Report of Assignment report

Introduction

I created a new interaction features from existing numerical variables in the dataset, focusing on how these features are expected to enhance the predictive performance of the model.

Problem 1:

1. **Categorization of Temperature (`temp_range` Feature)**
    - **Rationale**: Temperature is a continuous variable, but its relationship with the target variable might not be linear. By categorizing temperature into discrete ranges (`cold`, `cool`, `moderate`, `warm`, `hot`), we can capture non-linear interactions between temperature and other variables.
    - **Implementation**: A function `categorize_temp` was implemented to convert the continuous temperature values into categorical labels:

```python
def categorize_temp(temp):
    if temp < 0.2:
        return 'cold'
    elif 0.2 <= temp < 0.4:
        return 'cool'
    elif 0.4 <= temp < 0.6:
        return 'moderate'
    elif 0.6 <= temp < 0.8:
        return 'warm'
    else:
        return 'hot'
```

    - **Expected Improvement**: Categorizing temperature may help in identifying thresholds beyond which the relationship with the target variable significantly changes, improving the model's ability to make accurate predictions.
2. **Identification of Rush Hours (`rush_hour` Feature)**

- **Rationale**: Traffic congestion and other time-dependent factors often vary significantly during rush hours compared to non-rush hours. Capturing this temporal aspect can provide insights that improve the model's predictive capabilities.
- **Implementation**: A function `categorize_rush_hour` was created to classify hours into `rush_hour` and `non_rush_hour`:

```python
def categorize_rush_hour(hr):
    if 7 <= hr <= 9 or 16 <= hr <= 19:
        return 'rush_hour'
    else:
        return 'non_rush_hour'
```

- **Expected Improvement**: This feature captures the impact of rush hour on the target variable, potentially highlighting time periods where certain behaviors or outcomes are more prevalent.

3. **Creation of Day/Night Feature (`day_night`)**
    - **Rationale**: The time of day can significantly influence many phenomena, such as energy consumption, traffic, or human activity. By creating a `day_night` feature, the model can differentiate between behaviors that occur during the day versus the night.
    - **Implementation**: A new feature `day_night` was generated by categorizing the hour of the day:

```python
data['day_night'] = data['hr'].apply(lambda x: 'day' if 6 <= x <= 18 else 'night')
```

    - **Expected Improvement**: This feature may help in capturing patterns specific to day and night, thereby enhancing model accuracy.

Problem 2

1. **Numerical Feature Processing**
    - **Features**: `temp`, `hum`, `windspeed`
    - **Pipeline**: A numerical pipeline was applied that involved imputing missing values using the mean and scaling the features using MinMaxScaler to ensure they are within the same range.
    - **Implementation**:

```python
numerical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', MinMaxScaler())])
X[numerical_features] =
numerical_pipeline.fit_transform(X[numerical_features]
)
```

2. **Categorical Feature Processing**
    - **Features**: `season`, `weathersit`, `day_night`, `temp_range`, `rush_hour`
    - **Pipeline**: Instead of using OneHotEncoder, which converts categorical variables into binary columns, TargetEncoder was used.
    - **Implementation**:

```python
categorical_pipeline = Pipeline([
    ('imputer',
SimpleImputer(strategy='most_frequent')),
    ('target_encoder', TargetEncoder())
])
X_encoded =
categorical_pipeline.fit_transform(X[categorical_featu
res], y)
X = X.drop(columns=categorical_features)
X = X.join(X_encoded)
X.columns = X.columns.astype(str)
```

Impact on Model Performance

- **OneHotEncoder**: This approach generates an array of binary columns for each category, resulting in a sparse matrix, particularly when dealing with high-cardinality characteristics. Despite its effective capture of interactions between categories, this approach can lead to overfitting and higher computational expenses.
- **TargetEncoder**: The TargetEncoder algorithm decreases the dimensionality of the dataset by substituting categories with their respective target mean. This reduction in dimensionality helps to avoid overfitting and optimize model performance. This is especially advantageous when handling categorical features with high-cardinality.

Evaluation

- **Model Comparison**: After applying TargetEncoder, the model's performance was evaluated against the performance using OneHotEncoder. The evaluation metrics (e.g., accuracy, RMSE, AUC) showed whether the reduced dimensionality and preserved ordinal relationship from **TargetEncoder** led to better generalization and improved prediction accuracy.

Result [RandomForestRegressor]

- **OneHotEncoder**:
  - **MSE**: 42.46
  - **R^2**: 0.943
- **TargetEncoder**
  - **MSE**: 41.80
  - **R^2**: 0.944

Problem 3

1. **Training Using `sklearn` Package**:
   - **Implementation**: The `LinearRegression` class from `sklearn` was employed to fit the model on the training data.
   - **Prediction**: The model was used to predict the target values for the test set.
   - **Metrics**: MSE and R^2 were computed to evaluate model performance.

```python
from sklearn.linear_model import LinearRegression

sklearn_model = LinearRegression()
sklearn_model.fit(X_train, y_train)
y_pred_sklearn = sklearn_model.predict(X_test)

mse_sklearn = mean_squared_error(y_test, y_pred_sklearn)
r2_sklearn = r2_score(y_test, y_pred_sklearn)
```

2. **Training from Scratch**:
   - **Implementation**:
     - Added a bias term (a column of ones) to the feature matrix.
     - Computed the coefficients (theta) using the Normal Equation.
     - Predicted the target values for the test set using the computed coefficients.
   - **Metrics**: MSE and R^2 were calculated similarly to the `sklearn` model.

```python
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score

X_train_b = np.hstack([np.ones((X_train.shape[0], 1)),
X_train])
X_test_b = np.hstack([np.ones((X_test.shape[0], 1)),
X_test])

theta_best = np.linalg.inv(X_train_b.T @ X_train_b) @
X_train_b.T @ y_train
y_pred_scratch = X_test_b @ theta_best

mse_scratch = mean_squared_error(y_test, y_pred_scratch)
r2_scratch = r2_score(y_test, y_pred_scratch)
```

Mathematical Explanation for Linear Regression Implementation from Scratch

The Linear Regression model aims to find the best-fitting line through a dataset to predict the target variable. The mathematical steps involved in implementing linear regression from scratch:

1. Adding the Bias Term

In linear regression, the model is typically represented as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Where:

- y is the target variable.
- β_0 is the bias term (intercept).
- β_1, β_2, ..., β_n are the coefficients for the features $x_1, x_2, \ldots, x_n$.

To simplify the calculation and include the bias term in matrix operations, I added a column of ones to the feature matrix X . This allows to represent of the bias term as part of the coefficient vector.

Implementation:

```python
X_train_b = np.hstack([np.ones((X_train.shape[0], 1)),
X_train])
X_test_b = np.hstack([np.ones((X_test.shape[0], 1)), X_test])
```

Here:

- `np.ones((X_train.shape[0], 1))` creates a column vector of ones (bias term) with the same number of rows as `X_train`.
- `np.hstack` horizontally stacks this column with the original feature matrix `X_train`.

2. Computing the Coefficients (Theta) Using the Normal Equation

The Normal Equation provides a closed-form solution to finding the optimal coefficients θ for the linear regression model. It is derived from minimizing the cost function, which is the sum of squared differences between the predicted values and the actual values.

Normal Equation Formula:

$$\theta = (X^T X)^{-1} X^T y$$

Where:

- X is the feature matrix (with the bias term added).
- y is the vector of target values.
- θ is the vector of coefficients (including the bias term).

Implementation:

```python
theta_best = np.linalg.inv(X_train_b.T @ X_train_b) @
X_train_b.T @ y_train
```

Here:

- `X_train_b.T` computes the transpose of `X_train_b`.
- `X_train_b.T @ X_train_b` computes the dot product of the transposed feature matrix and the feature matrix itself.
- `np.linalg.inv(...)` calculates the inverse of the resulting matrix.
- `@ X_train_b.T @ y_train` performs matrix multiplication to obtain the coefficients.

3. Making Predictions

Once we have the coefficients (θ), I can make predictions for new data using the equation:

$$\hat{y} = X\theta$$

Where $\hat{y}$ is the predicted target variable.

Implementation:

```python
y_pred_scratch = X_test_b @ theta_best
```

Here:

- `X_test_b` is the test feature matrix (with the bias term included).
- `@ theta_best` performs matrix multiplication to obtain the predicted values.

4. Calculating Performance Metrics

Finally, I calculated the Mean Squared Error (MSE) and R-squared (R^2) to evaluate the model's performance.

- **MSE**: Measures the average squared difference between the predicted and actual values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **R^2**: Measures the proportion of variance explained by the model.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

  Where $\bar{y}$ is the mean of the actual values.

Implementation:

```python
mse_scratch = mean_squared_error(y_test, y_pred_scratch)
r2_scratch = r2_score(y_test, y_pred_scratch)
```

Results

- **Sklearn Linear Regression**:

    - **MSE**: 12264.0873
    - **R^2**: 0.6127
- **Scratch Linear Regression**:
    - **MSE**: 12264.0873
    - **R^2**: 0.6127

The near-perfect match in performance metrics between the `sklearn` implementation and the model trained from scratch demonstrates the correctness of the manual approach.

ML Pipeline