Feature Engineering Report

1. Importing Libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import zscore
from statsmodels.tsa.stattools import adfuller
from sklearn.preprocessing import LabelEncoder
import seaborn as sns
```

- **pandas:** A library for data manipulation and analysis, used for handling and cleaning data.
- **numpy**: A library for numerical operations, especially with arrays and mathematical operations.
- **matplotlib** and **seaborn**: Used for data visualization, especially for creating plots.
- **scipy.stats.zscore**: Used to calculate the **Z-score** for detecting outliers in the dataset.
- **statsmodels**: Provides tools for statistical models, like the **ADF test** (Augmented Dickey-Fuller) to test stationarity in time-series data.
- **LabelEncoder**: Converts categorical text data into numeric labels.

2. Reading and Preparing the Data:

```
python
CopyEdit
df = pd.read_parquet(r'C:\Users\alqay\OneDrive\Desktop\store-sales-time-
series-forecasting')
df.head()
```

- Here, we load the dataset from a **parquet** file and use `.head()` to display the first 5 rows of the data.

```
python
CopyEdit
df.dropna(inplace=True)
df.info()
```

- `dropna()`: Removes rows containing any missing (NaN) values.
- `df.info()`: Displays information about the DataFrame, including the number of non-null entries and data types of each column.

3. Converting String Columns to Object Type:

```
python
CopyEdit
for col in df.select_dtypes(include='string').columns:
    df[col] = df[col].astype('object')
```

- Here, the code checks for columns with string data types and converts them to **object** type, which is the appropriate type for categorical text data in pandas.

4. Detecting and Removing Outliers:

```
python
CopyEdit
numerical_cols = ['unit_sales', 'transactions']
Q1 = df[numerical_cols].quantile(0.25)
Q3 = df[numerical_cols].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

- The **Interquartile Range (IQR)** is calculated by subtracting the first quartile (Q1) from the third quartile (Q3). This range helps us identify outliers.
- **Lower and Upper Bounds** are calculated using the formula:

  Lower Bound=Q1−1.5×IQRandUpper Bound=Q3+1.5×IQR\text{Lower Bound} = Q1 - 1.5 \times \text{IQR} \quad \text{and} \quad \text{Upper Bound} = Q3 + 1.5 \times \text{IQR}Lower Bound=Q1−1.5×IQRandUpper Bound=Q3+1.5×IQR

```
python
CopyEdit
outliers = (df[numerical_cols] < lower_bound) | (df[numerical_cols] >
upper_bound)
outlier_rows = outliers.any(axis=1)
df = df[~outlier_rows]
```

- **Outliers** are detected by checking if any values in the `unit_sales` or `transactions` columns are outside the calculated bounds.
- We then filter the DataFrame to remove these outliers.

5. Visualizing the Data with Boxplot:

```
python
CopyEdit
sns.boxplot(df)
plt.xticks(rotation=90)
plt.show()
```

- A **Boxplot** is plotted to visualize the distribution of the numerical columns after removing outliers. It helps to identify the spread of data and any remaining extreme values.

6. Adding Time-Based Features:

```
python
CopyEdit
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['day_of_week'] = df['date'].dt.dayofweek
df['quarter'] = df['date'].dt.quarter
df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)
df['is_month_start'] = df['date'].dt.is_month_start.astype(int)
df['is_month_end'] = df['date'].dt.is_month_end.astype(int)
```

- Additional time-related features are added based on the `date` column. These include the **year**, **month**, **day**, **day of the week**, **quarter**, and indicators for whether the date is at the start or end of the month or a weekend.

7. Encoding Categorical Columns with LabelEncoder:

```
python
CopyEdit
categorical_cols = ['family', 'store_type', 'city', 'state', 'day_type',
'Event Scale', 'locale_name']
label_encoders = {}

for col in categorical_cols:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col].astype(str))
    label_encoders[col] = le
```

- **LabelEncoder** is used to convert categorical variables (such as 'family', 'store_type', etc.) into numerical labels so that the machine learning model can process them. These mappings are stored in a dictionary called `label_encoders`.

```
python
CopyEdit
with open('label_encoders.pkl', 'wb') as f:
    pickle.dump(label_encoders, f)
```

- The **label_encoders** are saved to a file using **pickle**, allowing them to be used later for prediction or inference.

8. ADF Test for Stationarity:

```
python
CopyEdit
result = adfuller(df['unit_sales'].dropna())
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')

if result[1] > 0.05:
    print("Data is not stationary; consider differencing or transformation.")
```

```
else:
    print("Data is stationary.")
```

- The **Augmented Dickey-Fuller (ADF) test** is used to test if the time-series data (in this case, `unit_sales`) is stationary.
- If the **p-value** is greater than 0.05, the data is considered non-stationary, and you may need to apply transformations or differencing to make it stationary.

9. Correlation Matrix:

```
python
CopyEdit
correlation_matrix = df.corr()
print("Correlation Matrix:\n", correlation_matrix)

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```

- **Correlation Matrix**: This shows the relationship between numerical features, helping to identify strong correlations.
- A **heatmap** is used to visualize the correlation values, with darker colors representing stronger correlations.

10. Feature Engineering:

- **create_date_features**: Extracts time-based features such as **day**, **week of the year**, **month**, **year**, and whether the date is the start or end of the month.
- **create_sales_lag_features**: Creates lag features for sales data, e.g., sales from 1 day, 7 days, 14 days, and 28 days ago.
- **create_sales_rolling_features**: Creates rolling window features, such as the 7-day, 14-day, and 28-day moving average of sales.
- **create_transaction_features**: Similar to sales lag and rolling features, but for transaction data.
- **create_group_stats**: Adds group statistics for **mean transactions** by family, store, and family/store combinations.

11. Removing Rows with Missing Values for Important Features:

```
python
CopyEdit
data_model = df.dropna(subset=[
    'sales_lag_1', 'sales_lag_7', 'sales_lag_14', 'sales_lag_28',
    'sales_roll_mean_7', 'sales_roll_mean_14', 'sales_roll_mean_28',
    'trans_lag_1', 'trans_lag_7', 'trans_lag_14', 'trans_roll_mean_7',
    'trans_roll_mean_14'
], inplace=True)
```

- Drops rows that have missing values in specific lag and rolling features to ensure the data is clean and ready for model training.

12. Splitting Data for Training and Validation:

```
python
CopyEdit
X = data[features]
y = data[target]

X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2,
shuffle=False)
```

- **train_test_split**: Splits the dataset into training (80%) and validation (20%) sets. The data is not shuffled, which is important for time-series data where temporal order should be maintained.

13. Linear Regression Model:

```
python
CopyEdit
lr = LinearRegression()
lr.fit(X_train, y_train)

y_pred = lr.predict(X_valid)
```

- A **Linear Regression model** is trained on the training data (`X_train` and `y_train`).
- The model then predicts the target (`unit_sales`) on the validation set.

14. Evaluating the Model:

```
python
CopyEdit
rmse = np.sqrt(mean_squared_error(y_valid, y_pred))
r2 = r2_score(y_valid, y_pred)
mean_sales = y_valid.mean()

print("=== Linear Regression Baseline ===")
print(f"RMSE: {rmse:.2f}")
print(f"Mean unit_sales: {mean_sales:.2f}")
print(f"Relative RMSE: {(rmse / mean_sales):.2%}")
print(f"R² Score: {r2:.4f}")
```

- **RMSE (Root Mean Squared Error)**: A metric to measure the prediction error of the model.
- **R^2 Score**: Represents how well the model explains the variance in the target variable.
- The model's **Relative RMSE** shows how the error compares to the average unit sales.

15. Saving the Final Data (optional):

```python
python
CopyEdit
# df.to_parquet(r'C:\Users\alqay\OneDrive\Desktop\store-sales-time-series-forecasting-normalized-final')
```

- Optionally, you can save the final processed DataFrame to a **parquet** file for later use.

Summary:

The code processes time-series sales data, cleaning and engineering features like time-based attributes and lag/rolling features. It then uses a **Linear Regression model** to predict sales and evaluates its performance using **RMSE** and **R^2 score**. It also provides a method for saving and reusing preprocessed data and label encoders.