Metaculus Model

Imports

In [2]:
```
import pandas as pd
import sys
from pathlib import Path
import subprocess
import os
import gc
from glob import glob

import numpy as np
import pandas as pd
#import polars as pl
from datetime import datetime
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import TimeSeriesSplit, GroupKFold, StratifiedGroupKFold
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.metrics import roc_auc_score
import lightgbm as lgb
#import xgboost as xgb
#from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import OrdinalEncoder
from sklearn.impute import KNNImputer
import optuna as optuna
from optuna.samplers import TPESampler, NSGAIISampler, CmaEsSampler, GridSampler
from sklearn.metrics import root_mean_squared_error
```

Import data

Train set

In [3]:
```
import os
import pandas as pd
from datetime import datetime

# Define the path to the CSV files
directory = r"C:\Users\maorb\CSVs\Metaculus_Csvs_train"
# Dictionary to hold individual DataFrames
df_dict = {}

# List to hold individual DataFrames for merging
df_list = []

# Iterate through all files in the directory
for filename in os.listdir(directory):
    if filename.endswith(".csv"):
        file_path = os.path.join(directory, filename)

            # Read the CSV file into a DataFrame
        df = pd.read_csv(file_path)

            # Extract date from filename
            # Adjust the slicing based on your filename pattern
            # Example: If filename is 'data_2023-07-something.csv', and date is at positions 4 to 12
            # Modify this according to your actual filename structure
        date_str = filename.split('.')[0][4:12]  # e.g., '2023-07'
        df_dict[date_str] = df
        df_list.append(df)


# Example: Accessing individual DataFrames
# For example, to access July 2023 data:
# july_2023_df = df_dict.get("07_2023")

# Merge all DataFrames
if df_list:
    merged_df = pd.concat(df_list, ignore_index=True)

    # Define the path for the merged DataFrame
    merged_file_path = os.path.join(directory, "merged_df.csv")

    # Save the merged DataFrame to a CSV file
    #merged_df.to_csv(merged_file_path, index=False)
    print(f"Merged DataFrame saved as {merged_file_path}")
else:
    print("No CSV files were processed.")

# Optional: Display the keys (dates) of the loaded DataFrames
print("Loaded DataFrames for the following dates:")
for key in df_dict.keys():
    print(key)
```

```
Merged DataFrame saved as C:\Users\maorb\CSVs\Metaculus_Csvs_train\merged_df.csv
Loaded DataFrames for the following dates:
01_01_20
01_01_21
01_01_22
01_01_23
01_01_24
05_020_2
05_20_20
05_20_22
05_20_23
05_20_24
09_20_20
09_20_21
09_20_22
09_20_23
```

Test set

In [4]:
```python
import os
import pandas as pd
from datetime import datetime

# Define the path to the CSV files
directory = r"C:\Users\maorb\CSVs\Metaculus_Csvs_test"
# Dictionary to hold individual DataFrames
df_dict = {}

# List to hold individual DataFrames for merging
df_list = []

# Iterate through all files in the directory
for filename in os.listdir(directory):
    if filename.endswith(".csv"):
        file_path = os.path.join(directory, filename)

            # Read the CSV file into a DataFrame
        df = pd.read_csv(file_path)

            # Extract date from filename
            # Adjust the slicing based on your filename pattern
            # Example: If filename is 'data_2023-07-something.csv', and date is at positions 4 to 12
            # Modify this according to your actual filename structure
        date_str = filename.split('.')[0][4:12]  # e.g., '2023-07'
        df_dict[date_str] = df
        df_list.append(df)


# Example: Accessing individual DataFrames
# For example, to access July 2023 data:
# july_2023_df = df_dict.get("07_2023")

# Merge all DataFrames
if df_list:
    merged_df_test = pd.concat(df_list, ignore_index=True)

    # Define the path for the merged DataFrame
    merged_file_path = os.path.join(directory, "merged_df.csv")

    # Save the merged DataFrame to a CSV file
    #merged_df.to_csv(merged_file_path, index=False)
    print(f"Merged DataFrame saved as {merged_file_path}")
else:
    print("No CSV files were processed.")

# Optional: Display the keys (dates) of the loaded DataFrames
print("Loaded DataFrames for the following dates:")
for key in df_dict.keys():
    print(key)
```

```
Merged DataFrame saved as C:\Users\maorb\CSVs\Metaculus_Csvs_test\merged_df.csv
Loaded DataFrames for the following dates:
01_01_17
01_01_18
01_01_19
01_06_18
05_20_19
06_01_17
09_20_19
```

In [5]:
```python
merge_test_dup_rows = merged_df_test[merged_df_test.duplicated()]
merged_test = merged_df_test.drop(merge_test_dup_rows.index)
merged_test['Response_date'] = pd.to_datetime(merged_test['Response_date'], errors='coerce')
```

In [6]:
```python
#find duplicate rows
duplicate_rows = merged_df[merged_df.duplicated()]
merged_df1 = merged_df.copy() #df without title
merged_df1['Response_date'] = pd.to_datetime(merged_df1['Response_date'],errors = 'coerce')

merged_df1.drop(columns = ['title'],inplace = True)
merged_no_duplicates = merged_df1.drop(duplicate_rows.index)
```

In [811…
```
#merged_df.to_csv(r"C:\Users\maorb\CSVs\Metaculus_Csvs_train\merged_df.csv")
```

Remove columns and tidy

In [7]:
```
merged_df1 = merged_df.copy() #df without title
merged_df1['Response_date'] = pd.to_datetime(merged_df1['Response_date'],errors = 'coerce')

merged_df1.drop(columns = ['title'],inplace = True)
```

Preprocessing function

In [8]:
```
import pandas as pd
import numpy as np

def get_half_rows(group):
    """Get the first half of rows for each group."""
    return group.iloc[:len(group) // 2]

def extract_time_features(df1):
    """Extract basic time-based features from the Response_date column."""
    df1['year'] = df1['Response_date'].dt.year
    df1['month'] = df1['Response_date'].dt.month
    df1['day'] = df1['Response_date'].dt.day
    df1['weekday'] = df1['Response_date'].dt.weekday
    df1['hour'] = df1['Response_date'].dt.hour
    df1['minute'] = df1['Response_date'].dt.minute
    df1['quarter'] = df1['Response_date'].dt.quarter
    df1['weekofyear'] = df1['Response_date'].dt.isocalendar().week
    df1['dayofyear'] = df1['Response_date'].dt.dayofyear
    df1['is_weekend'] = df1['weekday'].isin([5, 6]).astype(int)
    df1['is_month_start'] = df1['Response_date'].dt.is_month_start.astype(int)
    df1['is_month_end'] = df1['Response_date'].dt.is_month_end.astype(int)
    return df1

def generate_cyclical_features(df1):
    """Generate cyclical features for time variables."""
    cyclical_cols = {
        'hour': 24, 'weekday': 7, 'month': 12, 'day': 31,
        'minute': 60, 'weekofyear': 53, 'dayofyear': 365
    }
    for col, max_val in cyclical_cols.items():
        df1[f'{col}_sin'] = np.sin(df1[col] * 2 * np.pi / max_val)
        df1[f'{col}_cos'] = np.cos(df1[col] * 2 * np.pi / max_val)
    return df1

def calculate_lag_features(df1, lag_intervals):
    """Calculate lag features using merge_asof with correct sorting order."""
    # Create a copy of the original DataFrame for merging
    original_df = df1[['id', 'Response_date', 'Unique_predictors']].copy()

    # Iterate over each lag interval and compute lag features using merge_asof
    for lag_name, lag_delta in lag_intervals.items():
        # Calculate the lagged time in the left DataFrame
        df1[f'{lag_name}_time'] = df1['Response_date'] - lag_delta

        # Prepare the right DataFrame from the original data
        right_df = original_df.copy()
        right_df.rename(columns={
            'Response_date': f'{lag_name}_response_date',
            'Unique_predictors': f'{lag_name}'
        }, inplace=True)

        # Sort both DataFrames by the datetime feature first, then 'id'
        df1 = df1.sort_values([f'{lag_name}_time', 'id'])
        right_df = right_df.sort_values([f'{lag_name}_response_date', 'id'])

        # Perform as-of merge within each 'id' group
        df1 = pd.merge_asof(
            df1,
            right_df,
            left_on=f'{lag_name}_time',
            right_on=f'{lag_name}_response_date',
            by='id',
            direction='backward'
        )

        # Drop unnecessary columns
        df1.drop(columns=[f'{lag_name}_time', f'{lag_name}_response_date'], inplace=True)

    return df1

def calculate_agg_features(df1, groupby_col='id'):
    """Calculate mean and median cyclical features."""
    agg_cols = ['hour', 'weekday', 'month', 'day', 'minute', 'weekofyear', 'dayofyear']
    mean_dict = {f'{col}_sin': 'mean' for col in agg_cols}
    mean_dict.update({f'{col}_cos': 'mean' for col in agg_cols})
```

```
    means = df1.groupby(groupby_col).agg(mean_dict).reset_index()

    median_dict = {f'{col}_sin': 'median' for col in agg_cols}
    median_dict.update({f'{col}_cos': 'median' for col in agg_cols})
    medians = df1.groupby(groupby_col).agg(median_dict).reset_index()

    return means, medians

def generate_bedtime_feature(df1, bedtime_hour=12, bedtime_minute=30):
    """Generate after_bedtime feature based on given bedtime hour and minute."""
    df1['after_bedtime'] = (
        (df1['hour'] < bedtime_hour) |
        ((df1['hour'] == bedtime_hour) & (df1['minute'] > bedtime_minute))
    ).astype(int)
    return df1

def combine_features(df1, means, medians, time_between_mean, unique_predictors, lag_features):
    """Combine all generated features into the final dataframe using both means and medians."""
    # Merge mean and median features on 'id'
    combined = means.merge(medians, on='id', suffixes=('_mean', '_median'))

    # Create a DataFrame from unique_predictors and lag_features
    other_features = pd.DataFrame({
        'id': unique_predictors.index,
        'unique_predictors': unique_predictors.values,
        'after_bedtime': df1.groupby('id')['after_bedtime'].first().values,
        'Time_between_response_mean': time_between_mean['Time_between_response_mean'].values
    })

    # Add lag features
    for lag_name, lag_series in lag_features.items():
        other_features[lag_name] = lag_series.values

    # Merge all features
    combined = combined.merge(other_features, on='id', how='left')

    return combined

def Feature_generation(df1):
    """Main function to generate all features."""
    df1 = df1.copy()

    #df1['publish_time'] = pd.to_datetime(df1['publish_time'], errors='coerce')
    df1['Response_date'] = pd.to_datetime(df1['Response_date'], errors='coerce')

    # Step 1: Extract time features
    df1 = extract_time_features(df1)

    # Step 2: Generate cyclical features
    df1 = generate_cyclical_features(df1)

    # Step 3: Generate bedtime feature
    df1 = generate_bedtime_feature(df1)

    # Step 4: Calculate mean and median for full and half datasets
    full_means, full_medians = calculate_agg_features(df1)
    half_df = df1.groupby('id').apply(get_half_rows).reset_index(drop=True)
    half_means, half_medians = calculate_agg_features(half_df)

    # Step 5: Add lag features using the optimized method
    # Define lag intervals using pandas Timedelta objects
    lag_intervals = {
        'lag_2day': pd.Timedelta(days=2),
        'lag_1day': pd.Timedelta(days=1),
        'lag_5hours': pd.Timedelta(hours=5),
        'lag_1hour': pd.Timedelta(hours=1)
    }

    # Calculate lag features for the full dataset
    df1 = calculate_lag_features(df1, lag_intervals)

    # Calculate lag features for the half dataset
    half_df = calculate_lag_features(half_df, lag_intervals)

    # Step 6: Calculate Time_between_response_mean per 'id' for full_df1
    df1 = df1.sort_values(by=['id', 'Response_date'])
    df1['Time_between_response'] = df1.groupby('id')['Response_date'].diff().dt.total_seconds()
    time_between_mean_full = df1.groupby('id')['Time_between_response'].mean().reset_index(name='Time_between_response_mean')
    df1 = df1.merge(time_between_mean_full, on='id', how='left')

    # Step 6.1: Calculate Time_between_response_mean for half_df
    half_df = half_df.sort_values(by=['id', 'Response_date'])
    half_df['Time_between_response'] = half_df.groupby('id')['Response_date'].diff().dt.total_seconds()
    time_between_mean_half = half_df.groupby('id')['Time_between_response'].mean().reset_index(name='Time_between_response_mean')
    half_df = half_df.merge(time_between_mean_half, on='id', how='left')

    # Step 7: Get unique_predictors for full and half datasets
    unique_predictors_full = df1.groupby('id')['Unique_predictors'].max()
    unique_predictors_half = half_df.groupby('id')['Unique_predictors'].max()

    # Step 8: Combine lag features for full and half datasets
    lag_features_full = {
```

```
        lag_name: df1.groupby('id')[lag_name].last() for lag_name in lag_intervals.keys()
    }

    lag_features_half = {
        lag_name: half_df.groupby('id')[lag_name].last() for lag_name in lag_intervals.keys()
    }

    # Step 9: Combine both full and half data aggregation
    df1_n_full = combine_features(
        df1,
        full_means, full_medians,
        time_between_mean_full,
        unique_predictors_full,
        lag_features_full
    )

    df1_n_half = combine_features(
        half_df,
        half_means, half_medians,
        time_between_mean_half,
        unique_predictors_half,
        lag_features_half
    )

    # Step 10: Rename half features with '_half' suffix, excluding 'id'
    half_feature_cols = [col for col in df1_n_half.columns if col != 'id']
    df1_n_half = df1_n_half.rename(columns={col: f"{col}_half" for col in half_feature_cols})

    # Step 11: Merge full and half features on 'id'
    df1_n = df1_n_full.merge(df1_n_half, on='id', how='left')


    # Step 12: Return the final aggregated feature set
    return df1_n, df1
```

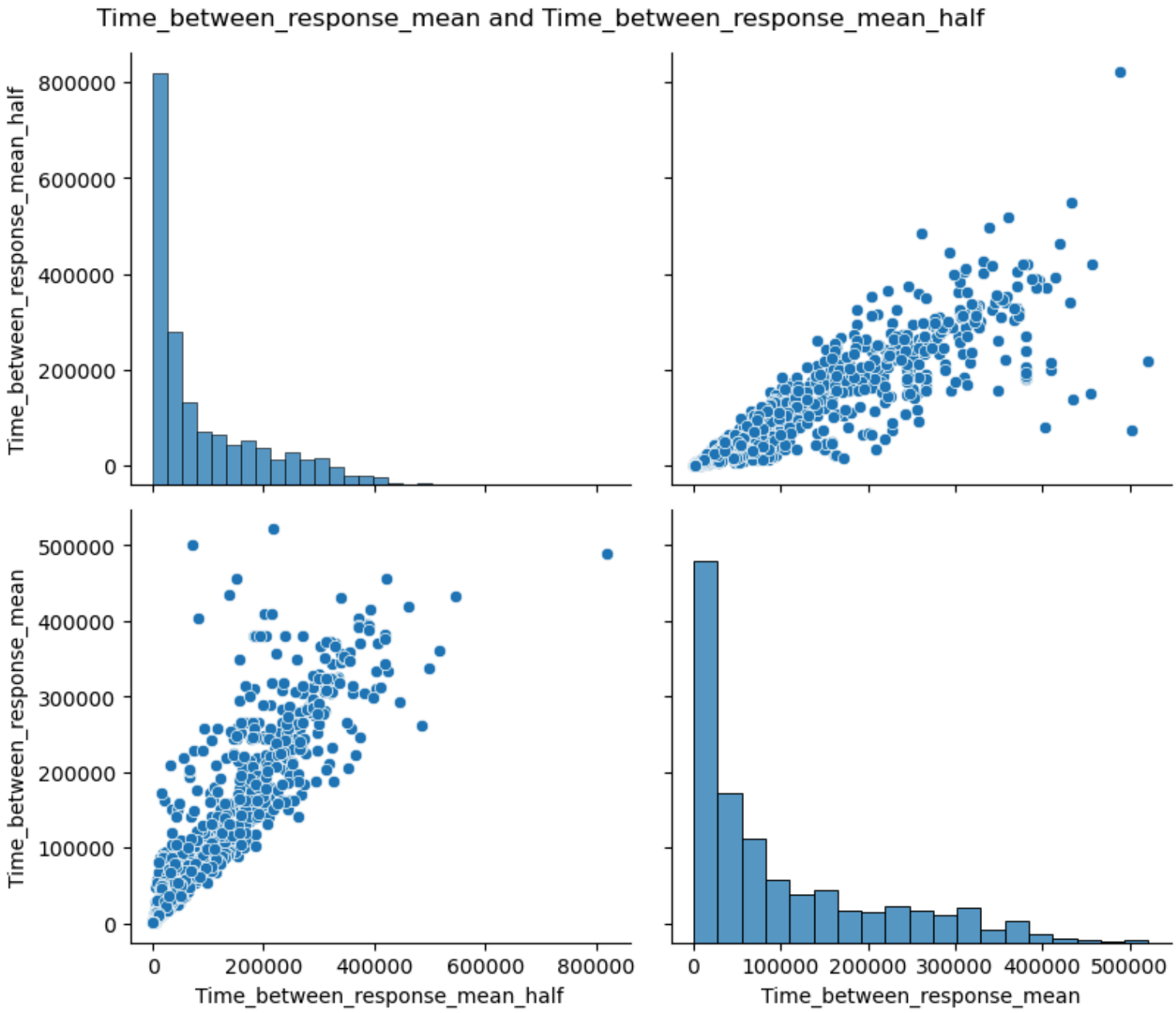
In [9]:
```
df1_n_dup,df1_no_dup = Feature_generation(merged_no_duplicates)
```

```
C:\Users\maorb\AppData\Local\Temp\ipykernel_32760\2332901589.py:132: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation. Either pass `include_groups=False` to exclude the groupings or explicitly select the grouping columns after groupby to silence this warning.
  half_df = df1.groupby('id').apply(get_half_rows).reset_index(drop=True)
```

In [975…
```
df1_n,df1_post = Feature_generation(merged_df1)
```

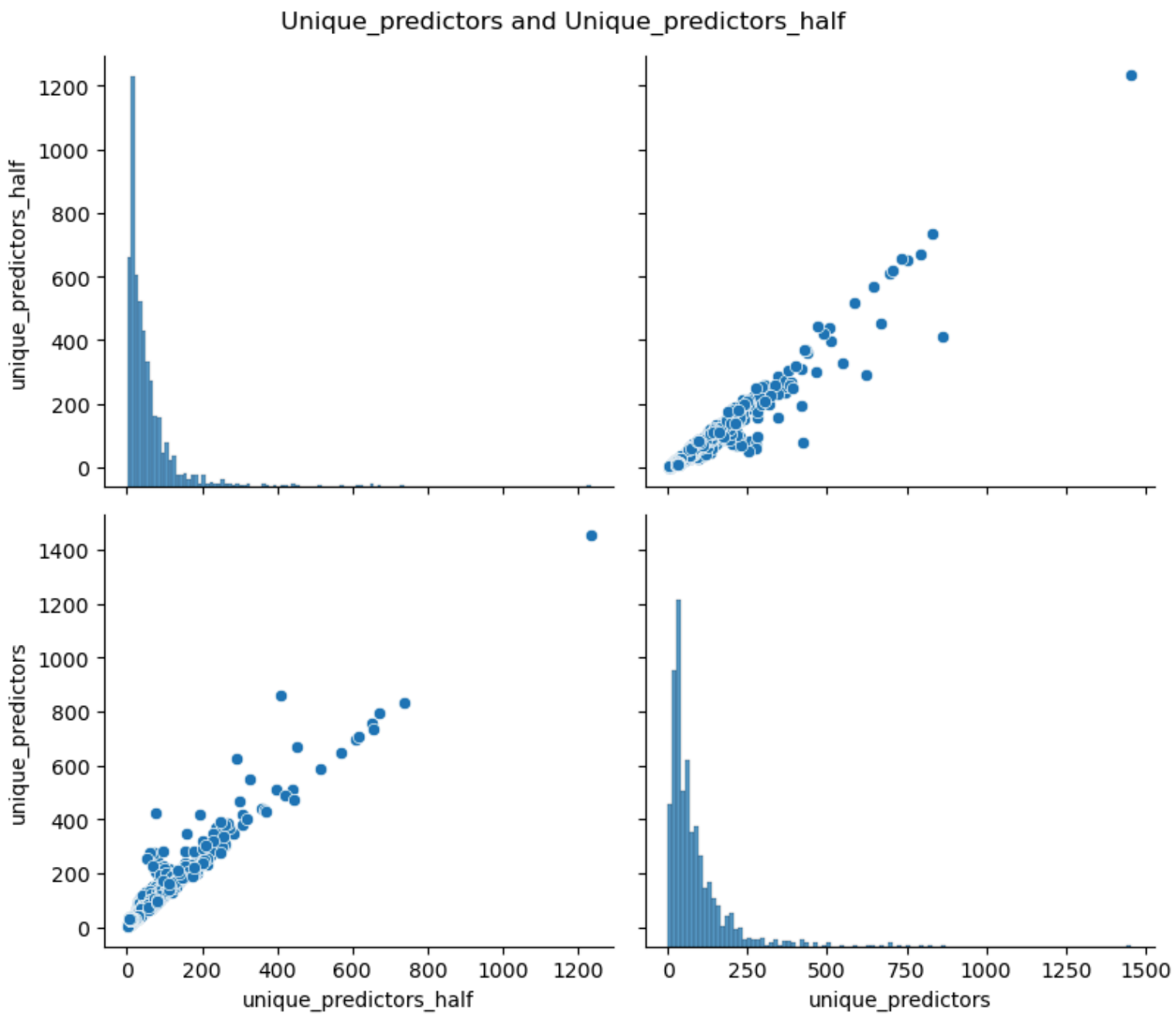
Comparing pivotal features- Full time and half time for each question:

In [175…
```
ax1 = sns.pairplot(df1_n_dup[['Time_between_response_mean_half', 'Time_between_response_mean']])
plt.suptitle('Time_between_response_mean and Time_between_response_mean_half', y=1.02)
ax1.fig.set_size_inches(9, 7)
```



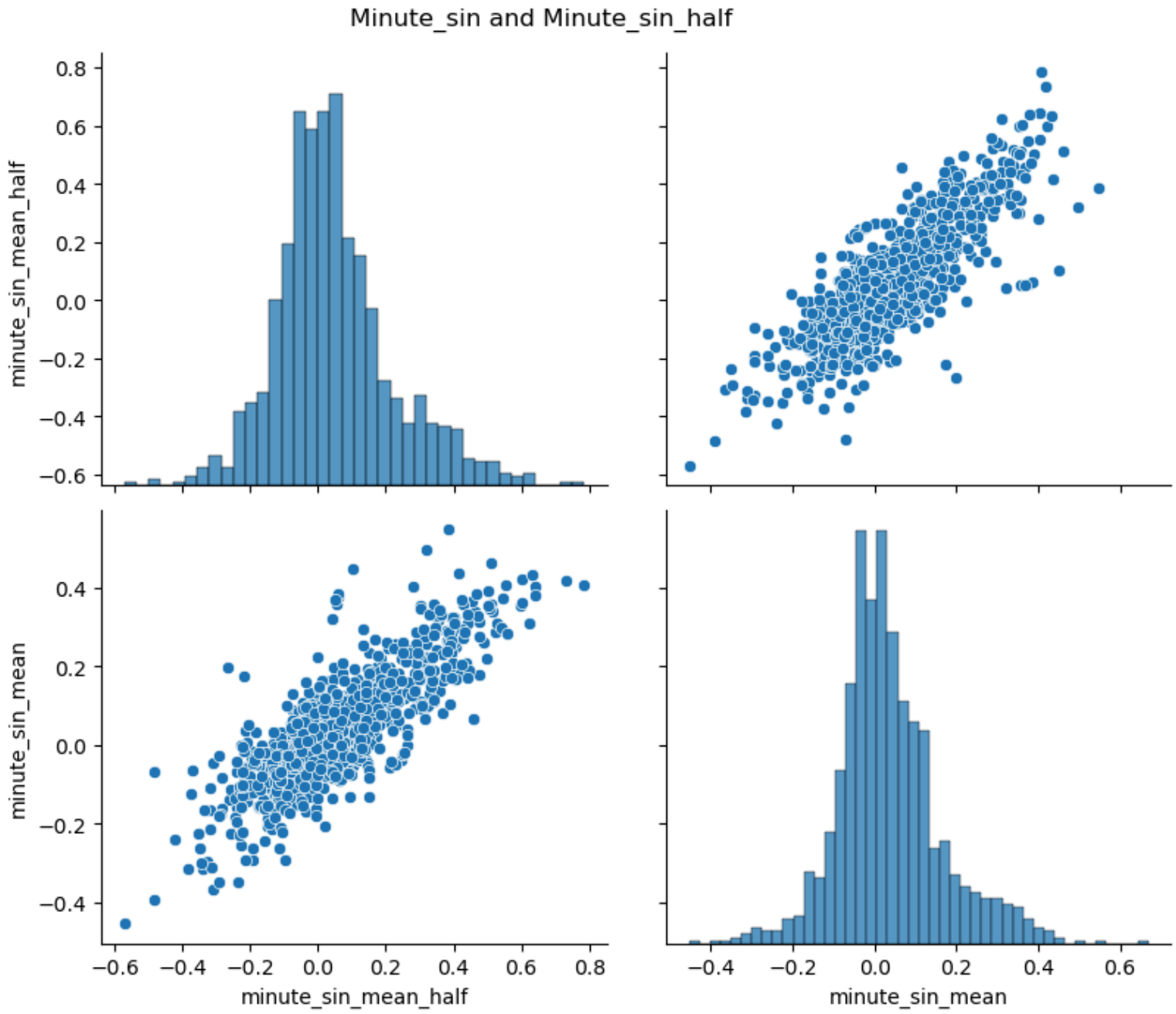
In [176…

```
ax1 = sns.pairplot(df1_n_dup[['unique_predictors_half', 'unique_predictors']])
plt.suptitle('Unique_predictors and Unique_predictors_half', y=1.02)
ax1.fig.set_size_inches(9, 7)
```



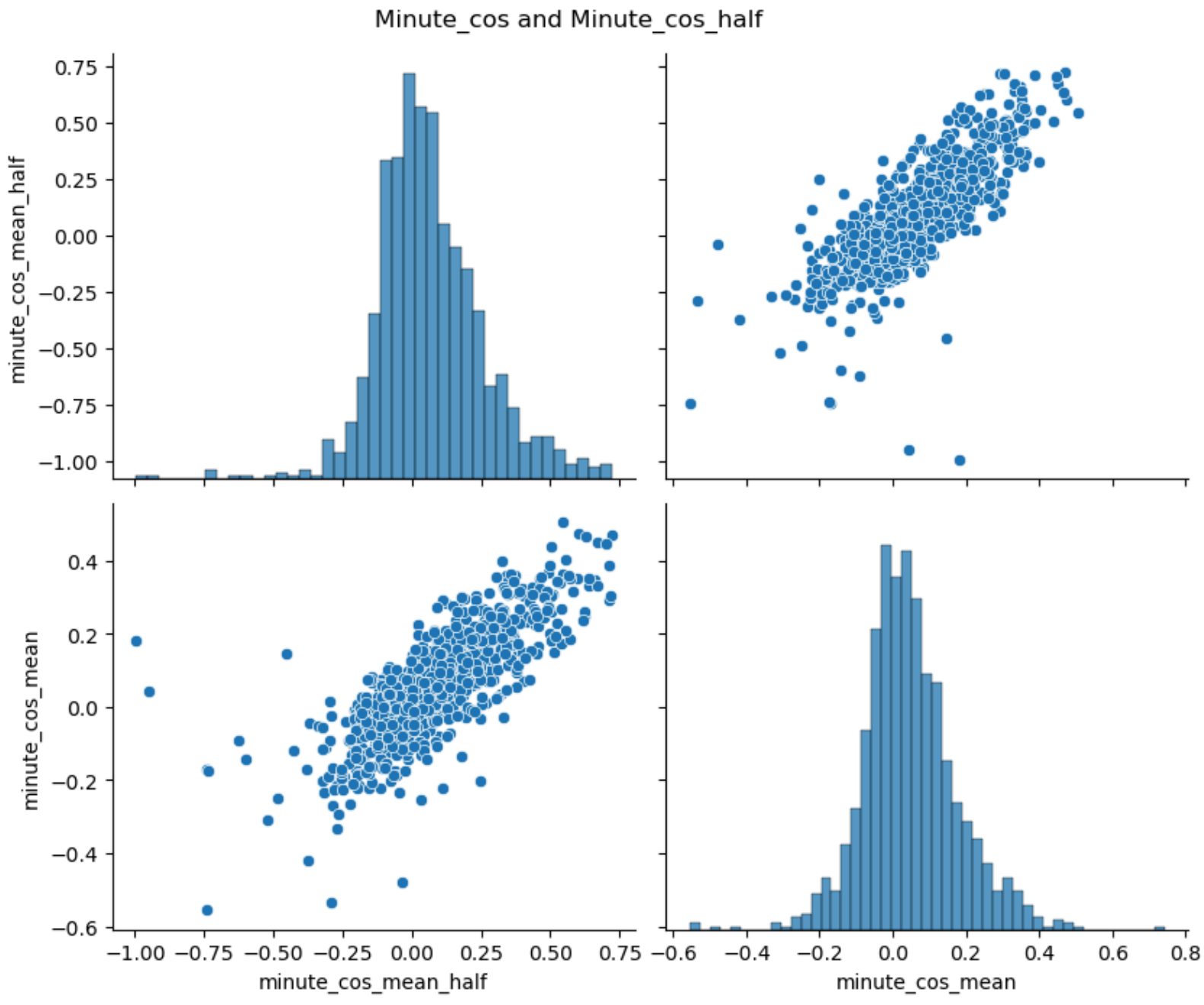
In [174…

```
ax1 = sns.pairplot(df1_n_dup[['minute_sin_mean_half', 'minute_sin_mean']])
plt.suptitle('Minute_sin and Minute_sin_half', y=1.02)
ax1.fig.set_size_inches(9, 7)
```



In [173…

```
ax1 = sns.pairplot(df1_n_dup[['minute_cos_mean_half', 'minute_cos_mean']])
plt.suptitle('Minute_cos and Minute_cos_half', y=1.02)
ax1.fig.set_size_inches(9, 7)
```

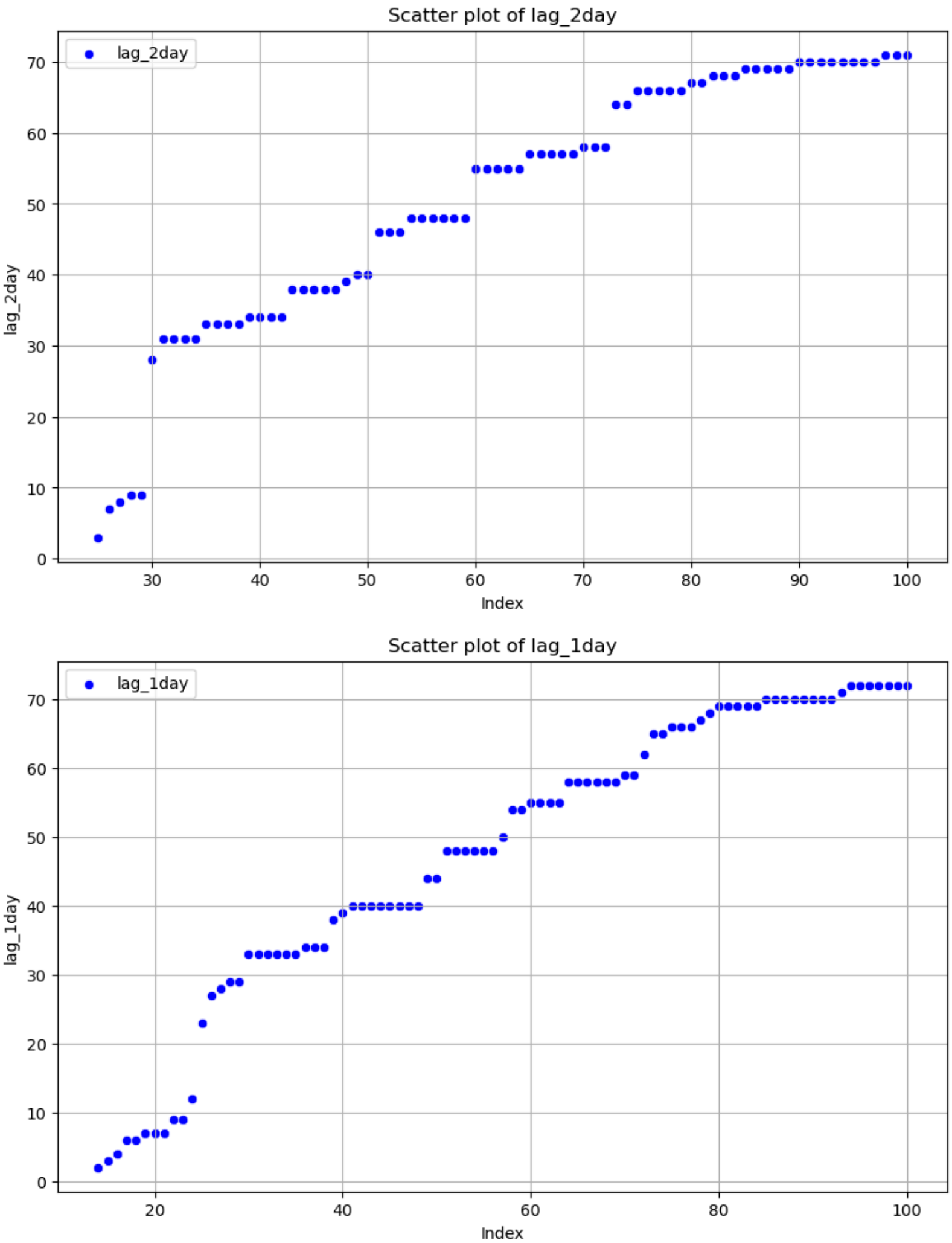


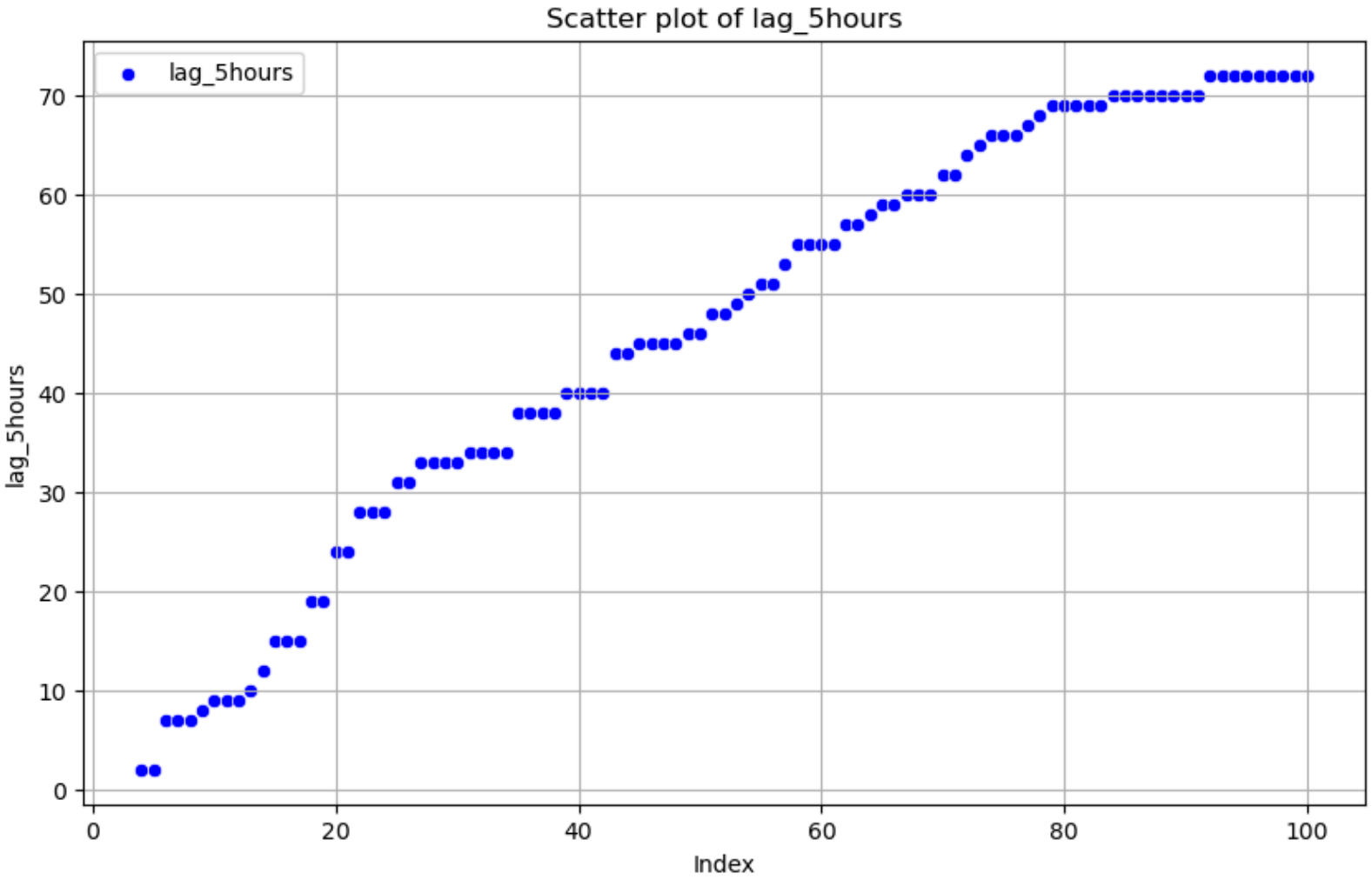
We argue that we can capture more information from looking at half time features, because the changes from some point may be minor and the answers may be biased more biased as we get close to the closing time.

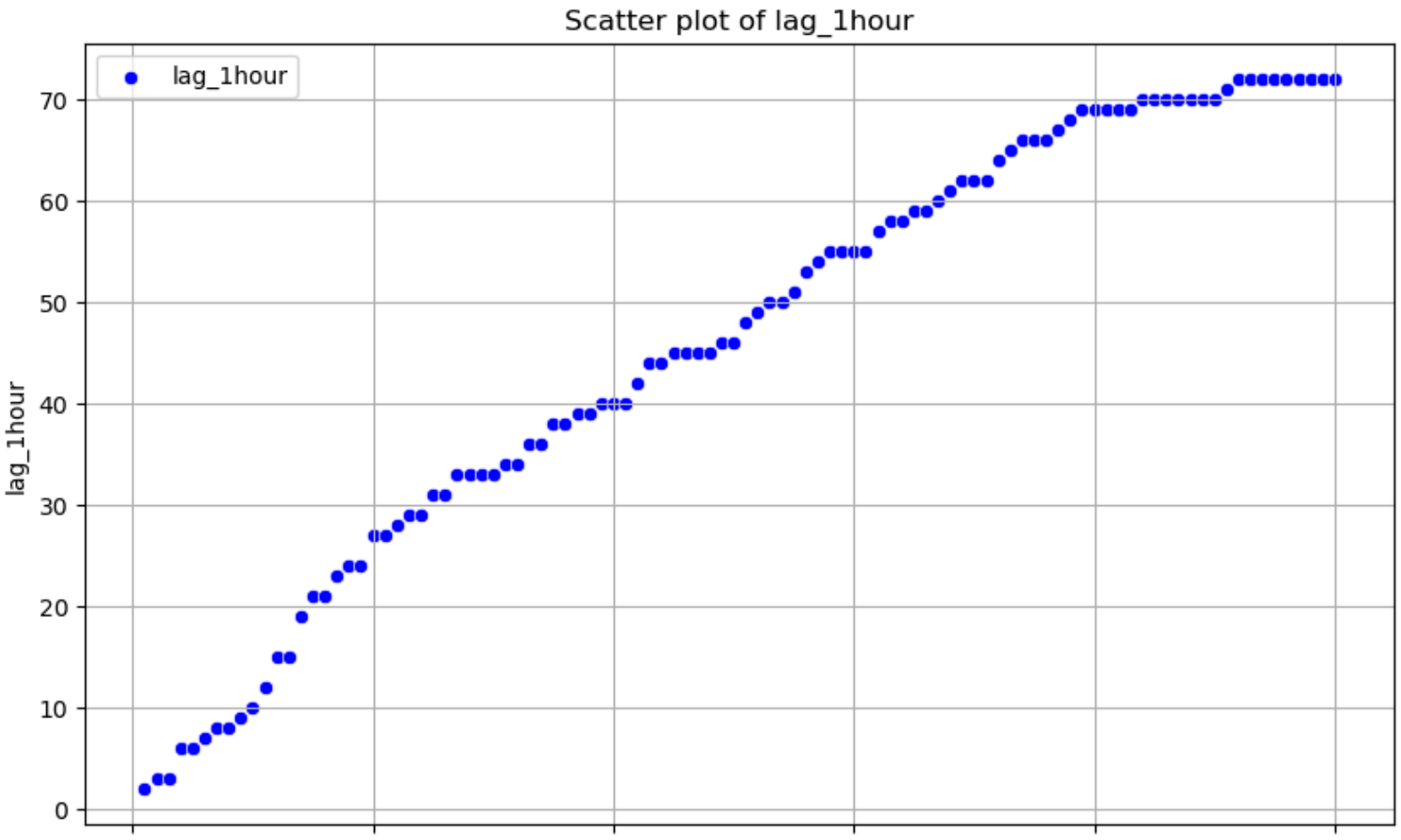
Plotting the lags for a question

In [165…

```
df1_id = df1_no_dup.copy()
df1_id = df1_id[df1_id['id'] == 3444]
import matplotlib.pyplot as plt
lag_cols = ['lag_2day', 'lag_1day', 'lag_5hours', 'lag_1hour']
for col in lag_cols:
    plt.figure(figsize=(10, 6))
    sns.scatterplot(x=df1_id.index, y=df1_id[col], label=col, color='blue')
    plt.title(f'Scatter plot of {col}')
    plt.xlabel('Index')
    plt.ylabel(col)
    plt.legend()
    plt.grid(True)
    plt.show()
```







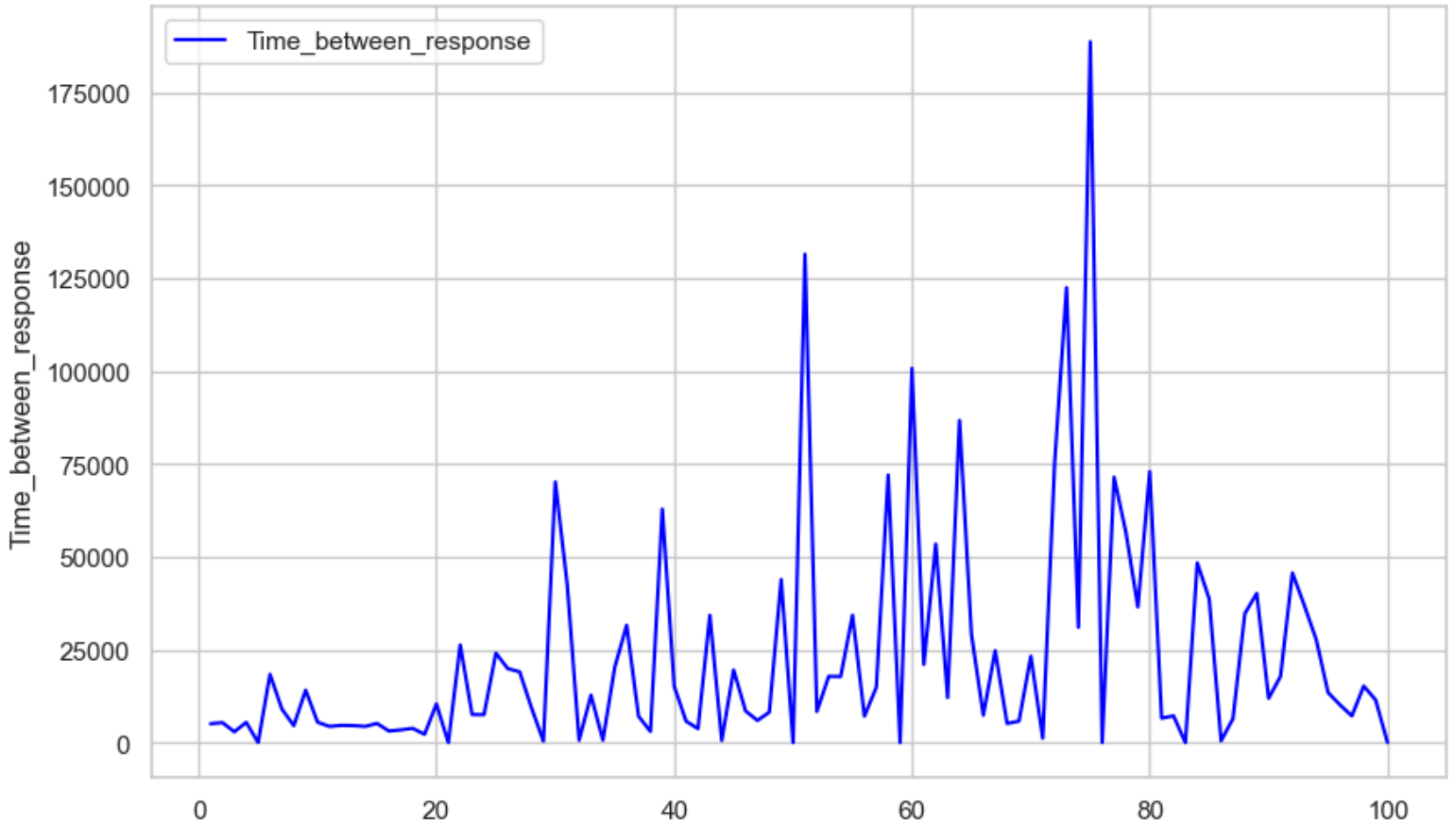
Here we can see that as we approach the last days left for the question unique predictors get reduced each day. People mainly change their current predictions

---

We capture the time between response to learn about the trend

In [18]:
```
#plot time between response over time
plt.figure(figsize=(10, 6))
sns.lineplot(x=df1_id.index, y=df1_id['Time_between_response'], label='Time_between_response', color='blue')
```

Out[18]:
```
<Axes: xlabel='None', ylabel='Time_between_response'>
```



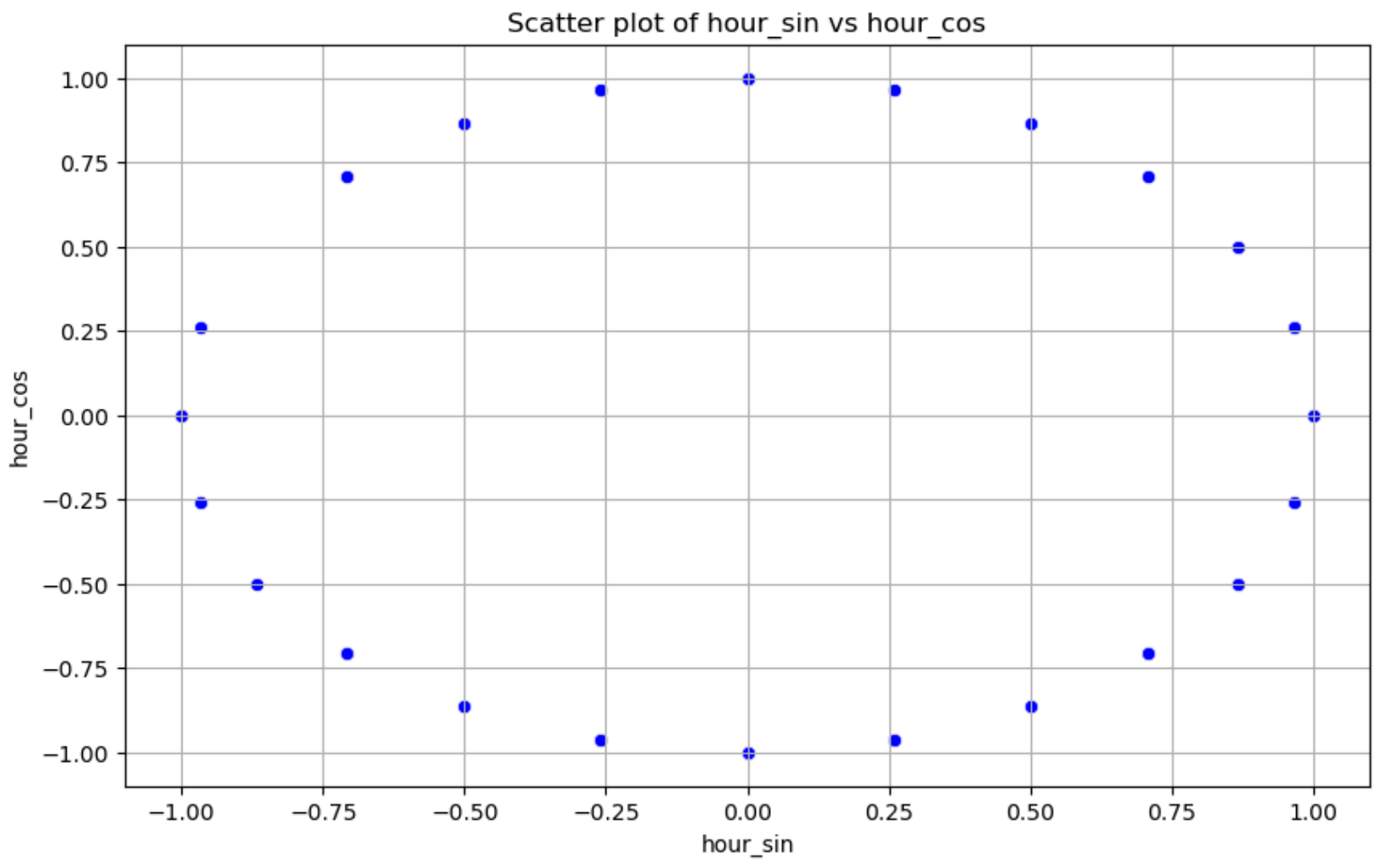
None

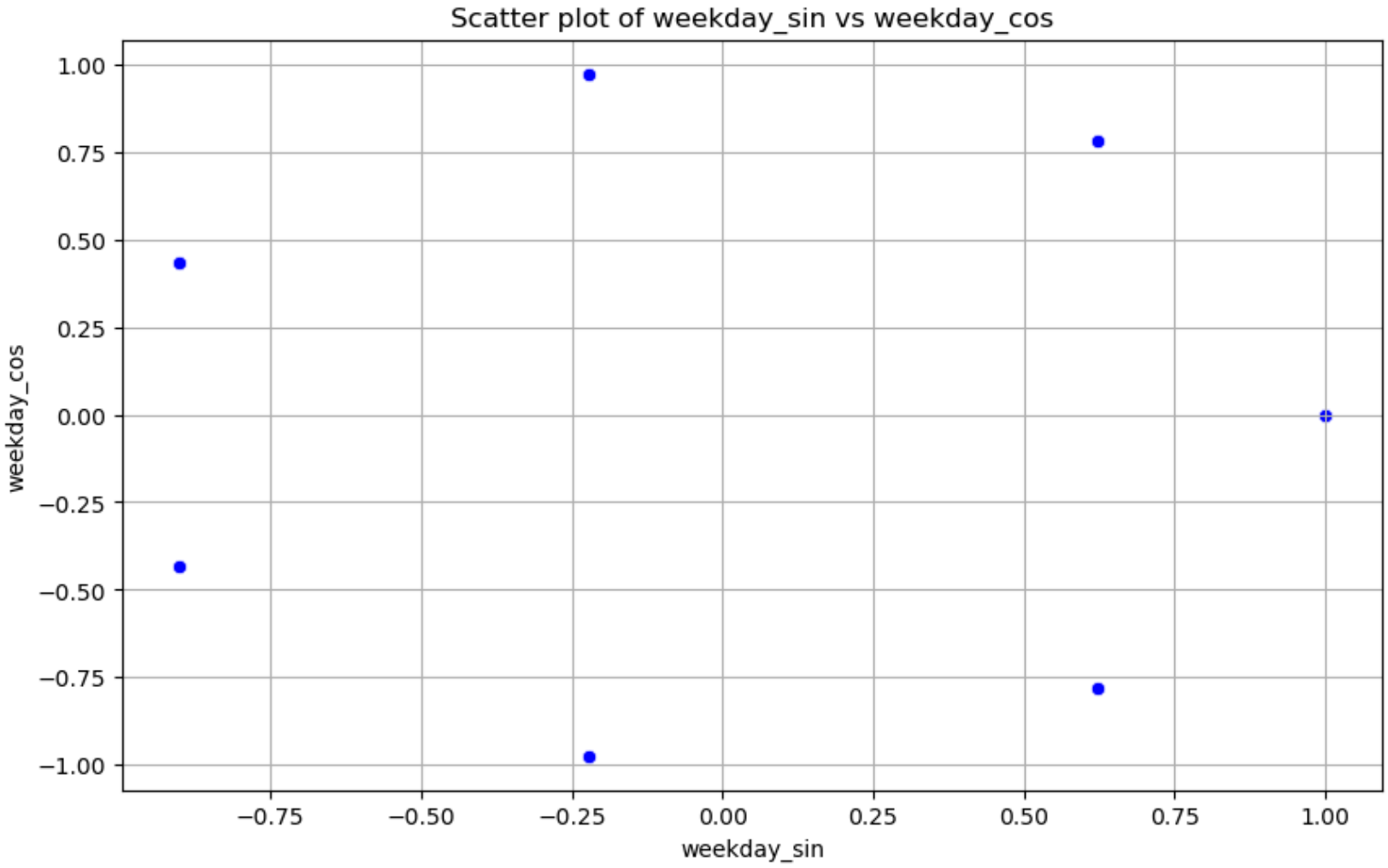
Gets increased as time progresses

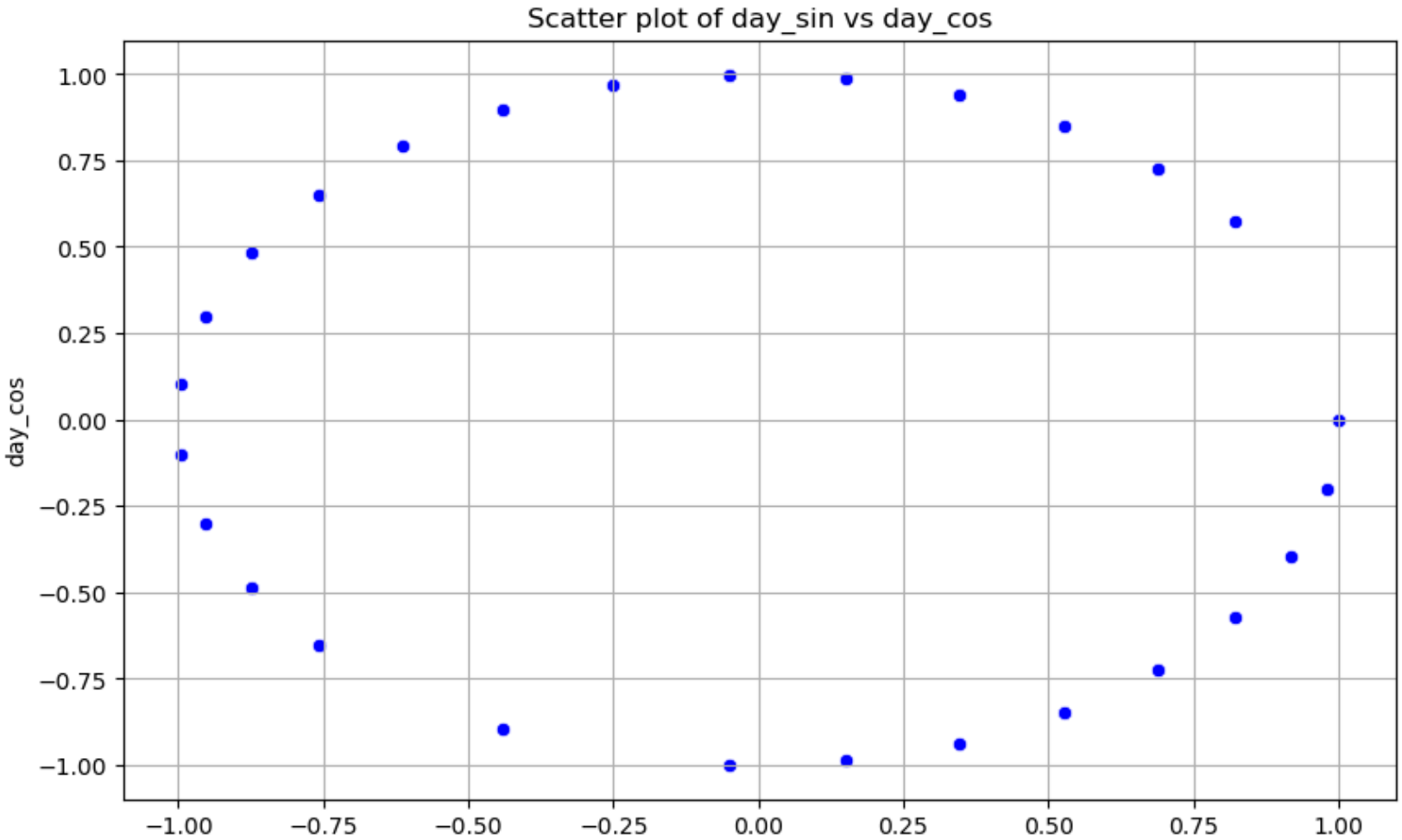
Sin and Cos features create cycle together

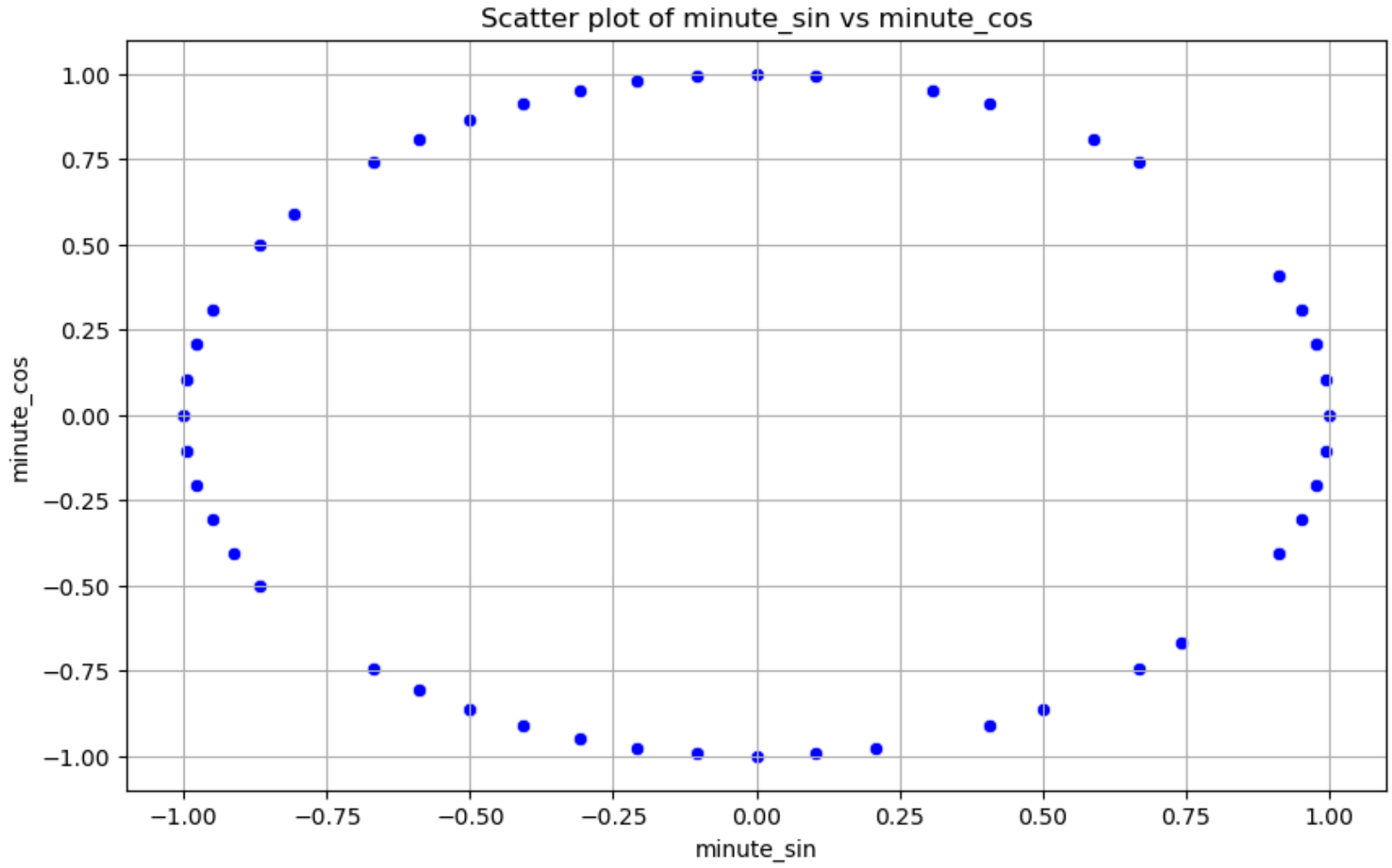
In [166…

```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming testdf is already defined as your filtered dataframe (id == 3444)

# List of cyclical feature columns (sine and cosine pairs)
list_cols_sine = ['hour_sin', 'weekday_sin', 'day_sin',
                  'minute_sin']
list_cols_cosine = ['hour_cos', 'weekday_cos',  'day_cos', 'minute_cos']

# Plot scatter plots of each pair of sine and cosine features
for sine_col, cosine_col in zip(list_cols_sine, list_cols_cosine):
    plt.figure(figsize=(10, 6))
    sns.scatterplot(x=df1_id[cosine_col], y=df1_id[sine_col], color='blue')
    plt.title(f'Scatter plot of {sine_col} vs {cosine_col}')
    plt.xlabel(sine_col)
    plt.ylabel(cosine_col)
    plt.grid(True)
    plt.show()
```




Scatter plot of weekday_sin vs weekday_cos
weekday_cos
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
weekday_sin


Scatter plot of day_sin vs day_cos
day_cos
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
day_sin



Mean time point of peak in time between response

In [21]:

```python
import pandas as pd
import numpy as np

# Sample DataFrame (Replace this with your actual df1_post)


# Step 1: Data Preparation
df1_post_cop = df1_no_dup.copy()
df1_post_cop = df1_post_cop.fillna(0)
df1_post_cop.reset_index(inplace=True)  # 'index' column added

# Step 2: Assign relative indices within each 'id'
df1_post_cop['relative_index'] = df1_post_cop.groupby('id').cumcount() + 1  # Starts at 1

# Step 3: Identify the Maximum 'Time_between_response' per 'id'
df_max = df1_post_cop.groupby('id')['Time_between_response'].max().reset_index()
df_max = df_max[df_max['Time_between_response'] != 0]  # Exclude max=0 if desired

# Step 4: Merge to Identify Max Rows
df_merged = pd.merge(
    df1_post_cop,
    df_max,
    on='id',
    how='inner',
    suffixes=('', '_max')
)

# Step 5: Filter Rows with Max 'Time_between_response'
df_max_rows = df_merged[
    (df_merged['Time_between_response'] == df_merged['Time_between_response_max']) &
    (df_merged['Time_between_response'] != 0)
]

# Step 6: Count Number of Max Rows and Get Relative Indices per 'id'
max_count_and_indices = df_max_rows.groupby('id').agg(
    num_max_rows=('Time_between_response', 'size'),
    max_relative_indices=('relative_index', list)  # Collect relative indices as lists
).reset_index()

# Step 7: Calculate Total Number of Rows per 'id'
total_rows_per_id = df1_post_cop.groupby('id').size().reset_index(name='total_rows')

# Step 8: Merge Counts and Calculate Quantile
result = pd.merge(
    max_count_and_indices,
    total_rows_per_id,
    on='id',
    how='left'
)

# Step 9: Calculate Quantile
```

```
# Step 10: Handle 'id's with All 'Time_between_response' as 0
# Identify 'id's not present in max_count_and_indices (i.e., all Time_between_response are 0)
ids_with_zero_max = total_rows_per_id[~total_rows_per_id['id'].isin(max_count_and_indices['id'])]['id'].unique()

if len(ids_with_zero_max) > 0:
    df_zero = pd.DataFrame({
        'id': ids_with_zero_max,
        'num_max_rows': 0,
        'max_relative_indices': [[] for _ in ids_with_zero_max],
        'total_rows': df1_post_cop.groupby('id').size()[ids_with_zero_max].values,
        'quantile': 0
    })
    result = pd.concat([result, df_zero], ignore_index=True)
result['max_relative_indices'] = result['max_relative_indices'].apply(lambda x: x[0] if len(x) > 0 else 0)
result['quantile'] = result['max_relative_indices'] / result['total_rows']
# Step 11: Sort the result for better readability
result = result.sort_values(by='id').reset_index(drop=True)

# Display the final result
print(result)
```

```
        id  num_max_rows  max_relative_indices  total_rows  quantile
0     3444             1                    76         101  0.752475
1     3446             1                    12         101  0.118812
2     3462             1                    80         101  0.792079
3     3483             1                    43         101  0.425743
4     3487             1                    36         101  0.356436
...    ...           ...                   ...         ...       ...
1184  27980            1                    48          50  0.960000
1185  27981            1                    50          52  0.961538
1186  27982            1                    50          56  0.892857
1187  27983            1                    50          53  0.943396
1188  28006            1                    45          59  0.762712

[1189 rows x 5 columns]
```

In [22]:
```
result['quantile'].mean() # mean of quantile
```

Out[22]: 0.5642883604534464

Train, Test sets

In [10]:
```
# Train

from sklearn.model_selection import TimeSeriesSplit, GroupKFold, StratifiedGroupKFold,StratifiedKFold
last_entries_1_no_dup = merged_no_duplicates.groupby('id').last()
last_q2_1_no_dup = last_entries_1_no_dup['q2']
y = last_q2_1_no_dup
df_train= df1_n_dup.drop(columns= ['id'])
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

In [11]:
```
df1_n_test,df1_test = Feature_generation(merged_df_test)
```

```
C:\Users\maorb\AppData\Local\Temp\ipykernel_32760\2332901589.py:132: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation. Either pass `include_groups=False` to exclude the groupings or explicitly select the grouping columns after groupby to silence this warning.
  half_df = df1.groupby('id').apply(get_half_rows).reset_index(drop=True)
```

For LGB, XGBoost,CatBoost

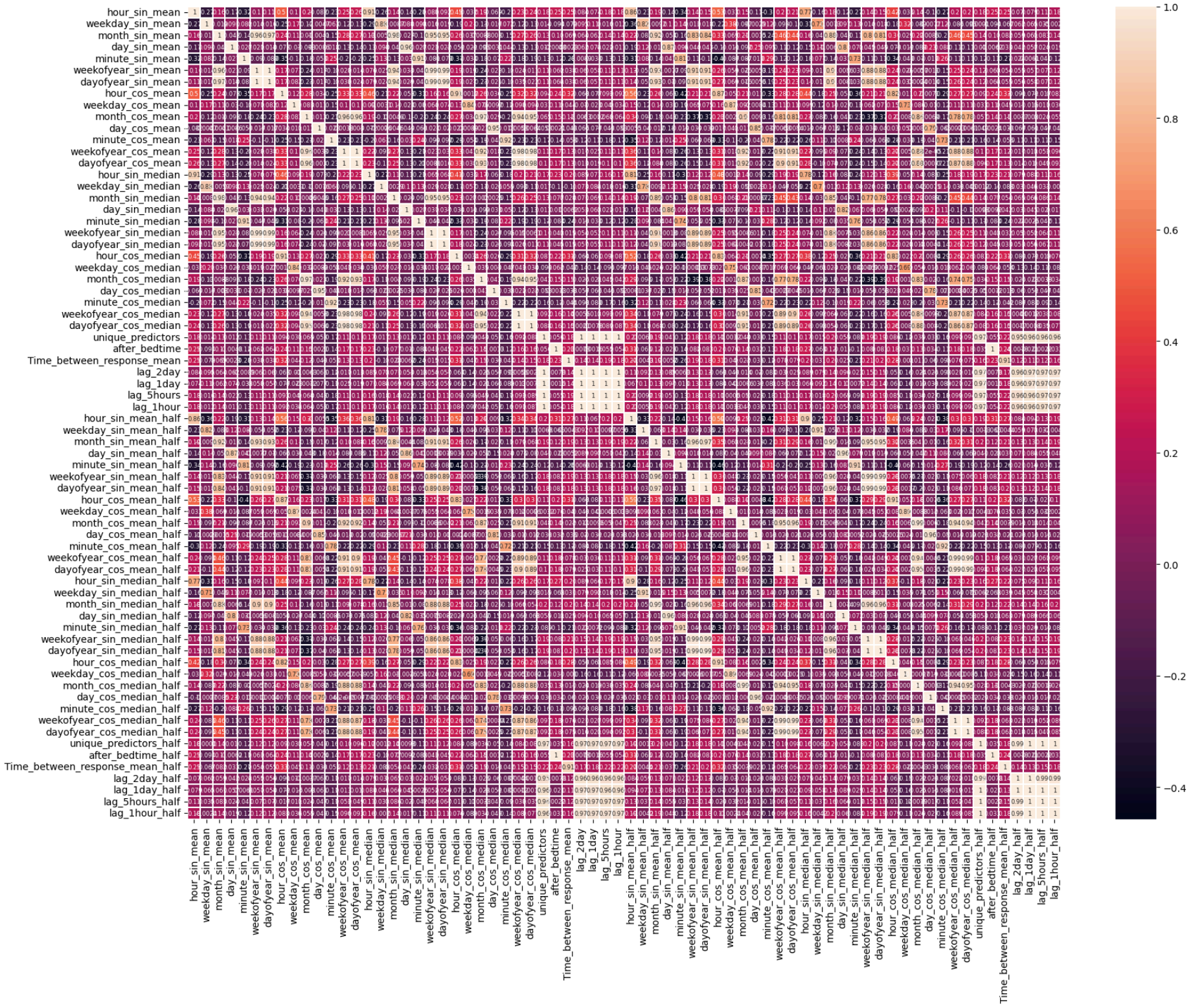
In [12]:
```
# Test

last_entries_1_test = merged_test.groupby('id').last()
last_q2_1_test = last_entries_1_test['q2']
y_test = last_q2_1_test
df_test= df1_n_test.drop(columns= ['id'])
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

For NGBoost

In [13]:
```
df_test1 = df_test.copy()
df_test1.fillna(0, inplace=True)
```

Correlation Matrix

In [14]:
```
roar = df_train.corr()

# Plot heatmap with adjusted font size
plt.figure(figsize=(20, 15))  # You can adjust the figure size if needed
sns.heatmap(roar, annot=True, linewidths=.3, annot_kws={"size": 6})  # Change the size value for smaller font

# Show the plot
plt.show()
```



Note: High correlation between different lags. We'll address it when inferring each model.

Tuning LGB

In [87]:
```python
import optuna
from lightgbm import LGBMRegressor
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from optuna.samplers import PartialFixedSampler, TPESampler
import warnings
import logging

# Suppress warnings and set logging level
warnings.filterwarnings('ignore')
logging.getLogger('lightgbm').setLevel(logging.ERROR)

# Set random seed for reproducibility
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)

# Define the objective function
def objective(trial):
    # Define parameters to be tuned by Optuna
    params = {
        "extra_trees": True,
        "reg_alpha": trial.suggest_float('reg_alpha', 0.5, 1, log=True),
        "reg_lambda": trial.suggest_float('reg_lambda', 0.4, 0.7, log=True),
        #'max_depth': trial.suggest_int('max_depth', 5, 9),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 0.95, log=True),
        'colsample_bynode': trial.suggest_float('colsample_bynode', 0.6, 0.88, log=True),
        'subsample': trial.suggest_float('subsample', 0.7, 0.95, log=True),
        #'subsample_freq': trial.suggest_int('subsample_freq', 15, 20),
        #'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1, log=True)
        # Add more parameters if needed
    }
```

```
    rmse_list = []
    kf = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

    for idx_train, idx_valid in kf.split(df_train):
        X_train, y_train_fold = df_train.iloc[idx_train], y.iloc[idx_train]
        X_valid, y_valid_fold = df_train.iloc[idx_valid], y.iloc[idx_valid]

        # Initialize and train the model
        model = LGBMRegressor(
            num_leaves=fixed_params['num_leaves'],  # Fixed parameter
            n_estimators=fixed_params['n_estimators'],  # Fixed parameter
            verbose=fixed_params['verbose'],
            boosting_type=fixed_params['boosting_type'],
            objective=fixed_params['objective'],
            metric=fixed_params['metric'],
            learning_rate=fixed_params['learning_rate'],  # Fixed parameter
            max_depth=fixed_params['max_depth'],  # Fixed parameter
            subsample_freq=fixed_params['subsample_freq'],  # Fixed parameter
            min_child_samples=fixed_params['min_child_samples'],  # Fixed parameter
            **params,  # Tuned parameters
            random_state=RANDOM_SEED
        )

        model.fit(
            X_train, y_train_fold,
            eval_set=[(X_valid, y_valid_fold)],
            eval_metric='rmse',

        )

        # Predict and calculate RMSE
        y_pred = model.predict(X_valid, num_iteration=model.best_iteration_)
        rmse = mean_squared_error(y_valid_fold, y_pred, squared=False)
        rmse_list.append(rmse)

    # Return the average RMSE across folds
    mean_rmse = np.mean(rmse_list)
    return mean_rmse

# Define fixed parameters
fixed_params = {'max_depth':8,
    'verbose': -1,
    "boosting_type": "gbdt",
    'num_leaves': 80,  # Fixed parameter
    #"objective": "regression",
     'learning_rate': 0.01,
     'subsample_freq': 200,
    #"metric": "rmse",
    'objective': 'regression',
    'metric': 'rmse',
    'min_child_samples': 4,


    'n_estimators': 2000,  # Fixed parameter
    'early_stopping_round': 1000,
}

# Create a study with PartialFixedSampler
study = optuna.create_study(direction='minimize', sampler= TPESampler(seed=42))
fixed_sampler = PartialFixedSampler(fixed_params, study.sampler)
study.sampler = fixed_sampler

# Optimize the study
study.optimize(objective, n_trials=15)

# Display the best trial
print("Best Trial:")
trial = study.best_trial
print(f"  Value (RMSE): {trial.value}")
print("  Params:")
for key, value in trial.params.items():
    print(f"    {key}: {value}")
```

```
[I 2024-10-13 12:31:07,101] A new study created in memory with name: no-name-7705f21a-7b4d-4814-bdad-3de296ef4825
[I 2024-10-13 12:31:14,650] Trial 0 finished with value: 0.30623711380010654 and parameters: {'reg_alpha': 0.6482131165247735, 'reg_la
mbda': 0.6809570824453028, 'colsample_bytree': 0.8399182254029082, 'colsample_bynode': 0.7546176645481937, 'subsample': 0.734158954785
735}. Best is trial 0 with value: 0.30623711380010654.
[I 2024-10-13 12:31:23,563] Trial 1 finished with value: 0.30453385137350353 and parameters: {'reg_alpha': 0.5570947094409952, 'reg_la
mbda': 0.41321541936307926, 'colsample_bytree': 0.8933385256691748, 'colsample_bynode': 0.7553279663380414, 'subsample': 0.86897356680
14167}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:31:32,030] Trial 2 finished with value: 0.3057208014452812 and parameters: {'reg_alpha': 0.5071851795705947, 'reg_lam
bda': 0.688311442289429, 'colsample_bytree': 0.8795970993799664, 'colsample_bynode': 0.6508335262476165, 'subsample': 0.73996755708395
59}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:31:40,144] Trial 3 finished with value: 0.30556475861999355 and parameters: {'reg_alpha': 0.5677802257385179, 'reg_la
mbda': 0.47424464138936934, 'colsample_bytree': 0.7636214641422735, 'colsample_bynode': 0.7079413496300051, 'subsample': 0.76510751793
75757}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:31:46,996] Trial 4 finished with value: 0.3052874772602209 and parameters: {'reg_alpha': 0.7641103434341517, 'reg_lam
bda': 0.43247629423349243, 'colsample_bytree': 0.6862071609794556, 'colsample_bynode': 0.6903808506067229, 'subsample': 0.804608431945
385}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:31:53,611] Trial 5 finished with value: 0.30695270464532837 and parameters: {'reg_alpha': 0.8616512464445057, 'reg_la
mbda': 0.44728910283201556, 'colsample_bytree': 0.7599381166821831, 'colsample_bynode': 0.7528152495809792, 'subsample': 0.71000033275
35128}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:00,542] Trial 6 finished with value: 0.305954285755685 and parameters: {'reg_alpha': 0.7618320309633697, 'reg_lamb
da': 0.4400518403490205, 'colsample_bytree': 0.6182067604078432, 'colsample_bynode': 0.8629402593209994, 'subsample': 0.94008156066334
65}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:07,095] Trial 7 finished with value: 0.30575605475816997 and parameters: {'reg_alpha': 0.8756324638155135, 'reg_la
mbda': 0.47434325276125844, 'colsample_bytree': 0.6275435991320644, 'colsample_bynode': 0.7797595501402845, 'subsample': 0.80070679361
3676}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:14,545] Trial 8 finished with value: 0.3054300967535293 and parameters: {'reg_alpha': 0.5441356415310549, 'reg_lam
bda': 0.5277239704808746, 'colsample_bytree': 0.6095568955227525, 'colsample_bynode': 0.8499625791881501, 'subsample': 0.7575632200366
137}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:21,308] Trial 9 finished with value: 0.3057365370671264 and parameters: {'reg_alpha': 0.791423760502661, 'reg_lamb
da': 0.47623097901249845, 'colsample_bytree': 0.7619780308939705, 'colsample_bynode': 0.7397523414256926, 'subsample': 0.7406524549407
941}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:29,645] Trial 10 finished with value: 0.3050540139031622 and parameters: {'reg_alpha': 0.6404883247499067, 'reg_la
mbda': 0.4013447025342788, 'colsample_bytree': 0.9367939013317815, 'colsample_bynode': 0.6041417853916213, 'subsample': 0.902940038098
7028}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:38,208] Trial 11 finished with value: 0.30496842384593636 and parameters: {'reg_alpha': 0.6372613330439074, 'reg_l
ambda': 0.40330210779977116, 'colsample_bytree': 0.9412267498198826, 'colsample_bynode': 0.6199929291176889, 'subsample': 0.8840689554
290414}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:46,656] Trial 12 finished with value: 0.3052869861527301 and parameters: {'reg_alpha': 0.6150823851877983, 'reg_la
mbda': 0.4019513141970534, 'colsample_bytree': 0.938540613634478, 'colsample_bynode': 0.6120374842437368, 'subsample': 0.8638252244436
901}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:32:55,174] Trial 13 finished with value: 0.30547129948406404 and parameters: {'reg_alpha': 0.6805885103969489, 'reg_l
ambda': 0.5884771626763932, 'colsample_bytree': 0.8537843275410335, 'colsample_bynode': 0.8116537464120723, 'subsample': 0.85261921436
53494}. Best is trial 1 with value: 0.30453385137350353.
[I 2024-10-13 12:33:04,089] Trial 14 finished with value: 0.30548847364314459 and parameters: {'reg_alpha': 0.5803752592643534, 'reg_la
mbda': 0.5374404419492861, 'colsample_bytree': 0.892974118331707, 'colsample_bynode': 0.6665100867042353, 'subsample': 0.8671418079267
429}. Best is trial 1 with value: 0.30453385137350353.
Best Trial:
  Value (RMSE): 0.30453385137350353
  Params:
    reg_alpha: 0.5570947094409952
    reg_lambda: 0.41321541936307926
    colsample_bytree: 0.8933385256691748
    colsample_bynode: 0.7553279663380414
    subsample: 0.8689735668014167
```

In [88]:
```
best_params = trial.params
all_fParams = {**fixed_params, **best_params}
all_fParams
```

Out[88]:
```
{'max_depth': 8,
 'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 80,
 'learning_rate': 0.01,
 'subsample_freq': 200,
 'objective': 'regression',
 'metric': 'rmse',
 'min_child_samples': 4,
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.5570947094409952,
 'reg_lambda': 0.41321541936307926,
 'colsample_bytree': 0.8933385256691748,
 'colsample_bynode': 0.7553279663380414,
 'subsample': 0.8689735668014167}
```

Tuning XGB

In [ ]:
```
import optuna
from xgboost import XGBRegressor
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
import warnings
import logging
from optuna.samplers import TPESampler

# Suppress warnings and set logging level
warnings.filterwarnings('ignore')
logging.getLogger('xgboost').setLevel(logging.ERROR)
```

```
# Set random seed for reproducibility
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)

# Define fixed parameters for XGBoost
fixed_params_xgb = {
    'max_depth': 8,
    'verbosity': 0,
    'objective': 'reg:squarederror',  # Use 'binary:logistic' for classification
    'eval_metric': 'rmse',
    'learning_rate': 0.03,
    'min_child_weight': 4,
    'n_estimators': 2000,
    'early_stopping_rounds': 50,  # Reasonable value to prevent long training
}

# Define the objective function for XGBoost Regressor
def objective_xgb(trial):
    # Define parameters to be tuned by Optuna
    params = {
        "booster": trial.suggest_categorical('booster', ['gbtree', 'gblinear', 'dart']),
        "reg_alpha": trial.suggest_float('reg_alpha', 0.5, 1.0, log=True),
        "reg_lambda": trial.suggest_float('reg_lambda', 0.4, 0.7, log=True),
        "colsample_bytree": trial.suggest_float('colsample_bytree', 0.6, 0.95, log=True),
        "colsample_bylevel": trial.suggest_float('colsample_bylevel', 0.6, 0.95, log=True),
        "subsample": trial.suggest_float('subsample', 0.6, 0.95, log=True),
        # Add more parameters if needed
    }
    rmse_list = []
    kf = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

    for idx_train, idx_valid in kf.split(df_train):
        X_train, y_train_fold = df_train.iloc[idx_train], y.iloc[idx_train]
        X_valid, y_valid_fold = df_train.iloc[idx_valid], y.iloc[idx_valid]


        # Initialize and train the model
        model = XGBRegressor(
            max_depth=fixed_params_xgb['max_depth'],
            n_estimators=fixed_params_xgb['n_estimators'],
            verbosity=fixed_params_xgb['verbosity'],
            objective=fixed_params_xgb['objective'],
            eval_metric=fixed_params_xgb['eval_metric'],
            learning_rate=fixed_params_xgb['learning_rate'],
            min_child_weight=fixed_params_xgb['min_child_weight'],
            early_stopping_rounds= fixed_params_xgb['early_stopping_rounds'],
            **params,  # Tuned parameters
            random_state=RANDOM_SEED
        )

        model.fit(
            X_train, y_train_fold,
            eval_set=[(X_valid, y_valid_fold)],
            #early_stopping_rounds=50,
            verbose=False
        )

        # Predict and calculate RMSE
        y_pred = model.predict(X_valid)
        rmse = mean_squared_error(y_valid_fold, y_pred, squared=False)
        rmse_list.append(rmse)

    # Return the average RMSE across folds
    mean_rmse = np.mean(rmse_list)
    return mean_rmse

# Create a study with TPE sampler
study_xgb = optuna.create_study(direction='minimize', sampler=TPESampler(seed=RANDOM_SEED))

# Optimize the study
study_xgb.optimize(objective_xgb, n_trials=20)

# Display the best trial
print("Best Trial for XGBoost Regressor:")
trial = study_xgb.best_trial
print(f"  Value (RMSE): {trial.value}")
print("  Params:")
for key, value in trial.params.items():
    print(f"    {key}: {value}")
```

LGB Model

In [125…

```
np.random.seed(42)
import warnings
warnings.filterwarnings('ignore')
# df_train is the training dataframe and y is the target variable
epsilon = 1e-6
```

```
params = {'max_depth': 8,
 'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 80,
 'learning_rate': 0.05,
 'subsample_freq': 200,
 'objective': 'regression',
 'metric': 'rmse',
 'min_child_samples': 4,
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.5424194717109394,
 'reg_lambda': 0.49550064897571744,
 'colsample_bytree': 0.9366788569612681,
 'colsample_bynode': 0.6301043124936729,
 'subsample': 0.831476319086343}

famParams = {'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 64,
 'objective': 'regression',
 'learning_rate': 0.01,
 'metric': 'rmse',
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.12776485418442662,
 'reg_lambda': 1.7023282716867387,
 'max_depth': 7,
 'colsample_bytree': 0.9138159148710222,
 'colsample_bynode': 0.8879864738697725,
 'subsample': 0.7747174812920734,
 'subsample_freq': 10}


This_the_one = {'max_depth': 8,
 'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 64,
 'objective': 'regression',
 'learning_rate': 0.02,
 'subsample_freq': 200,
 'metric': 'rmse',
 'min_child_samples': 4,
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.6150823851877983,
 'reg_lambda': 0.4019513141970534,
 'colsample_bytree': 0.9381694254851193,
 'colsample_bynode': 0.6144874410854747,
 'subsample': 0.8365551463780401}

This_no_dup = {'max_depth': 8,
 'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 64,
 'objective': 'regression',
 'learning_rate': 0.01,
 'subsample_freq': 200,
 'metric': 'rmse',
 'min_child_samples': 4,
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.510843433276398,
 'reg_lambda': 0.5758815394477095,
 'colsample_bytree': 0.6603126150898777,
 'colsample_bynode': 0.8625942585381968,
 'subsample': 0.8764818166434032}

second_try_with_full = {'max_depth': 7,
 'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 62,
 'objective': 'regression',
 'learning_rate': 0.01,
 'metric': 'rmse',
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.09099694632498756,
 'reg_lambda': 0.03124565071260872,
 'colsample_bytree': 0.656967498247354,
 'colsample_bynode': 0.634913994829205,
 'subsample': 0.8477107305826747,
 'subsample_freq': 17}

all_fParams = {'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 64,
 'objective': 'regression',
 'learning_rate': 0.01,
 'metric': 'rmse',
 'n_estimators': 2000,
 'early_stopping_round': 1000,
```

```
 'reg_alpha': 2.275151647466187,
 'reg_lambda': 0.5450853190672589,
 'max_depth': 8,
 'colsample_bytree': 0.5070512158501231,
 'colsample_bynode': 0.731892041822454,
 'subsample': 0.8477107305826747,
 'subsample_freq': 200,
 'min_child_samples': 4}

yes = {
    'boosting_type': 'gbdt',
    'objective': 'binary',            # Using 'binary' objective
    'metric': 'binary_logloss',       # Appropriate metric for binary objective
    'learning_rate': 0.01,            # Lower learning rate for better performance
    'num_leaves': 64,                 # Controls the complexity of the tree
    'max_depth': -1,
    'subsample_freq':10,                # No limit on depth; adjust as needed
    'min_data_in_leaf': 60,           # Minimum number of data in one leaf
    'feature_fraction': 0.8,          # Randomly select a subset of features
    'bagging_fraction': 0.8,          # Randomly select a subset of data
    'bagging_freq': 5,                # Perform bagging at every 5 iterations
    'lambda_l1': 0.1,                 # L1 regularization
    'lambda_l2': 0.1,                 # L2 regularization
    'verbose': -1,
    'early_stopping_rounds': 1000,     # Early stopping to prevent overfitting
    'seed': 42                        # For reproducibility
}

lst_try = {'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 64,
 'objective': 'regression',
 'learning_rate': 0.05,
 'metric': 'rmse',
 'n_estimators': 2000,
 'reg_alpha': 2.4992188841592546,
 'reg_lambda': 0.3302239707082332,
 'max_depth': 4,
 'colsample_bytree': 0.5250094467242431,
 'colsample_bynode': 0.7932460915376365,
 'subsample': 0.7537675614533189,
 'subsample_freq': 2}
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error, root_mean_squared_error, log_loss
import lightgbm as lgb
import numpy as np

# Assuming df_trainis the training dataframe and y_new is the target variable
kf = KFold(n_splits=5, shuffle=True, random_state=42)

fitted_models_lgb = []
cv_scores_lgb = []
iters = []
learning_curves= []
for idx_train, idx_valid in kf.split(df_train):
    X_train, y_train = df_train.iloc[idx_train], y.iloc[idx_train]
    X_valid, y_valid = df_train.iloc[idx_valid], y.iloc[idx_valid]
    #X_train, y_transformed_train = df_train.iloc[idx_train], y_transformed.iloc[idx_train]
    #X_valid, y_origin_valid = df_train.iloc[idx_valid], y.iloc[idx_valid]
    # Define LightGBM datasets
    train_data = lgb.Dataset(X_train, label=y_train)
    valid_data = lgb.Dataset(X_valid, label=y_valid)
    #train_data = lgb.Dataset(X_train, label=y_transformed_train)
    #valid_data = lgb.Dataset(X_valid, label=y_origin_valid)
    # Train LightGBM model
    lgb_model = lgb.train(This_the_one, train_data, valid_sets=[valid_data],
                      callbacks=[lgb.log_evaluation(200), lgb.early_stopping(1000)])
    # Evaluate model on validation data
    y_pred_valid = lgb_model.predict(X_valid)
    #y_pred_transformed = lgb_model.predict(X_valid, num_iteration=lgb_model.best_iteration)
    #y_pred_valid = 1 / (1 + np.exp(-1 * y_pred_transformed))

    rmse_score = root_mean_squared_error(y_valid, y_pred_valid)  # RMSE



    # Store model and score
    fitted_models_lgb.append(lgb_model)
    cv_scores_lgb.append(rmse_score)
    learning_curves.append(lgb_model.best_iteration)
    iters.append(idx_train)

# Display CV RMSE scores
print("CV RMSE scores: ", cv_scores_lgb)
print("min CV RMSE score: ", min(cv_scores_lgb))
print("Max CV RMSE score: ", max(cv_scores_lgb))
print("Mean CV RMSE score: ", np.mean(cv_scores_lgb))
```

```
Training until validation scores don't improve for 1000 rounds
[200]   valid_0's rmse: 0.294335
[400]   valid_0's rmse: 0.300548
[600]   valid_0's rmse: 0.30054
[800]   valid_0's rmse: 0.300413
[1000]  valid_0's rmse: 0.30036
Early stopping, best iteration is:
[128]   valid_0's rmse: 0.29137
Training until validation scores don't improve for 1000 rounds
[200]   valid_0's rmse: 0.332461
[400]   valid_0's rmse: 0.33735
[600]   valid_0's rmse: 0.336904
[800]   valid_0's rmse: 0.335438
[1000]  valid_0's rmse: 0.335053
Early stopping, best iteration is:
[55]    valid_0's rmse: 0.325594
Training until validation scores don't improve for 1000 rounds
[200]   valid_0's rmse: 0.301851
[400]   valid_0's rmse: 0.302874
[600]   valid_0's rmse: 0.301819
[800]   valid_0's rmse: 0.301168
[1000]  valid_0's rmse: 0.300807
[1200]  valid_0's rmse: 0.300807
[1400]  valid_0's rmse: 0.300807
[1600]  valid_0's rmse: 0.300807
[1800]  valid_0's rmse: 0.300807
Training until validation scores don't improve for 1000 rounds
[200]   valid_0's rmse: 0.295981
[400]   valid_0's rmse: 0.297627
[600]   valid_0's rmse: 0.296361
[800]   valid_0's rmse: 0.29567
[1000]  valid_0's rmse: 0.295769
[1200]  valid_0's rmse: 0.2961
Early stopping, best iteration is:
[223]   valid_0's rmse: 0.294671
Training until validation scores don't improve for 1000 rounds
[200]   valid_0's rmse: 0.299065
[400]   valid_0's rmse: 0.302348
[600]   valid_0's rmse: 0.302933
[800]   valid_0's rmse: 0.302063
[1000]  valid_0's rmse: 0.302063
Early stopping, best iteration is:
[191]   valid_0's rmse: 0.298878
CV RMSE scores:  [0.29137011926343054, 0.32559393847899637, 0.30080019673357755, 0.29467101463511375, 0.29887753521334215]
min CV RMSE score:  0.29137011926343054
Max CV RMSE score:  0.32559393847899637
Mean CV RMSE score:  0.30226256086489206
```

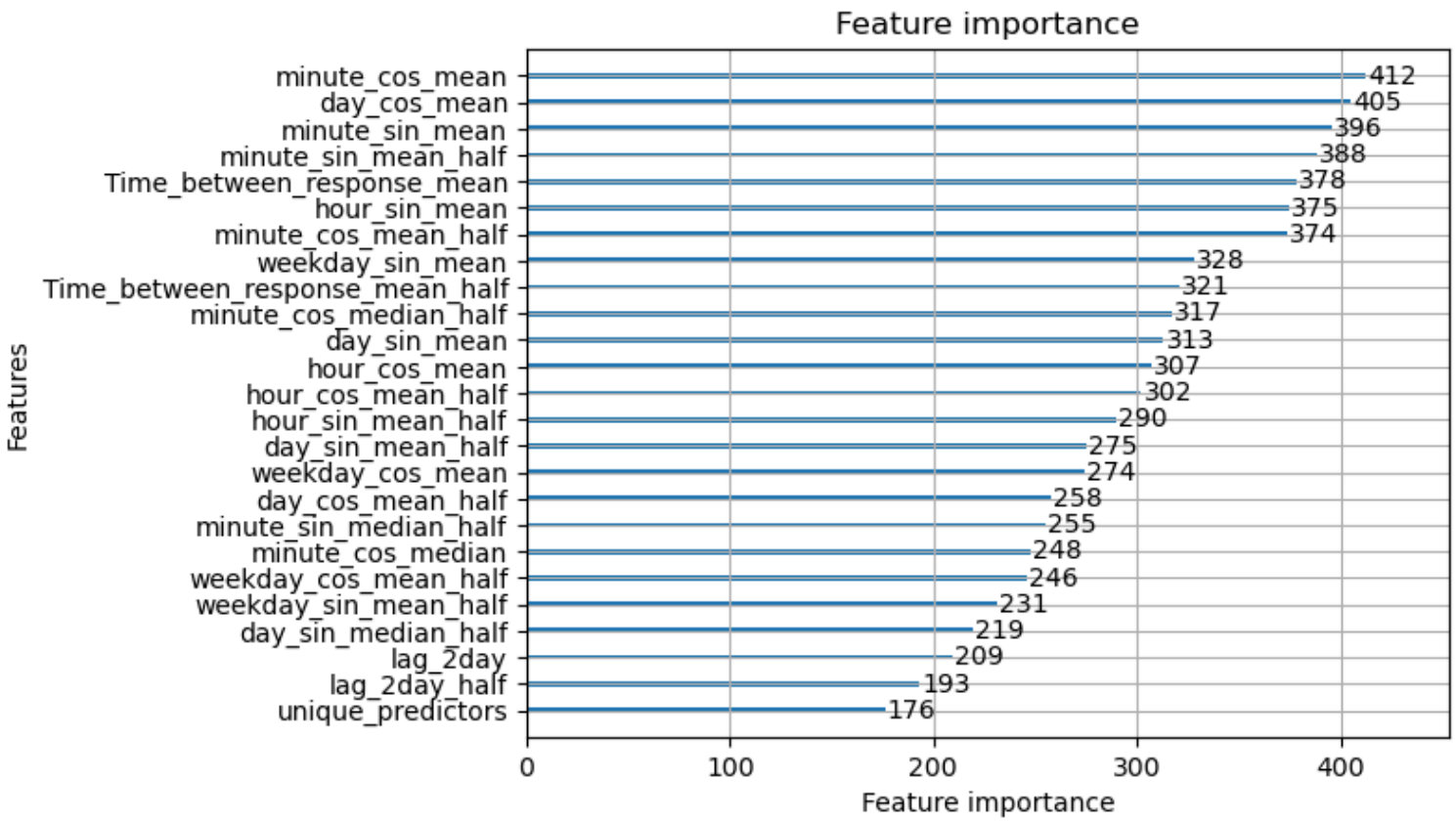
Importance

In [127…

```
# Take only the top 10 features:
lgb.plot_importance(lgb_model, max_num_features=25)
```

Out[127…

```
<Axes: title={'center': 'Feature importance'}, xlabel='Feature importance', ylabel='Features'>
```



Time between response is not that crucial, minute features seems most important. Relatively low importance to lags (only 23-24 in importance) and only captured 2 days feature.

XGB Model

In [258…

```python
import pandas as pd
import numpy as np
import xgboost as xgb
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.preprocessing import LabelEncoder

# Function to generate features
def Feature_generation(df1):
    # Convert 'publish_time' and 'Response_date' to datetime
    df1['publish_time'] = pd.to_datetime(df1['publish_time'])
    df1['Response_date'] = pd.to_datetime(df1['Response_date'], errors='coerce')

    # Step 1: Extract time features
    df1 = extract_time_features(df1)

    # Step 2: Generate cyclical features
    df1 = generate_cyclical_features(df1)

    # Step 3: Generate bedtime feature
    df1 = generate_bedtime_feature(df1)

    # Step 4: Calculate mean and median for full and half datasets
    full_means, full_medians = calculate_agg_features(df1)
    half_df = df1.groupby('id').apply(get_half_rows).reset_index(drop=True)
    half_means, half_medians = calculate_agg_features(half_df)

    # Step 5: Add lag features
    lag_intervals = {'lag_2day': 2 * 24 * 60 * 60, 'lag_1day': 24 * 60 * 60, 'lag_5hours': 5 * 60 * 60, 'lag_1hour': 60 * 60}
    df1 = calculate_lag_features(df1, lag_intervals)

    # Step 6: Last q2 and unique predictors
    last_entries = df1.groupby('id').last()
    last_q2 = last_entries['q2']
    unique_predictors = df1.groupby('id')['Unique_predictors'].max().reset_index().iloc[:,1]

    # Step 7: Combine all features
    lag_features = {
        'lag_2day': df1.groupby('id')['lag_2day'].first().reset_index().iloc[:,1],
        'lag_1day': df1.groupby('id')['lag_1day'].first().reset_index().iloc[:,1],
        'lag_5hours': df1.groupby('id')['lag_5hours'].first().reset_index().iloc[:,1],
        'lag_1hour': df1.groupby('id')['lag_1hour'].first().reset_index().iloc[:,1],
    }

    # Combine both full and half data aggregation
    df1_n_full = combine_features(df1, full_means, full_medians, unique_predictors, lag_features)
    df1_n_half = combine_features(half_df, half_means, half_medians, unique_predictors, lag_features)
    df1_n_half = df1_n_half.drop(columns = ['id','unique_predictors'])

    #Add half suffix to the half features
    df1_n_half.columns = [f'{col}_half' if col not in ['id', 'unique_predictors'] else col for col in df1_n_half.columns]
    #combinig the two dataframes
    df1_n = pd.concat([df1_n_full, df1_n_half], axis=1)

    return df1_n

# Preparing target variable
last_entries_1 = merged_no_duplicates.groupby('id').last()
last_q2_1 = last_entries_1['q2']
y = last_q2_1
df_train = df1_n_dup.drop(columns= ['id'])

# Convert the training set to DMatrix (XGBoost's preferred structure)
dtrain = xgb.DMatrix(df_train, label=y)

# XGBoost parameters
xgb_params = {
    'objective': 'reg:squarederror',
    'eval_metric': 'rmse',
    'max_depth': 6,
    'learning_rate': 0.01,
    'colsample_bytree': 0.6,
    'colsample_bynode': 0.8,
    'alpha': 0.1,
    'lambda': 10,
    'n_estimators': 2000,
    'seed': 42
}

all_xgb = {'max_depth': 8,
 'verbosity': 0,
 'objective': 'reg:squarederror',
 'eval_metric': 'rmse',
 'learning_rate': 0.03,
 'min_child_weight': 4,
 'n_estimators': 3000,
 'early_stopping_rounds': 50,
 'booster': 'dart',
 'reg_alpha': 0.5524056913641981,
 'reg_lambda': 0.40422510843726456,
 'colsample_bytree': 0.7047303196728362,
```

```
 'colsample_bylevel': 0.8128091754365336,
 'subsample': 0.8565508022637018}

This_the_one = {'max_depth': 8,
 'verbose': -1,
 'objective': 'reg:squarederror',
 'num_leaves': 64,
 'learning_rate': 0.02,
 'subsample_freq': 200,
 'metric': 'rmse',
 'min_child_samples': 4,
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.6150823851877983,
 'reg_lambda': 0.4019513141970534,
 'colsample_bytree': 0.9381694254851193,
 'colsample_bynode': 0.6144874410854747,
 'subsample': 0.8365551463780401}

# Setting up cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize lists to store models and CV scores
fitted_models_xgb = []
cv_scores_xgb = []
learning_curves = []

# Training and evaluating the model using cross-validation
for idx_train, idx_valid in kf.split(df_train, y):
    X_train, y_train = df_train.iloc[idx_train], y.iloc[idx_train]
    X_valid, y_valid = df_train.iloc[idx_valid], y.iloc[idx_valid]

    dtrain = xgb.DMatrix(X_train, label=y_train)
    dvalid = xgb.DMatrix(X_valid, label=y_valid)

    xgb_model = xgb.train(params=all_xgb, dtrain=dtrain, num_boost_round=2000, evals=[(dvalid, 'eval')],
                          early_stopping_rounds=50, verbose_eval=200)

    y_pred_valid = xgb_model.predict(dvalid)
    rmse_score = np.sqrt(mean_squared_error(y_valid, y_pred_valid))  # RMSE

    # Store model and CV score
    fitted_models_xgb.append(xgb_model)
    cv_scores_xgb.append(rmse_score)

# Display the cross-validation RMSE scores
print("CV RMSE scores: ", cv_scores_xgb)
print("Min CV RMSE score: ", min(cv_scores_xgb))
print("Max CV RMSE score: ", max(cv_scores_xgb))
print("Mean CV RMSE score: ", np.mean(cv_scores_xgb))
```

```
[0]     eval-rmse:0.30832
[94]    eval-rmse:0.29301
[0]     eval-rmse:0.33480
[79]    eval-rmse:0.32979
[0]     eval-rmse:0.31958
[151]   eval-rmse:0.29521
[0]     eval-rmse:0.31076
[112]   eval-rmse:0.29738
[0]     eval-rmse:0.32512
[166]   eval-rmse:0.29390
CV RMSE scores:  [0.2930343176821247, 0.329788489514118, 0.2952106889068333, 0.2973849399214489, 0.2939243520072687]
Min CV RMSE score:  0.2930343176821247
Max CV RMSE score:  0.329788489514118
Mean CV RMSE score:  0.3018685576063588
```

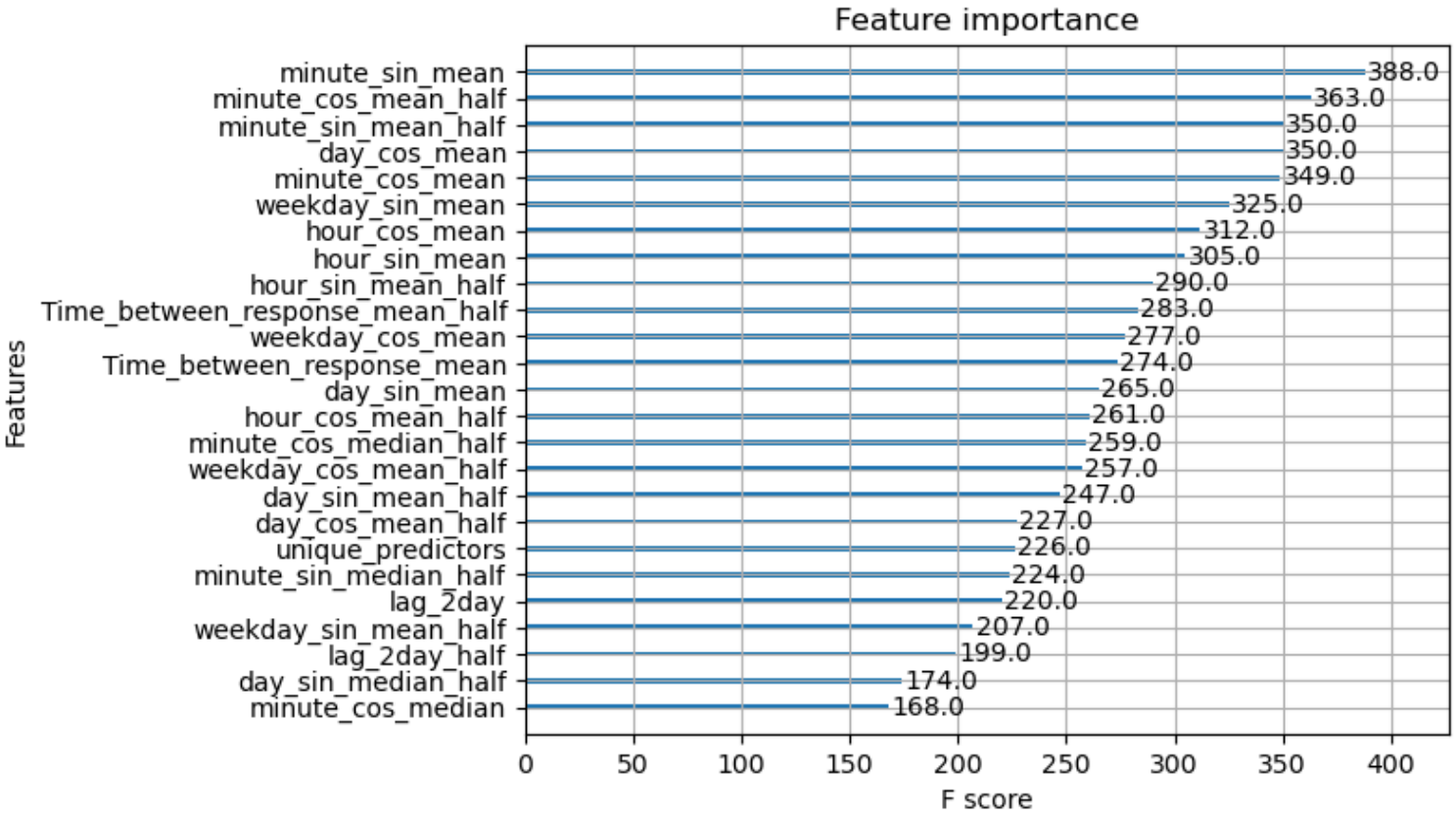
Importance

In [129…

```
xgb.plot_importance(xgb_model, max_num_features=25)
```

Out[129…

```
<Axes: title={'center': 'Feature importance'}, xlabel='F score', ylabel='Features'>
```



Time between response is not that major and the half feature is more important, minute features seems most important. Relatively low importance to lags (only 21,23 in importance) and only captured 2 days feature.

Catboost Model

In [161…

```
import pandas as pd
import numpy as np
from catboost import CatBoostRegressor, Pool
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

# Assuming df_train is the training dataframe and y is the target variable
kf = KFold(n_splits=5, shuffle=True, random_state=42)

fitted_models_catboost = []
cv_scores_catboost = []
learning_curves = []
par = {'max_depth': 8,
 'verbose': -1,
 'boosting_type': 'gbdt',
 'num_leaves': 64,
 'objective': 'regression',
 'learning_rate': 0.02,
 'subsample_freq': 200,
 'metric': 'rmse',
 'min_child_samples': 4,
 'n_estimators': 2000,
 'early_stopping_round': 1000,
 'reg_alpha': 0.6150823851877983,
 'reg_lambda': 0.4019513141970534,
 'colsample_bytree': 0.9381694254851193,
 'colsample_bynode': 0.6144874410854747,
 'subsample': 0.8365551463780401}
# Define CatBoost parameters
catboost_params = {
    'iterations': 2000,
    'learning_rate': 0.05,
    'depth': 8,
    'subsample': 0.8365551463780401,
    'loss_function': 'RMSE',  # For regression
    'eval_metric': 'RMSE',
    'random_seed': 42,
    'verbose': 200,
    'early_stopping_rounds': 50,
    'reg_lambda': 0.4019513141970534,
}

# Training and evaluating the model using cross-validation
for idx_train, idx_valid in kf.split(df_train):
    X_train, y_train = df_train.iloc[idx_train], y.iloc[idx_train]
    X_valid, y_valid = df_train.iloc[idx_valid], y.iloc[idx_valid]

    # Convert the data to Pool objects for CatBoost
    train_pool = Pool(X_train, y_train)
    valid_pool = Pool(X_valid, y_valid)

    # Train the CatBoost model
    cat_model = CatBoostRegressor(**catboost_params)
    cat_model.fit(train_pool, eval_set=valid_pool, use_best_model=True)
```

```
    # Predict and calculate RMSE on validation data
    y_pred_valid = cat_model.predict(X_valid)
    rmse_score = np.sqrt(mean_squared_error(y_valid, y_pred_valid))  # RMSE

    # Store model and score
    fitted_models_catboost.append(cat_model)
    cv_scores_catboost.append(rmse_score)

# Display the cross-validation RMSE scores
print("CV RMSE scores: ", cv_scores_catboost)
print("Min CV RMSE score: ", min(cv_scores_catboost))
print("Max CV RMSE score: ", max(cv_scores_catboost))
print("Mean CV RMSE score: ", np.mean(cv_scores_catboost))
```

```
0:      learn: 0.3205914        test: 0.3080278 best: 0.3080278 (0)     total: 171ms    remaining: 5m 42s
Stopped by overfitting detector  (50 iterations wait)

bestTest = 0.2995865813
bestIteration = 26

Shrink model to first 27 iterations.
0:      learn: 0.3135048        test: 0.3348001 best: 0.3348001 (0)     total: 11ms     remaining: 22s
Stopped by overfitting detector  (50 iterations wait)

bestTest = 0.3276777632
bestIteration = 86

Shrink model to first 87 iterations.
0:      learn: 0.3182343        test: 0.3190782 best: 0.3190782 (0)     total: 16.6ms   remaining: 33.3s
200:    learn: 0.1085128        test: 0.2927155 best: 0.2919117 (177)   total: 2.04s    remaining: 18.3s
Stopped by overfitting detector  (50 iterations wait)

bestTest = 0.2919116927
bestIteration = 177

Shrink model to first 178 iterations.
0:      learn: 0.3189420        test: 0.3116704 best: 0.3116704 (0)     total: 12.5ms   remaining: 25s
200:    learn: 0.1092571        test: 0.2852606 best: 0.2850102 (157)   total: 2.1s     remaining: 18.8s
Stopped by overfitting detector  (50 iterations wait)

bestTest = 0.2850102441
bestIteration = 157

Shrink model to first 158 iterations.
0:      learn: 0.3159165        test: 0.3244848 best: 0.3244848 (0)     total: 13.5ms   remaining: 26.9s
Stopped by overfitting detector  (50 iterations wait)

bestTest = 0.2971234243
bestIteration = 119

Shrink model to first 120 iterations.
CV RMSE scores:  [0.29958658097655344, 0.3276777620234348, 0.2919116913229105, 0.28501024294925287, 0.29712342351318904]
Min CV RMSE score:  0.28501024294925287
Max CV RMSE score:  0.3276777620234348
Mean CV RMSE score:  0.30026194015706814
```

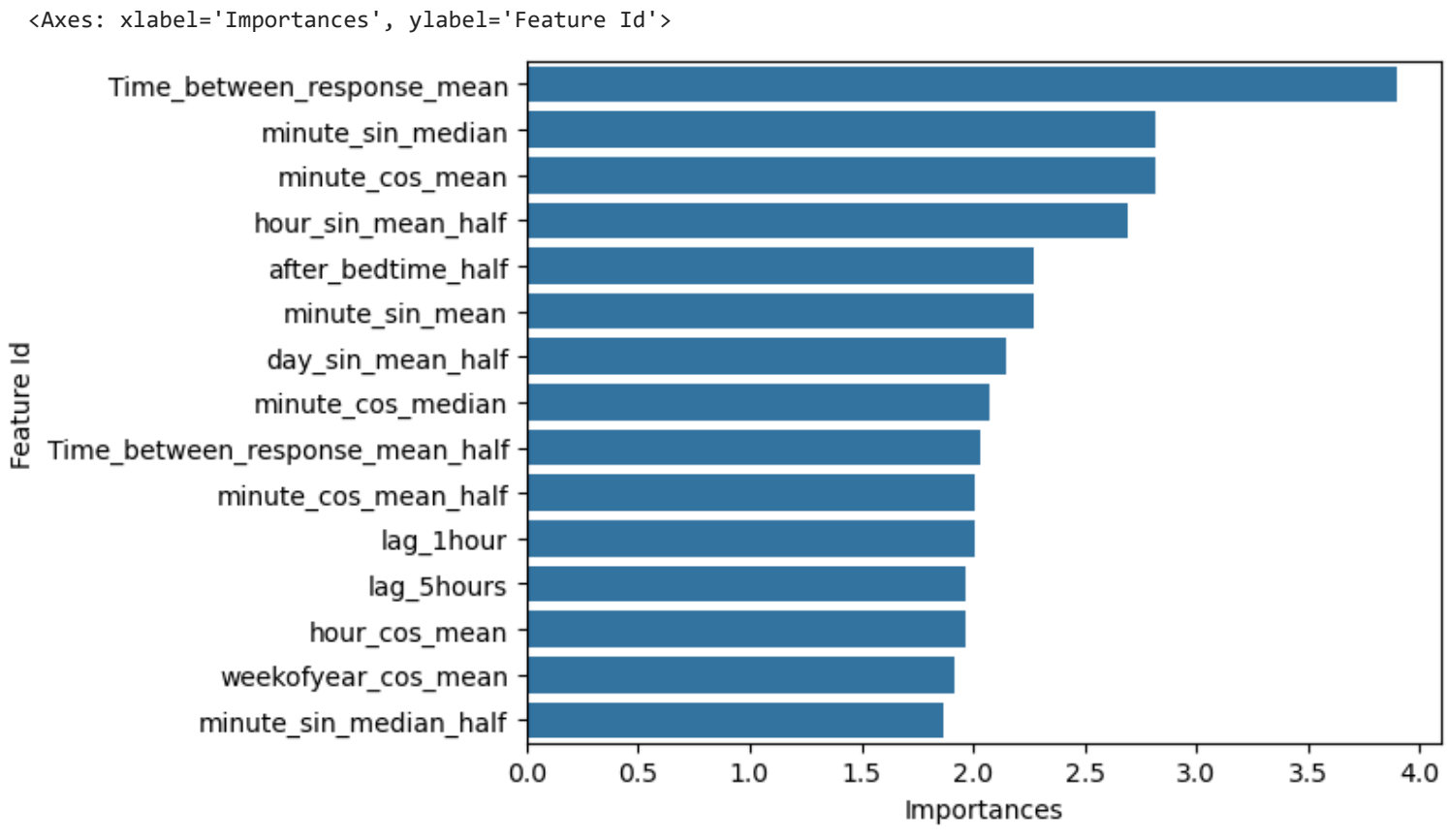
Importance

In [163…

```
a = cat_model.get_feature_importance(prettified=True)
sns.barplot(x='Importances', y='Feature Id', data=a[:15])
```

Out[163…

```
<Axes: xlabel='Importances', ylabel='Feature Id'>
```

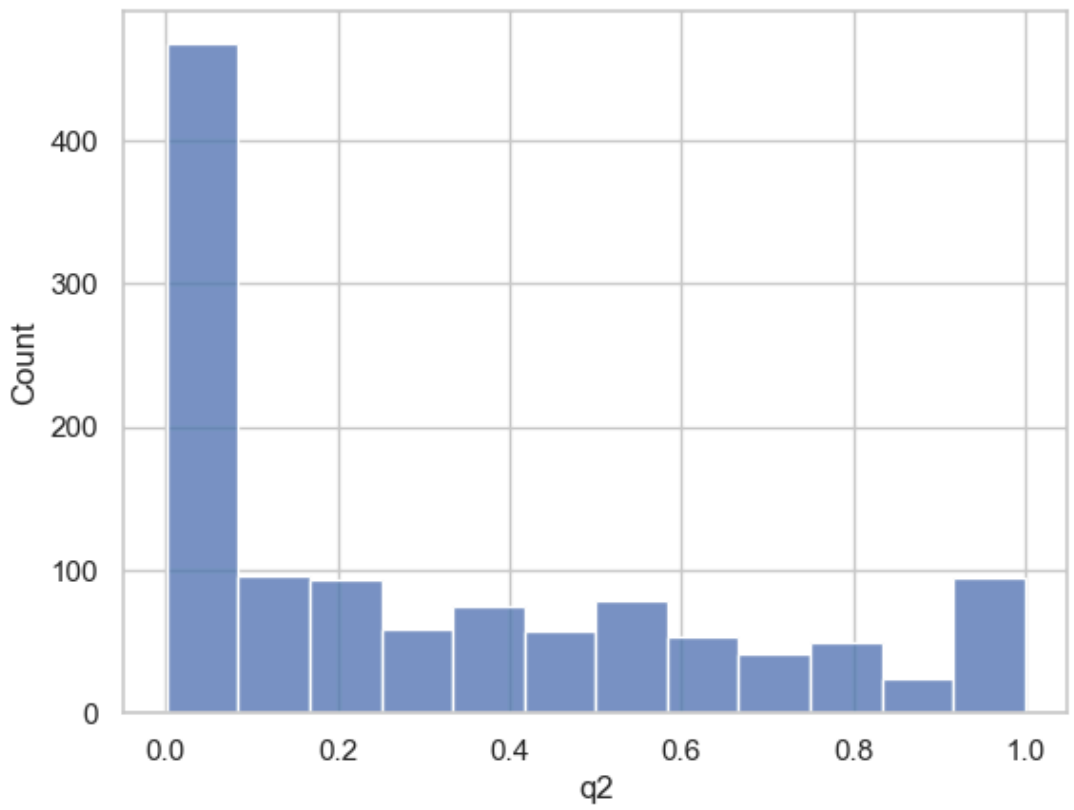


Time Between response is crucial here, the lags have importance as well and half features are important. Minute features are important as well

NGboost Model

In [178…
```
sns.histplot(y)
```

Out[178…
```
<Axes: xlabel='q2', ylabel='Count'>
```



Doesnt seem like a known distribution, after experimenting Normal distribution was chosen

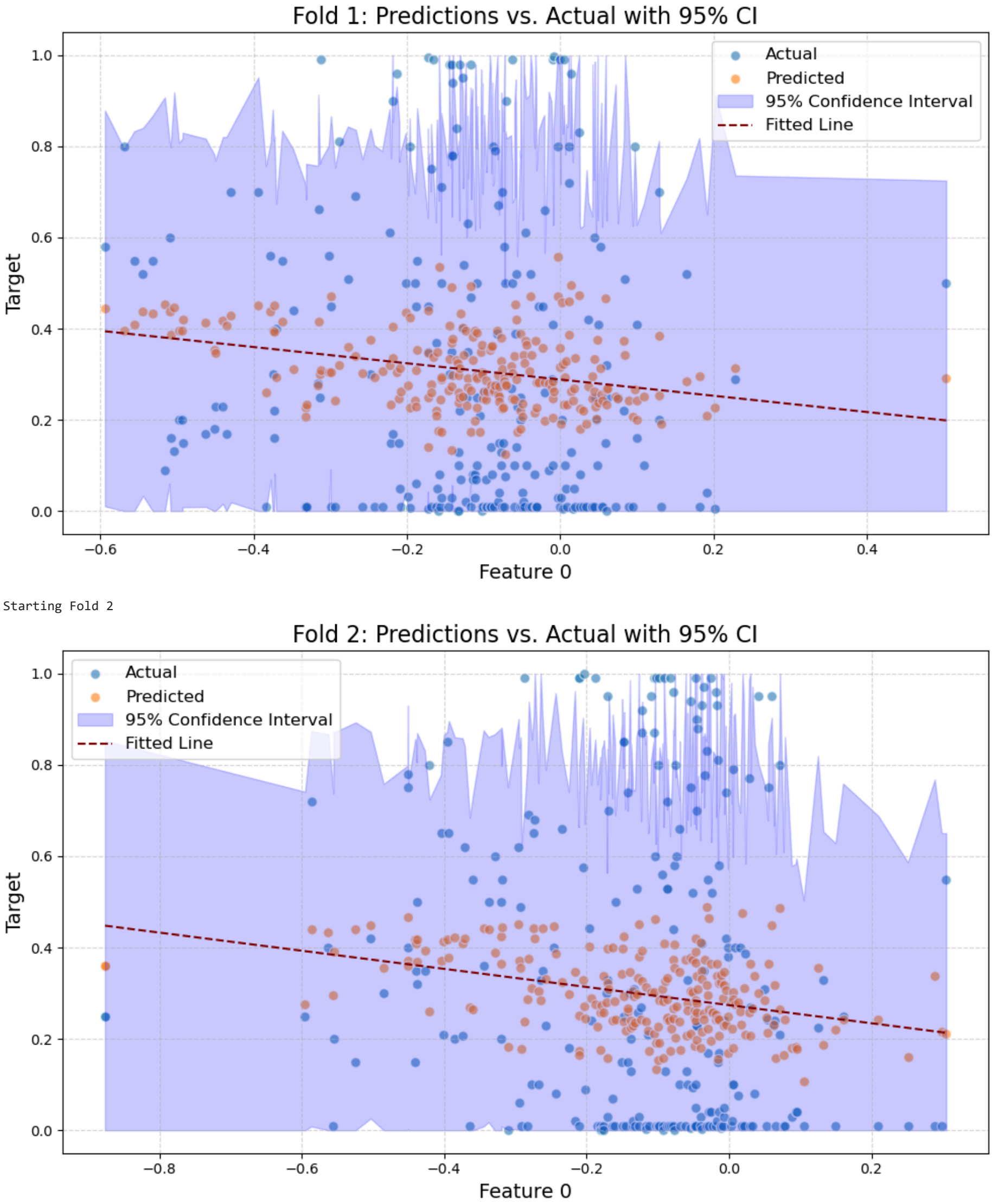
In [15]:
```
# Doesn't work with NaN values
df_train1 = df_train.copy()
df_train1.fillna(0, inplace=True)
```

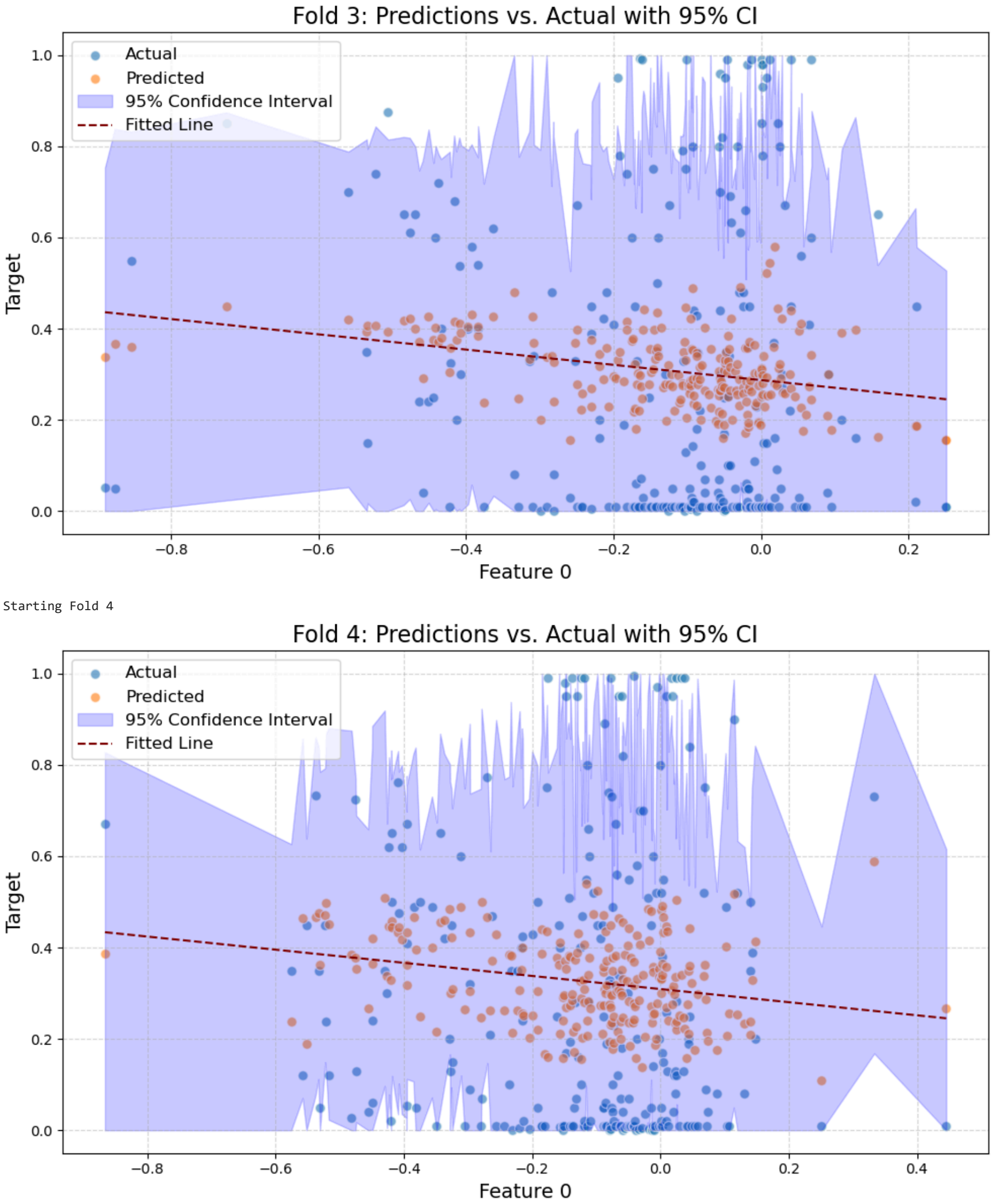
In [201…
```
import pandas as pd
import numpy as np
from ngboost import NGBRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
import warnings
from ngboost.distns import Normal
from ngboost.scores import LogScore
import matplotlib.pyplot as plt
import seaborn as sns

ngb_params = {
    'n_estimators': 2000,
    'minibatch_frac': 0.8,        # Use 75% of the data at each step
    'natural_gradient': True,
    'Score': LogScore,              # Scoring function
    'Dist': Normal,                 # Distribution
    'verbose': False,               # Set to False to reduce verbosity
    'random_state': 42,
    'learning_rate': 0.02,
}
# Training and evaluating the model using cross-validation
fitted_models_ngboost = []
cv_scores_ngboost = []
all_predictions = []
all_actuals = []
all_upper = []
all_lower = []
all_features = []


feature_index = 0
for fold, (idx_train, idx_valid) in enumerate(kf.split(df_train1)):
    print(f"\nStarting Fold {fold + 1}")
    X_train, y_train = df_train1.iloc[idx_train], y.iloc[idx_train]
    X_valid, y_valid = df_train1.iloc[idx_valid], y.iloc[idx_valid]

    # Train the NGBoost model
    model = NGBRegressor(**ngb_params)
    model.fit(
        X_train, y_train,
        X_val=X_valid, Y_val=y_valid,
        early_stopping_rounds=50
    )

    # Store the fitted model
    fitted_models_ngboost.append(model)
```

```
    # Predict on validation data
    y_pred_valid = model.predict(X_valid)

    # Compute RMSE
    rmse_score = root_mean_squared_error(y_valid, y_pred_valid)
    cv_scores_ngboost.append(rmse_score)

    # Extract loc and scale from the predicted distribution
    dist = model.pred_dist(X_valid)
    model_loc = dist.loc
    model_scale = dist.scale

    # Calculate 95% confidence intervals
    upper = model_loc + model_scale * 1.96
    lower = model_loc - model_scale * 1.96

    # Clip the confidence intervals to [0, 1] if necessary
    upper = np.clip(upper, 0, 1)
    lower = np.clip(lower, 0, 1)
    # Select the feature to plot against (replace '0' with the desired feature index)
    feature = X_valid.iloc[:, feature_index]


    # Sort the data by the selected feature for better visualization
    sorted_indices = np.argsort(feature)
    feature_sorted = feature.iloc[sorted_indices]
    y_valid_sorted = y_valid.iloc[sorted_indices]
    y_pred_valid_sorted = y_pred_valid[sorted_indices]

    upper_sorted = upper[sorted_indices]
    lower_sorted = lower[sorted_indices]

    # Append to aggregate lists
    all_predictions.append(y_pred_valid_sorted)
    all_actuals.append(y_valid_sorted)
    all_upper.append(upper_sorted)
    all_lower.append(lower_sorted)
    all_features.append(feature_sorted)

    # Plot the predictions and the 95% confidence interval for this fold
    plt.figure(figsize=(10, 6))
    plt.scatter(feature_sorted, y_valid_sorted, alpha=0.6, label='Actual', edgecolors='w', s=50)
    plt.scatter(feature_sorted, y_pred_valid_sorted, edgecolors='w', s=50, alpha=0.6, label='Predicted')

    plt.fill_between(feature_sorted, lower_sorted, upper_sorted, color='blue', alpha=0.2, label='95% Confidence Interval')

    # Plot the mean prediction line
    mean_pred_sorted = np.mean([y_pred_valid_sorted], axis=0)
    coef = np.polyfit(feature_sorted, mean_pred_sorted, 1)
    poly1d_fn = np.poly1d(coef)

    # Plot the fitted line
    plt.plot(feature_sorted, poly1d_fn(feature_sorted), '--k', label='Fitted Line', color='maroon')

    plt.xlabel('Feature {}'.format(feature_index), fontsize=14)
    plt.ylabel('Target', fontsize=14)
    plt.title(f'Fold {fold + 1}: Predictions vs. Actual with 95% CI', fontsize=16)
    plt.legend(fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()
```

```
# After cross-validation, display the overall RMSE scores
print("\nCross-Validation RMSE Scores:")
for i, score in enumerate(cv_scores_ngboost, 1):
    print(f"Fold {i}: RMSE = {score:.6f}")
print(f"Minimum CV RMSE: {min(cv_scores_ngboost):.6f}")
print(f"Maximum CV RMSE: {max(cv_scores_ngboost):.6f}")
print(f"Mean CV RMSE: {np.mean(cv_scores_ngboost):.6f}")
```

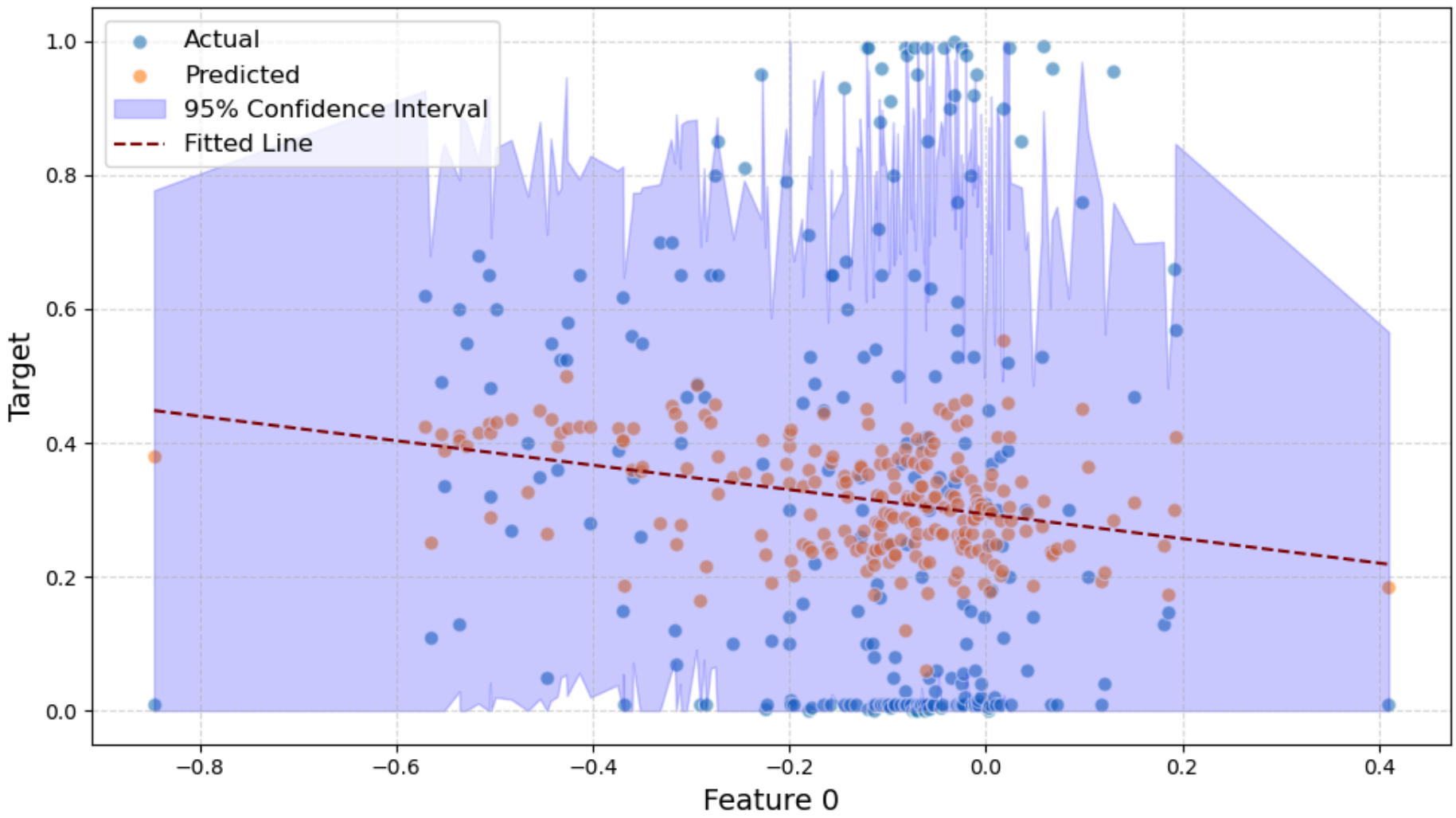
```
Starting Fold 1
```


Fold 1: Predictions vs. Actual with 95% CI
Actual
Predicted
95% Confidence Interval
Fitted Line
Target
Feature 0
Starting Fold 2
Fold 2: Predictions vs. Actual with 95% CI
Actual
Predicted
95% Confidence Interval
Fitted Line
Target
Feature 0
Starting Fold 3



Starting Fold 4

Starting Fold 5



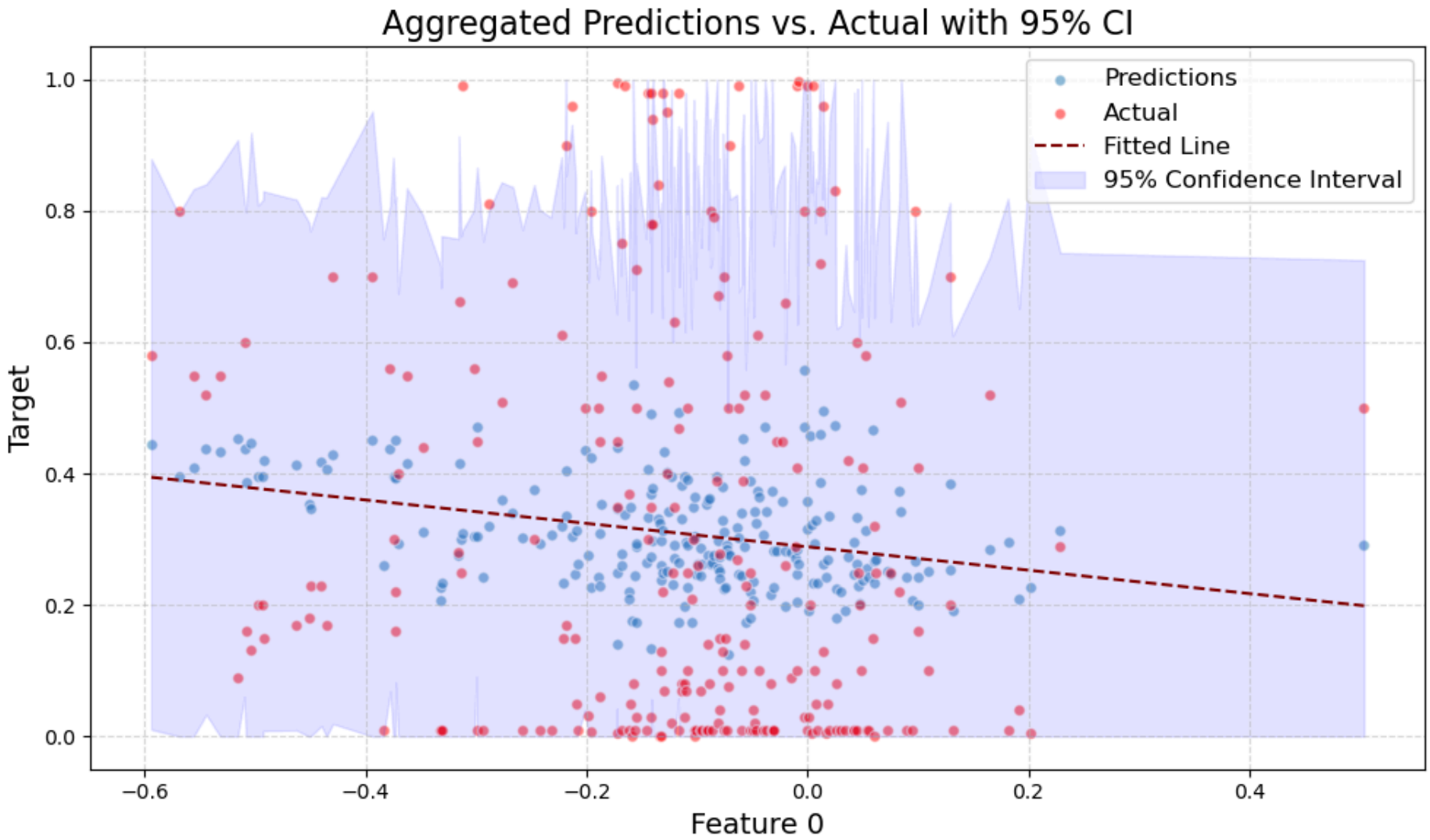

```
Cross-Validation RMSE Scores:
Fold 1: RMSE = 0.299140
Fold 2: RMSE = 0.337034
Fold 3: RMSE = 0.306653
Fold 4: RMSE = 0.301081
Fold 5: RMSE = 0.306202
Minimum CV RMSE: 0.299140
Maximum CV RMSE: 0.337034
Mean CV RMSE: 0.310022
```

Plot best fold

In [203…

```python
all_predictions1 = np.array(all_predictions[0])
all_actuals1 = np.array(all_actuals[0])
all_upper1 = np.array(all_upper[0])
all_lower1 = np.array(all_lower[0])
all_features1 = np.array(all_features[0])


# Sort the data by the selected feature for better visualization
sorted_indices = np.argsort(all_features[0])
feature_sorted = all_features1[sorted_indices]
y_valid_sorted = all_actuals1[sorted_indices]
y_pred_sorted = all_predictions1[sorted_indices]
upper_sorted = all_upper1[sorted_indices]
lower_sorted = all_lower1[sorted_indices]
mean_pred_sorted_all = np.mean([y_pred_sorted], axis=0)
coef_all = np.polyfit(feature_sorted, mean_pred_sorted_all, 1)
poly1d_fn_all = np.poly1d(coef_all)
# Plot aggregated predictions
plt.figure(figsize=(10, 6))
plt.scatter(feature_sorted, y_pred_sorted, alpha=0.5, label='Predictions', edgecolors='w', s=30)
print(len(feature_sorted), len(y_valid_sorted))
print(type(feature_sorted), type(y_valid_sorted))
plt.scatter(feature_sorted, y_valid_sorted, color='red', alpha=0.5, label='Actual', edgecolors='w', s=30)
plt.plot(feature_sorted, poly1d_fn_all(feature_sorted), '--k', label='Fitted Line', color='maroon')

plt.fill_between(feature_sorted, lower_sorted, upper_sorted, color='blue', alpha=0.1, label='95% Confidence Interval')
plt.xlabel('Feature {}'.format(feature_index), fontsize=14)
plt.ylabel('Target', fontsize=14)
plt.title('Aggregated Predictions vs. Actual with 95% CI', fontsize=16)
plt.legend(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

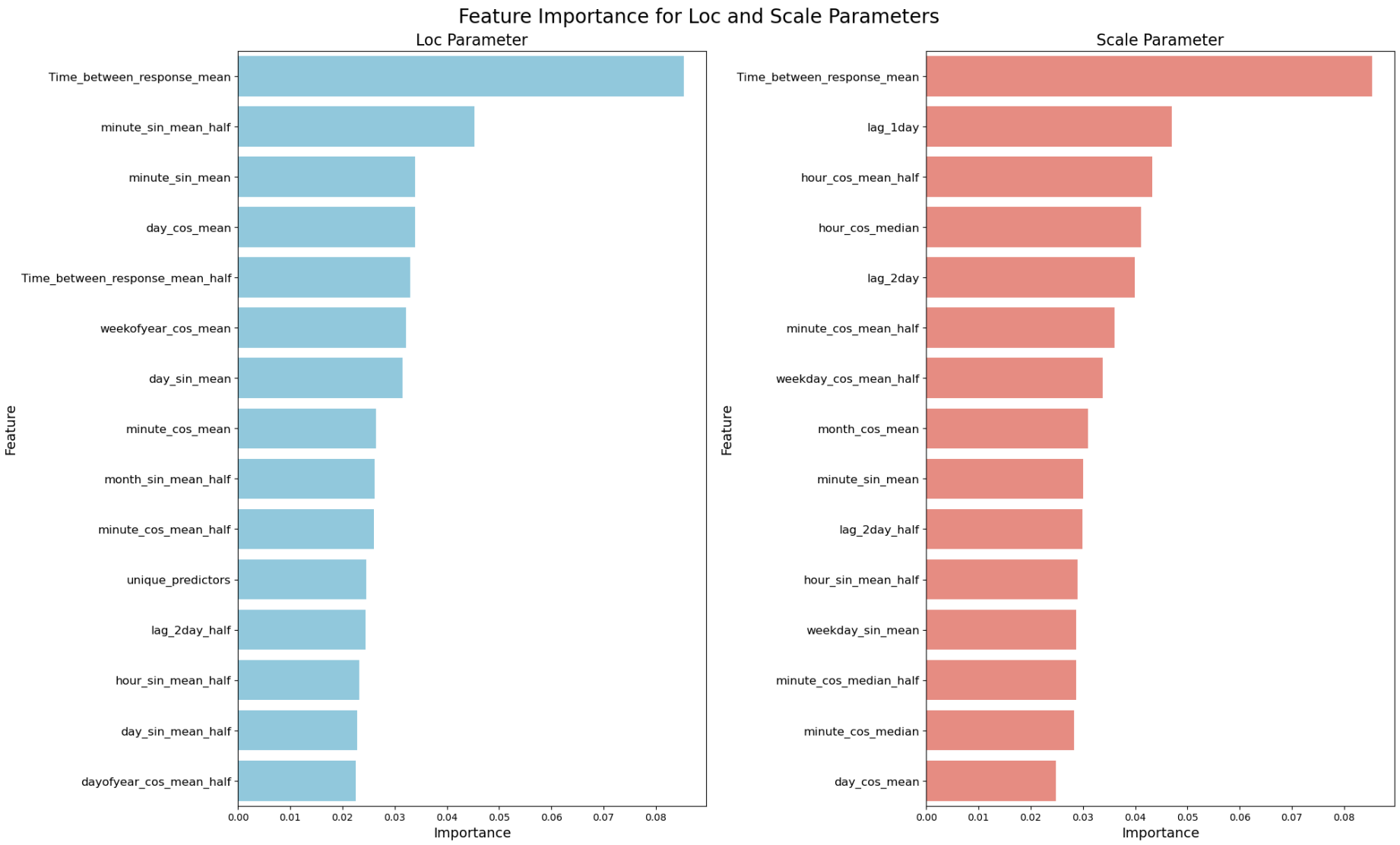
```
238 238
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```



In [205…

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming 'model' is your trained NGBRegressor model
# and 'df_train1' is your training dataframe with features

# Extract feature importances for loc and scale parameters
# Ensure that 'feature_importances_' has two elements: [loc_importances, scale_importances]
feature_importance_loc = model.feature_importances_[0]
feature_importance_scale = model.feature_importances_[1]

# Create DataFrames for loc and scale feature importances
df_loc = pd.DataFrame({
    'feature': df_train1.columns,
    'importance': feature_importance_loc
}).sort_values('importance', ascending=False)

df_scale = pd.DataFrame({
    'feature': df_train1.columns,
    'importance': feature_importance_scale
}).sort_values('importance', ascending=False)

# Select top 15 features for each parameter
top_n = 15
df_loc_top = df_loc.head(top_n)
df_scale_top = df_scale.head(top_n)

# Initialize the matplotlib figure with increased width
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 12), constrained_layout=True)

# Set the overall title for the figure

# Plot Feature Importance for 'loc' parameter
sns.barplot(
    x='importance',
    y='feature',
    data=df_loc_top,
    color="skyblue",
    ax=ax1
)
ax1.set_title('Loc Parameter', fontsize=16)
ax1.set_xlabel('Importance', fontsize=14)
ax1.set_ylabel('Feature', fontsize=14)
ax1.tick_params(axis='y', labelsize=12)  # Adjust y-tick label size

# Plot Feature Importance for 'scale' parameter
sns.barplot(
    x='importance',
    y='feature',
    data=df_scale_top,
    color="salmon",
    ax=ax2
)
ax2.set_title('Scale Parameter', fontsize=16)
ax2.set_xlabel('Importance', fontsize=14)
```

```
ax2.set_ylabel('Feature', fontsize=14)
ax2.tick_params(axis='y', labelsize=12)  # Adjust y-tick label size
fig.suptitle('Feature Importance for Loc and Scale Parameters', fontsize=20)

# Display the plots
plt.show()
```



We can see that time between responses was crucial for this model.

Moreover, Regarding *Loc* parameter (mean): The features for half of the time seems equal in importance to the full time features.

In addition, Regarding *Scale* parameter (SD): The features for half of the time seems equal in importance to the full time features as well and there's more emphasis on the lag features. minute features are important as well

Voting regressor

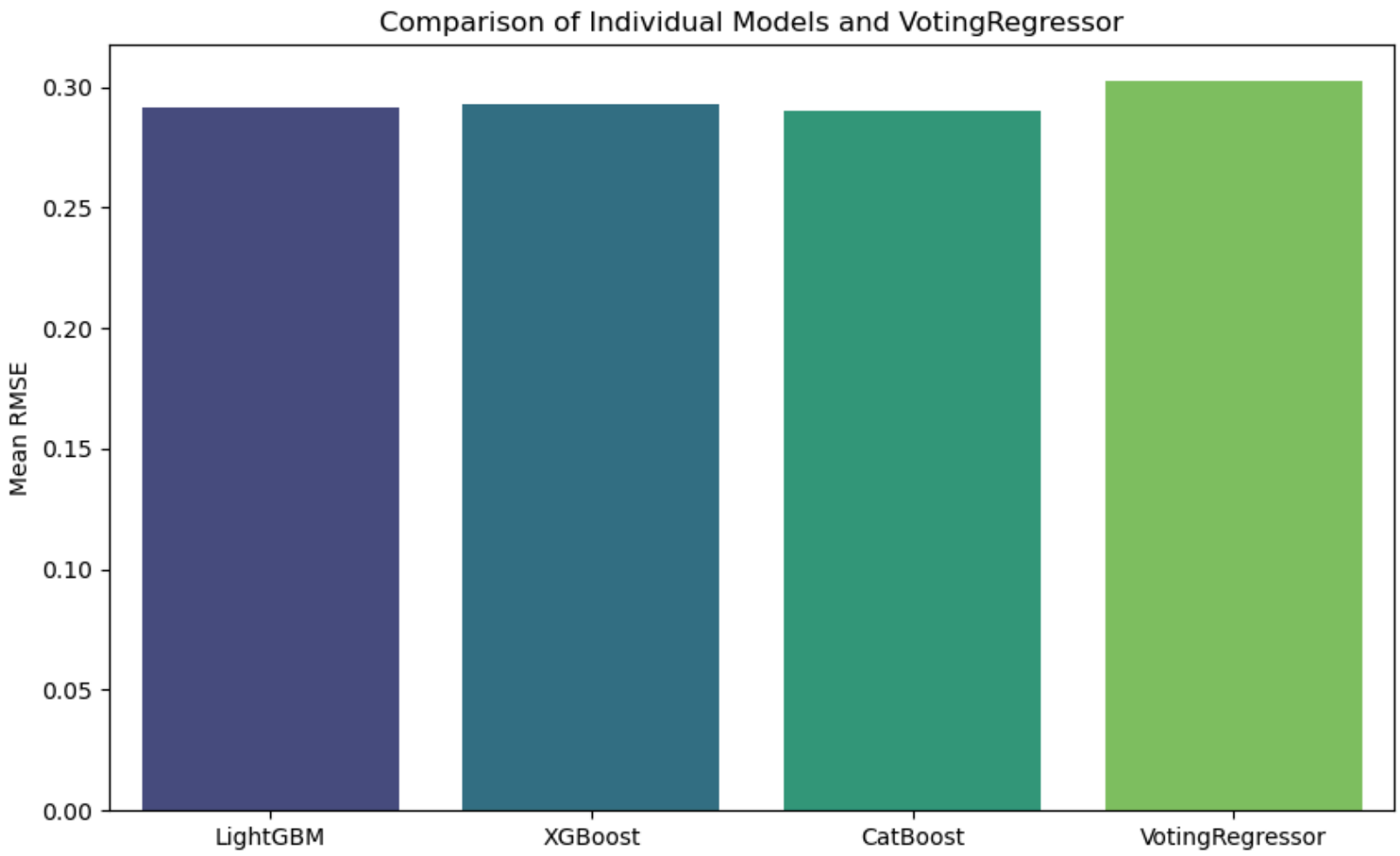
In [262…

```
import pandas as pd
import numpy as np
from ngboost import NGBRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import VotingRegressor
from lightgbm import LGBMRegressor
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
import warnings
import logging
import matplotlib.pyplot as plt
import seaborn as sns

# Suppress warnings and set logging level
warnings.filterwarnings('ignore')
logging.getLogger('lightgbm').setLevel(logging.ERROR)
logging.getLogger('xgboost').setLevel(logging.ERROR)
logging.getLogger('catboost').setLevel(logging.ERROR)

# Define the Root Mean Squared Error function
def root_mean_squared_error(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

# --- Assumptions ---
# Ensure that 'df_train1' (features) and 'y' (target) are already defined and preprocessed.
# Example:
# df_train1 = pd.read_csv('your_features.csv')
# y = pd.read_csv('your_target.csv')['target_column']
# ----------------------

# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store models and scores
fitted_models_ngboost = []
cv_scores_ngboost = []
best_val_losses = []
```

```python
# Define NGBoost parameters without early_stopping_rounds
ngb_params = {
    'n_estimators': 2000,
    'learning_rate': 0.05,
    'minibatch_frac': 0.85,  # Set to 1.0 to use the full dataset at each step
    'natural_gradient': True,
    'Score': LogScore,         # Choose appropriate scoring function
    'Dist': Normal,            # Choose appropriate distribution
    # 'max_depth': 7,          # Uncomment if needed
    'verbose': False,          # Set to False to reduce verbosity
    'random_state': 42
}

# Define parameters for VotingRegressor base estimators
lgb_params = {
    'max_depth': 8,
    'verbose': -1,
    'boosting_type': 'gbdt',
    'num_leaves': 64,
    'objective': 'regression',
    'learning_rate': 0.02,
    'subsample_freq': 200,
    'metric': 'rmse',
    'min_child_samples': 4,
    'n_estimators': 2000,
    # 'early_stopping_round': 1000,  # Removed as per your request
    'reg_alpha': 0.6150823851877983,
    'reg_lambda': 0.4019513141970534,
    'colsample_bytree': 0.9381694254851193,
    'colsample_bynode': 0.6144874410854747,
    'subsample': 0.8365551463780401
}

xgb_params = {
    'max_depth': 8,
    'verbosity': 0,
    'objective': 'reg:squarederror',
    'eval_metric': 'rmse',
    'learning_rate': 0.03,
    'min_child_weight': 4,
    'n_estimators': 2000,
    # 'early_stopping_rounds': 50,    # Removed as per your request
    'booster': 'dart',
    'reg_alpha': 0.5524056913641981,
    'reg_lambda': 0.40422510843726456,
    'colsample_bytree': 0.7047303196728362,
    'colsample_bylevel': 0.8128091754365336,
    'subsample': 0.8565508022637018
}

catboost_params = {
    'iterations': 2000,
    'learning_rate': 0.05,
    'depth': 8,
    'subsample': 0.8365551463780401,
    'loss_function': 'RMSE',  # For regression
    'eval_metric': 'RMSE',
    'random_seed': 42,
    'verbose': 200,
    # 'early_stopping_rounds': 50,   # Removed as per your request
    'reg_lambda': 0.4019513141970534,
}

# Create base estimators (unfitted)
lgb_model1 = LGBMRegressor(**lgb_params)
xgb_model1 = XGBRegressor(**xgb_params)
cat_model1 = CatBoostRegressor(**catboost_params)

# Create VotingRegressor with unfitted models
voting_reg = VotingRegressor([
    ('lgb', lgb_model1),
    ('xgb', xgb_model1),
    ('cat', cat_model1)
])

# Lists to store cross-validation results
cv_scores_voting = []

# Training and evaluating the ensemble using cross-validation
for fold, (idx_train, idx_valid) in enumerate(kf.split(df_train1)):
    X_train, y_train_fold = df_train1.iloc[idx_train], y.iloc[idx_train]
    X_valid, y_valid_fold = df_train1.iloc[idx_valid], y.iloc[idx_valid]

    # Fit the VotingRegressor
    voting_reg.fit(X_train, y_train_fold)

    # Predict on validation data
    y_pred_valid = voting_reg.predict(X_valid)

    # Calculate RMSE
    rmse_score = root_mean_squared_error(y_valid_fold, y_pred_valid)
```

```
    cv_scores_voting.append(rmse_score)

    print(f"Fold {fold+1}: VotingRegressor RMSE = {rmse_score:.4f}")

# Display the cross-validation RMSE scores for VotingRegressor
print("\nVotingRegressor CV RMSE Scores:")
print(cv_scores_voting)
print(f"Minimum CV RMSE: {min(cv_scores_voting):.4f}")
print(f"Maximum CV RMSE: {max(cv_scores_voting):.4f}")
print(f"Mean CV RMSE: {np.mean(cv_scores_voting):.4f}")

# Compare with individual model performances
# Assuming you have stored individual RMSE scores separately
# For illustration, here's how you might have them
# Replace these with your actual scores
individual_rmse = {
    'LightGBM': 0.2912172102064591,
    'XGBoost': 0.29306985006070535,
    'CatBoost': 0.2901387759790261
}

print("\nIndividual Model RMSE Scores on Validation Folds:")
for model_name, rmse in individual_rmse.items():
    print(f"{model_name}: {rmse}")

# Optional: Visualize the results
plt.figure(figsize=(10, 6))
models = list(individual_rmse.keys()) + ['VotingRegressor']
rmse_values = list(individual_rmse.values()) + [np.mean(cv_scores_voting)]
sns.barplot(x=models, y=rmse_values, palette='viridis')
plt.ylabel('Mean RMSE')
plt.title('Comparison of Individual Models and VotingRegressor')
plt.show()
```

```
0:      learn: 0.3206195        total: 38.4ms   remaining: 1m 16s
200:    learn: 0.1011774        total: 6.2s     remaining: 55.5s
400:    learn: 0.0366016        total: 12.4s    remaining: 49.6s
600:    learn: 0.0150246        total: 18.5s    remaining: 43.1s
800:    learn: 0.0062365        total: 24.5s    remaining: 36.7s
1000:   learn: 0.0028921        total: 30.3s    remaining: 30.2s
1200:   learn: 0.0013773        total: 36.5s    remaining: 24.3s
1400:   learn: 0.0006735        total: 42.8s    remaining: 18.3s
1600:   learn: 0.0003281        total: 48.8s    remaining: 12.2s
1800:   learn: 0.0001647        total: 54.9s    remaining: 6.06s
1999:   learn: 0.0000824        total: 1m 1s    remaining: 0us
Fold 1: VotingRegressor RMSE = 0.2963
0:      learn: 0.3130922        total: 35.5ms   remaining: 1m 10s
200:    learn: 0.0996993        total: 6.18s    remaining: 55.3s
400:    learn: 0.0384847        total: 12.3s    remaining: 49.2s
600:    learn: 0.0163741        total: 18.4s    remaining: 42.7s
800:    learn: 0.0071929        total: 24.5s    remaining: 36.6s
1000:   learn: 0.0032031        total: 30.7s    remaining: 30.6s
1200:   learn: 0.0015222        total: 36.9s    remaining: 24.5s
1400:   learn: 0.0007298        total: 43s      remaining: 18.4s
1600:   learn: 0.0003569        total: 49.5s    remaining: 12.3s
1800:   learn: 0.0001801        total: 55.7s    remaining: 6.16s
1999:   learn: 0.0000968        total: 1m 1s    remaining: 0us
Fold 2: VotingRegressor RMSE = 0.3285
0:      learn: 0.3176227        total: 149ms    remaining: 4m 58s
200:    learn: 0.1039220        total: 6.72s    remaining: 1m
400:    learn: 0.0420161        total: 12.7s    remaining: 50.7s
600:    learn: 0.0176260        total: 18.7s    remaining: 43.6s
800:    learn: 0.0080631        total: 25s      remaining: 37.4s
1000:   learn: 0.0037265        total: 31.1s    remaining: 31.1s
1200:   learn: 0.0017615        total: 35.4s    remaining: 23.5s
1400:   learn: 0.0008550        total: 37.3s    remaining: 16s
1600:   learn: 0.0004288        total: 39.1s    remaining: 9.75s
1800:   learn: 0.0002178        total: 41.1s    remaining: 4.54s
1999:   learn: 0.0001120        total: 43s      remaining: 0us
Fold 3: VotingRegressor RMSE = 0.2966
0:      learn: 0.3189420        total: 13.8ms   remaining: 27.6s
200:    learn: 0.1049453        total: 2.56s    remaining: 22.9s
400:    learn: 0.0412178        total: 5.18s    remaining: 20.7s
600:    learn: 0.0173741        total: 7.73s    remaining: 18s
800:    learn: 0.0075685        total: 10.5s    remaining: 15.7s
1000:   learn: 0.0034936        total: 13.3s    remaining: 13.3s
1200:   learn: 0.0016550        total: 15.9s    remaining: 10.6s
1400:   learn: 0.0007945        total: 18.4s    remaining: 7.88s
1600:   learn: 0.0003989        total: 21s      remaining: 5.22s
1800:   learn: 0.0002033        total: 23.4s    remaining: 2.59s
1999:   learn: 0.0000997        total: 26s      remaining: 0us
Fold 4: VotingRegressor RMSE = 0.2927
0:      learn: 0.3160385        total: 32.5ms   remaining: 1m 5s
200:    learn: 0.1067380        total: 6.05s    remaining: 54.1s
400:    learn: 0.0396848        total: 11.9s    remaining: 47.7s
600:    learn: 0.0164604        total: 18s      remaining: 41.9s
800:    learn: 0.0072515        total: 24.1s    remaining: 36s
1000:   learn: 0.0033309        total: 30.1s    remaining: 30s
1200:   learn: 0.0016526        total: 36.1s    remaining: 24s
1400:   learn: 0.0008383        total: 42.3s    remaining: 18.1s
1600:   learn: 0.0004239        total: 45s      remaining: 11.2s
1800:   learn: 0.0002179        total: 46.9s    remaining: 5.19s
1999:   learn: 0.0001123        total: 48.9s    remaining: 0us
Fold 5: VotingRegressor RMSE = 0.2970

VotingRegressor CV RMSE Scores:
[0.2962999673692794, 0.3284989734463774, 0.29659345871456017, 0.29267791136009286, 0.2970020078843013]
Minimum CV RMSE: 0.2927
Maximum CV RMSE: 0.3285
Mean CV RMSE: 0.3022

Individual Model RMSE Scores on Validation Folds:
LightGBM: 0.2912172102064591
XGBoost: 0.29306985006070535
CatBoost: 0.2901387759790261
```



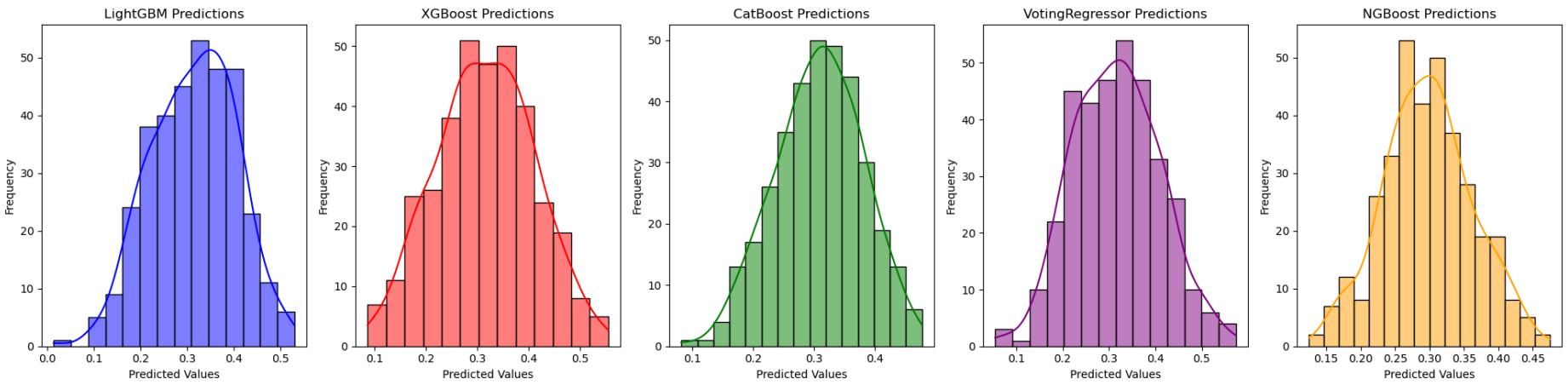
In [264…
```
xg_df_test = xgb.DMatrix(df_test, label=y_test)

lgb_pred_new = lgb_model.predict(df_test)

xgb_pred_new = xgb_model.predict(xg_df_test)
cat_pred_new = cat_model.predict(df_test)
voting_pred_new = voting_reg.predict(df_test)
ngb_pred = model.predict(df_test1)
```

In [265…
```
from sklearn.metrics import root_mean_squared_error
print("LightGBM RMSE on new data: ", root_mean_squared_error(y_test, lgb_pred_new))
print("XGBoost RMSE on new data: ", root_mean_squared_error(y_test, xgb_pred_new))
print("CatBoost RMSE on new data: ", root_mean_squared_error(y_test, cat_pred_new))
print("VotingRegressor RMSE on new data: ", root_mean_squared_error(y_test, voting_pred_new))
print('NGBoost RMSE on new data: ', root_mean_squared_error(y_test, ngb_pred))
```

```
LightGBM RMSE on new data:  0.2912172102064591
XGBoost RMSE on new data:  0.29306985006070535
CatBoost RMSE on new data:  0.2901387759790261
VotingRegressor RMSE on new data:  0.29221936432525114
NGBoost RMSE on new data:  0.2875321461705481
```

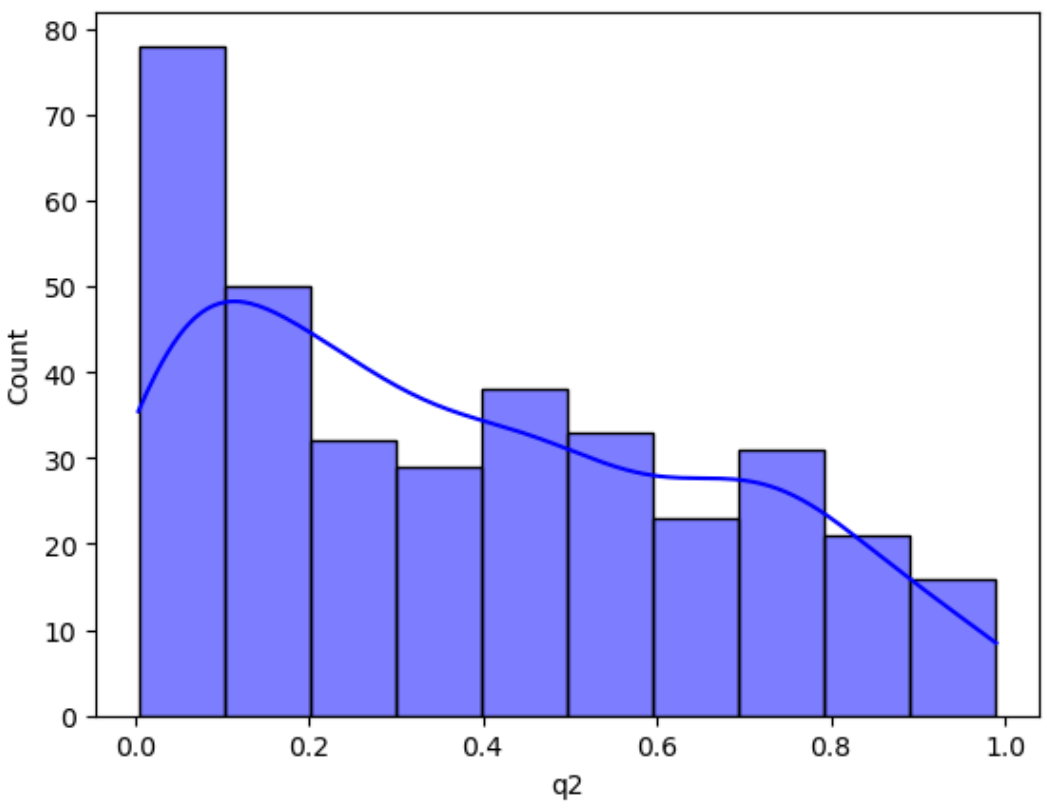
NGBoost is a relatively new algorithm published in 2019. It captures uncertainty and provides credible intervals in addition to the point estimate. It predicts the probability. For this task, it predicted with the best accuracy.

In [269…
```
fig, (ax1,ax2, ax3,ax4, ax5) = plt.subplots(1, 5, figsize=(20, 5))

# Plot the predictions
sns.histplot(lgb_pred_new, kde=True, color='blue', ax=ax1)
ax1.set_title('LightGBM Predictions')
ax1.set_xlabel('Predicted Values')
ax1.set_ylabel('Frequency')

sns.histplot(xgb_pred_new, kde=True, color='red', ax=ax2)
ax2.set_title('XGBoost Predictions')
ax2.set_xlabel('Predicted Values')
ax2.set_ylabel('Frequency')

sns.histplot(cat_pred_new, kde=True, color='green', ax=ax3)
ax3.set_title('CatBoost Predictions')
ax3.set_xlabel('Predicted Values')
ax3.set_ylabel('Frequency')

sns.histplot(voting_pred_new, kde=True, color='purple', ax=ax4)
ax4.set_title('VotingRegressor Predictions')
ax4.set_xlabel('Predicted Values')
ax4.set_ylabel('Frequency')

sns.histplot(ngb_pred, kde=True, color='orange', ax=ax5)
ax5.set_title('NGBoost Predictions')
ax5.set_xlabel('Predicted Values')
ax5.set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



In [213…
```
sns.histplot(y_test, kde=True, color='blue')
```

Out[213…
```
<Axes: xlabel='q2', ylabel='Count'>
```



Our models seems to be bounded by approximatley 0.5. Moving on, we night want to change our target function and regularization parameters in order to increase our upper bound