DK Housing Price Prediction

1 DATA ANALYSIS AND PREPROCESSING

1.1 Dataset Overview:

The dataset comprises **100,000 records** and **19 features**, including a mix of categorical (e.g., house_type, region) and numerical variables (e.g., sqm, sqm_price, purchase_price). It captures housing transactions in Denmark from 1992 to 2024, providing insights into property sizes, prices, and economic trends. The average purchase price is approximately **1.9 million DKK**, with significant variation. The dataset's diversity in property types and regions makes it well-suited for analyzing housing price determinants.

1.2 Exploratory Data Analysis

1.2.1 Data Statistics

Summary statistics for numerical columns were obtained using df.describe(). The year build ranges from 1000 to 2024, indicating properties built over a wide historical period, with a mean year of 1955 (±45.7 years). The purchase_price varies significantly, from 250,200 DKK to 45,955,000 DKK, with a mean price of approximately 1.93 million DKK (±1.78 million DKK). The sqm_price ranges from 374.5 to 75,000 DKK, with a mean of 16,406 DKK (±13,655 DKK). The %_change between offer and purchase shows price adjustments between -49% and +49%, with a mean change of -2.08% (±4.85%).

```
summary_stats = df.describe()
summary_stats
```

	house_id	year_build	purchase_price	%_change_between_offer_and_purchase	no_rooms	sqm	sqm_price	zip_code	nom
count	1.000000e+05	100000.000000	1.000000e+05	100000.000000	100000.000000	100000.000000	100000.000000	100000.000000	
mean	7.531364e+05	1955.078530	1.925992e+06	-2.084740	4.375790	129.387820	16406.648074	5951.612680	
std	4.348902e+05	45.703585	1.777832e+06	4.852926	1.661686	57.170619	13655.184405	2369.938171	
min	1.300000e+01	1000.000000	2.502000e+05	-49.000000	1.000000	26.000000	374.549800	1051.000000	
25%	3.770722e+05	1931.000000	8.000000e+05	-3.000000	3.000000	89.000000	6792.940775	4000.000000	
50%	7.511035e+05	1966.000000	1.400000e+06	0.000000	4.000000	123.000000	12070.707000	5970.000000	
75%	1.128904e+06	1980.000000	2.450000e+06	0.000000	5.000000	160.000000	21343.201500	8250.000000	
max	1.507901e+06	2024.000000	4.595500e+07	49.000000	15.000000	984.000000	75000.000000	9990.000000	

1.2.2 Checking Missing Values

The dataset has missing values in city (11 records) and in dk_ann_infl_rate% and yield_on_mortgage_credit_bonds% (77 records each), identified using df.isnull().sum().

```
df.isnull().sum()
```

```
date                                  0
quarter                               0
house_id                              0
house_type                            0
sales_type                            0
year_build                            0
purchase_price                        0
%_change_between_offer_and_purchase   0
no_rooms                              0
sqm                                   0
sqm_price                             0
address                               0
zip_code                              0
city                                 11
area                                  0
region                                0
nom_interest_rate%                    0
dk_ann_infl_rate%                    77
yield_on_mortgage_credit_bonds%      77
dtype: int64
```

1.2.3 Dataset Length and Shape:

The dataset contains **100,000 records** with **19 columns**, including both categorical and numerical variables. It captures property details like house_type, year_build, and purchase_price, along with geographic data and economic indicators, providing a rich resource for analyzing housing trends and price factors.

```
print(f'length of the dataset: {len(df)}')
print()

print(f'Shape of the dataset: {df.shape}')
```

```
length of the dataset: 100000

Shape of the dataset: (100000, 19)
```

1.2.4 Checking Information about the data

The df.info() method reveals 100,000 entries and 19 columns, including 6 numerical, 5 integer, and 8 categorical columns. Missing values are present in the city column (11 entries), and both dk_ann_infl_rate% and yield_on_mortgage_credit_bonds% columns (77 entries each).

```
: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 19 columns):
 #   Column                                Non-Null Count   Dtype
---  ------                                --------------   -----
 0   date                                  100000 non-null  object
 1   quarter                               100000 non-null  object
 2   house_id                              100000 non-null  int64
 3   house_type                            100000 non-null  object
 4   sales_type                            100000 non-null  object
 5   year_build                            100000 non-null  int64
 6   purchase_price                        100000 non-null  int64
 7   %_change_between_offer_and_purchase   100000 non-null  float64
 8   no_rooms                              100000 non-null  int64
 9   sqm                                   100000 non-null  float64
 10  sqm_price                             100000 non-null  float64
 11  address                               100000 non-null  object
 12  zip_code                              100000 non-null  int64
 13  city                                  99989 non-null   object
 14  area                                  100000 non-null  object
 15  region                                100000 non-null  object
 16  nom_interest_rate%                    100000 non-null  float64
 17  dk_ann_infl_rate%                     99923 non-null   float64
 18  yield_on_mortgage_credit_bonds%       99923 non-null   float64
dtypes: float64(6), int64(5), object(8)
memory usage: 14.5+ MB
```

1.2.5 Data Pre-processing and Validation

Dropping Null Values: Rows with missing values are removed using df.dropna(axis=0) to ensure only complete data is used, preventing inaccuracies or biases during analysis or model training.

```
df = df.dropna(axis=0)

df
```

Converting Categorical Columns into Numerical Values Using Label Encoding: It applies Label Encoding to transform categorical columns into numerical values. The LabelEncoder is used to convert the categories in the columns (house_type, sales_type, city, area, and region) into integer labels. This ensures that categorical data is in a format suitable for machine learning models. The encoded values are then stored back in the encoded_df DataFrame, replacing the original categorical data.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# Assuming df is your dataframe and these are the remaining categorical columns
categorical_columns = ['house_type', 'sales_type', "city", "area", 'region']

encoded_df = pd.DataFrame()
encoded_df = df

# Apply Label Encoding for ordinal columns
label_encoder = LabelEncoder()

encoded_df[categorical_columns[0]] = label_encoder.fit_transform(encoded_df[categorical_columns[0]])
encoded_df[categorical_columns[1]] = label_encoder.fit_transform(encoded_df[categorical_columns[1]])
encoded_df[categorical_columns[2]] = label_encoder.fit_transform(encoded_df[categorical_columns[2]])
encoded_df[categorical_columns[3]] = label_encoder.fit_transform(encoded_df[categorical_columns[3]])
encoded_df[categorical_columns[4]] = label_encoder.fit_transform(encoded_df[categorical_columns[4]])

encoded_df
```

Dropping Unnecessary Columns: The columns ['date', 'quarter', 'address', 'zip_code', 'house_id'] are dropped using df_dropped = encoded_df.drop(['date', 'quarter', 'address', 'zip_code', 'house_id'], axis=1). These columns are removed as they do not add predictive value and are not needed for model training, streamlining the dataset for more effective analysis.

```
df_dropped = encoded_df.drop(['date', 'quarter', 'address', 'zip_code', 'house_id'], axis=1)

df_dropped
```

Scaling the dataset: The StandardScaler () standardizes the features by scaling them to have a mean of 0 and a standard deviation of 1. It uses fit_transform() on the training set and transform() on the test set for consistent scaling.

```
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Correlation Matrix: The corr() method is used to calculate the correlation matrix for the numeric columns in df_dropped. This matrix shows the relationship between the numerical features of the dataset, indicating how each feature correlates with others, helping to identify patterns and dependencies between variables.

```
corr_matrix = df_dropped.corr()

# Print the correlation matrix
corr_matrix
```

	house_type	sales_type	year_build	purchase_price	%_change_between_offer_and_purchase	no_rooms	sqm	sqm_
house_type	1.000000	0.072742	0.159274	-0.131039	-0.076444	0.417286	0.392590	-0.4
sales_type	0.072742	1.000000	0.087464	0.076686	-0.105141	-0.006235	0.009888	0.0
year_build	0.159274	0.087464	1.000000	-0.013662	0.012253	-0.039880	-0.039005	-0.0
purchase_price	-0.131039	0.076686	-0.013662	1.000000	0.026697	0.176836	0.259483	0.7
%_change_between_offer_and_purchase	-0.076444	-0.105141	0.012253	0.026697	1.000000	-0.028875	-0.025606	0.0
no_rooms	0.417286	-0.006235	-0.039880	0.176836	-0.028875	1.000000	0.770669	-0.2
sqm	0.392590	0.009888	-0.039005	0.259483	-0.025606	0.770669	1.000000	-0.2
sqm_price	-0.421465	0.077847	-0.023975	0.758828	0.060001	-0.258309	-0.252218	1.0
city	0.056396	0.005173	0.006506	-0.030535	-0.004594	0.027363	0.026732	-0.0
area	0.144824	-0.034357	0.009332	-0.261624	-0.018097	0.081645	0.070271	-0.3
region	-0.110407	0.012419	0.054861	0.191864	0.014911	-0.078868	-0.078397	0.2
nom_interest_rate%	0.038655	-0.013669	-0.097540	-0.187266	0.170407	0.044918	0.029685	-0.1
dk_ann_infl_rate%	0.001125	0.016353	-0.025250	0.025443	0.014895	-0.008460	-0.011281	0.0
yield_on_mortgage_credit_bonds%	0.050391	-0.029668	-0.108138	-0.232718	0.160200	0.055913	0.041694	-0.2

2 Modeling:

2.1 Regression Models Performance Metrics

Model	RMSE	MSE	MAE	R^2 Score
Linear Regression	794,097.23	630,590,413,395.69	437,554.76	0.8020
Ridge Regression	794,097.05	630,590,123,212.19	437,551.17	0.8020
Lasso Regression	794,097.25	630,590,447,606.48	437,554.32	0.8020
Random Forest Regression	77,535.99	6,011,829,140.15	9,066.43	0.9981
Gradient Boosting Regressor	84,106.37	7,073,881,016.29	49,953.07	0.9978
K-Nearest Neighbors (KNN)	696,301.34	484,835,558,733.38	275,488.88	0.8478

2.2 Model Performance Insights:

- Linear, Ridge, and Lasso Regression perform similarly, with high RMSE, MSE, and MAE values, indicating difficulty in accurate predictions. Their R^2 score of 0.8020 shows they explain about 80% of the variance but struggle with complex relationships.
- Random Forest performs excellently, with low RMSE (77,535.99), MSE (6B), and MAE (9,066.43), and a high R^2 score of 0.9981, indicating it captures complex relationships well.
- Gradient Boosting also performs well with an RMSE of 84,106.37, MSE of 7B, and R^2 of 0.9978, close to Random Forest.
- KNN, with an R^2 of 0.8478, shows poor performance due to sensitivity to outliers and high-dimensional spaces, leading to higher error values.

2.3 Best Performing Models

The best-performing models in this analysis are the Random Forest and Gradient Boosting Regressors. Both models stand out with high R^2 scores and low error metrics, demonstrating their ability to effectively capture complex relationships within the dataset. While linear models and K-Nearest Neighbors struggle to provide accurate predictions due to their limitations in handling non-linearities and high-dimensional spaces, Random Forest and Gradient Boosting excel in this regard. Their robust performance underscores their suitability for datasets with intricate patterns, making them the ideal choice for achieving accurate and reliable predictions in this context.

R^2 scores, RMSE, MAE and MSE of each model:

```
Linear Regression R2 Score: 0.8020
Ridge Regression R2 Score: 0.8020
Lasso Regression R2 Score: 0.8020
Random Forest Regression R2 Score: 0.9981
Gradient Boosted Regression R2 Score: 0.9978
K nearest neighbour Regression R2 Score: 0.8478
```

```
Linear Regressor Metrics:
Mean Absolute Error: 437554.7617389363
Mean Squared Error: 630590413395.6881
Root Mean Squared Error: 794097.2317013126

Ridge Regressor Metrics:
Mean Absolute Error: 437551.16751764563
Mean Squared Error: 630590123212.1896
Root Mean Squared Error: 794097.0489884656

Lasso Regressor Metrics:
Mean Absolute Error: 437554.318071137
Mean Squared Error: 630590447606.4788
Root Mean Squared Error: 794097.2532419935

Random Forest Regressor Metrics:
Mean Absolute Error: 9066.425581711906
Mean Squared Error: 6011829140.149226
Root Mean Squared Error: 77535.9860977419
```

```
Gradient Boosting Regressor Metrics:
Mean Absolute Error: 49953.07242325686
Mean Squared Error: 7073881016.2861595
Root Mean Squared Error: 84106.3672755289
```

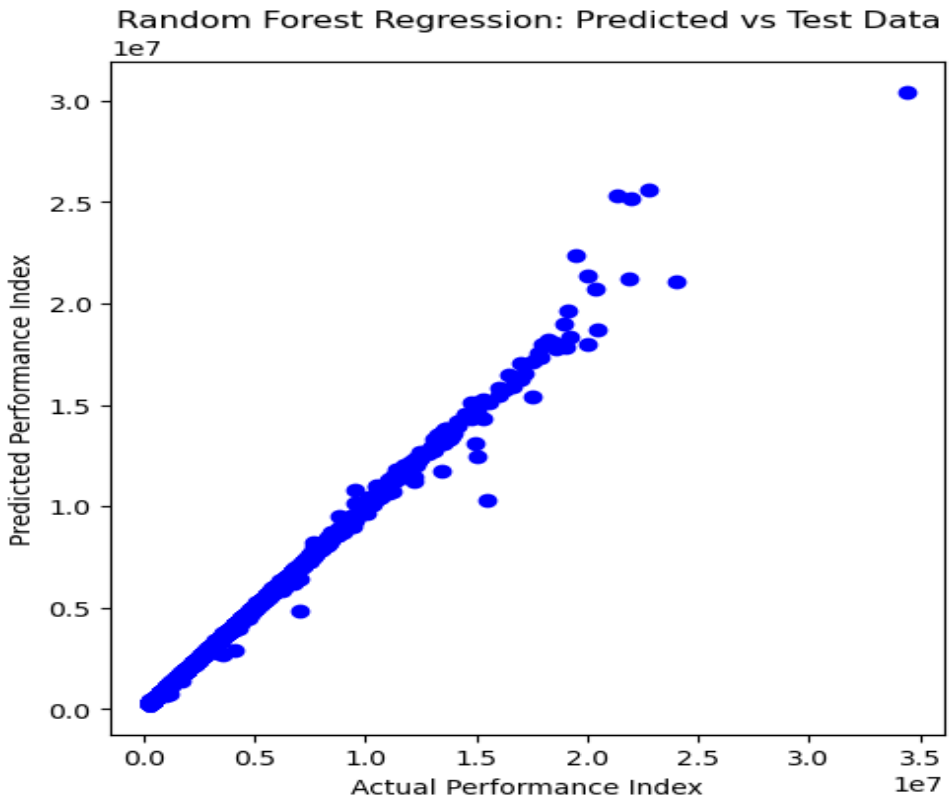
3 Evaluation

Visualising the best model results:

3.1 Scatter Plot for Random Forest Regression

We have visualized the performance of the Random Forest Regression models using scatter plots. The first plot is generated for Random Forest Regression, where the actual performance index values (y_test) are plotted on the x-axis and the predicted performance index values (rf_reg_pred) on the y-axis.

```
# Scatter plot for Random Forest Regression on Test Data
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(y_test, rf_reg_pred, color='blue')
plt.title('Random Forest Regression: Predicted vs Test Data')
plt.xlabel('Actual Performance Index')
plt.ylabel('Predicted Performance Index')
```

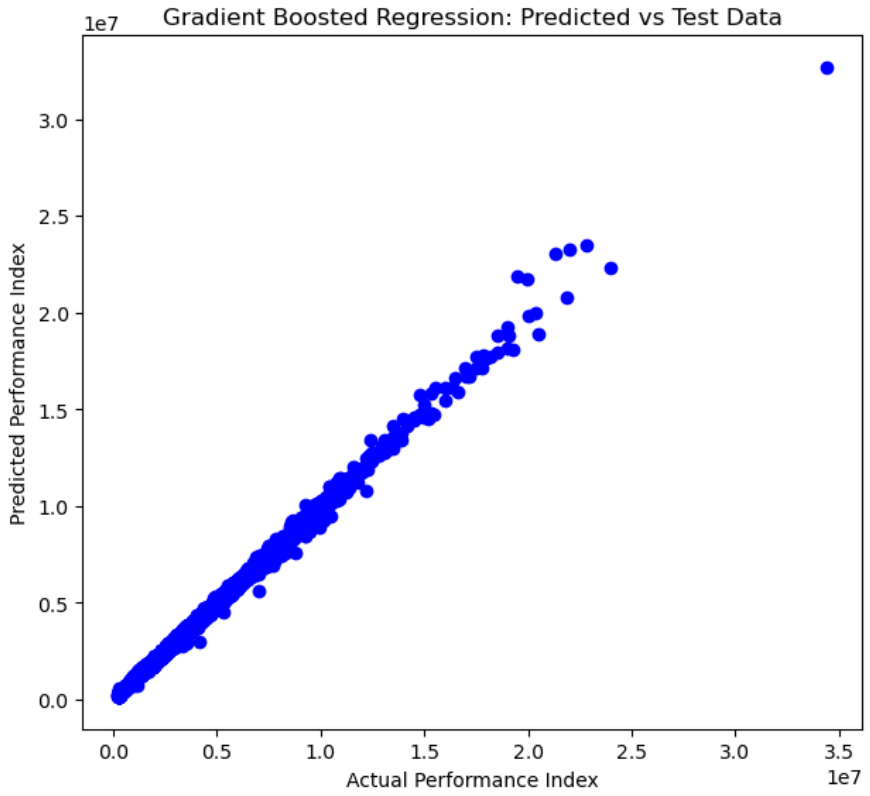


3.2 Scatter Plot for Gradient Boosted Regression

The second plot does the same for Gradient Boosted Regression, using the predicted values (y_pred_gbr) for comparison with the actual values (y_test). Both plots are displayed side by side for comparison, with blue markers representing the data points. The title and axis labels are set for both plots to indicate that they represent the predicted versus actual performance index values. Finally, plt.tight_layout() is used to adjust the spacing between the plots, and plt.show() displays the visualizations.

```
# Scatter plot for Gradient Boosted Regression on Test Data
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred_gbr, color='blue')
plt.title('Gradient Boosted Regression: Predicted vs Test Data')
plt.xlabel('Actual Performance Index')
plt.ylabel('Predicted Performance Index')


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



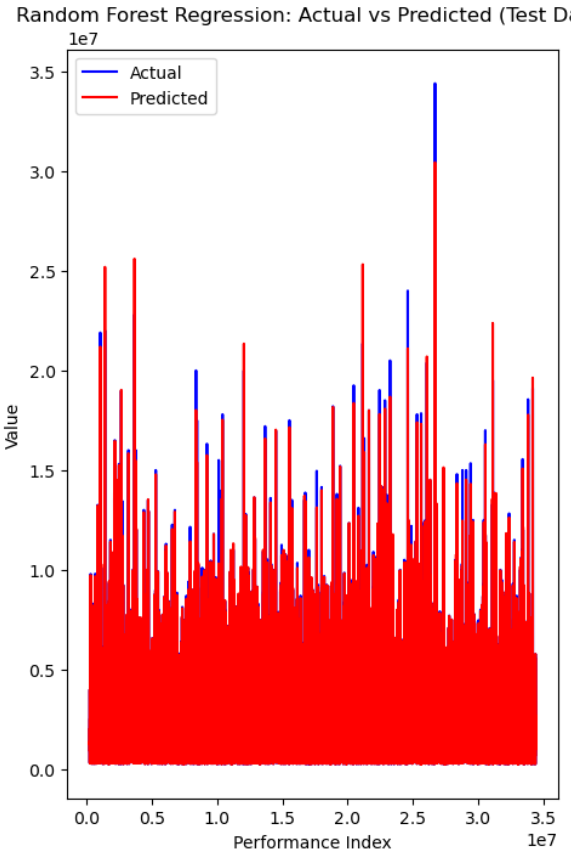
3.3 Line plotting for Random Forest Regression and Gradient Boosted Regression

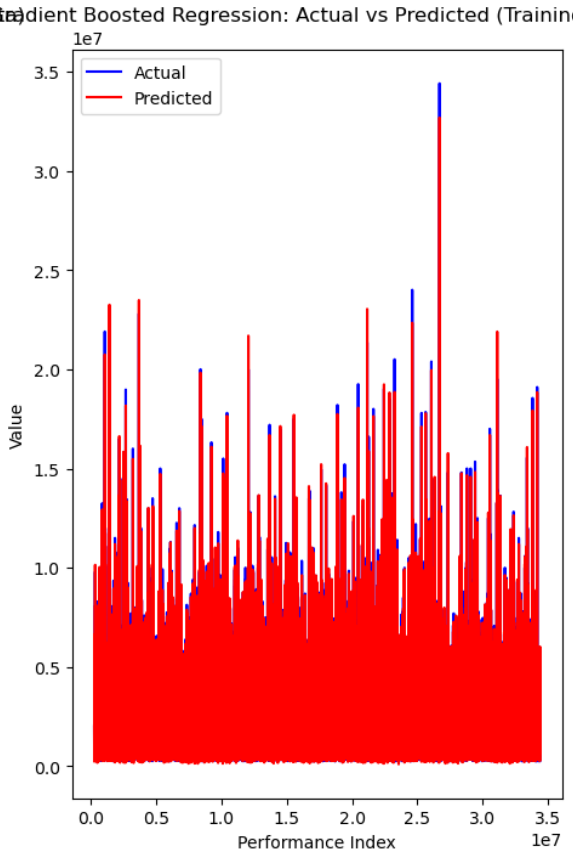
We have visualized the performance of the Random Forest Regression and Gradient Boosted Regression models using line plots. The first plot is for Random Forest Regression on the test data, where the actual values (y_test) and predicted values (rf_reg_pred) are plotted against a linear space that spans the range of the actual values. The actual values are plotted in blue, and the predicted values are plotted in red. This plot helps us visually compare how closely the predicted values match the actual values. The second plot is for Gradient Boosted Regression, but on the training data. Similarly, it shows the actual values (y_test) and predicted values (y_pred_gbr) in blue and red, respectively. The plots are labeled with titles, axis labels, and a legend to indicate which line represents the actual and predicted values. Finally, plt.show() is used to display the visualizations.

```
# Line plot for Random Forest Regression on Test Data
plt.figure(figsize=(24, 8))
plt.subplot(1, 4, 3)
plt.plot(np.linspace(min(y_test), max(y_test), len(y_test)), y_test, label='Actual', color='blue')
plt.plot(np.linspace(min(y_test), max(y_test), len(y_test)), rf_reg_pred, label='Predicted', color='red')
plt.title('Random Forest Regression: Actual vs Predicted (Test Data)')
plt.xlabel('Performance Index')
plt.ylabel('Value')
plt.legend()


# Line plot for Gradient Boosted Regression on Training Data
plt.subplot(1, 4, 4)
plt.plot(np.linspace(min(y_test), max(y_test), len(y_test)), y_test, label='Actual', color='blue')
plt.plot(np.linspace(min(y_test), max(y_test), len(y_test)), y_pred_gbr, label='Predicted', color='red')
plt.title('Gradient Boosted Regression: Actual vs Predicted (Training Data)')
plt.xlabel('Performance Index')
plt.ylabel('Value')
plt.legend()

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


Random Forest Regression: Actual vs Predicted (Test Data)
1e7
Actual
Predicted
Value
Performance Index
1e7


Gradient Boosted Regression: Actual vs Predicted (Training Data)
1e7
Actual
Predicted
Value
Performance Index
1e7

4 Interpretation

4.1 Insights and Conclusions

The performance evaluation of the models reveals significant findings regarding their suitability for the dataset:

Best-performing Models:

Random Forest Regression emerges as the most effective model, with an R^2 score of 0.9981 and the lowest RMSE, MSE, and MAE values. Its ability to handle non-linear patterns and complex feature interactions makes it highly suitable for this dataset.

Gradient Boosted Regression closely follows, achieving an R^2 score of 0.9978. It effectively captures the underlying data relationships while maintaining strong prediction accuracy, albeit slightly slower to train than Random Forest.

4.2 Limitations of Linear Models:

Linear Regression, Ridge Regression, and Lasso Regression

all show comparable results, with an R^2 score of 0.8020. However, their higher error metrics (RMSE, MSE, and MAE) indicate their inability to model the dataset's complexity due to the assumption of linear relationships.

The limitations of other models arise primarily from their inability to handle complex, non-linear relationships in the data. Linear Regression, Ridge, and Lasso assume a linear relationship between features and the target variable, making them unsuitable for datasets with intricate interactions and patterns. Additionally, Ridge and Lasso, despite their regularization benefits, may either overfit or eliminate relevant features, further impacting performance. K-Nearest Neighbors (KNN) struggles in high-dimensional spaces due to the "curse of dimensionality" and is highly sensitive to outliers, which can distort predictions.

Challenges with KNN Regression:

K-Nearest Neighbours (KNN) achieves an R^2 score of 0.8478, but its performance is hampered by high error metrics. The model struggles with the dataset's high-dimensionality and sensitivity to outliers, resulting in reduced prediction accuracy.

Moreover, KNN is computationally intensive and requires careful hyperparameter tuning, which makes it less practical for large datasets. These limitations highlight why ensemble models like Random Forest and Gradient Boosting, with their ability to model complex interactions and non-linear relationships, significantly outperform these simpler models.

Conclusion

The analysis underscores that **Random Forest Regression** and **Gradient Boosted Regression** are the best models for predicting the target variable in this dataset. Their strong performance is attributed to their ability to handle non-linear relationships and complex data interactions. The linear models and KNN, while explaining some variance, are not as effective due to their inherent limitations in modeling non-linear and high-dimensional data. For tasks requiring high accuracy and reliability, leveraging ensemble methods like Random Forest and Gradient Boosting is essential.