1. Importing Libraries

```
python
Copy code
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
```

Explanation:

- **`import pandas as pd`**:
  - **Pandas** is a library for data manipulation. We use it to load and manage datasets, perform transformations, and handle missing data.
- **`import numpy as np`**:
  - **NumPy** provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- **`from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV`**:
  - **`train_test_split`**: Used to split the dataset into training and testing sets.
  - **`cross_val_score`**: Performs cross-validation, evaluating the model on different splits to get a more robust performance measure.
  - **`GridSearchCV`**: Performs hyperparameter tuning by exhaustively searching over a specified parameter grid.
- **`from sklearn.linear_model import Ridge, Lasso, ElasticNet`**:
  - These are the models we'll use. Each performs regularized linear regression:
    - **Ridge** uses L2 regularization (penalty on large coefficients).
    - **Lasso** uses L1 regularization (penalizes the absolute value of coefficients).
    - **ElasticNet** is a combination of both L1 and L2 regularization.
- **`from sklearn.metrics import mean_squared_error, r2_score`**:
  - These are the evaluation metrics used to assess model performance.
    - **MSE (Mean Squared Error)** quantifies the error between predicted and actual values.
    - **R^2 (R-squared)** measures the proportion of variance in the dependent variable explained by the model.
- **`from sklearn.preprocessing import StandardScaler`**:
  - **StandardScaler** is used to standardize the features by removing the mean and scaling to unit variance. This is important because models like Ridge and Lasso are sensitive to the scale of the input data.
- **`import matplotlib.pyplot as plt`**:
  - **Matplotlib** is used for visualizations, although it isn't used explicitly in this example. It helps in visualizing data and model performance.

---

2. Loading and Preprocessing Data

```
python
Copy code
df = pd.read_csv("data.csv")
df.replace([np.inf, -np.inf], np.nan, inplace=True)
df.fillna(method='ffill', inplace=True)
```

Explanation:

- **`df = pd.read_csv("data.csv")`**:
  - Loads the dataset into a Pandas DataFrame. The `.csv` file contains the data to be analyzed.
- **`df.replace([np.inf, -np.inf], np.nan, inplace=True)`**:
  - This step replaces any **infinite** values (positive or negative) in the dataset with **NaN** (Not a Number), which is necessary because models cannot handle infinite values directly.
- **`df.fillna(method='ffill', inplace=True)`**:
  - **`fillna`** is used to handle missing values (NaNs). The **`method='ffill'`** argument specifies forward filling, where missing values are replaced with the most recent non-null value in the column.

---

3. Feature Engineering

```
python
Copy code
df['Global_active_power_lag1'] = df['Global_active_power'].shift(1)
df['Global_active_power_lag2'] = df['Global_active_power'].shift(2)
df['Global_active_power_lag3'] = df['Global_active_power'].shift(3)
df['power_ratio'] = df['Global_active_power'] / df['Global_reactive_power']
df.dropna(inplace=True)
```

Explanation:

- **`df['Global_active_power_lag1'] = df['Global_active_power'].shift(1)`**:
  - Creates a lag feature by shifting the column **`Global_active_power`** by 1 row. This captures the value from the previous time step.
- **`df['Global_active_power_lag2'] = df['Global_active_power'].shift(2)`**:
  - Similar to the previous line, but shifts by 2 rows. This helps capture additional temporal dependencies.
- **`df['Global_active_power_lag3'] = df['Global_active_power'].shift(3)`**:
  - Creates another lag feature, this time capturing the value from 3 rows earlier.
- **`df['power_ratio'] = df['Global_active_power'] / df['Global_reactive_power']`**:

    - Creates a new feature that is the ratio of **`Global_active_power`** to **`Global_reactive_power`**. This feature could be useful if there's a relationship between these two variables.
- **`df.dropna(inplace=True)`**:
    - Drops any rows that contain **NaN** values, which could have been introduced by shifting columns for lag features or any missing data.

---

4. Defining Features and Target Variable

```
python
Copy code
X = df[['Global_active_power', 'Voltage', 'Global_reactive_power',
'Global_active_power_lag1', 'Global_active_power_lag2',
'Global_active_power_lag3']]
y = df['power_ratio']
```

Explanation:

- **`X`** represents the **features** (independent variables). It includes the columns:
    - `Global_active_power`, `Voltage`, `Global_reactive_power`, and the lag features we created.
- **`y`** represents the **target variable** (dependent variable), which is `power_ratio`. This is what we are trying to predict.

---

5. Train-Test Split

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

Explanation:

- **`train_test_split`** is used to split the data into **training** and **testing** sets. Here:
    - **`X_train, X_test`** are the feature sets for training and testing.
    - **`y_train, y_test`** are the target variable sets for training and testing.
    - **`test_size=0.2`** means 20% of the data will be used for testing and 80% for training.
    - **`random_state=42`** ensures reproducibility of the split (i.e., you get the same split every time you run the code).

---

6. Standardizing the Data

```
python
Copy code
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Explanation:

- **`StandardScaler()`** creates an instance of the scaler.
  - **`fit_transform(X_train)`**: Calculates the mean and standard deviation of `X_train` and scales the data so that it has a mean of 0 and a standard deviation of 1.
  - **`transform(X_test)`**: Applies the same transformation (mean and standard deviation) to `X_test` without recalculating the parameters.

---

7. Defining Evaluation Function

```
python
Copy code
def evaluate_model_metrics(model, X_test, y_test):
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"Mean Squared Error: {mse}")
    print(f"R² Score: {r2}")
```

Explanation:

- **`evaluate_model_metrics()`**: This function evaluates the performance of a model on the test data.
  - **`model.predict(X_test)`**: Uses the trained model to make predictions on the test data.
  - **`mean_squared_error(y_test, y_pred)`**: Calculates the **Mean Squared Error (MSE)**, a common metric for regression problems.
  - **`r2_score(y_test, y_pred)`**: Calculates the **R^2 score**, a metric indicating how well the model explains the variance in the target variable.

---

8. Fitting and Evaluating Models

```
python
Copy code
ridge.fit(X_train_scaled, y_train)
```

```
print("Ridge Model Evaluation:")
evaluate_model_metrics(ridge, X_test_scaled, y_test)

lasso.fit(X_train_scaled, y_train)
print("\nLasso Model Evaluation:")
evaluate_model_metrics(lasso, X_test_scaled, y_test)

elasticnet.fit(X_train_scaled, y_train)
print("\nElasticNet Model Evaluation:")
evaluate_model_metrics(elasticnet, X_test_scaled, y_test)
```

Explanation:

- **Model Fitting**:
  - **`ridge.fit(X_train_scaled, y_train)`**: Trains the Ridge model using the scaled training data.
  - Similarly, Lasso and ElasticNet are trained with their respective **`fit()`** methods.
- **Model Evaluation**:
  - After training each model, we evaluate it using the `evaluate_model_metrics` function, which computes MSE and R^2 on the test data.

---

9. Cross-Validation for Model Robustness

```
python
Copy code
ridge_cv_scores = cross_val_score(ridge, X_train_scaled, y_train, cv=5,
scoring='neg_mean_squared_error')
lasso_cv_scores = cross_val_score(lasso, X_train_scaled, y_train, cv=5,
scoring='neg_mean_squared_error')
elasticnet_cv_scores = cross_val_score(elasticnet, X_train_scaled, y_train,
cv=5, scoring='neg_mean_squared_error')

print(f"Ridge Cross-Validation MSE: {np.mean(ridge_cv_scores)}")
print(f"Lasso Cross-Validation MSE: {np.mean(lasso_cv_scores)}")
print(f"ElasticNet Cross-Validation MSE: {np.mean(elasticnet_cv_scores)}")
```

Explanation:

- **`cross_val_score()`**: Performs **cross-validation** to assess the model's performance on different subsets of the training data.
  - **`cv=5`** means 5-fold cross-validation (splitting the data into 5 parts and training/evaluating the model 5 times).
  - **`scoring='neg_mean_squared_error'`**: The negative sign is because `cross_val_score()` returns scores where higher is better. In MSE, lower is better, so it returns negative MSE.
- **Mean of Cross-Validation Scores**: We print the mean MSE for each model's cross-validation performance.

10. Hyperparameter Tuning using GridSearchCV

```
python
Copy code
param_grid = {
    'alpha': [0.01, 0.1, 1, 10, 100],
    'fit_intercept': [True, False]
}
ridge_grid_search = GridSearchCV(ridge, param_grid, cv=5,
scoring='neg_mean_squared_error')
ridge_grid_search.fit(X_train_scaled, y_train)
print(f"Best Ridge Parameters: {ridge_grid_search.best_params_}")
```

Explanation:

- **`param_grid`**: Defines the grid of hyperparameters to search over for tuning the model.
  - **`alpha`**: The regularization strength for Ridge regression.
  - **`fit_intercept`**: Determines if the model should include an intercept term.
- **`GridSearchCV`**: Performs an exhaustive search over the specified parameter grid with cross-validation to find the best combination of hyperparameters.