AI in Business Decision-Making : Using machine learning algorithms to assist managerial decision

Authors:

Syeda Saba Abedi, Nowsath Ahmed, Jithendar Amanaganti Reddy, Sai Charan Vasa

Importing Libraries

```
# Import necessary libraries
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import mean_squared_error
from sklearn.svm import SVC
import pandas as pd
import numpy as np
```

Load Dataset

```
df = pd.read_csv('diamonds.csv')
```

```
df.shape
```

```
(50000, 8)
```

The Dataset contains 50000 rows and 8 columns.

```python
from tabulate import tabulate

# Get columns and their data types
columns = df.columns
data_types = df.dtypes

# Create a DataFrame for better formatting
summary_df = pd.DataFrame({'   Column ': columns, 'Data_Type': data_types})
print(tabulate(summary_df, headers='keys', tablefmt='psql', showindex=False))
```

```
+--------------+-------------+
|    Column    | Data_Type   |
|--------------+-------------|
| Sr. No.      | int64       |
| carat        | float64     |
| cut          | object      |
| cut_ord      | int64       |
| color        | object      |
| clarity      | object      |
| clarity_ord  | int64       |
| price        | int64       |
+--------------+-------------+
```

Describing Data

```python
df.describe()
```

	Sr. No.	carat	cut_ord	clarity_ord	price
count	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000
mean	25000.500000	0.798597	3.903980	4.126700	3939.103500
std	14433.901067	0.474651	1.117043	1.665564	3995.879832
min	1.000000	0.200000	1.000000	1.000000	326.000000
25%	12500.750000	0.400000	3.000000	3.000000	948.000000
50%	25000.500000	0.700000	4.000000	4.000000	2402.500000
75%	37500.250000	1.040000	5.000000	5.000000	5331.000000
max	50000.000000	5.010000	5.000000	8.000000	18823.000000

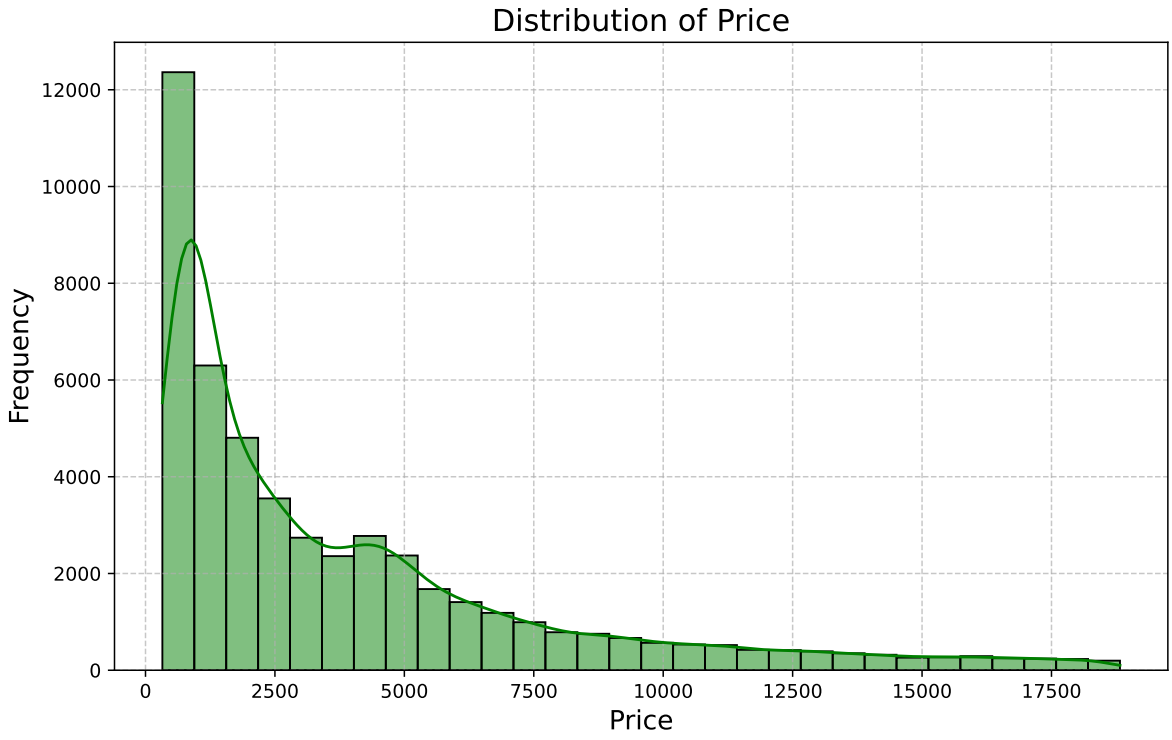
Dataset Preparation

Select Relevant Features

```
# Feature selection (drop target and unnecessary columns like 'Sr. No.')
X = df.drop(columns=['Sr. No.'])  # 'drop Sr.No. column
y = df['price']  # Target variable
```

Distribution Plot of Target Variable (Price)

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(10, 6))
sns.histplot(df['price'], kde=True, color='green', bins=30)
plt.title('Distribution of Price', fontsize=16)
plt.xlabel('Price', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



The price distribution is **right-skewed**, indicating that **most diamonds are priced in the lower range**. A large number of diamonds have prices concentrated around the lower to

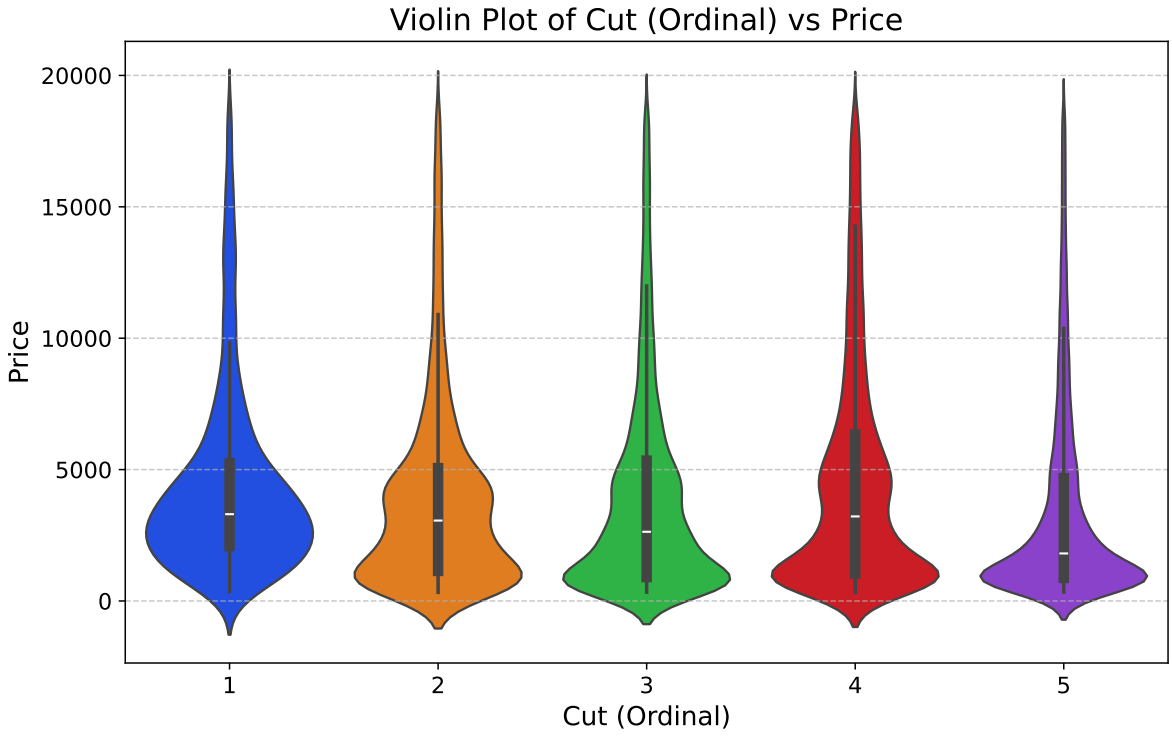
mid-range. Long tail extending to the right highlights that there are **some very expensive diamonds** in the dataset, which will be considered outliers.

Most diamonds are moderately priced, with a few high-priced diamonds.

Violin Plot

```
# Violin Plot of 'cut_ord' vs 'price'
plt.figure(figsize=(10, 6))
sns.violinplot(x='cut_ord', y='price', data=df, palette='bright')
plt.title('Violin Plot of Cut (Ordinal) vs Price', fontsize=16)
plt.xlabel('Cut (Ordinal)', fontsize=14)
plt.ylabel('Price', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

```
C:\Users\Saba\AppData\Local\Temp\ipykernel_12748\1440253991.py:3: FutureWarning:


Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assig
```



The **Ideal cut** (represented by cut_ord=5 has a **broader and taller violin plot** in the higher price range, indicating that diamonds with an Ideal cut are generally **more expensive**.

For lower cut qualities, such as cut_ord = 1 (Fair cut), the violin plot is **narrower and concentrated at lower prices**, meaning that **diamonds with lower cut quality are generally cheaper.**

Diamonds with a Very Good cut (cut_ord = 3) exhibit a **broad price distribution** from lower to moderately higher prices, indicating significant variation within that cut category.

Some cut_ord = 5 (Ideal cut) diamonds have **thin tails extending into the highest price ranges**, indicating that a few Ideal-cut diamonds are priced much higher than most.

Identify Categorical and Numerical Variables

```
# Identify categorical and numerical columns
categorical_cols = ['cut', 'color', 'clarity']
numerical_cols = ['carat', 'cut_ord', 'clarity_ord']
```

Define Features and Variables

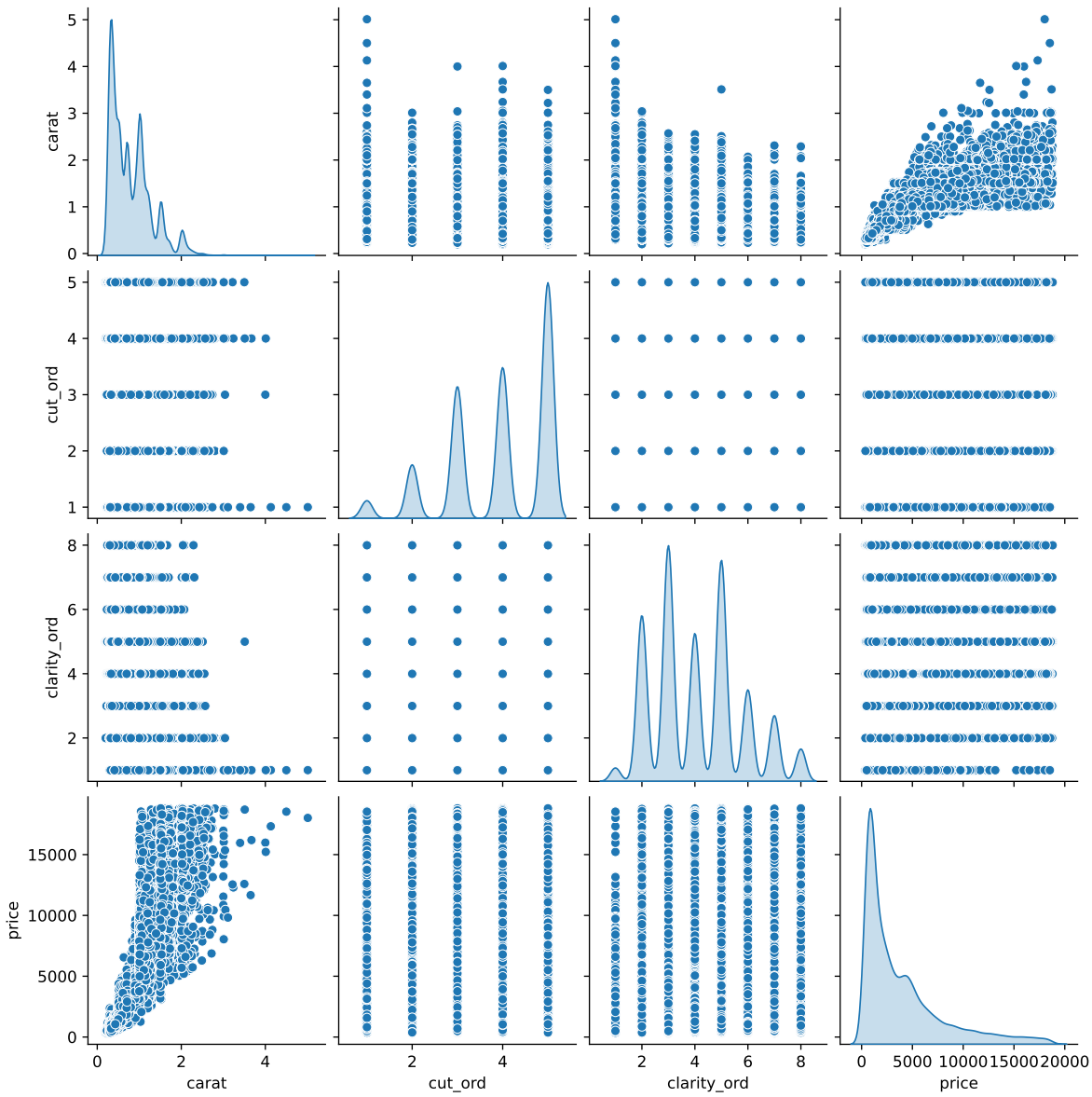
```
# Define features and target variable

# Features
X = df[['carat', 'cut', 'cut_ord', 'color', 'clarity', 'clarity_ord']]

# Target variable
y = df['price']
```

Pair Plot (for Feature Relationships)

```
# Scatter plot of the dataset
sns.pairplot(df[['carat', 'cut_ord', 'clarity_ord', 'price']], diag_kind='kde', palette='pas
plt.suptitle('Pair Plot of Features', y=1.02, fontsize=1)
plt.show()
```

```
C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:
```

```
Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.

C:\Users\Saba\AppData\Local\Programs\Python\Python312\Lib\site-packages\seaborn\axisgrid.py:

Ignoring `palette` because no `hue` variable has been assigned.
```



The scatter plot between carat and price shows a **strong positive correlation**. As carat weight increases, prices rise significantly. Larger diamonds (higher carat values) are typically much more expensive. Diamonds with carats greater than 1.5 tend to have much higher prices compared to smaller diamonds.

The scatter plot between cut_ord and price shows a **weaker correlation** compared to carat, but still suggests that diamonds with better cuts (higher cut_ord values) tend to be more expensive. However, there is **more variability** in prices within the same cut category. Some

diamonds with lower cut quality (e.g., cut_ord = 2) can still be relatively expensive, suggesting that other factors like carat may have more influence.

The relationship between clarity and price is weaker, with more scattered points. This suggests that clarity does not have a strong impact on price when compared to carat or cut. Example: Diamonds with higher clarity (clarity_ord = 7) do not show a significant increase in price compared to those with moderate clarity.

The distribution of carat shows that most diamonds in the dataset are **smaller**, and only a few are larger. This matches the right-skewed distribution seen in the price.

Convert categorical columns

```
# Convert categorical columns to dummy variables
X = pd.get_dummies(X, drop_first=True)
```

Split Dataset

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Linear Regression

```
# Linear Regression

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
lin_reg_predictions = lin_reg.predict(X_test)

# Evaluation
print("Linear Regression MSE:", mean_squared_error(y_test, lin_reg_predictions))
print("Linear Regression R^2:", r2_score(y_test, lin_reg_predictions))
```

```
Linear Regression MSE: 1300915.5847875
Linear Regression R^2: 0.9158762211084442
```

Polynomial Regression

```
# Polynomial Regression

from sklearn.preprocessing import PolynomialFeatures

# Create polynomial features
degree = 2  # Set the degree of the polynomial
poly = PolynomialFeatures(degree=degree)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Fit the Linear Regression model on polynomial features
poly_reg = LinearRegression()
poly_reg.fit(X_train_poly, y_train)

# Predictions
poly_predictions = poly_reg.predict(X_test_poly)

# Evaluation
mse = mean_squared_error(y_test, poly_predictions)
r2 = r2_score(y_test, poly_predictions)

# Display results
print("Polynomial Regression MSE:", mse)
print("Polynomial Regression R²:", r2)
```

```
Polynomial Regression MSE: 604729.8695419376
Polynomial Regression R²: 0.9608951092374106
```

Ridge Regression

```
# Ridge Regression

from sklearn.linear_model import Ridge

ridge_reg = Ridge(alpha=1.0)  # Adjust alpha as needed
ridge_reg.fit(X_train, y_train)
ridge_predictions = ridge_reg.predict(X_test)

# Evaluation
```

```
print("Ridge Regression MSE:", mean_squared_error(y_test, ridge_predictions))
print("Ridge Regression R^2:", r2_score(y_test, ridge_predictions))
```

```
Ridge Regression MSE: 1300566.1602876715
Ridge Regression R^2: 0.9158988166632263
```

Lasso Regression

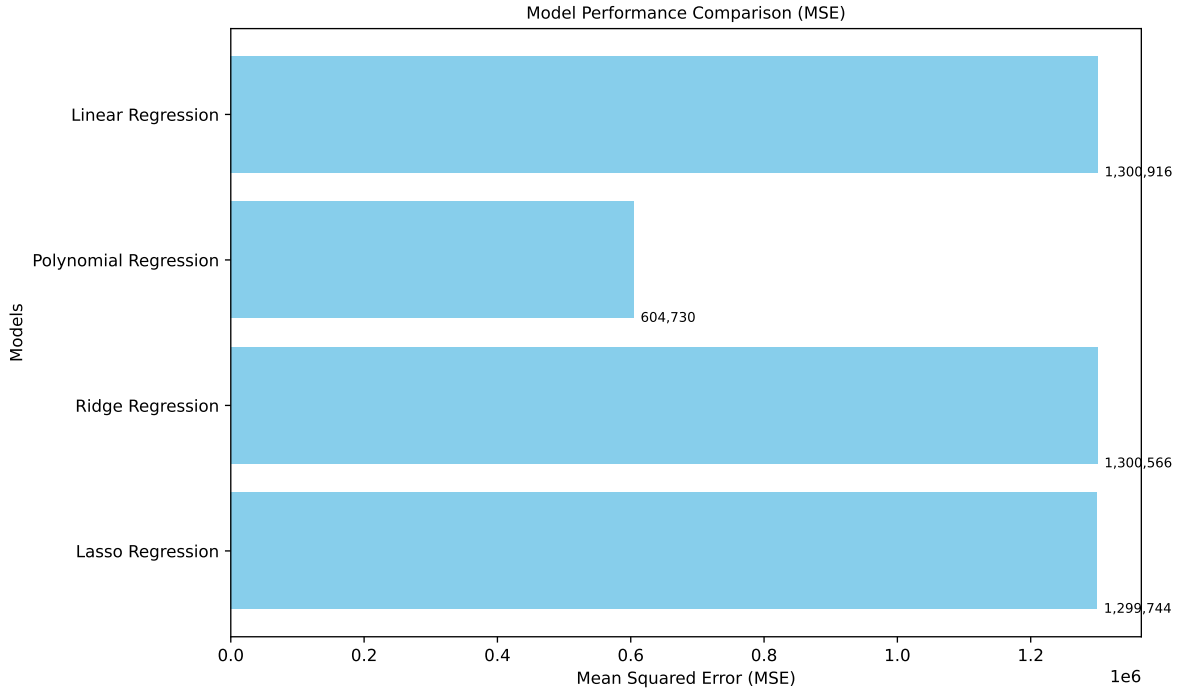
```
# Lasso Regression

from sklearn.linear_model import Lasso

lasso_reg = Lasso(alpha=1.0)  # Adjust alpha as needed
lasso_reg.fit(X_train, y_train)
lasso_predictions = lasso_reg.predict(X_test)

# Evaluation
print("Lasso Regression MSE:", mean_squared_error(y_test, lasso_predictions))
print("Lasso Regression R^2:", r2_score(y_test, lasso_predictions))
```

```
Lasso Regression MSE: 1299743.8320066174
Lasso Regression R^2: 0.915951992567413
```

MSE Plot (Mean Squared Error comparison for models)

```
import matplotlib.pyplot as plt

# Model names and corresponding MSE values
models = ['Linear Regression', 'Polynomial Regression', 'Ridge Regression', 'Lasso Regressio
mse_values = [1300915.58, 604729.87, 1300566.16, 1299743.83]

# Create a bar plot for MSE values
plt.figure(figsize=(10, 6))
bars = plt.barh(models, mse_values, color='skyblue')

# Add title and labels
plt.title('Model Performance Comparison (MSE)', fontsize=10)
plt.xlabel('Mean Squared Error (MSE)', fontsize=10)
plt.ylabel('Models', fontsize=10)
```

```
# Display MSE values on each bar for better readability
for bar in bars:
    plt.text(bar.get_width() + 10000, bar.get_y() + bar.get_height() / 1,
             f'{bar.get_width():,.0f}', va='center', fontsize=8)

# Invert y-axis to have the best model at the top
plt.gca().invert_yaxis()

# Automatically adjust subplot parameters for better fit
plt.tight_layout()

# Show the plot
plt.show()
```



Polynomial Regression has the **lowest MSE** value of **604,729.87**. This indicates that it provides the best fit for the data among the models tested, suggesting that it captures the underlying patterns more effectively than the other models.

Linear Regression and **Ridge Regression** exhibit significantly higher MSE values (**1,300,915.58** and **1,300,566.16**, respectively). This indicates that these models do not fit the data as well, and their predictions are less accurate compared to the Polynomial Regression model.

Lasso Regression has an MSE of **1,299,743.83**, which is slightly better than Linear and Ridge Regression but still falls short of the performance of Polynomial Regression.

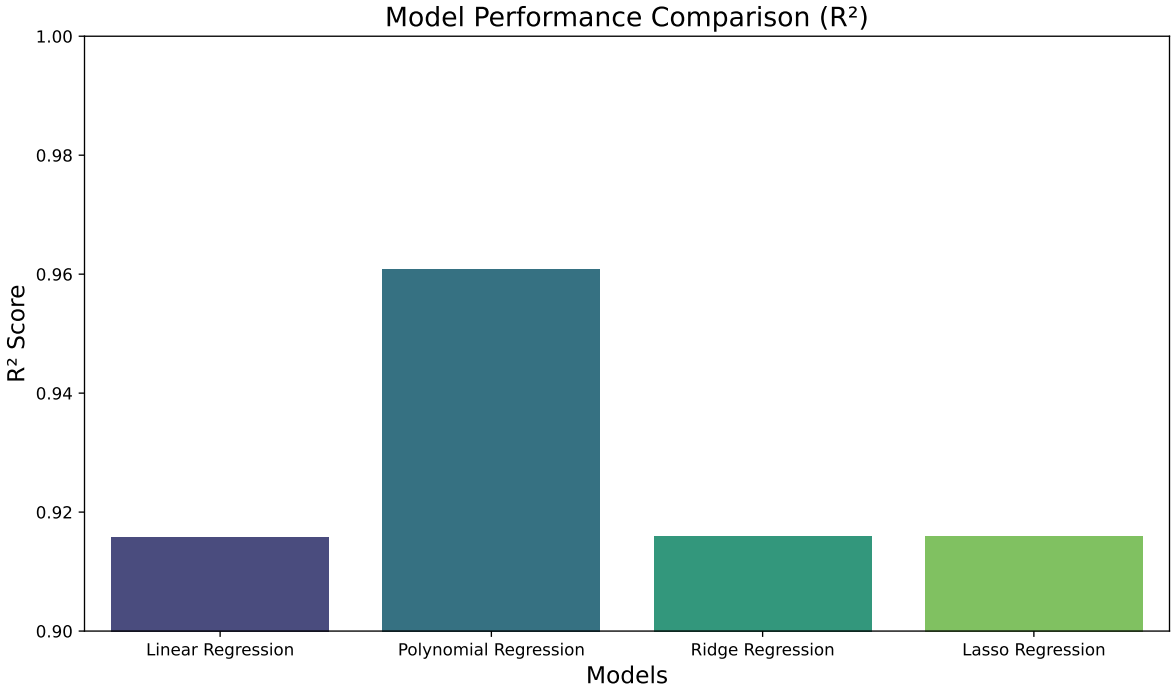
The higher MSE values for Linear and Ridge Regression suggest potential **underfitting**—indicating that these models might be too simple to capture the complexity of the relationship between features and the target variable (price).

The **lower MSE of Polynomial Regression** highlights the importance of selecting a more complex model when dealing with non-linear relationships in the data.

R^2 Bar Chart (R-squared values comparison for models)

```
import seaborn as sns

# R² values for the models
r2_values = [0.9158, 0.9609, 0.9159, 0.9160]

# Create a bar plot for R² values
plt.figure(figsize=(10, 6))
sns.barplot(x=models, y=r2_values, palette='viridis')
plt.title('Model Performance Comparison (R²)', fontsize=16)
plt.ylabel('R² Score', fontsize=14)
plt.xlabel('Models', fontsize=14)
plt.ylim(0.9, 1)  # Setting limit for R² to focus on differences

# Automatically adjust subplot parameters for better fit
plt.tight_layout()

# Show the plot
plt.show()
```

```
C:\Users\Saba\AppData\Local\Temp\ipykernel_12748\3535816124.py:8: FutureWarning:


Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assig
```



Polynomial Regression achieves the highest R^2 value of **0.9609**, meaning it explains approximately **96.09%** of the variance in diamond prices. This demonstrates that it is highly effective for this dataset.

All models have R^2 values above **0.90**, indicating that they explain a substantial portion of the variance in diamond prices. However, **Linear Regression** and **Ridge Regression** show the lowest R^2 values (**0.9158** and **0.9159**, respectively), which suggests they are less effective than Polynomial Regression.

Lasso Regression achieves an R^2 of **0.9160**, slightly better than Linear and Ridge Regression, indicating that it captures some additional variance but still does not match the performance of Polynomial Regression.

The high R^2 values across all models indicate that they all have a reasonable fit to the data, but the differences highlight the superiority of the **Polynomial Regression model** in this analysis.

The relatively lower R^2 values for Linear and Ridge Regression suggest that these models might benefit from incorporating more complex relationships or interactions between variables to improve their predictive accuracy.

Polynomial Regression consistently outperforms the other models in both metrics (MSE and R^2), suggesting it is the best choice for this dataset. **Linear and Ridge Regression** models may be too simplistic for the underlying complexity in the data, as indicated by their

higher MSE values and lower R^2 scores. The **Lasso Regression** provides a slight improvement over Linear and Ridge, but still lags Polynomial Regression.