Group Writeup - Machine Learning project - Spring 2024

Anzhelika Belozerova (238852), Isabella Urbano-Trujillo (233239), Camilo Pedraza Jimenez (226679), Luis Fernando Ramirez Ruiz (222819), Milton Mier (223594)

In [28]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
```

1. Context

What is the context? Refine the setup you proposed in the project memo and explain it in a paragraph.

In the context of global energy markets, predicting energy prices is a critical task that influences economic policy, business strategy, and individual decision-making. The European countries present a very interesting case for such analysis due to its diverse energy sources, varying levels of energy dependency, and differing national energy policies. Our project focuses on predicting the energy prices of different countries within Europe using machine learning models.

- Unit of Analysis: Our units of analysis are the hourly energy prices of individual European countries. These prices are measured in Euros per megawatt-hour (€/MWh) and are sourced from historical data provided by governmental and industry databases.
- Prediction Objective: The primary objective of this project is to predict future energy prices for Eropean countries, particularly energ prices 12 hours ahead. Accurate predictions will help in understanding the potential cost fluctuations and the factors driving these changes. This can be particularly valuable for stakeholders such as policymakers, energy companies, and consumers.
- Features: To make these predictions, we use past energy prices, some dummys that describe the period of time such as hour, day, week number, day of week and month, and finally, sources of energy such as biomass and gas.
- Application in Decision-Making: The predictions generated can be used in several decision-making contexts:

1. Policymaking: Governments can use the predictions to formulate energy policies, subsidies, and tariffs to stabilize the market.
2. Strategic Planning for Companies: Energy companies can plan their operations, investments, and pricing strategies based on predicted price trends.
3. Consumer Decision-Making: Businesses and residential consumers can make informed choices about energy consumption, contract negotiations, and investments in energy-efficient technologies.
4. Investment Strategies: Financial analysts and investors can use the predictions to guide investments in energy markets and related sectors.

2. EDA

What are some of the most important patterns in your data that you can observe through exploratory data analysis? For instance, what does the label look like? What do the simple bivariate relationships in the data look like? Will this affect any of your modeling choices

In [29]:
```
df = pd.read_csv('/Users/isdc/Library/CloudStorage/OneDrive-HertieSchool/2024-1/Machine Learning/ML project/dates_cleaned.csv',
```

In [30]:
```
df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d %H:%M:%S', errors='coerce')
```

In [31]:
```
print(df)
```

```
              Date Country  energy_price     biomass          gas  \
0       2015-01-01      CH         44.94  252.200902  1180.283774
1       2015-01-01      CZ         26.48  135.000000   172.000000
2       2015-01-01    DK_1         25.02   18.000000   233.000000
3       2015-01-01    DK_2         27.38   25.000000   304.000000
4       2015-01-01      EE         27.38  252.200902  1180.283774
...            ...     ...           ...         ...         ...
3294775 2023-12-31    SE_2         44.87  252.200902  1180.283774
3294776 2023-12-31    SE_3         44.87  252.200902  1180.283774
3294777 2023-12-31    SE_4         44.87  252.200902  1180.283774
3294778 2023-12-31      SI         33.30  252.200902  1180.283774
3294779 2023-12-31      SK         35.14  252.200902  1180.283774

             nuclear  year  month  day  hour  week_number  day_of_week
0        7185.089448  2015      1    1     0            1            3
1        2596.000000  2015      1    1     0            1            3
2        7185.089448  2015      1    1     0            1            3
3        7185.089448  2015      1    1     0            1            3
4        7185.089448  2015      1    1     0            1            3
...              ...   ...    ...  ...   ...          ...          ...
3294775  7185.089448  2023     12   31     0           52            6
3294776  7185.089448  2023     12   31     0           52            6
3294777  7185.089448  2023     12   31     0           52            6
3294778  7185.089448  2023     12   31     0           52            6
3294779  7185.089448  2023     12   31     0           52            6

[3294780 rows x 12 columns]
```

In [32]:
```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3294780 entries, 0 to 3294779
Data columns (total 12 columns):
 #   Column        Dtype
---  ------        -----
 0   Date          datetime64[ns]
 1   Country       object
 2   energy_price  float64
 3   biomass       float64
 4   gas           float64
 5   nuclear       float64
 6   year          int64
 7   month         int64
 8   day           int64
 9   hour          int64
 10  week_number   int64
 11  day_of_week   int64
dtypes: datetime64[ns](1), float64(4), int64(6), object(1)
memory usage: 326.8+ MB
None
```

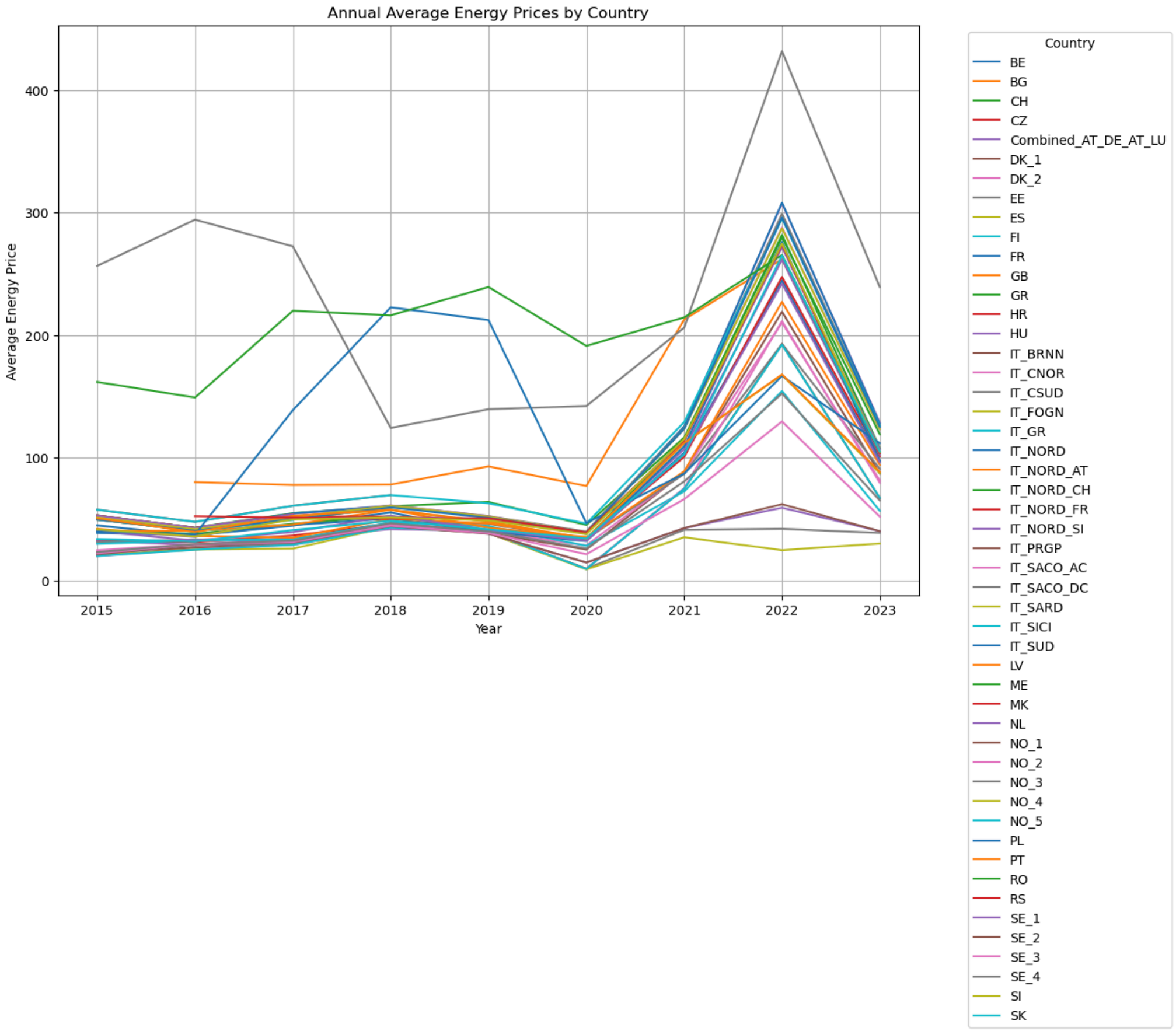
In [33]:
```
print(df.describe())
```

```
                                Date  energy_price       biomass  \
count                        3294780  3.294780e+06  3.294780e+06
mean   2019-05-11 19:56:08.041812224  7.931905e+01  2.522009e+02
min              2015-01-01 00:00:00 -5.000000e+02  0.000000e+00
25%              2017-02-07 07:00:00  3.250000e+01  7.400000e+01
50%              2019-03-29 04:00:00  4.800000e+01  2.522009e+02
75%              2021-08-12 00:00:00  7.995000e+01  2.522009e+02
max              2023-12-31 00:00:00  6.101780e+03  5.368250e+03
std                              NaN  1.202656e+02  4.890990e+02

                 gas       nuclear          year         month           day  \
count  3.294780e+06  3.294780e+06  3.294780e+06  3.294780e+06  3.294780e+06
mean   1.180284e+03  7.185089e+03  2.018858e+03  6.528544e+00  1.573202e+01
min    0.000000e+00  0.000000e+00  2.015000e+03  1.000000e+00  1.000000e+00
25%    1.770000e+02  7.185089e+03  2.017000e+03  4.000000e+00  8.000000e+00
50%    1.180284e+03  7.185089e+03  2.019000e+03  7.000000e+00  1.600000e+01
75%    1.180284e+03  7.185089e+03  2.021000e+03  1.000000e+01  2.300000e+01
max    2.045400e+04  6.149000e+04  2.023000e+03  1.200000e+01  3.100000e+01
std    1.621084e+03  5.733848e+03  2.589092e+00  3.443276e+00  8.795178e+00

                hour   week_number   day_of_week
count  3.294780e+06  3.294780e+06  3.294780e+06
mean   1.147040e+01  2.666163e+01  2.989570e+00
min    0.000000e+00  1.000000e+00  0.000000e+00
25%    5.000000e+00  1.400000e+01  1.000000e+00
50%    1.100000e+01  2.700000e+01  3.000000e+00
75%    1.700000e+01  4.000000e+01  5.000000e+00
max    2.300000e+01  5.300000e+01  6.000000e+00
std    6.902406e+00  1.503643e+01  1.995233e+00
```

What does the label look like?

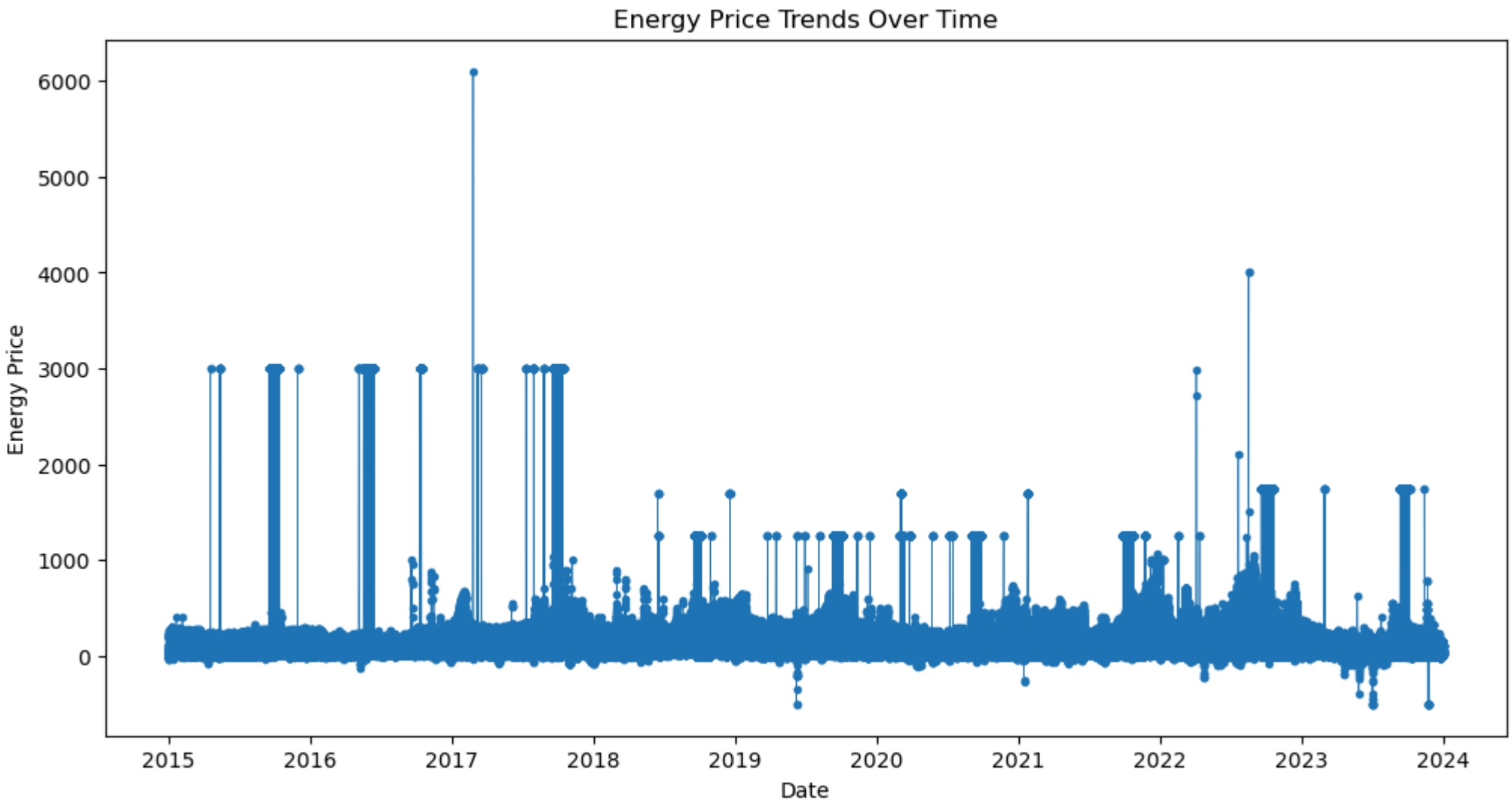
In [34]:
```
df['Year'] = df['Date'].dt.year
annual_avg_price = df.groupby(['Country', 'Year'])['energy_price'].mean().reset_index()
pivot_table = annual_avg_price.pivot(index='Year', columns='Country', values='energy_price')
```

In [35]:
```
plt.figure(figsize=(12, 8))
for column in pivot_table.columns:
    plt.plot(pivot_table.index, pivot_table[column], marker='', label=column)

plt.title('Annual Average Energy Prices by Country')
plt.xlabel('Year')
plt.ylabel('Average Energy Price')
plt.legend(title='Country', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.show()
```



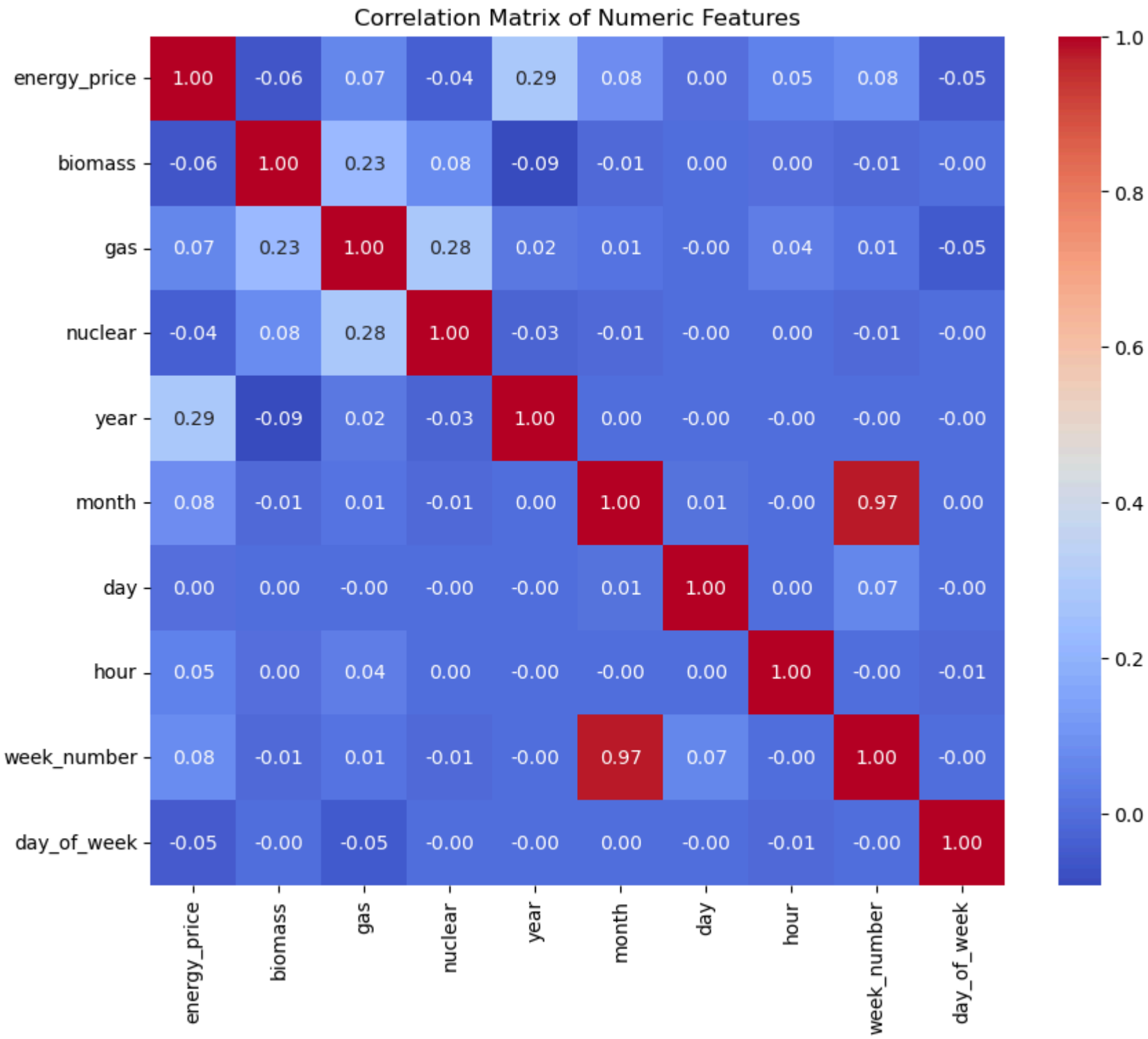
Time series analysis

In [36]:
```
plt.figure(figsize=(12, 6))
plt.plot(df['Date'], df['energy_price'], marker='.', linestyle='-', linewidth=0.5)
plt.title('Energy Price Trends Over Time')
plt.xlabel('Date')
plt.ylabel('Energy Price')
plt.show()
```



Correlation analysis

In [37]:
```
df = df.drop(columns=['Year'])

numeric_df = df.select_dtypes(include=[np.number])
correlation = numeric_df.corr()

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

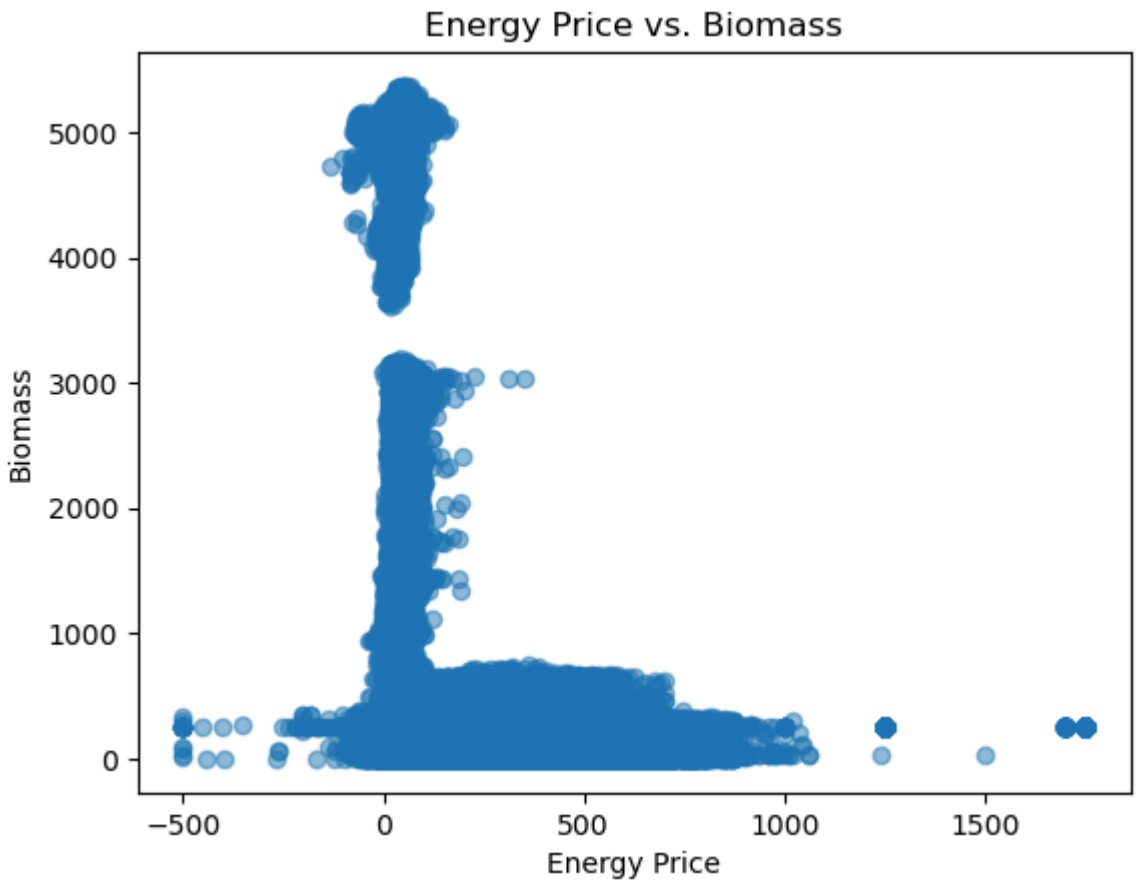


We dropped the variables sources of energy solar and wind because of high correlation.

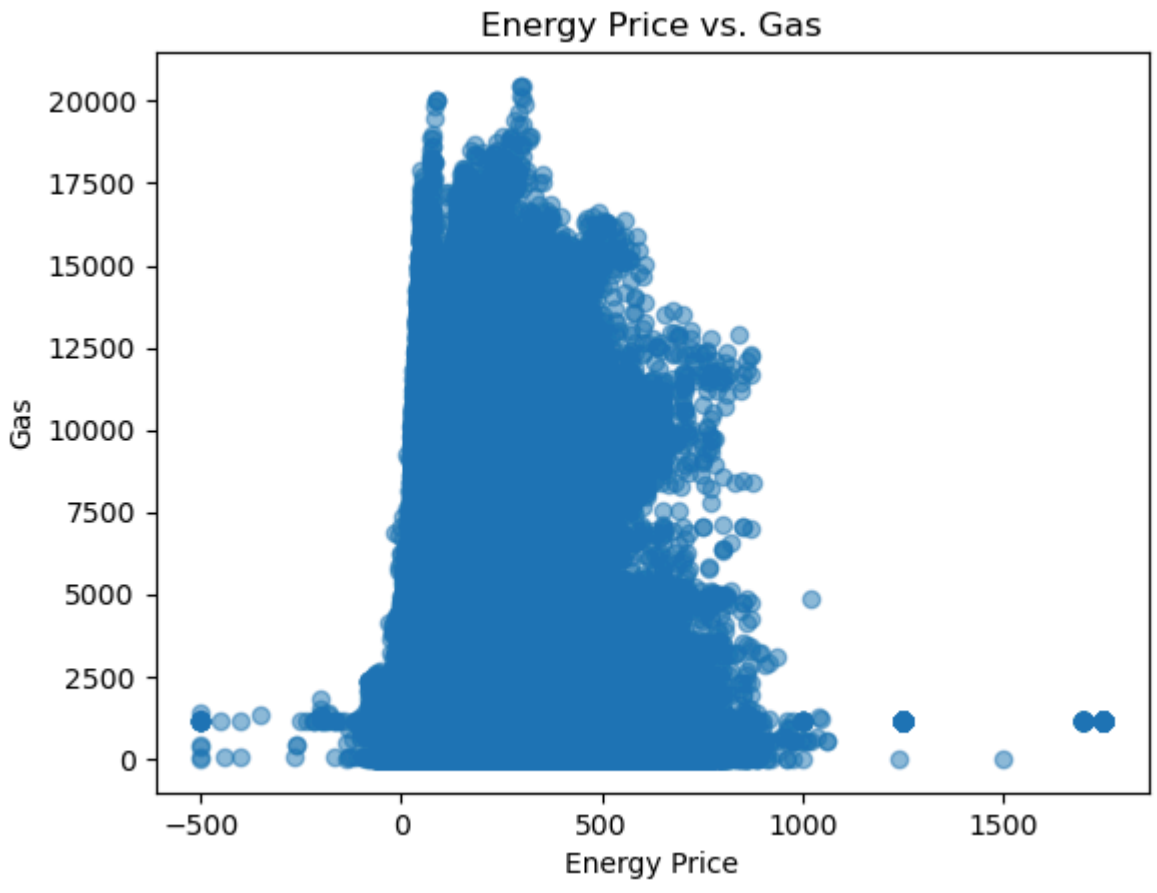
Exploring relationship

In [38]:
```
df_2000 = df[df['energy_price'] <= 2000] # because after 2000 there are outliers
```

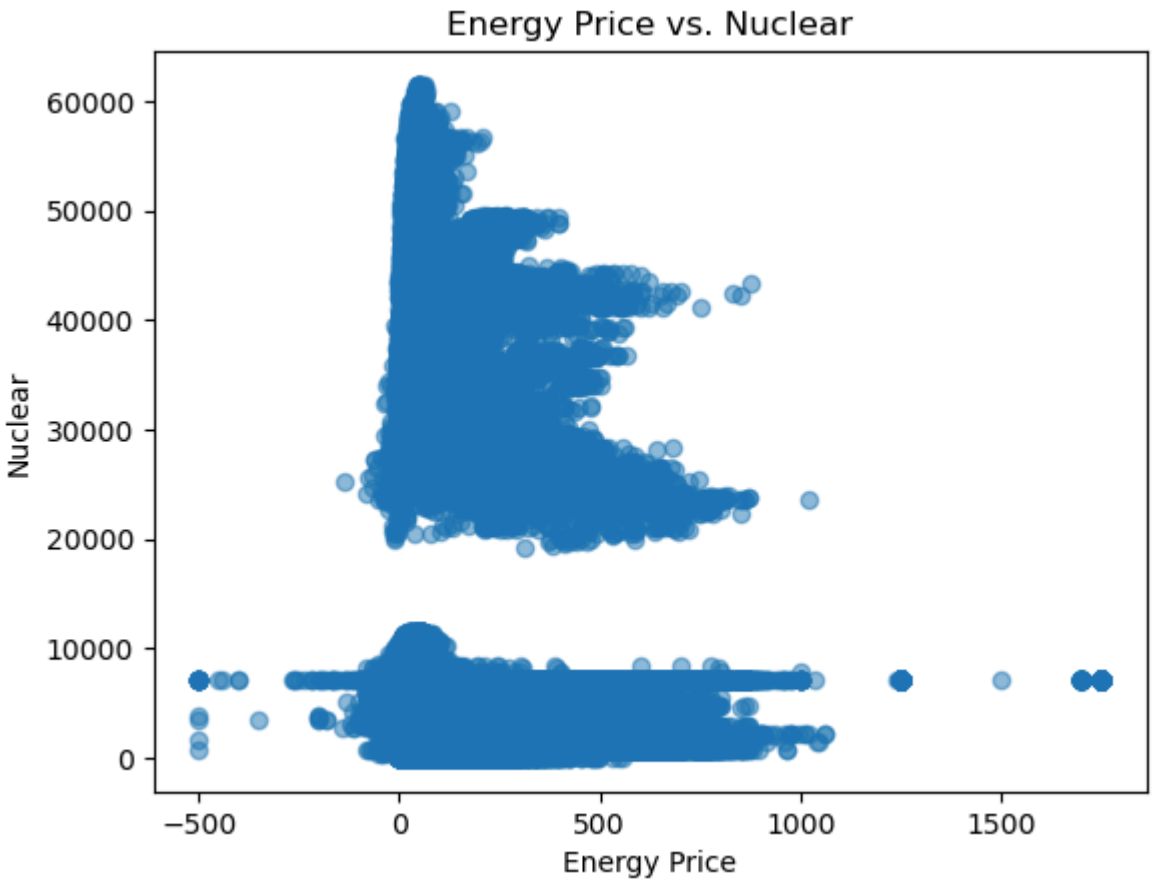
In [39]:
```
# Relationship target and biomass
plt.scatter(df_2000['energy_price'], df_2000['biomass'], alpha=0.5)
plt.title('Energy Price vs. Biomass')
plt.xlabel('Energy Price')
plt.ylabel('Biomass')
plt.show()
```



In [40]:
```
# Relationship target and gas
plt.scatter(df_2000['energy_price'], df_2000['gas'], alpha=0.5)
plt.title('Energy Price vs. Gas')
plt.xlabel('Energy Price')
plt.ylabel('Gas')
plt.show()
```

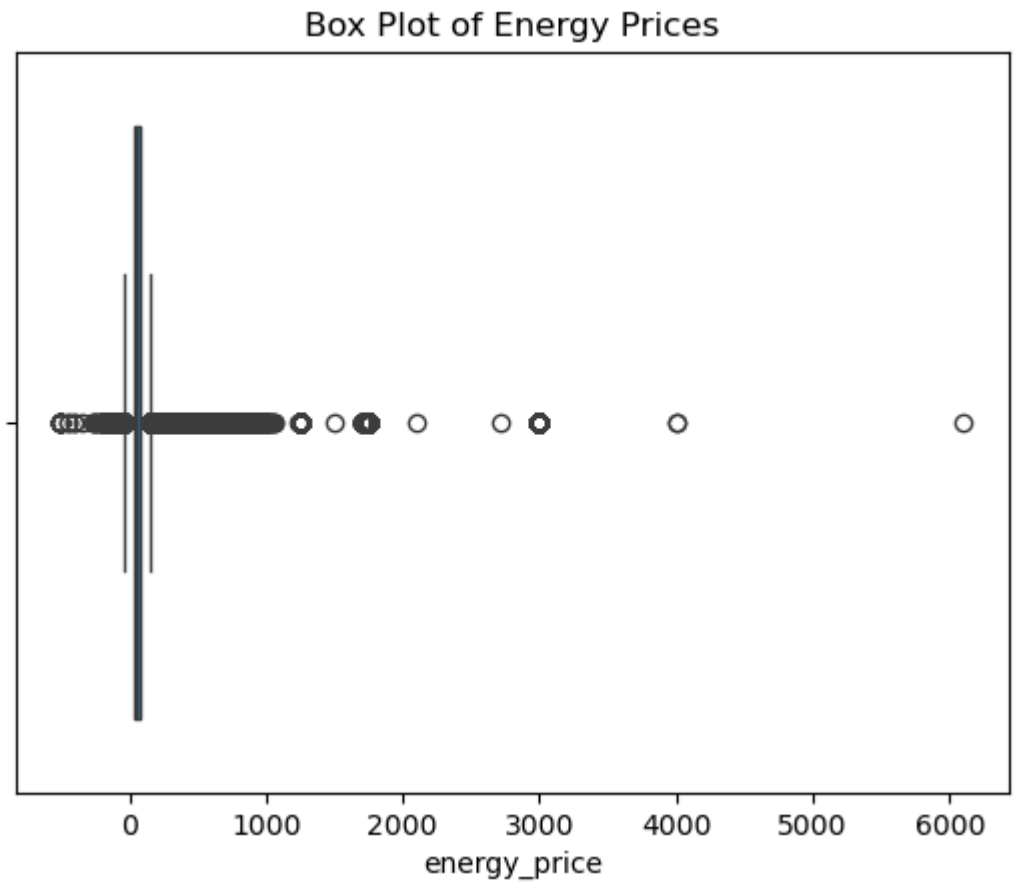


In [41]:
```
# Relationship target and nuclear
plt.scatter(df_2000['energy_price'], df_2000['nuclear'], alpha=0.5)
plt.title('Energy Price vs. Nuclear')
plt.xlabel('Energy Price')
plt.ylabel('Nuclear')
plt.show()
```



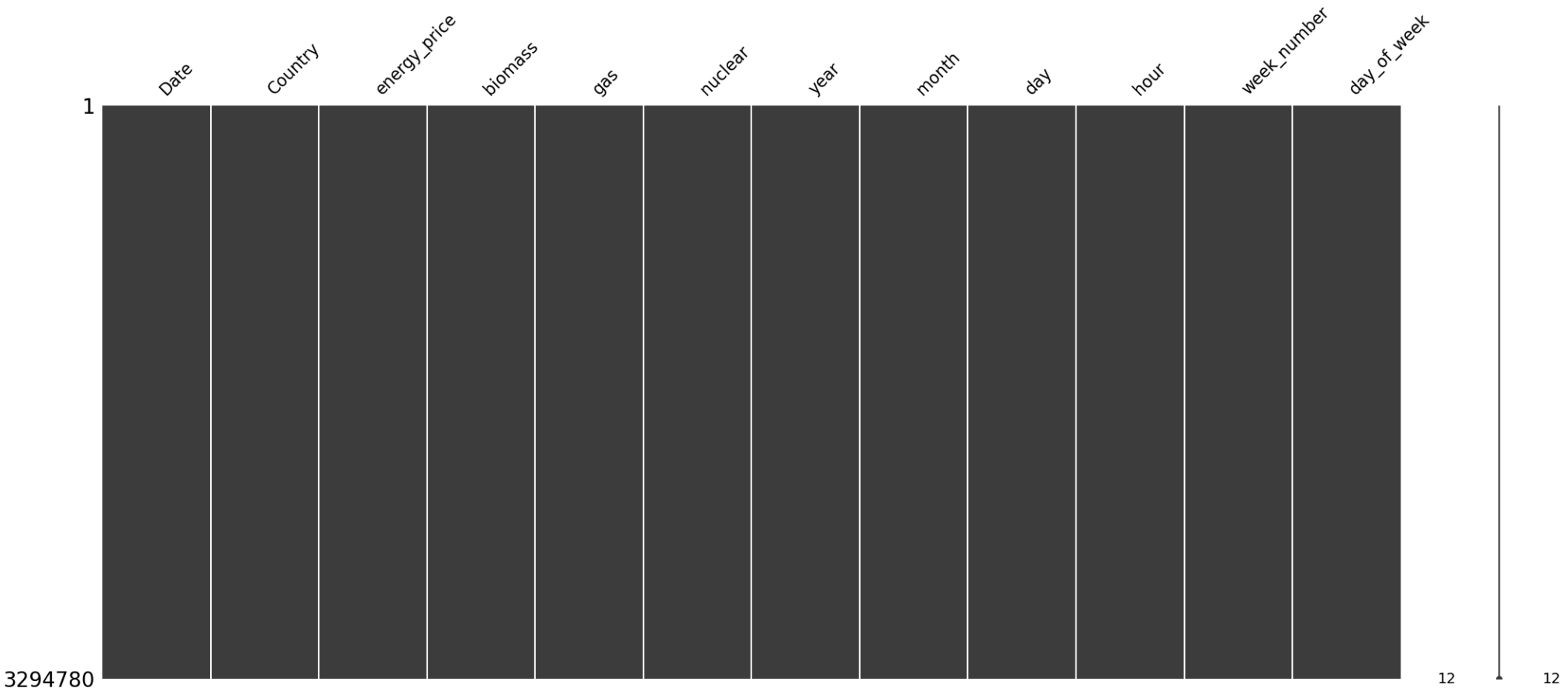
Outliers detection

In [42]:
```
sns.boxplot(x=df['energy_price'])
plt.title('Box Plot of Energy Prices')
plt.show()
```



Missing values

In [43]:
```
msno.matrix(df)
plt.show()
```



In [44]:
```
print(df.isnull().sum())
```

```
Date           0
Country        0
energy_price   0
biomass        0
gas            0
nuclear        0
year           0
month          0
day            0
hour           0
week_number    0
day_of_week    0
dtype: int64
```

We dropped the missing values for the energy_prices column because some of them were too spread in time so I didn't make much sense to use any fill-in strategy such as the mean or past/future values.

3. Models

What models did you try? How did they perform? I will want to see charts comparing the performance of a few different models. I will also want to see an exploration of which features should be included and proved to be particularly useful.

Linear regression

Linear Regression is a statistical model that examines the linear relationship between two or more variables — a dependent variable and independent variables. In its simplest form, linear regression uses a single independent variable to predict the values of the dependent variable. We chose it as a first approximation to the problem because it's a straightforward, easy-to-implement approach and provides a clear interpretation of how variables impact the target (energy prices). It can serve as a baseline to compare more complex models. If more sophisticated models do not perform significantly better than linear regression, the simpler model may be preferred due to its transparency and efficiency. This first try includes the linear model with all features in X (excluding variables "Date" and "Country" because they served as index and to avoid multicollinearity due to the fact that these factor variables were transformed into dummies) serves as a benchmark for the next models.

The accuracy score allows us to see that 83% of the predictions were correctly "classified" into the true values when both values (y_test and y_pred) are binarized based on the threshold (y_test mean). So 83% of predictions followed the same classification in the training set and in the test set (above or below the average price of the test set).

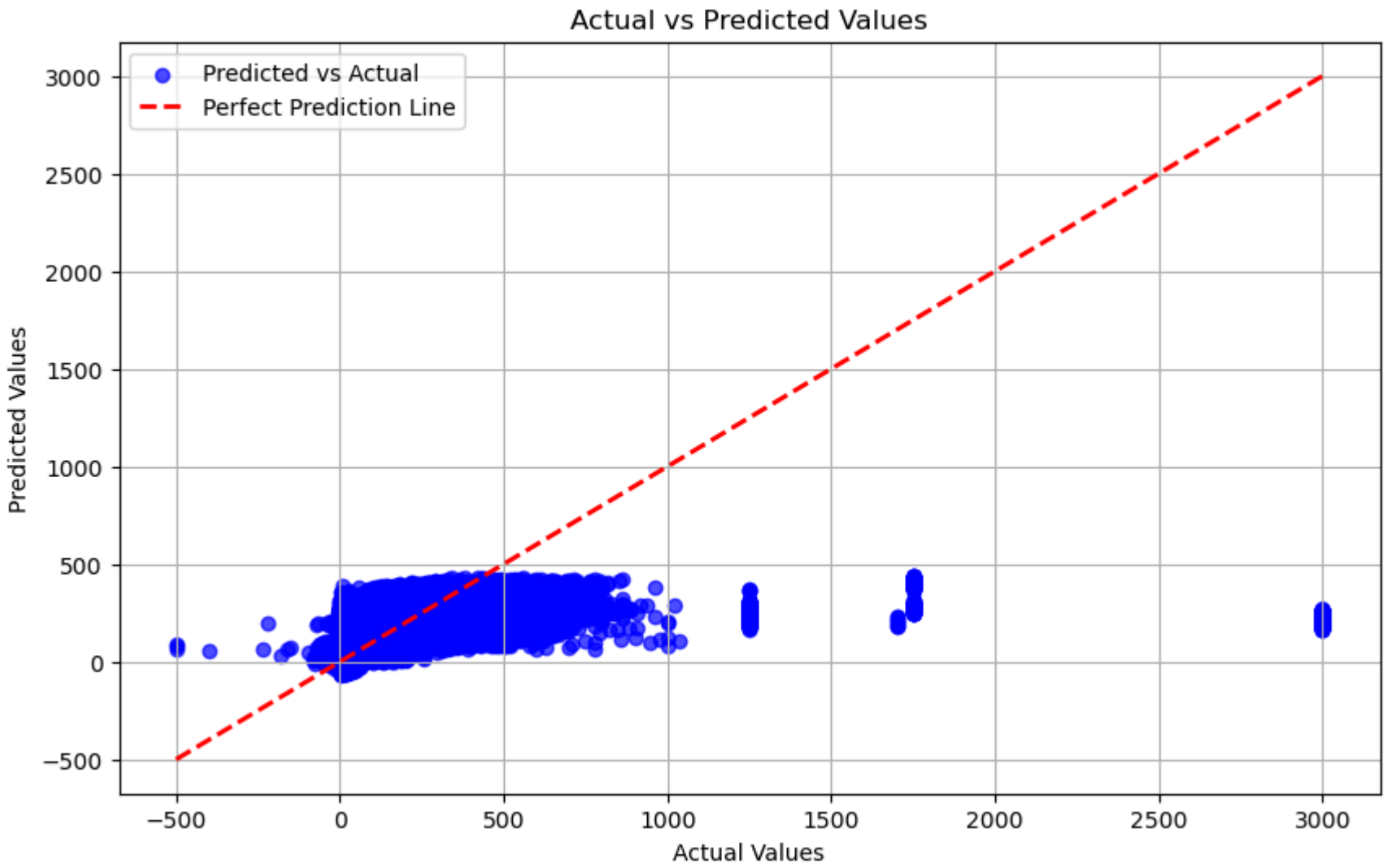
Code:

```
Calculate the MSE
#### Mean Squared Error: 9577.273403544055
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

#### R-squared: 0.33
r2 = r2_score(y_test, y_pred)
print(f'R-squared: {r2:.2f}')
```

Accuracy: 0.83

```
threshold = y_test.mean()
y_pred_binary = (y_pred >= threshold).astype(int)
y_test_binary = (y_test >= threshold).astype(int)

#### Calculate the accuracy score
```

```
accuracy = accuracy_score(y_test_binary, y_pred_binary)
print(f'Accuracy: {accuracy:.2f}')
```



We can observe a logarithmic function associating predicted values and actual values, so we did a log transformation on the features to see its impact on the output (since values over $1,000 per megawatt for energy price can be considered as outliers). Additionally, to avoid generating too big feature matrices (which requires huge memory space), the key variables selected were: source of energy and time features for three years: 2021,2022, and 2023. As shown by the accuracy score (0.85) and R^2 (0.24) the model's predictions are correctly estimating predicted values in 85% of the cases, although due to outliers, these results are not very precise for the entire model (MSE very large). Additionally, R^2 of 0.24 means that the model doesn´t explain the variability of the data around the mean, although since it is a non-linear model, it can hardly tell us more.

Code:

```
energy_features = ['biomass', 'gas', 'nuclear','weekend','year_2021','year_2022','year_2023']
features_to_keep = energy_features
X_train = X_train[features_to_keep]
X_test = X_test[features_to_keep]

degree = 3

# Create a pipeline that first transforms the data to polynomial features, then fits a linear model
model = Pipeline([
    ('poly', PolynomialFeatures(degree=degree)),
    ('linear', LinearRegression())
])

# Fit the model
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
```

Calculate the MSE

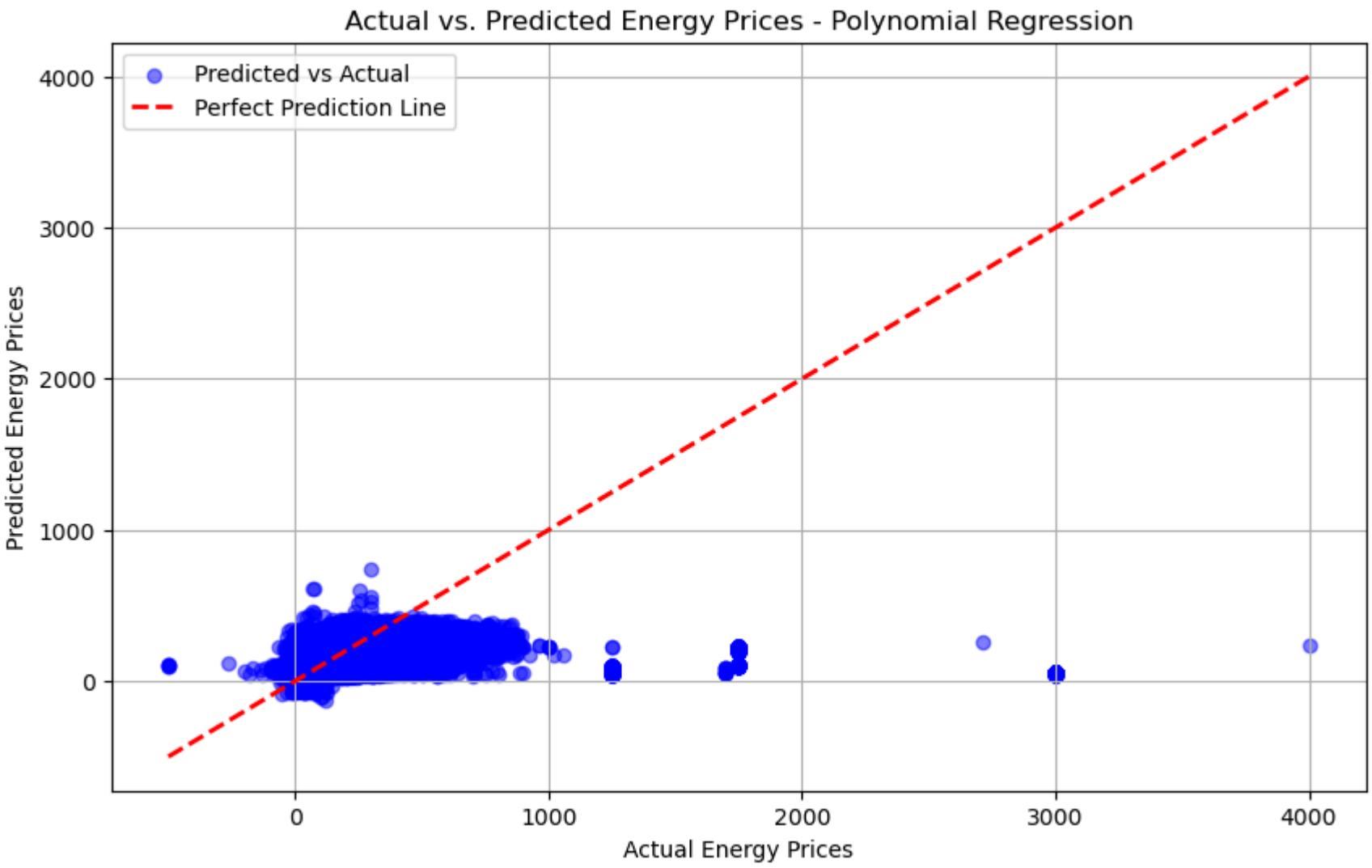
MSE: 11123.742394660521

mse = mean_squared_error(y_test, y_pred)

Calculate R-square: 0.24

Accuracy score: 0.85

```
Ensure y_pred and y_test are one-dimensional
y_pred = np.array(y_pred).flatten()
y_test = np.array(y_test).flatten()

Define a threshold for binary classification (e.g., mean of y_test or a specific value)
threshold = y_test.mean()

Binarize the predictions and the true values based on the threshold
y_pred_binary = (y_pred >= threshold).astype(int)
y_test_binary = (y_test >= threshold).astype(int)

Calculate the accuracy score
accuracy = accuracy_score(y_test_binary, y_pred_binary)
```



As shown in the plot, the difference between predicted values and observed values in the test set is larger compared to the the previous linear model we fitted.

Lasso Regression with interactions

Lastly, since we included dummies to account for different countries and time structure of the data (such as) then it made sense to use Lasso regression. The idea was to drop variables that have small or non incidence for predicting future prices. To restrict the amount of data space, we selected specific features. Energy features only included biomass, gas, and nuclear energy, which are all our energy sources for electricity. Then, to keep some of the time structure we restricted the creation of interaction terms for years from 2020 to 2024. The dummies that represent months were not considered as interactions but were still considered in the general model.

Here 85% of the predicted values followed the same classification in the training set and in the test set (above or below the average price of the test set)

Code:

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.linear_model import LassoCV

lasso_with_interactions = LassoCV(cv=TimeSeriesSplit(n_splits=20), random_state=42, alphas=np.logspace(-1, 1, 20))
lasso_with_interactions.fit(X_train_scaled, y_train.values.ravel())

Predictions with the model
y_pred_test_interactions = lasso_with_interactions.predict(X_test_scaled)
```

Evaluate the model

```
mse_test_interactions = mean_squared_error(y_test, y_pred_test_interactions)
print(f'MSE with Interactions: {mse_test_interactions}')
```

MSE with Interactions: 9552.11439556307

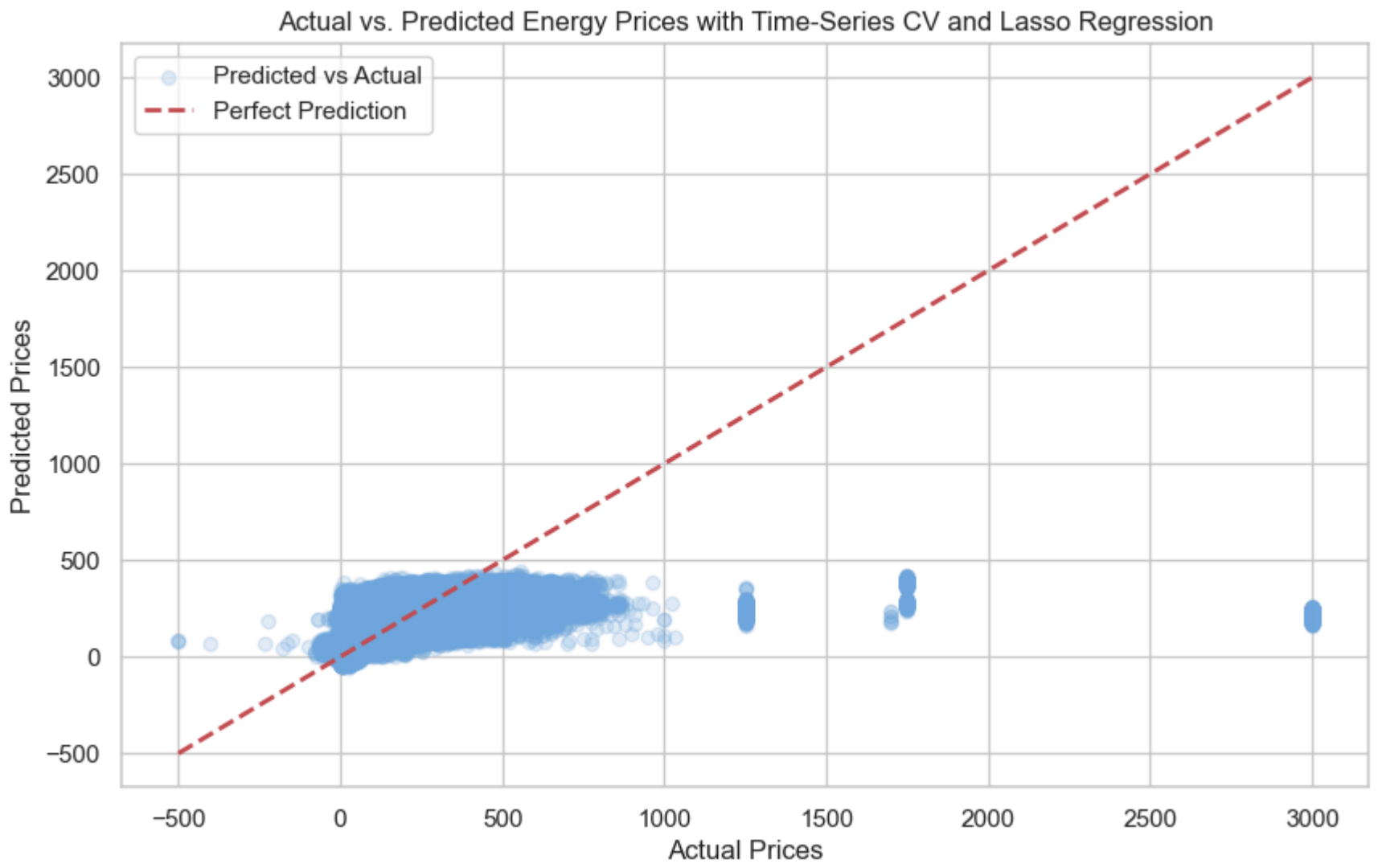
Optimal Alpha with Interactions: 0.1

R-squared of Lasso with interactions: 0.33

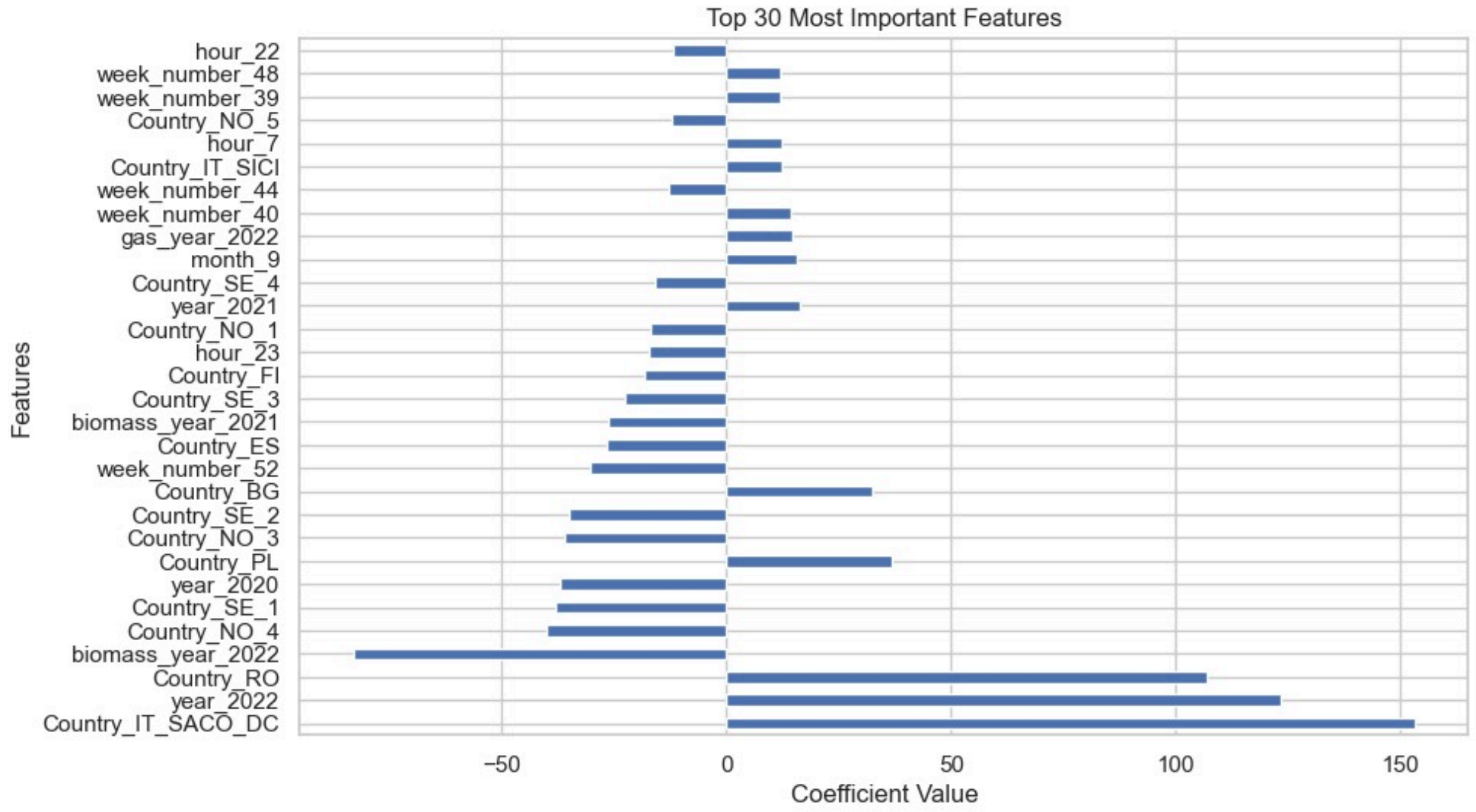
Accuracy score: 0.85

```
threshold = y_train.values.ravel().mean()  # Ensure threshold is a scalar

y_pred_binary = (y_pred_test_interactions >= threshold).astype(int)
y_test_binary = (y_test.values.ravel() >= threshold).astype(int)

accuracy = accuracy_score(y_test_binary, y_pred_binary)
```

Plot:



Relevant features for Lasso regression with CV and Time series split



Linear regression with a different dataset without conflicting outliers

Filtering the new dataset

```
df = pd.read_csv("dates_shift.csv")
Convert the 'Date' column to datetime format
df['Date'] = pd.to_datetime(df['Date'])

Filter out the rows where the year is 2022
df = df[df['Date'].dt.year != 2022]

Filter out rows where 'Country' is either 'RO', 'PL', or 'IT_SACO_DC'
df = df[~df['Country'].isin(['RO', 'PL', 'IT_SACO_DC'])]

Define the columns to drop
columns_to_drop = ['year_2022', 'Country_RO', 'Country_PL', 'Country_IT_SACO_DC']

Drop the specified columns
df = df.drop(columns=columns_to_drop)
```

Code for linear regression

```
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression

Initialize and train the Linear Regression model

model = LinearRegression()
model.fit(X_train_shift, y_train_shift)

y_pred = model.predict(X_test_shift)

Calculate the MSE
mse = mean_squared_error(y_test_shift, y_pred)
print(f'Test MSE: {mse}')
```

MSE with this dataframe with a linear regression that we'll use as benchmark.

Test MSE: 996.7128023011338

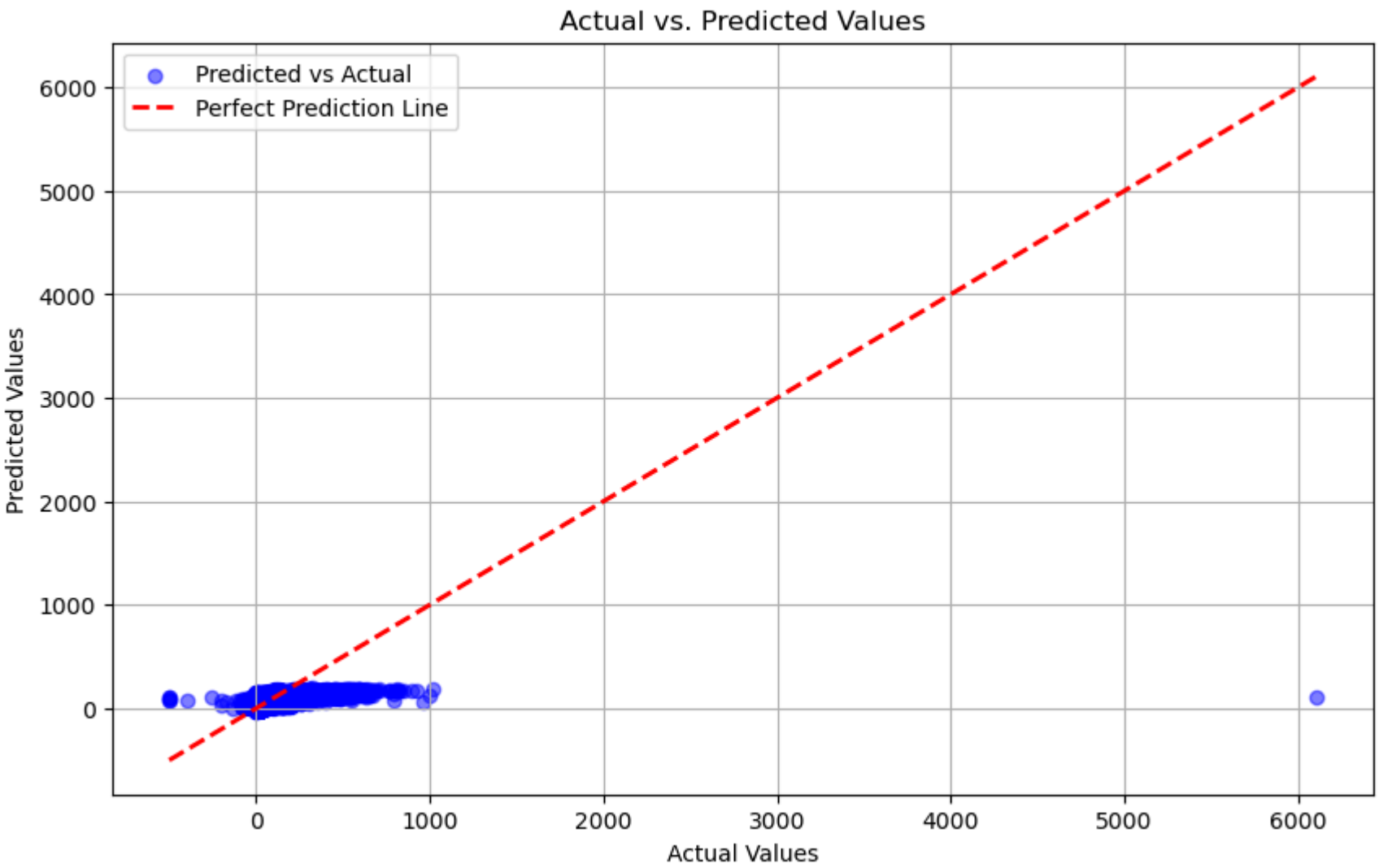
R-squared: 0.48

Accuracy score: 0.80

Code for calculating accuracy score

```
threshold = np.mean(y_test_shift)

Binarize the predictions and the true values based on the threshold
y_pred_binary = (y_pred >= threshold).astype(int)
y_test_binary = (y_test_shift >= threshold).astype(int)

Calculate the accuracy score
accuracy = accuracy_score(y_test_binary, y_pred_binary)
print(f'Accuracy: {accuracy:.2f}')
```

Plot:



Here we can observe less outliers by dropping 2022 as year and some countries that had abnormal price changes.

Lasso with Cross-validation and TimeSeriesSplit with a filtered dataset without conflicting outliers (same dataframe used with linear regression)

Code for Lasso with Cross validation and Time Series Split

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.linear_model import LassoCV
import numpy as np

lasso_with_interactions = LassoCV(cv=TimeSeriesSplit(n_splits=30), random_state=42, alphas=np.logspace(-10, 10,
30))
```

```
lasso_with_interactions.fit(X_train_shift, y_train_shift.values.ravel())

Predict using the model
y_pred_test_interactions = lasso_with_interactions.predict(X_test_shift)

Evaluate the model
mse_test_interactions = mean_squared_error(y_test_shift, y_pred_test_interactions)
print(f'MSE with Interactions: {mse_test_interactions}')
```

MSE with Interactions: 994.2483208749895

print(f'Optimal Alpha with Interactions: {lasso_with_interactions.alpha_}')

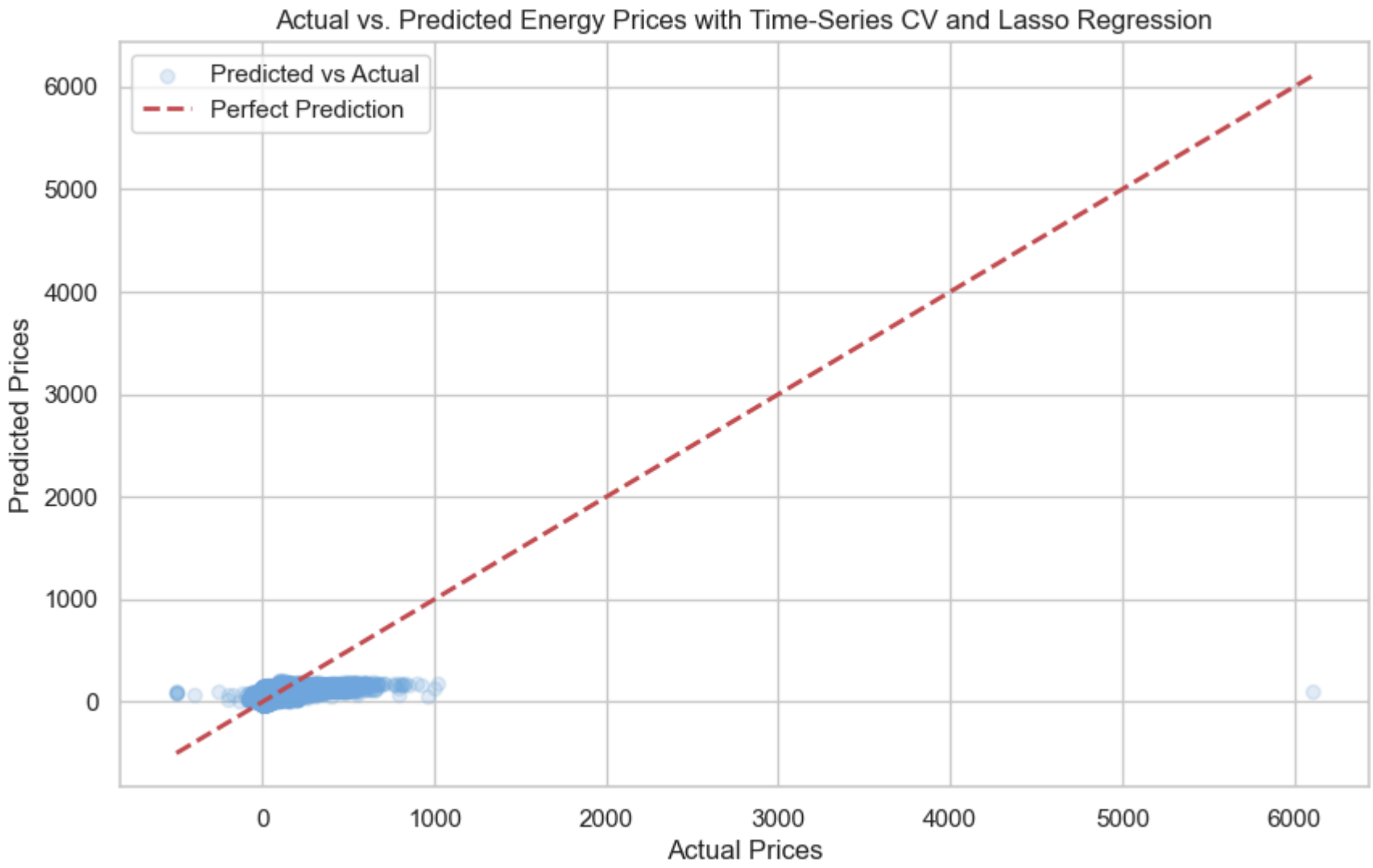
Optimal Alpha with Interactions: 3.290344562312671e-05

R-squared with Interactions: 0.48

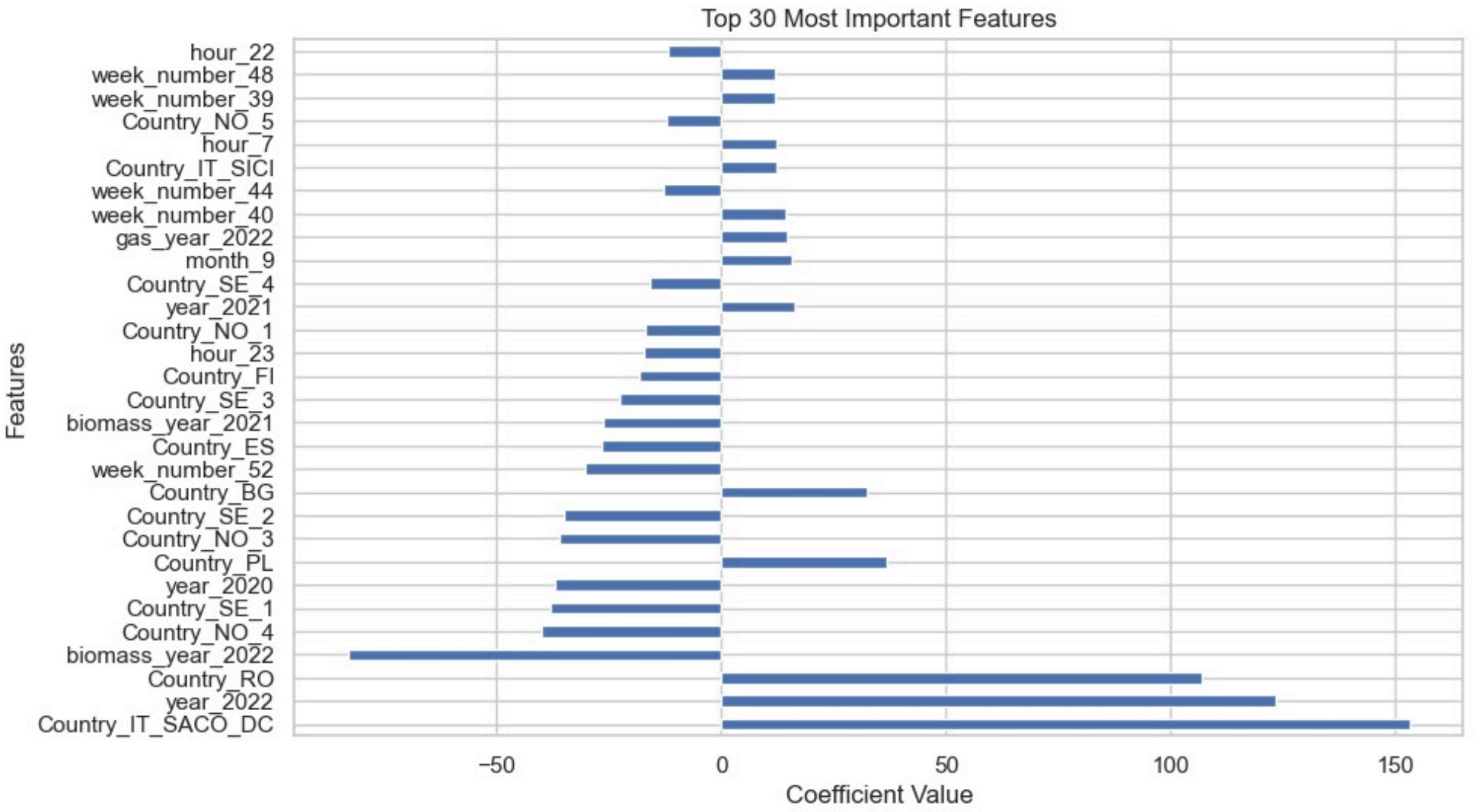
Accuracy score: 0.7975174666809759

```
Accuracy score using the mean of the test set
threshold = np.mean(y_test_shift)  # Ensure threshold is a scalar

y_pred_binary = (y_pred_test_interactions >= threshold).astype(int)
y_test_binary = (y_test_shift.values.ravel() >= threshold).astype(int)

accuracy = accuracy_score(y_test_binary, y_pred_binary)
print(f'Accuracy Score: {accuracy}')
```

Here we are using a dataset without conflicting outliers, although we still have them.



Feature-importance matrix



ARIMA (Autoregressive Integral Moving Average)

ARIMA is a popular statistical method for time series forecasting that uses data points from previous time steps as input to a regression equation to predict future values. This method combines autoregressive (AR), differencing (I), and moving average (MA) components. We chose it because it's especially useful for data with trends or seasonal patterns, which is typical in energy data. ARIMA is also suitable for univariate series with a reliance on past values, which makes it useful for initial exploration in time series forecasting.

First, TimeSeriesSplit is initialized with 20 splits. This is used to split the data into training and testing sets while preserving the temporal order, which is crucial for time-series data. Then an ARIMA model is created with specified parameters (p=5, d=1, q=0) where:

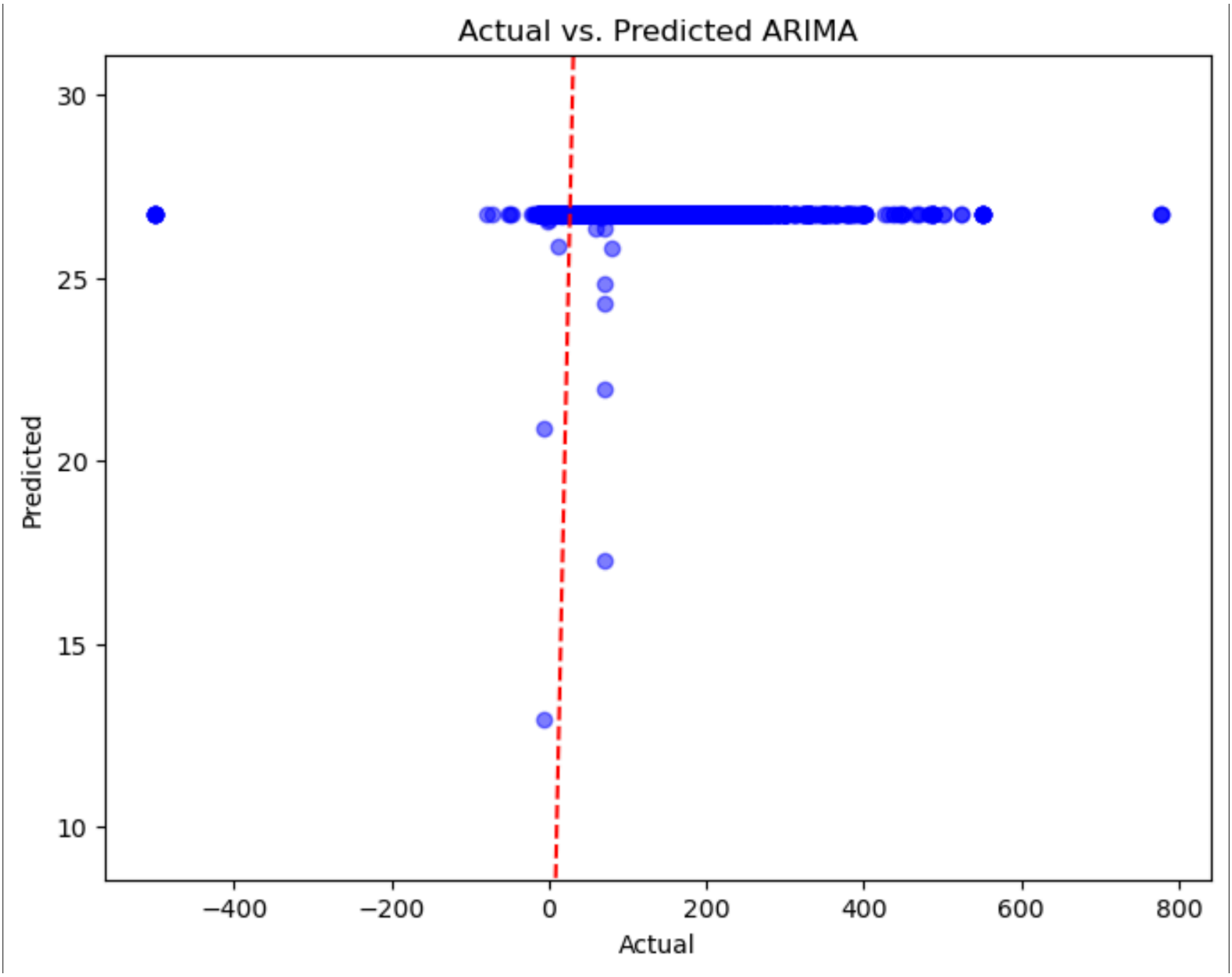
- p: Number of lag observations included in the model (lag order).
- d: Number of times that the raw observations are differenced (degree of differencing).
- q: Size of the moving average window (order of moving average).

The model is fitted to the training data. The model then forecasts the energy prices using the fitted ARIMA model and the forecasted values are converted into a pandas Series and indexed by the same indices as the test set for easy comparison.

Code:

```
#### Initialize TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=20)

#### Apply TimeSeriesSplit to df
for i, (train_index, test_index) in enumerate(tscv.split(X)):
   X_train, X_test = X.iloc[train_index], X.iloc[test_index]
   y_train, y_test = y.iloc[train_index], y.iloc[test_index]

   print(f"Fold {i}:")
   print(f"  Train Indices: {train_index}")
   print(f"  Test Indices: {test_index}")

test = df.iloc[test_index]

#### Fit model
model = ARIMA(y_train, order=(1,0,0))
fitted_model = model.fit()

#### Forecast
y_pred = fitted_model.forecast(steps=len(X_test))

#### Evaluate the model
mse = mean_squared_error(y_test, y_pred)
rs=r2_score(y_test, y_pred)
```

```
MSE: 3812.463852796268
R2: -0.4134327559037574
```



The plot shows a comparison between the actual energy prices (in blue) and the forecasted energy prices (in red). The actual prices display significant variability with frequent spikes and drops, indicating a highly volatile energy market. In contrast, the forecasted prices are relatively flat and fail to capture this variability. This suggests that the ARIMA model used is not effectively modeling the volatility of the energy prices. The forecasted values are almost constant, implying that the model might not be well-tuned for this data or that the chosen ARIMA parameters (5, 1, 0) might not be appropriate. The flat forecast line indicates that the model is not responsive to changes in actual energy prices, which can be due to reasons such as inappropriate differencing, lack of seasonality consideration, or the model order not capturing the underlying patterns. The large discrepancy between the actual and forecasted values suggests that further model tuning or alternative modeling approaches might be necessary. To improve the model's performance, it would be beneficial to re-evaluate the ARIMA parameters, possibly using auto_arima to find the best parameters, incorporate additional features or external factors that might influence energy prices, and explore other time-series models like SARIMA, SARIMAX, or machine learning approaches that can handle the complexity and volatility better.

Auto ARIMA

Now we use the auto_arima function to automatically determine the best parameters (p, d, q) for the ARIMA model. The start_p and start_q parameters set the starting values for the p and q parameters. The max_p and max_q parameters set the maximum values for p and q to be considered during the search. seasonal=False indicates that the model is non-seasonal. trace=True enables the function to print the steps it takes during the parameter search. error_action='ignore' and suppress_warnings=True are used to handle and suppress any warnings or errors that might occur during the search. stepwise=True indicates that the function should use a stepwise approach to search for the best parameters.

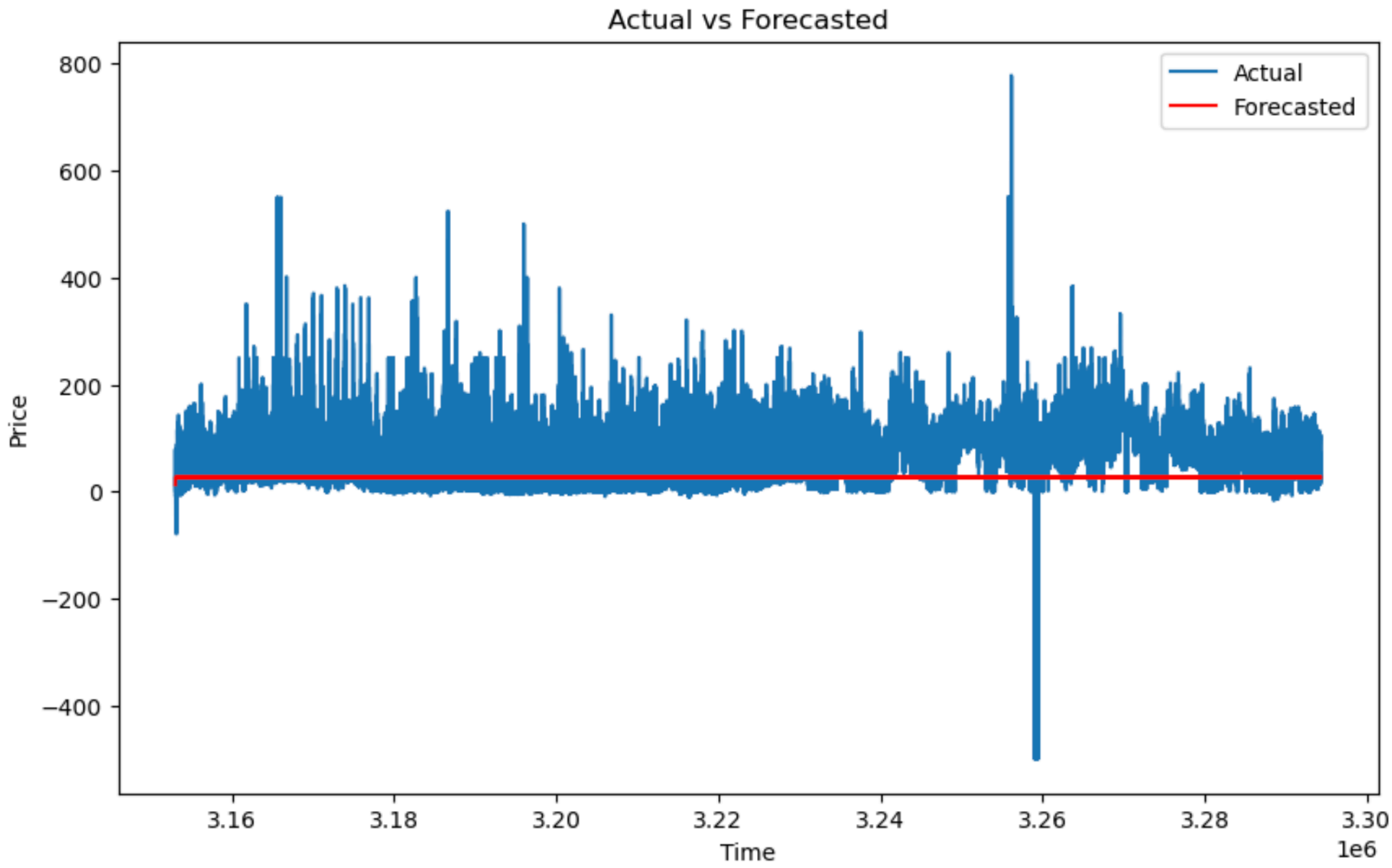
Code:

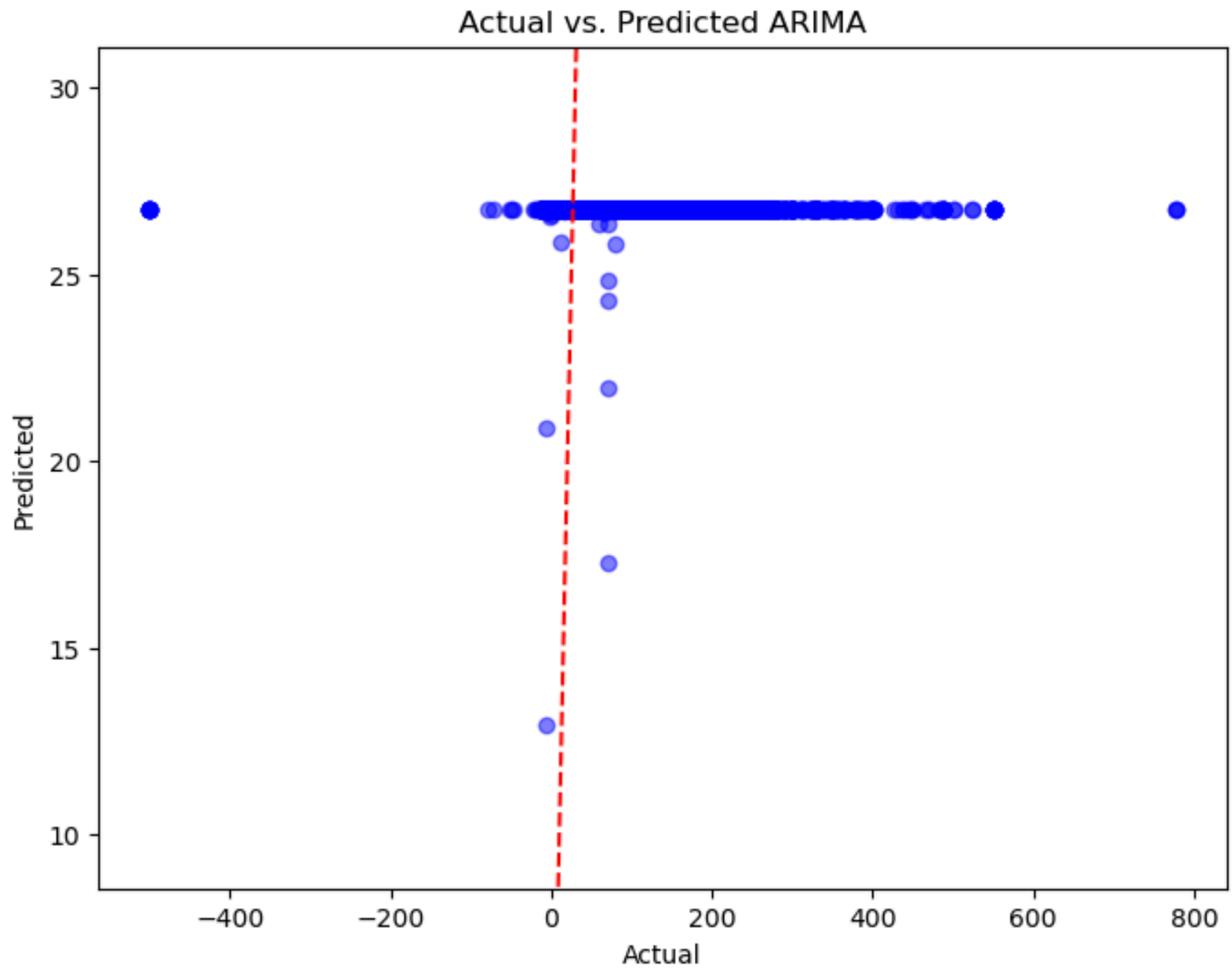
```
#### Fit auto_arima to find the best p, d, q
stepwise_model = auto_arima(y_test, start_p=1, start_q=1,
                            max_p=5, max_q=5, m=1,
                            start_P=0, seasonal=False,
                            d=None, D=0, trace=True,
                            error_action='ignore',
                            suppress_warnings=True,
                            stepwise=True)
#### Fit model
model2 = ARIMA(y_train, order=(5,1,2))
fitted_model2 = model.fit()

#### Forecast
y_pred2 = fitted_model2.forecast(steps=len(X_test))

#### Evaluate the model
```

```
mse2 = mean_squared_error(y_test, y_pred)
rs2=r2_score(y_test, y_pred)
```

MSE2: 6220.818165606518

R2 (2): -1.3063059751610044





The plots show the comparison between the actual energy prices (in blue) and the forecasted energy prices (in red) after using the `auto_arima` function to automatically determine the best ARIMA model parameters. The actual prices display significant variability with frequent spikes and drops, indicating a highly volatile energy market. In contrast, the forecasted prices remain relatively flat and fail to capture this variability. Despite using `auto_arima` to find the optimal ARIMA parameters, the model still struggles to capture the volatility of the energy prices. The forecasted values remain almost constant, suggesting that the best ARIMA model found by `auto_arima` is also not adequately tuned to the data or that ARIMA might not be the best model for this particular dataset. The flat forecast line indicates that the model is not responsive to changes in the actual energy prices. This could be due to the nature of the ARIMA model, which might not be well-suited to capture the high volatility and sudden changes in the energy prices. In conclusion, using `auto_arima` did not significantly improve the model's ability to forecast the highly volatile energy prices. The forecasted values remain relatively flat and fail to capture the actual price variability, suggesting that further tuning or different modeling approaches might be necessary. Exploring other models that can handle high volatility and sudden changes better, such as SARIMA, SARIMAX, or machine learning-based time-series models, could be more effective in improving the forecast accuracy.

Random Forest

Random Forest is an ensemble learning method for regression (and classification) that operates by constructing multiple decision trees during training and outputting the average prediction of the individual trees. It is a robust, versatile model that can handle both numerical and categorical data. We chose it because it's capable of capturing complex nonlinear relationships between the dependent and independent variables without extensive data transformation. It also provides insights into which features are most important in predicting the target variable, which can be valuable for understanding energy price dynamics.

For both models, TimeSeriesSplit is initialized with 20 splits to ensure that cross-validation is performed in a time-series aware manner, preserving the order of the data. A Random Forest Regressor is initialized with 100 trees (n_estimators=100) and a fixed random seed (random_state=42) to ensure reproducibility of the results. For Cross-Validation he data is split into 20 folds using TimeSeriesSplit. Then the model is trained and predictions are made on the test set. For the first model we ran it with the whole df (3294780 obs).

Model 1

Code:

```
# Initialize TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=20)

# Initialize the Random Forest Regressor
model = RandomForestRegressor(n_estimators=100, random_state=42)

# Initialize a list to store the mean squared error for each fold
mse_scores = []

# Apply TimeSeriesSplit to df
for i, (train_index, test_index) in enumerate(tscv.split(X)):
   X_train, X_test = X.iloc[train_index], X.iloc[test_index]
   y_train, y_test = y.iloc[train_index], y.iloc[test_index]

   # Train the Random Forest model
   model.fit(X_train, y_train)

   # Predict on the test data
   predictions = model.predict(X_test)

   # Calculate MSE for the current fold
   mse = mean_squared_error(y_test, predictions)
   mse_scores.append(mse)
```

MSE1: 15894.23672145299

For the second model we excluded features based on feature importance of first model and outlies were removed.

Model 2

Code:

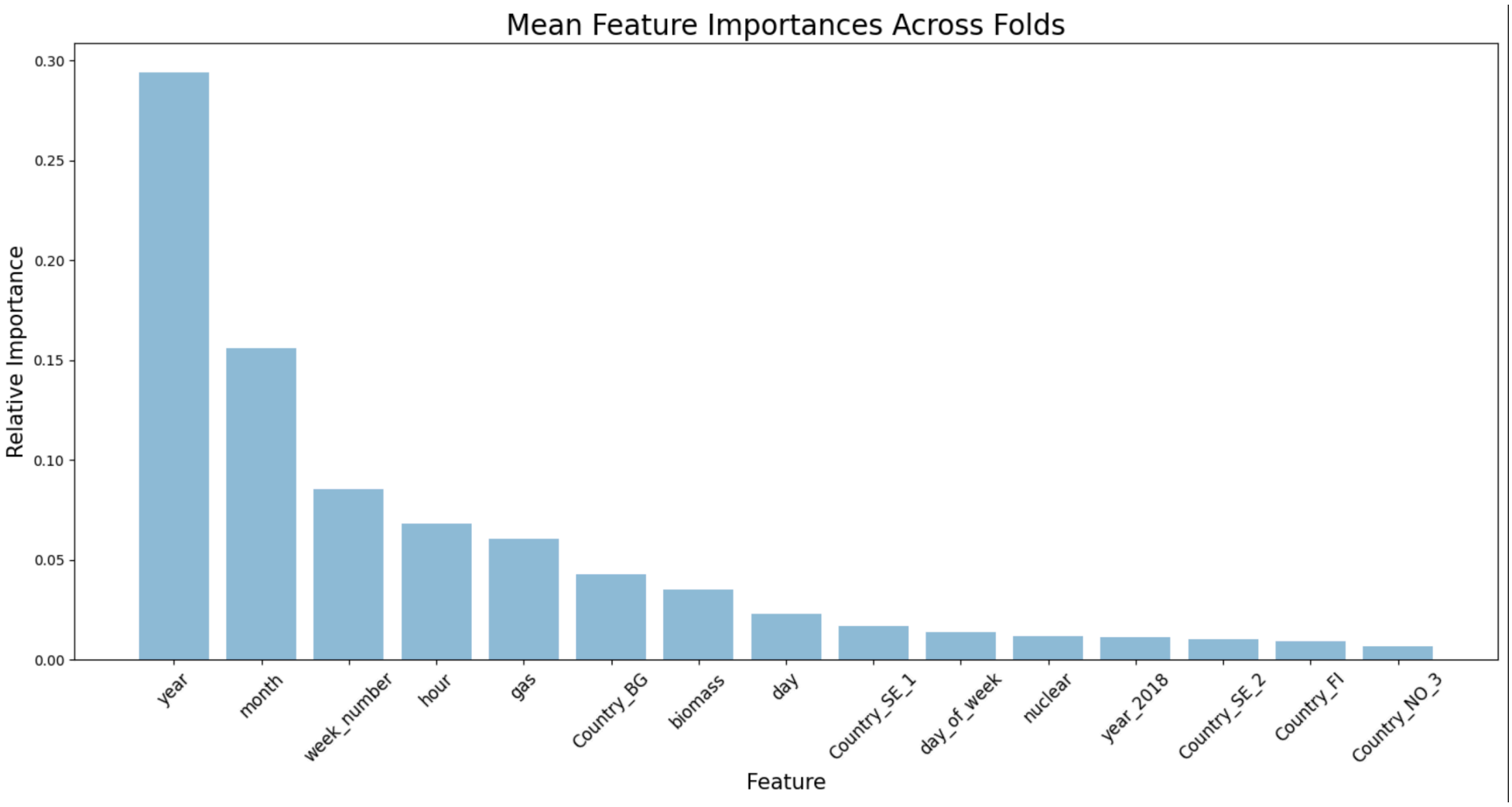
```
dates_shift_exclude.sort_index(level='Date', inplace=True)
X_2 = dates_shift_exclude.drop(['energy_price', 'energy_price_target', 'Country', 'Date'], axis=1)
y_2 = dates_shift_exclude['energy_price_target']

# Initialize TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=20)

# Initialize the Random Forest Regressor
model_2 = RandomForestRegressor(n_estimators=100, random_state=42)

# Initialize a list to store the mean squared error for each fold
mse_scores_2 = []

# Apply TimeSeriesSplit to df
for i, (train_index, test_index) in enumerate(tscv.split(X_2)):
   X_train_2, X_test_2 = X_2.iloc[train_index], X_2.iloc[test_index]
```

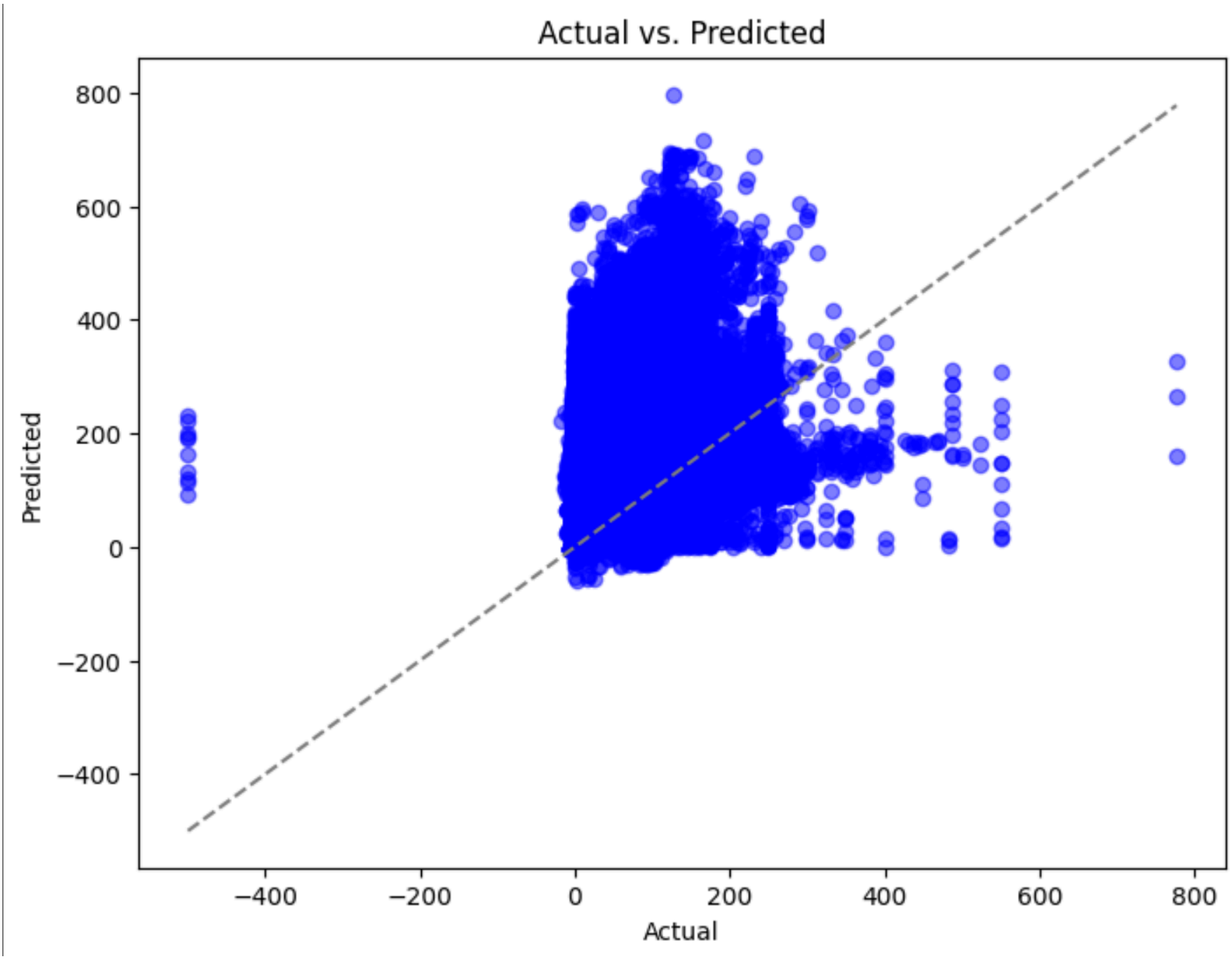
```
    y_train_2, y_test_2 = y_2.iloc[train_index], y_2.iloc[test_index]

    # Train the Random Forest model
    model_2.fit(X_train_2, y_train_2)

    # Predict on the test data
    predictions_2 = model_2.predict(X_test_2)

    # Calculate MSE for the current fold
    mse_2 = mean_squared_error(y_test_2, predictions_2)
    mse_scores_2.append(mse_2)

    # Print fold results
    print(f"Fold {i}:")
    print(f"  Train Indices: {train_index}")
    print(f"  Test Indices: {test_index}")
    print(f"  Mean Squared Error: {mse_2}")

    # Get feature importances
    feature_importances = model_2.feature_importances_

    # Append feature importances to the list
    all_feature_importances = []

    # Append feature importances to the list
    all_feature_importances.append(feature_importances)
```

MSE: 12467.268424190457

Mean Absolute Error: 84.05360051491654

R-squared: -3.640089618401806





The scatter plot displays the relationship between the actual energy prices (x-axis) and the predicted energy prices (y-axis) using the Random Forest model. Upon examining the plot, we observe a dense cluster of points around the center, indicating that the model is predicting many values within a narrow range despite the actual values being spread out. This suggests that the model might be underestimating the variability in the actual energy prices.

The dashed gray line represents the ideal scenario where the predicted values perfectly match the actual values (i.e., a 45-degree line). However, most points deviate significantly from this line, indicating a discrepancy between actual and predicted values. The spread of points away from the line suggests that the model is not accurately capturing the true values.

Additionally, there are noticeable outliers where actual values are both significantly under-predicted and over-predicted. Some points are far away from the ideal line, showing instances where the model predictions are very poor. The predicted values span a wide range, but many predictions are clustered around lower values (near the x-axis). This clustering suggests that the model is biased towards predicting lower values more frequently, possibly failing to capture higher energy prices effectively.

Overall, the scatter plot indicates that the Random Forest model has difficulty accurately predicting energy prices. The significant deviation from the ideal line and the concentration of predictions in a narrow range imply that the model is not effectively capturing the variability and complexity of the actual energy prices. The presence of outliers further underscores the model's limitations in providing precise predictions. These observations suggest that further model tuning or the use of alternative modeling techniques may be necessary to improve prediction accuracy.

In conclusion, for model 1 we get a crazy MSE from 752 to 68715, then we excluded some columns from descriptive statistics and based on the feature importance plot. We then ran model 2 and got a better MSE but still a negative R2.

XGBoost (Gradient Boosting Machines)

XGBoost is an implementation of gradient boosting frameworks that are used for building fast and accurate models. It uses tree-based learning algorithms that successively learn from errors of prior trees, improving accuracy incrementally. We chose it because they are known for delivering high performance and prediction accuracy. Also because it is designed to be efficient, scalable, and to work with large datasets such as ours that has over 3 million observations. It is also effective with datasets having numerous missing values or wide variations in feature scale, common in energy datasets.

NOTE: Due to computational and time constraints, we had to use a model with 80,000 observations (10,000 observations by each of the selected years) but it could be possible that a model with more observations could lead to a lower MSE better, hiegher R^2, thus to a better prediction performance.

Initial model run

```
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

```python
# Normalize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize and train the XGBoost classifier
xgb = XGBRegressor(objective='reg:squarederror', random_state=42)
xgb.fit(X_train_scaled, y_train)

#Optimal parameters
param_grid = {
    'max_depth': [3, 4, 5, 6, 7],
    'min_child_weight': [1, 2, 3, 4],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'n_estimators': [50, 100, 150, 200],
    'subsample': [0.6, 0.7, 0.8, 0.9],
    'colsample_bytree': [0.6, 0.7, 0.8, 0.9]
}
# Initialize GridSearchCV and fit to find the best parameters
grid_search = GridSearchCV(xgb, param_grid, scoring='neg_mean_squared_error', cv=tscv, verbose=1)
grid_search.fit(X_train_scaled, y_train)
# Print best parameters found by GridSearchCV
best_params = grid_search.best_params_
print("Best parameters:", best_params)
```

Best parameters: `{'colsample_bytree': 0.6, 'learning_rate': 0.2, 'max_depth': 3, 'min_child_weight': 4, 'n_estimators': 100, 'subsample': 0.9}`

Model with best parameters

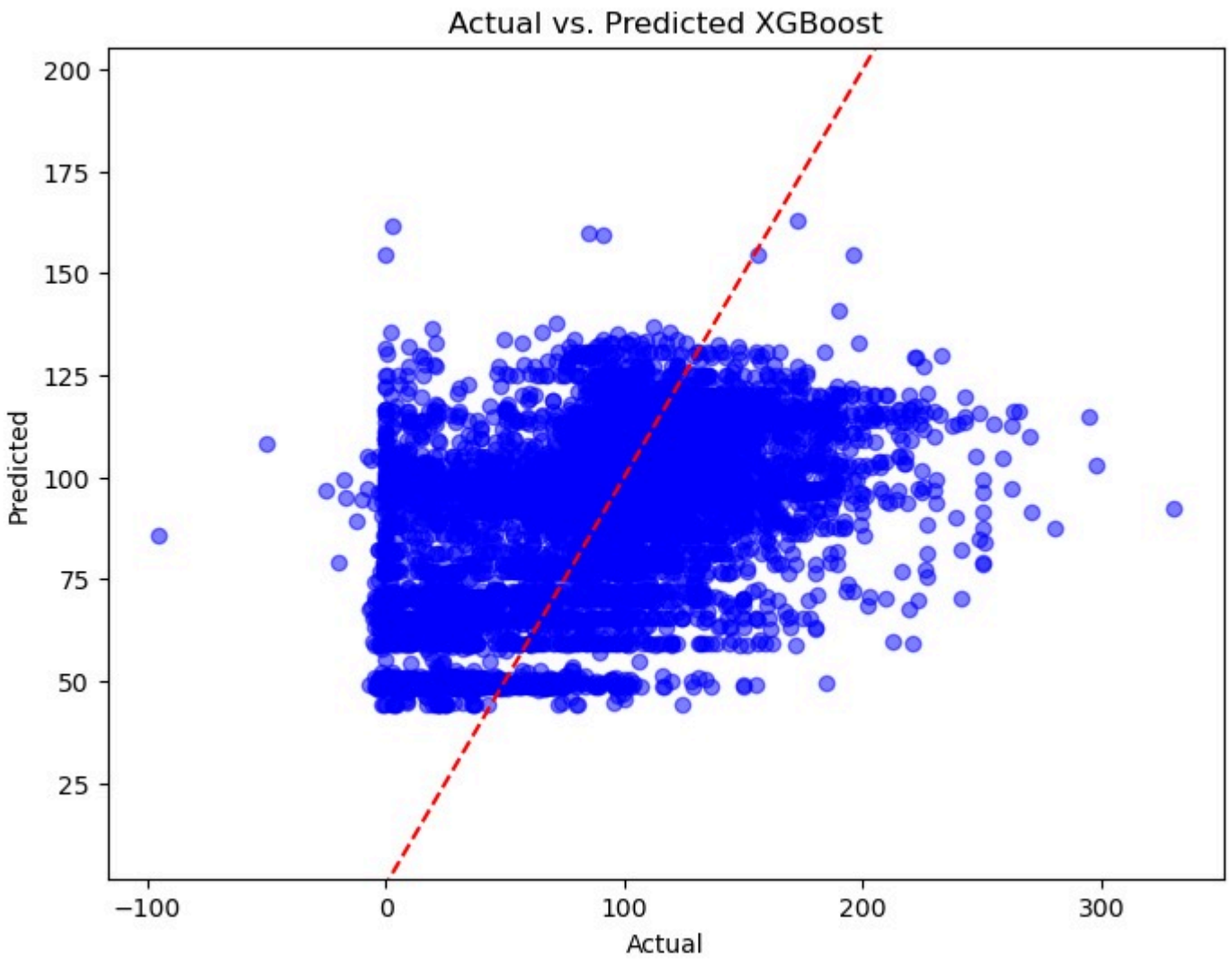
```python
# Re-train the model with the best parameters on the full training set
xgb_optimized = XGBRegressor(**best_params, objective='reg:squarederror', random_state=42)
xgb_optimized.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred_optimized = xgb_optimized.predict(X_test_scaled)

# Calculate and print the accuracy on the test set
mse_optimized = mean_squared_error(y_test, y_pred_optimized)
print("Mean Squared Error (MSE): {:.2f}".format(mse_optimized))

r2_optimized = r2_score(y_test, y_pred_optimized)
print("R-squared (R²): {:.2f}".format(r2_optimized))
```

Mean Squared Error (MSE): 1964.31

R-squared (R^2): 0.24

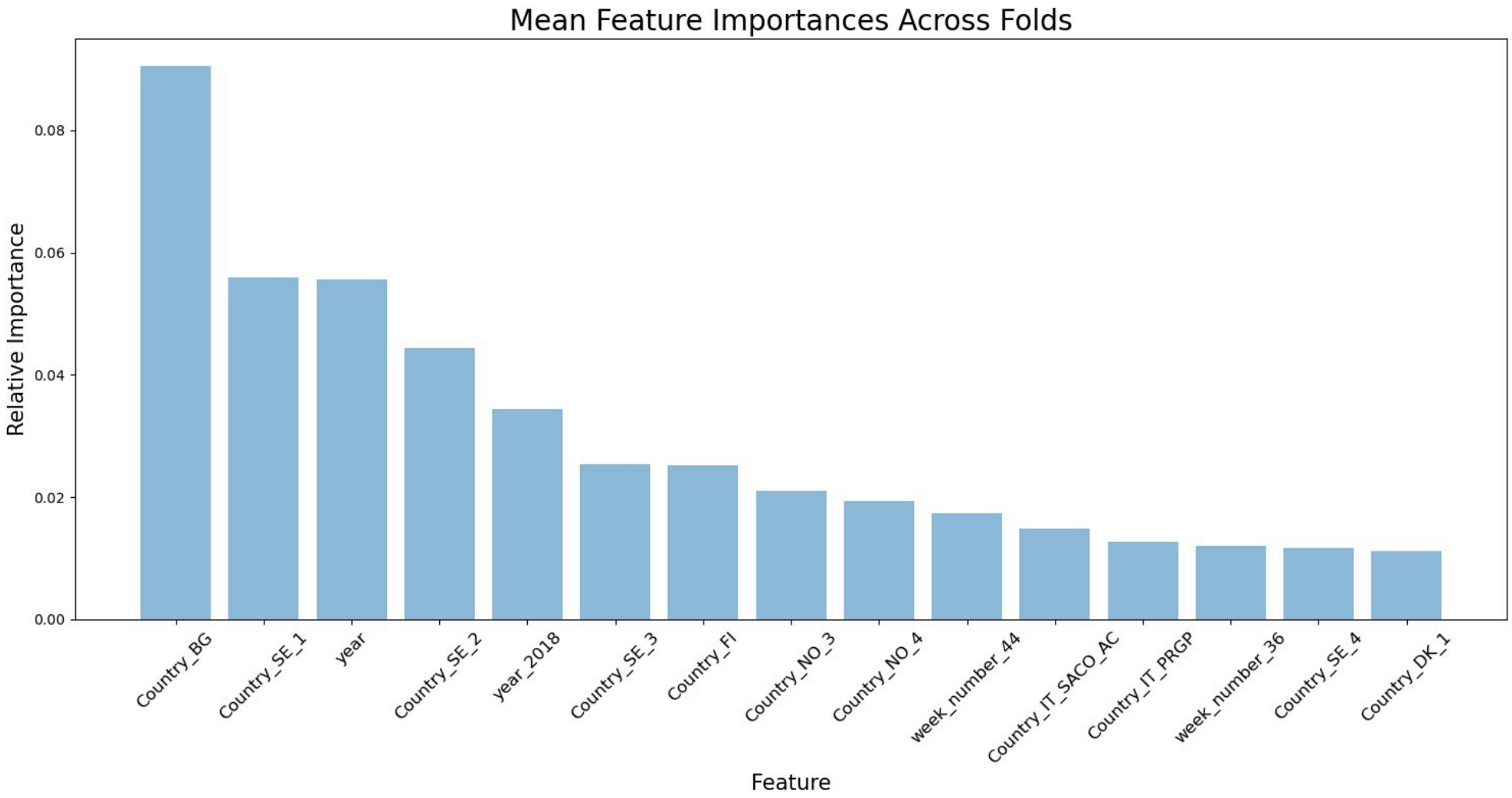
Visual representation of model with best parameters

```python
# Set axis limits
min_val_y = y_pred_optimized.min()
max_val_y = y_pred_optimized.max()
buffer = 0.1 * (max_val - min_val)  # Add a buffer for better visualization

plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred_optimized, color='blue', alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--')

# Set axis limits
plt.ylim(min_val_y - buffer, max_val_y + buffer)

plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs. Predicted XGBoost')
plt.show()
```



Optimal number of features

```
# Top 15 features
top_features = 15
plt.figure(figsize=(15, 8))
plt.title('Mean Feature Importances Across Folds XGBoost', fontsize=20)
bars = plt.bar(range(top_features), mean_feature_importance[sorted_indices][:top_features], align='center',
alpha=0.5)
plt.xticks(range(top_features), X_train.columns[sorted_indices][:top_features], rotation=45, fontsize=12)
plt.ylabel('Relative Importance', fontsize=15)
plt.xlabel('Feature', fontsize=15)
plt.tight_layout()
plt.show()
```



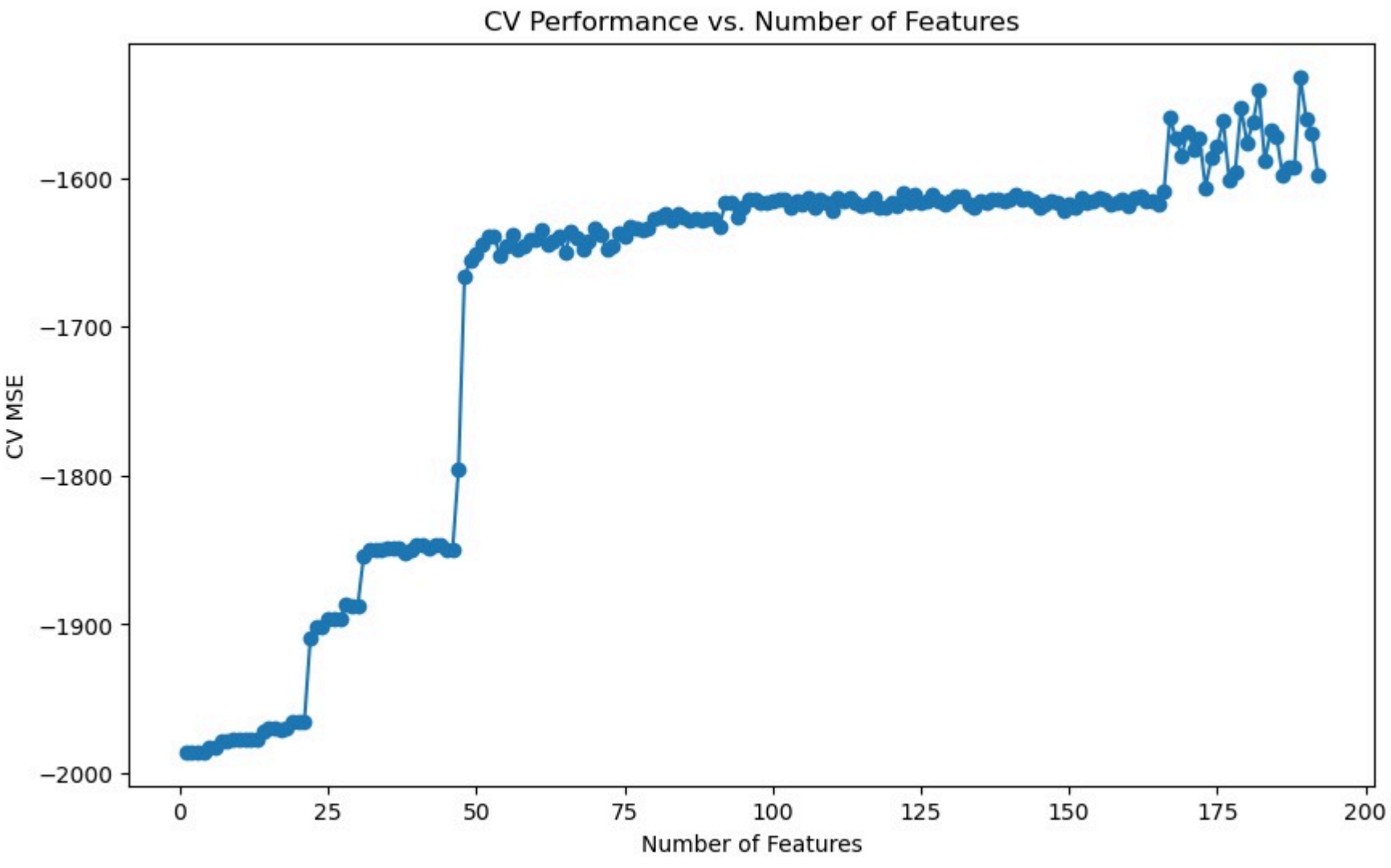

```
# Initialize a list to store the average cross-validation scores for different numbers of top features
cv_scores1 = []
cv_scores2 = []

# Iterate over the range of number of features, starting from 1 to the total number of features
for i in range(1, len(indices) + 1):
    # Select the top 'i' features
    top_features = [X_train.columns[indices[j]] for j in range(i)]

    # Create the training data using the selected features
    X_train_top = X_train_scaled[:, indices[:i]]
```

```
    # Initialize the model with the best parameters
    model = XGBRegressor(**best_params, objective='reg:squarederror', random_state=42)

    # Perform cross-validation and store the mean score
    scores1 = cross_val_score(model, X_train_top, y_train, cv=tscv, scoring='neg_mean_squared_error')
    cv_scores1.append(scores1.mean())

    scores2 = cross_val_score(model, X_train_top, y_train, cv=tscv, scoring='r2')
    cv_scores2.append(scores2.mean())
# Find the number of features that resulted in the highest mean CV score
optimal_features1 = np.argmax(cv_scores1) + 1  # Adding 1 because list indices start at 0
optimal_score1 = cv_scores1[optimal_features1 - 1]

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(indices) + 1), cv_scores1, marker='o')
plt.xlabel('Number of Features')
plt.ylabel('CV MSE')
plt.title('CV Performance vs. Number of Features')
plt.show()
```

MSE

Optimal number of features: 189

Highest CV (negative) MSE: -1531.99

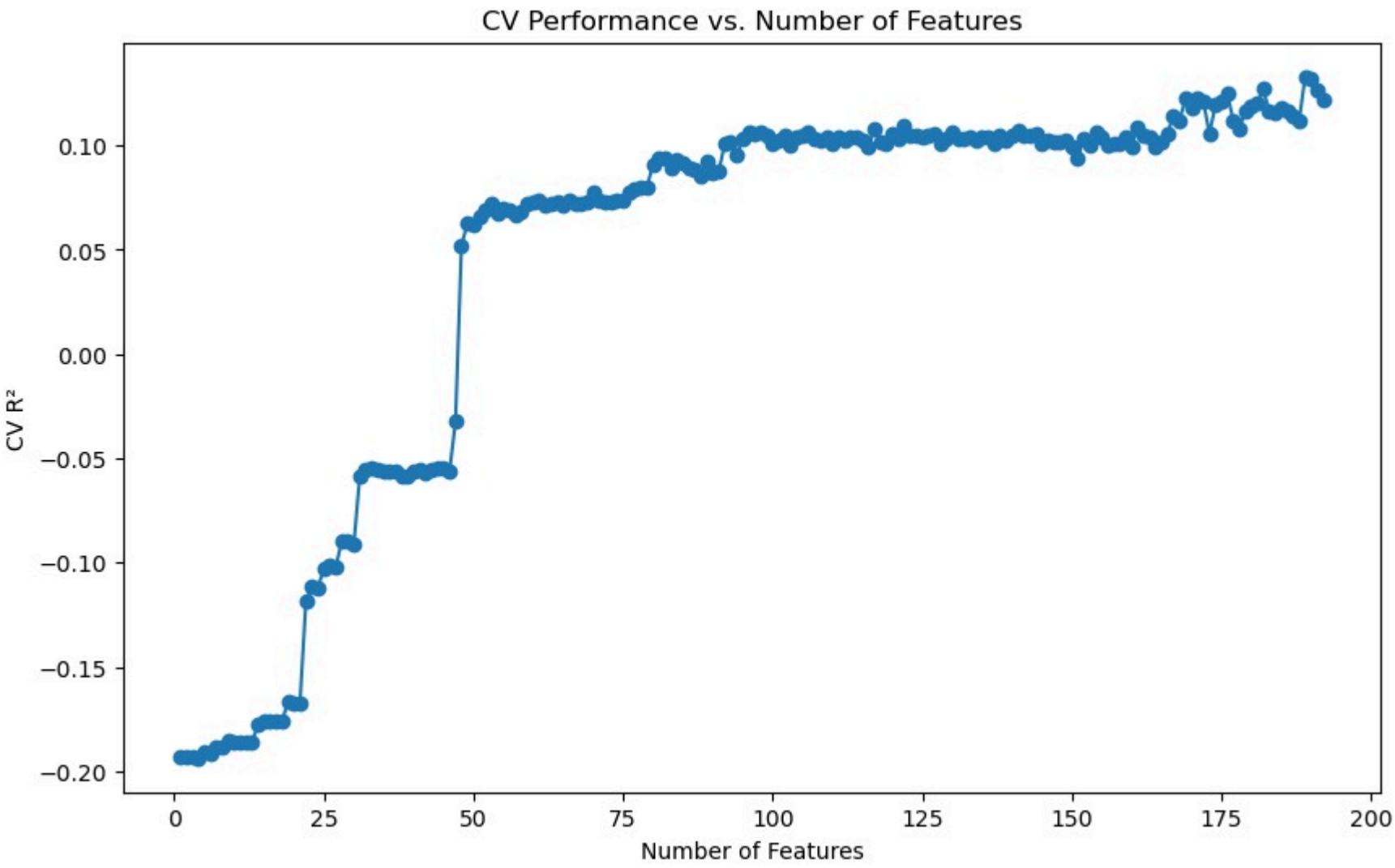



```
optimal_features2 = np.argmax(cv_scores2) + 1  # Adding 1 because list indices start at 0
optimal_score2 = cv_scores2[optimal_features2 - 1]

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(indices) + 1), cv_scores2, marker='o')
plt.xlabel('Number of Features')
plt.ylabel('CV R²')
plt.title('CV Performance vs. Number of Features')
plt.show()
```

R^2

Optimal number of features: 189

Highest CV R^2: -1531.99



We find that the optimal number of features is the same for both the (negative) MSE and the R^2 (189 features).

Model with optimal number of features

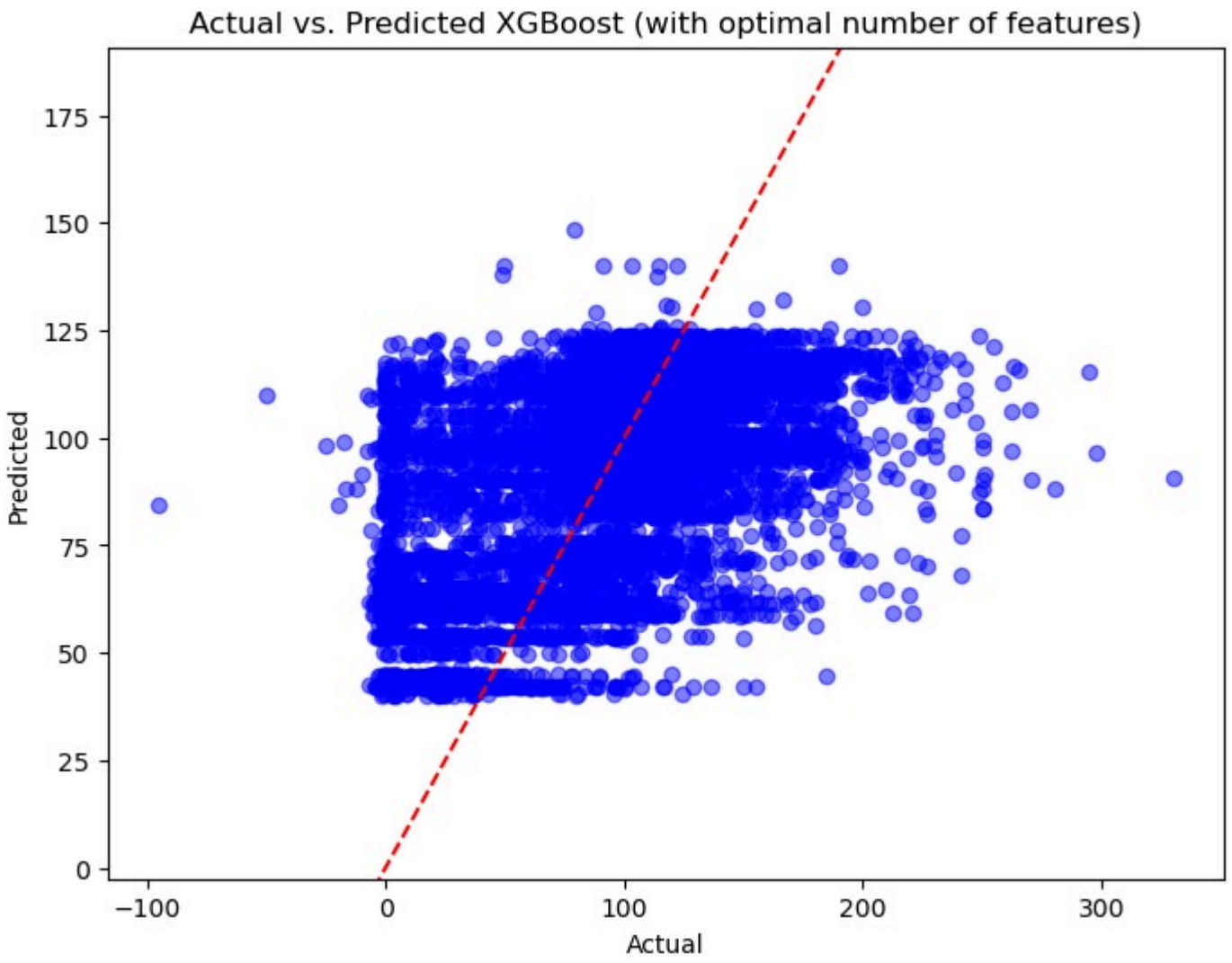
```
# Select the top 'optimal_features' features based on their importance rankings
top_features_indices = indices[:optimal_features1]
X_train_optimal = X_train_scaled[:, top_features_indices]
X_test_optimal = X_test_scaled[:, top_features_indices]

# Train the model using the selected subset of features
model_optimal = XGBRegressor(**best_params, use_label_encoder=False, objective='reg:squarederror', random_state=42)
model_optimal.fit(X_train_optimal, y_train)

# Evaluate the model on the test set
y_pred_optimal = model_optimal.predict(X_test_optimal)

# Calculate and print the accuracy on the test set
mse_optimal= mean_squared_error(y_test, y_pred_optimal)
print("Mean Squared Error (MSE): {:.2f}".format(mse_optimal))

r2_optimal = r2_score(y_test, y_pred_optimal)
print("R-squared (R²): {:.2f}".format(r2_optimal))
```

Mean Squared Error (MSE): 1938.34

R-squared (R^2): 0.25

Visual representation of model with best parameters and optimal number of features

```
# Set axis limits
min_val_y = y_pred_optimal.min()
max_val_y = y_pred_optimal.max()
buffer = 0.1 * (max_val - min_val)  # Add a buffer for better visualization

plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred_optimal, color='blue', alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--')

# Set axis limits
plt.ylim(min_val_y - buffer, max_val_y + buffer)

plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs. Predicted XGBoost (with optimal number of features)')
plt.show()
```



Although the performance of this model is lower than the previous ones, it is important to bear in mind the computational and time limitations that led to the use of a model with fewer observations. It is possible to argue that the use of more observations could have led to a better training for the model and, therefore, to more precise predictions.

4. Evaluation

How are you doing evaluation of your models? What is your test set? How did you choose it? How are you doing cross-validation or, more generally, choosing hyperparameters?

Model evaluation

The evaluation part of the models is considering first the difference between predictions and observed values. We achieve this by looking at the Mean squared error. However, we are aware that the presence of outliers or sudden rises in energy prices (as in 2022) mean a big difference between predicted and observed values, making MSE even larger. This is one of the main reasons why we played around another dataframe that excluded some observations we knew were problematic (as in 2022 for the energy market which meant the beginning of Russia's war with Ukraine). This then resulted in a MSE of around 900 (1/10 of the previos MSE with all observations).

The second main reason is finding a balance between space needed to save results and running time for each models. For instance, linear models without cross validation or time series splits ran faster than every other model, but they are subject to biases that impact the overall performance of the predictive power for our model.

Test set

The test set was built using the Time series split, available in sklearn modules, to account for the temporal structure of our dataset . We specified a test set of 20% of our initial dataset (for the complete this meant around 600k rows and for the reduced set around 400k rows), thus the remaining 80% was used for training the model. We use the random seed 42 for the process to be reproducible. Overall we chose this method because it was simpler and more reliable to go for a random split of the dataset since we have around 3 million rows (total observations depends on the amount of features).

Choosing hyperparameters

In all of our models, except the most naive linear one, we chose hyperparameters by estimating the most optimal alpha (in sklearn is alpha but in class we saw them as lambdas). This setup necessarily means iterations over different models with several parameters values, that then got different results for the estimated betas. In this way, the chosen hyperparameters were the ones that gave us a minimum difference between predicted and observed values across folds and iterations. Cross-validation was also performed using time series split to tune our hyperparameters even further while considering the time structure of the data and minimizing data leakage risk.

5. Choice of model

The final part of the writeup should be an argument for which model you believe best satisfies the needs of the context. You should buttress this argument with the charts and diagnostics you have prepared. This doesn’t necessarily mean the model with the lowest overall error! It’s entirely plausible that concerns such as fairness or explainability might be important considerations to you, in which case you should discuss the tradeoffs you have measured on these dimensions. Crucially, however, your determination should be backed by evidence.

When choosing the best model for predicting energy prices, several factors must be considered, including predictive accuracy, computational efficiency, and model interpretability. Random Forests show the lowest Mean Squared Error (MSE), indicating high predictive accuracy. However, the negative R-squared value raises concerns about the model's reliability and overfitting. Additionally, Random Forests are computationally expensive and require significant time and memory resources, which can be a limitation in practical scenarios.

Lasso regression, particularly with cross-validation (CV) and time series split, provides a balanced approach. While it may not have the lowest MSE, its R-squared value is reasonable, and it avoids the overfitting issues seen with Random Forests. Lasso regression is also less computationally demanding, making it a more practical choice for large datasets or limited computational resources.

Lasso regression offers the advantage of producing interpretable models. The simplicity and transparency of Lasso allow us to understand the influence of each predictor, which is crucial for making informed decisions and providing actionable insights. In contrast, models like Random Forests and XGBoost, while potentially more accurate, function more as "black boxes" with complex internal mechanisms that are more difficult to interpret.

Given the current constraints on computational power and time, Lasso regression stands out as the most feasible option for this project. It strikes a balance between predictive performance and practical implementation. Although XGBoost or Random Forests might perform better given unlimited computational resources, Lasso provides a more reasonable and manageable solution under current constraints.

While Random Forests and XGBoost show promise in terms of predictive accuracy, their computational demands and lack of interpretability make them less suitable for our current needs. Lasso regression, with its balance of reasonable predictive performance, computational efficiency, and interpretability, emerges as the safest and most practical choice. It allows us to deliver reliable predictions and maintain transparency in our modeling process.

In conclusion, selecting Lasso regression is justified by its balance of adequate predictive performance, computational efficiency, and model interpretability. It represents the most reasonable and practical choice given our current constraints, despite not having the best MSE.