Converting Categorical Values into Numerical ones

Topics and Outcomes

- Converting categorical values into numerical values

Introduction

When working with machine learning models, especially linear regression or most algorithms, **categorical variables** need to be converted into **numerical** ones because models usually require numeric input to compute distances, apply mathematical transformations, etc.

We present belowe some common techniques to convert categorical variables into numerical ones, along with examples using `scikit-learn`.

1. One-Hot Encoding

One-hot encoding creates binary (0 or 1) variables for each category. Each category gets its own column, and the value is 1 if the category is present in that sample, otherwise 0.

Example with Scikit-Learn

In [32]:

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Sample dataset with a categorical variable
data = {'Product': ['A', 'B', 'C', 'A', 'B', 'C', 'A']}
df = pd.DataFrame(data)

# Initialize OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)

# Apply the encoder to the 'Product' column
encoded_data = encoder.fit_transform(df[['Product']])

# Get the new column names from the encoder
column_names = encoder.get_feature_names_out(['Product'])

# Convert to DataFrame for better visualization
encoded_df = pd.DataFrame(encoded_data, columns=column_names)

print(encoded_df)
```

```
   Product_A  Product_B  Product_C
0        1.0        0.0        0.0
1        0.0        1.0        0.0
2        0.0        0.0        1.0
3        1.0        0.0        0.0
4        0.0        1.0        0.0
5        0.0        0.0        1.0
6        1.0        0.0        0.0
```

Pros:

- Simple and effective for categories with no ordinal relationship.
- Widely used in many machine learning models.

Cons:

- Can result in high dimensionality when there are many categories (curse of dimensionality).
- Doesn't capture the relationship between categories (e.g., rank or order).

2. Label Encoding

Label encoding assigns a unique integer to each category. This method is most appropriate when the categorical variables are **ordinal** (i.e., the categories have a meaningful order).

Example with Scikit-Learn

In [33]:
```
from sklearn.preprocessing import LabelEncoder

# Sample dataset with a categorical variable
data = {'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small', 'Large', 'Me
df = pd.DataFrame(data)

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Apply the encoder to the 'Size' column
df['Size_Encoded'] = label_encoder.fit_transform(df['Size'])

print(df)
```

```
     Size  Size_Encoded
0   Small             2
1  Medium             1
2   Large             0
3  Medium             1
4   Small             2
5   Large             0
6  Medium             1
```

Pros:

- Efficient and simple for ordinal variables where the order is meaningful.

- Doesn’t increase dimensionality.

Cons:

- Can mislead models when used on nominal variables (no inherent order), as models might assume a ranking based on the integer labels.

3. Ordinal Encoding

Ordinal encoding is similar to label encoding but explicitly used when the categories have a natural order (like `Low < Medium < High`).

Example with Scikit-Learn

In [34]:
```
from sklearn.preprocessing import OrdinalEncoder

# Sample dataset with an ordinal variable
data = {'Education_Level': ['High School', 'Bachelor', 'Master', 'PhD']}
df = pd.DataFrame(data)

# Define the ordering of the categories
education_order = [['High School', 'Bachelor', 'Master', 'PhD']]

# Initialize OrdinalEncoder
ordinal_encoder = OrdinalEncoder(categories=education_order)

# Apply the encoder to the 'Education_Level' column
df['Education_Level_Encoded'] = ordinal_encoder.fit_transform(df[['Educatior

print(df)
```

```
  Education_Level  Education_Level_Encoded
0     High School                      0.0
1        Bachelor                      1.0
2          Master                      2.0
3             PhD                      3.0
```

Pros:

- Maintains the order of categories.
- Works well with features that have an inherent ranking (ordinal data).

Cons:

- Should not be used for nominal data (no inherent order).
- May introduce assumptions about the relationship between categories that are not accurate.

4. Target Encoding (Mean Encoding)

Target encoding replaces each category with the mean of the target variable for that category. For example, if you're predicting house prices, and the "city" is a categorical feature, you could replace each city with the average house price in that city.

Example (Manual Implementation)

In [35]:
```python
# Sample dataset with categorical variable and target variable (Sales)
data = {'City': ['CityA', 'CityB', 'CityA', 'CityC', 'CityB', 'CityA'],
        'Sales': [200, 300, 250, 400, 320, 280]}
df = pd.DataFrame(data)

# Calculate the mean target value for each category
city_mean_encoding = df.groupby('City')['Sales'].mean()

# Apply mean encoding to the 'City' column
df['City_Encoded'] = df['City'].map(city_mean_encoding)

print(df)
```

```
    City  Sales  City_Encoded
0  CityA    200    243.333333
1  CityB    300    310.000000
2  CityA    250    243.333333
3  CityC    400    400.000000
4  CityB    320    310.000000
5  CityA    280    243.333333
```

Pros:

- Reduces dimensionality, which is beneficial for categorical variables with many unique values.
- Captures the relationship between the categorical variable and the target variable.

Cons:

- Can lead to **data leakage** if not done properly, meaning the model might learn from information it wouldn't have during prediction.
- Needs to be used carefully, especially in cross-validation.

5. Frequency or Count Encoding

This method replaces each category with the count (or frequency) of its occurrences in the dataset. It’s useful when the distribution of categories might have an impact on the target variable.

Example (Manual Implementation)

In [36]:
```python
# Sample dataset with a categorical variable
data = {'Product': ['A', 'B', 'C', 'A', 'B', 'C', 'A']}
df = pd.DataFrame(data)
```

```
# Count the occurrences of each category
product_counts = df['Product'].value_counts()

# Apply count encoding to the 'Product' column
df['Product_Encoded'] = df['Product'].map(product_counts)

print(df)
```

```
  Product  Product_Encoded
0       A                3
1       B                2
2       C                2
3       A                3
4       B                2
5       C                2
6       A                3
```

Pros:

- Simple to implement.
- Helps capture the impact of category frequency on the target variable.

Cons:

- Doesn't capture relationships between categories or their intrinsic properties.
- Might not work well if counts don’t represent a meaningful relationship with the target.

Comparison and Use Cases

Method	Use Case	Advantages	Disadvantages
One-Hot Encoding	Nominal variables with no inherent order	Works well with nominal data	Increases dimensionality
Label Encoding	Ordinal variables with inherent order	Simple to implement	Misleading for nominal variables
Ordinal Encoding	Ordinal variables with meaningful ranks	Preserves order of categories	Assumes linear relationship
Target Encoding	Categorical variables where a relationship with target exists	Can capture complex relationships	Prone to data leakage
Frequency/Count Encoding	High-cardinality categorical variables	Reduces dimensionality	May not capture category meaning

---

Conclusion

Choosing the right encoding method depends on the type of categorical variable (nominal or ordinal), the relationship between the feature and target, and the characteristics of the dataset.

For example:

- **One-Hot Encoding** works best for variables with no natural order (e.g., Product types).
- **Label or Ordinal Encoding** is useful when there's a clear order (e.g., Education levels).
- **Target Encoding** can be helpful when a category has a strong influence on the target variable, but you need to be careful about overfitting.

Practical Example

Predicting Apartment Price based on Appartment Features

In this example, we'll predict **price** of an apartment based on its area size, number of rooms, age of the building, floor number,**and we will convert the city from categorical into numerical value using one hot encoding method.**

Step 1: Import Libraries

Step 1: Import Libraries and Open the dataset

In [37]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_scor
import matplotlib.pyplot as plt

df = pd.read_csv("../datasets/apartment_prices.csv")
df[df['Price'] < 0]
```

Out[37]:

Square_Area	Num_Rooms	Age_of_Building	Floor_Level	City	Price

Step 2: One-Hot Encoding the Categorical Variable

To properly include the categorical **'Region'** feature, we need to convert it to a numerical format using **One-Hot Encoding**.

In [38]:
```
from sklearn.preprocessing import OneHotEncoder

# Initialize OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)

# Apply the encoder to the 'City' column
encoded_city = encoder.fit_transform(df[['City']])
```

```
# Get the new column names for the encoded 'Region' variable
city_encoded_df = pd.DataFrame(encoded_city, columns=encoder.get_feature_nam

# Combine the original dataset with the encoded 'Region' variable
df = pd.concat([df, city_encoded_df], axis=1)

# Drop the original 'Region' column as it's now encoded
df = df.drop('City', axis=1)

# Display the updated DataFrame with one-hot encoded regions
df
```

Out[38]:

	Square_Area	Num_Rooms	Age_of_Building	Floor_Level	Price	City_Amman	City_
0	162	1	15	12	74900.0	1.0	
1	152	5	8	8	79720.0	0.0	
2	74	3	2	8	43200.0	0.0	
3	166	1	3	18	69800.0	0.0	
4	131	3	14	15	63160.0	0.0	
...	...	...	...	...	...	...	
495	177	1	6	12	64100.0	0.0	
496	79	5	9	13	52700.0	0.0	
497	106	3	7	14	60160.0	0.0	
498	108	3	9	18	72600.0	1.0	
499	73	1	18	6	19280.0	0.0	

500 rows × 8 columns

Step 3: Define Features and Target Including the Encoded Variables

Now, the dataset includes the **one-hot encoded** region columns along with the original advertising spend features. We will include these encoded columns in our feature set for the model.

In [39]:

```
# Features and Target
X = df[['Square_Area', 'Num_Rooms', 'Age_of_Building','Floor_Level','City_Am
y = df['Price']  # Dependent variable (Sales)

df['Log_Price'] = np.log(df['Price'])
y = df['Log_Price']

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
```

Step 4: Train the Model

We can now train the model using the expanded feature set, which includes both numerical and the one-hot encoded categorical variables.

In [40]:
```
# Initialize the Linear Regression model
model = LinearRegression()

# Train the model on the training data
model.fit(X_train, y_train)

# Coefficients and Intercept
print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)
```

```
Coefficients: [ 0.00633132  0.08428514 -0.01736378  0.01640284  0.17079697 -
0.1491896
 -0.02160737]
Intercept: 9.931971496910505
```

Step 5: Make Predictions and Evaluate the Model

Evaluate the model's performance using the test set.

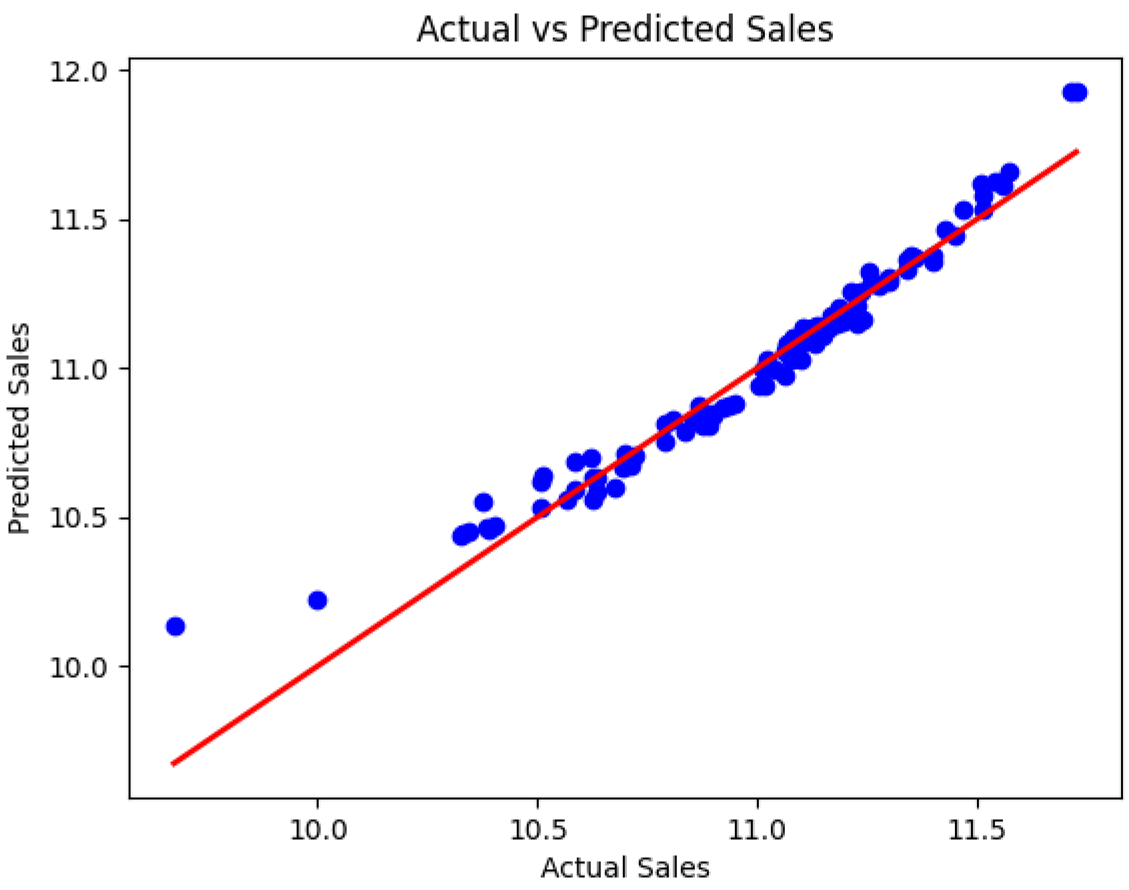
In [41]:
```
# Predict the target variable for the test set
y_pred = model.predict(X_test)

# Evaluate the model using MSE, MAE, and R²
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", round(mse,2))
print("Mean Absolute Error (MAE):", round(mae,2))
print("R-squared (R²):", round(r2,2))
```

```
Mean Squared Error (MSE): 0.01
Mean Absolute Error (MAE): 0.05
R-squared (R²): 0.95
```

Step 6: Visualizing Performance

Finally, we can plot the actual vs predicted values to visualize the model's accuracy.

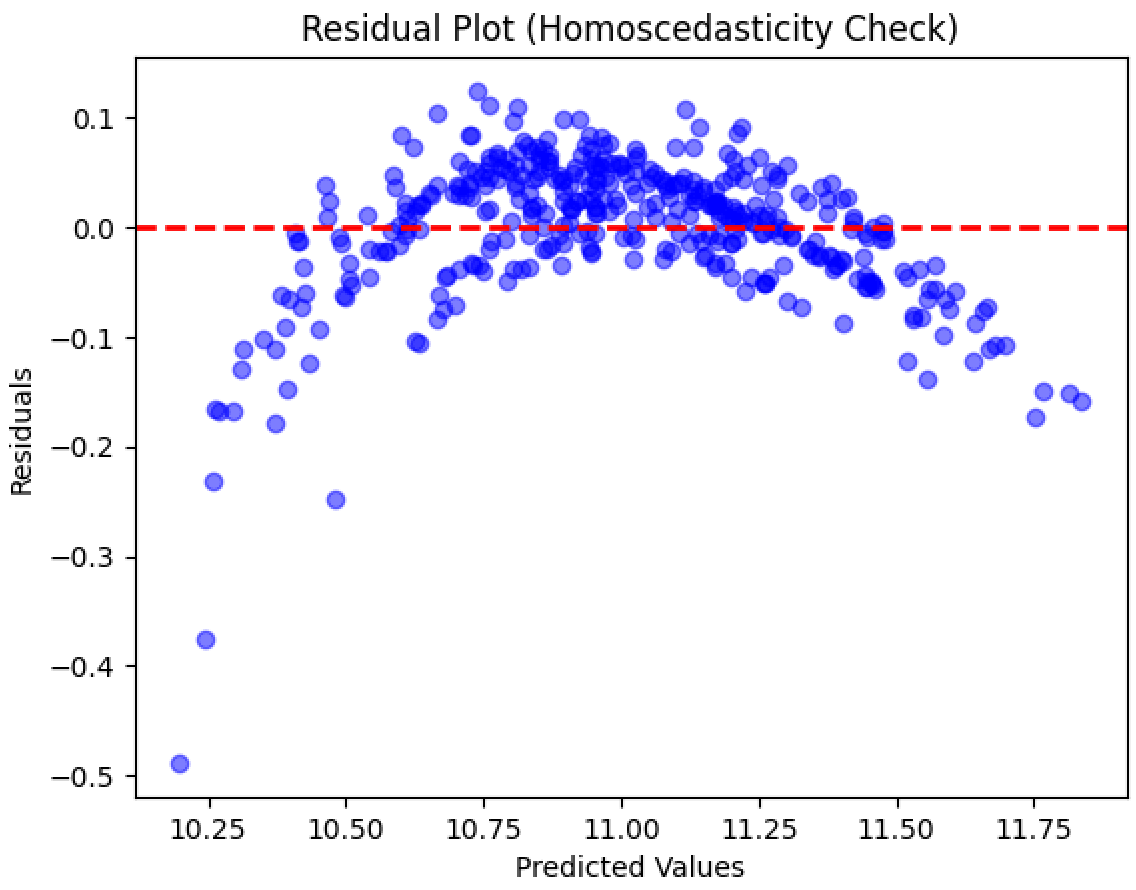
In [42]:
```
# Plot actual vs predicted values
plt.scatter(y_test, y_pred, color='blue')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red'
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Actual vs Predicted Sales')
plt.show()
```



Impact of Including the Categorical Variable

The inclusion of the **city** variable improved the model's performance, as prices are affected by the geographical location. By **one-hot encoding** the city, we allowed the model to account for differences in prices patterns across regions.

In [43]:
```
import matplotlib.pyplot as plt
import numpy as np

# Predict the target variable for the training set
y_train_pred = model.predict(X_train)

# Calculate residuals
residuals = y_train - y_train_pred

# Plot residuals vs. predicted values
plt.scatter(y_train_pred, residuals, color='blue', alpha=0.5)
plt.axhline(y=0, color='red', linestyle='--', linewidth=2)
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residual Plot (Homoscedasticity Check)")
plt.show()
```



The residual plot has a **non-random pattern** with a noticeable **curvature** or **arch shape**. Here's what this shape indicates about the model:

Interpretation of the Residual Plot

1. **Curvature in the Residuals**:
   - The upward and downward curve in the residuals around the horizontal axis suggests a **non-linear relationship** between the predictors and the target variable.
   - This shape indicates that a **linear model might not be capturing the true underlying relationship** in the data.
   - **The residual plot is telling us that there is likely un-captured information in the current features that the linear model is not effectively modeling.**
2. **Violation of Linearity Assumption**:
   - Linear regression assumes a linear relationship between the independent variables and the dependent variable.
   - The arching pattern here shows that the relationship may be **non-linear**, which violates this assumption.
3. **Possible Remedies**:
   - **Add Polynomial or Interaction Terms**: Introduce non-linear terms (e.g., squared or cubed terms for predictors) to capture the non-linear relationship.

- **Try a Non-Linear Model**: Consider using a non-linear regression model, such as decision trees, random forests, or gradient boosting, which can capture complex relationships.
- **Transform the Target Variable**: If the target variable has a skewed distribution, applying a transformation (e.g., logarithmic) might help.

Next Steps

- We could start by adding polynomial terms (e.g., `Square_Area^2`) or interaction terms between predictors to see if they help reduce the curvature in the residuals.
- Alternatively, experiment with non-linear models, as they are generally more flexible and can capture relationships that linear models cannot.

This residual pattern strongly suggests that the current linear model is not well-suited to the data and would benefit from additional complexity to account for the non-linearity.