the imputed subset
print("\nNaN counts in 'df_imputed_subset' after imputation:")
print(df_imputed_subset.isna().sum())

# Ensure the indices of the imputed subset match those of the original Data

df_imputed_subset.index = dataToKnn.index

# Reassign the imputed values back to the original DataFrame
data[columns_to_impute] = df_imputed_subset

# Check if the reassignment worked and there are no NaN values in the origi
nan_counts = data.isna().sum()
print("\nNaN counts per column in 'data' after reassignment:")
print(nan_counts)
```

```
Initial NaN counts per column in 'data':
Bedrooms               328
Bathrooms              165
SquareFootageHouse       0
Location               230
Age                    130
PoolQuality              0
HasPhotovoltaics       258
HeatingType              0
HasFiberglass            0
IsFurnished              0
DateSinceForSale         0
HasFireplace             0
KitchensQuality          0
BathroomsQuality         0
BedroomsQuality          0
LivingRoomsQuality       0
SquareFootageGarden      0
PreviousOwnerRating      0
HeatingCosts           452
WindowModelNames         0
Price                    0
dtype: int64

NaN counts in 'df_imputed_subset' after imputation:
Bedrooms        0
Bathrooms       0
Age             0
HeatingCosts    0
dtype: int64

NaN counts per column in 'data' after reassignment:
Bedrooms                 0
Bathrooms                0
SquareFootageHouse       0
Location               230
Age                      0
PoolQuality              0
HasPhotovoltaics       258
HeatingType              0
HasFiberglass            0
IsFurnished              0
DateSinceForSale         0
HasFireplace             0
KitchensQuality          0
BathroomsQuality         0
BedroomsQuality          0
LivingRoomsQuality       0
SquareFootageGarden      0
PreviousOwnerRating      0
HeatingCosts             0
WindowModelNames         0
Price                    0
dtype: int64
```

Lets apply a decision tree algorithim to Location. Decision tree looks at other

patterns in the data and imputes the values

In [17]:
```
# Encode 'Location' and 'HasPhotovoltaics' columns
label_encoder_loc = LabelEncoder()
label_encoder_pv = LabelEncoder()
data['Location_encoded'] = label_encoder_loc.fit_transform(data['Location']
data['HasPhotovoltaics_encoded'] = label_encoder_pv.fit_transform(data['Has

# Separate rows with and without missing values
train_data_loc = data[data['Location'].notna()]
test_data_loc = data[data['Location'].isna()]

train_data_pv = data[data['HasPhotovoltaics'].notna()]
test_data_pv = data[data['HasPhotovoltaics'].isna()]

# Identify categorical columns, excluding the target columns
categorical_columns = data.select_dtypes(include=['object']).columns.differ

# Apply one-hot encoding to categorical columns on the entire dataset
one_hot_encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
one_hot_encoded_full = one_hot_encoder.fit_transform(data[categorical_colum

# Create DataFrame from the one-hot encoded array
one_hot_encoded_full_df = pd.DataFrame(one_hot_encoded_full, index=data.ind

# Split the one-hot encoded DataFrame into train and test sets
one_hot_encoded_train_loc_df = one_hot_encoded_full_df.loc[train_data_loc.i
one_hot_encoded_test_loc_df = one_hot_encoded_full_df.loc[test_data_loc.ind

one_hot_encoded_train_pv_df = one_hot_encoded_full_df.loc[train_data_pv.ind
one_hot_encoded_test_pv_df = one_hot_encoded_full_df.loc[test_data_pv.index

# Combine one-hot encoded columns with the rest of the features
train_data_loc = train_data_loc.drop(columns=categorical_columns).join(one_
test_data_loc = test_data_loc.drop(columns=categorical_columns).join(one_ho

train_data_pv = train_data_pv.drop(columns=categorical_columns).join(one_ho
test_data_pv = test_data_pv.drop(columns=categorical_columns).join(one_hot_

# Features for training the decision tree (excluding the target columns)
features_loc = train_data_loc.columns.drop(['Location', 'Location_encoded']
features_pv = train_data_pv.columns.drop(['HasPhotovoltaics', 'HasPhotovolt

# Train a decision tree classifier for 'Location'
classifier_loc = DecisionTreeClassifier()
classifier_loc.fit(train_data_loc[features_loc], train_data_loc['Location_e

# Train a decision tree classifier for 'HasPhotovoltaics'
classifier_pv = DecisionTreeClassifier()
classifier_pv.fit(train_data_pv[features_pv], train_data_pv['HasPhotovoltai

# Predict missing values
predicted_locations = classifier_loc.predict(test_data_loc[features_loc])
predicted_pvs = classifier_pv.predict(test_data_pv[features_pv])

# Convert predicted encoded labels back to original labels
predicted_locations_labels = label_encoder_loc.inverse_transform(predicted_
predicted_pvs_labels = label_encoder_pv.inverse_transform(predicted_pvs)

# Fill missing values in the original DataFrame
data.loc[data['Location'].isna(), 'Location'] = predicted_locations_labels
data.loc[data['HasPhotovoltaics'].isna(), 'HasPhotovoltaics'] = predicted_p
```

```
# Drop the encoded columns
data.drop(columns=['Location_encoded', 'HasPhotovoltaics_encoded'], inplace

# Check NaN counts again
nan_counts = data.isna().sum()
print("NaN counts per column in 'data' after decision tree imputation:")
print(nan_counts)
```

```
C:\Users\Vasco\anaconda3\Lib\site-packages\sklearn\preprocessing\_encoder
s.py:972: FutureWarning: `sparse` was renamed to `sparse_output` in versio
n 1.2 and will be removed in 1.4. `sparse_output` is ignored unless you le
ave `sparse` to its default value.
  warnings.warn(

NaN counts per column in 'data' after decision tree imputation:
Bedrooms               0
Bathrooms              0
SquareFootageHouse     0
Location               0
Age                    0
PoolQuality            0
HasPhotovoltaics       0
HeatingType            0
HasFiberglass          0
IsFurnished            0
DateSinceForSale       0
HasFireplace           0
KitchensQuality        0
BathroomsQuality       0
BedroomsQuality        0
LivingRoomsQuality     0
SquareFootageGarden    0
PreviousOwnerRating    0
HeatingCosts           0
WindowModelNames       0
Price                  0
dtype: int64
```

In [85]:
```
def printNAN(df):
    nan_counts = df.isna().sum()
    print("NaN counts per column in 'data' after decision tree imputation:"
    print(nan_counts)

# printNAN(data)
dataTarget = data['Price']
dataTarget

data.drop(columns='Price', inplace=True)
# data
#data now looks very nice and free of NaN, using two different imputation te
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
File ~\anaconda3\Lib\site-packages\pandas\core\indexes\base.py:3653, in In
dex.get_loc(self, key)
   3652 try:
-> 3653     return self._engine.get_loc(casted_key)
   3654 except KeyError as err:

File ~\anaconda3\Lib\site-packages\pandas\_libs\index.pyx:147, in pandas._
libs.index.IndexEngine.get_loc()

File ~\anaconda3\Lib\site-packages\pandas\_libs\index.pyx:176, in pandas._
libs.index.IndexEngine.get_loc()

File pandas\_libs\hashtable_class_helper.pxi:7080, in pandas._libs.hashtab
le.PyObjectHashTable.get_item()

File pandas\_libs\hashtable_class_helper.pxi:7088, in pandas._libs.hashtab
le.PyObjectHashTable.get_item()

KeyError: 'Price'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call las
t)
Cell In[85], line 7
      4     print(nan_counts)
      6 # printNAN(data)
----> 7 dataTarget = data['Price']
      8 dataTarget
     10 data.drop(columns='Price', inplace=True)

File ~\anaconda3\Lib\site-packages\pandas\core\frame.py:3761, in DataFram
e.__getitem__(self, key)
   3759 if self.columns.nlevels > 1:
   3760     return self._getitem_multilevel(key)
-> 3761 indexer = self.columns.get_loc(key)
   3762 if is_integer(indexer):
   3763     indexer = [indexer]

File ~\anaconda3\Lib\site-packages\pandas\core\indexes\base.py:3655, in In
dex.get_loc(self, key)
   3653     return self._engine.get_loc(casted_key)
   3654 except KeyError as err:
-> 3655     raise KeyError(key) from err
   3656 except TypeError:
   3657     # If we have a listlike key, _check_indexing_error will raise
   3658     #  InvalidIndexError. Otherwise we fall through and re-raise
   3659     #  the TypeError.
   3660     self._check_indexing_error(key)

KeyError: 'Price'
```

In [84]:
```
# Encode categorical columns
one_hot_encoded = pd.get_dummies(data.select_dtypes(include=['object']), dr

# # # Standardize numerical columns
scaler = StandardScaler()
scaled_numerical = scaler.fit_transform(data.select_dtypes(include=['int',

# # Create DataFrame from standardized numerical data
scaled_numerical_df = pd.DataFrame(scaled_numerical, columns=data.select_dt

# # Concatenate one-hot encoded and standardized numerical columns
one_hot_encoded.reset_index(drop=True, inplace=True)
scaled_numerical_df.reset_index(drop=True, inplace=True)
processed_data = pd.concat([one_hot_encoded, scaled_numerical_df], axis=1)

# Apply PCA
pca = PCA(n_components=20)  # You can adjust the number of components as ne
pca_features = pca.fit_transform(processed_data)

# Create DataFrame from PCA features
pca_df = pd.DataFrame(data=pca_features, columns=[f"PC{i+1}" for i in range

# Concatenate PCA features with original DataFrame if needed
final_data = pd.concat([data.drop(columns=data.select_dtypes(include=['obje

# # Check the final data
# final_data.head()

cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)

# Plot the cumulative explained variance ratio
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(cumulative_variance_ratio) + 1), cumulative_variance_
plt.xlabel('Number of Components')
plt.xlim(0, 25)
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Cumulative Explained Variance Ratio vs. Number of Components')
plt.grid(True)
plt.show()

pca_df
```

0	-1.227502	-0.059873	0.052722	-0.844185	-0.528346	0.434486	-0.092810	0.092141
1	0.434062	-1.983958	0.634324	0.754995	0.916626	0.958605	0.743830	0.018498
2	-1.622454	0.167973	-0.009296	0.870788	-0.638236	-0.912327	-0.124111	-0.267908
3	0.018352	0.986225	0.190199	-1.898025	0.310746	1.107815	0.467609	-0.298120
4	-1.200022	0.734276	1.367843	1.078088	0.371911	0.247162	-0.706451	1.702153
...	...	...	...	...	...	...	...	...
990	1.162144	-0.759954	-1.577112	-0.087712	0.287328	-0.462111	-0.243883	0.789793
991	-1.065731	0.596910	-1.351680	0.539245	-0.792007	-0.789664	0.101235	0.123574
992	-0.451532	-2.777263	1.968970	1.376977	0.866465	-1.399882	0.644863	-0.327028
993	-0.278493	-1.825864	0.025692	-0.141651	-0.687769	1.161957	-0.110039	0.400147
994	-0.363053	0.926627	0.955411	0.433808	0.254947	-0.583558	-0.193718	1.079177

995 rows × 20 columns

In [ ]:

In [ ]:

In [ ]:

In [91]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(pca_df, dataTarget, tes

# Train linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

# Evaluate model
train_rmse = mean_squared_error(y_train, train_predictions, squared=False)
test_rmse = mean_squared_error(y_test, test_predictions, squared=False)

threshold = 200  # Adjust the threshold as needed

# Convert predictions to binary classification
train_predictions_binary = (train_predictions > threshold).astype(int)
test_predictions_binary = (test_predictions > threshold).astype(int)

# Convert true labels to binary classification
y_train_binary = (y_train > threshold).astype(int)
y_test_binary = (y_test > threshold).astype(int)

# Compute F1 score
train_f1 = f1_score(y_train_binary, train_predictions_binary)
test_f1 = f1_score(y_test_binary, test_predictions_binary)

print(f"Train F1 Score: {train_f1}")
print(f"Test F1 Score: {test_f1}")

print(f"Train RMSE: {train_rmse}")
print(f"Test RMSE: {test_rmse}")
```

```
Train F1 Score: 0.6653225806451614
Test F1 Score: 0.638655462184874
Train RMSE: 671.3294448428381
Test RMSE: 317.25072822428854
```

In [94]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, f1_score

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(pca_df, dataTarget, tes

# Train Random Forest model
model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)

# Make predictions
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

# Evaluate model
train_rmse = mean_squared_error(y_train, train_predictions, squared=False)
test_rmse = mean_squared_error(y_test, test_predictions, squared=False)

threshold = 200  # Adjust the threshold as needed

# Convert predictions to binary classification
train_predictions_binary = (train_predictions > threshold).astype(int)
test_predictions_binary = (test_predictions > threshold).astype(int)

# Convert true labels to binary classification
y_train_binary = (y_train > threshold).astype(int)
y_test_binary = (y_test > threshold).astype(int)

# Compute F1 score
train_f1 = f1_score(y_train_binary, train_predictions_binary)
test_f1 = f1_score(y_test_binary, test_predictions_binary)

print(f"Train F1 Score: {train_f1}")
print(f"Test F1 Score: {test_f1}")

print(f"Train RMSE: {train_rmse}")
print(f"Test RMSE: {test_rmse}")
```

```
Train F1 Score: 0.8398384925975774
Test F1 Score: 0.6934097421203438
Train RMSE: 305.2136567537085
Test RMSE: 403.71234095444356
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [86]:
```
dataTarget
```

Out[86]:
```
0      208.13382
1      333.75130
2       52.30557
3      256.17149
4      252.23226
         ...
995    235.10908
996    103.91421
997    230.80934
998    129.25993
999    149.25619
Name: Price, Length: 995, dtype: float64
```

In [77]:
```
print(pca_df.dtypes)
```

```
PC1     float64
PC2     float64
PC3     float64
PC4     float64
PC5     float64
PC6     float64
PC7     float64
PC8     float64
PC9     float64
PC10    float64
PC11    float64
PC12    float64
PC13    float64
PC14    float64
PC15    float64
PC16    float64
PC17    float64
PC18    float64
PC19    float64
PC20    float64
dtype: object
```

In [ ]: