In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.preprocessing import StandardScaler
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import learning_curve
```

In [ ]:
```
data = pd.read_csv("sickness_table.csv")
```

In [ ]:
```
data.head(10)
```

Out[ ]:

| | Unnamed: 0 | date | n_sick | calls | n_duty | n_sby | sby_need | dafted |
|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 2016-04-01 | 73 | 8154.0 | 1700 | 90 | 4.0 | 0.0 |
| **1** | 1 | 2016-04-02 | 64 | 8526.0 | 1700 | 90 | 70.0 | 0.0 |
| **2** | 2 | 2016-04-03 | 68 | 8088.0 | 1700 | 90 | 0.0 | 0.0 |
| **3** | 3 | 2016-04-04 | 71 | 7044.0 | 1700 | 90 | 0.0 | 0.0 |
| **4** | 4 | 2016-04-05 | 63 | 7236.0 | 1700 | 90 | 0.0 | 0.0 |
| **5** | 5 | 2016-04-06 | 70 | 6492.0 | 1700 | 90 | 0.0 | 0.0 |
| **6** | 6 | 2016-04-07 | 64 | 6204.0 | 1700 | 90 | 0.0 | 0.0 |
| **7** | 7 | 2016-04-08 | 62 | 7614.0 | 1700 | 90 | 0.0 | 0.0 |
| **8** | 8 | 2016-04-09 | 51 | 5706.0 | 1700 | 90 | 0.0 | 0.0 |
| **9** | 9 | 2016-04-10 | 54 | 6606.0 | 1700 | 90 | 0.0 | 0.0 |

Removing Redundant Columns

In [ ]:
```
# 1.1 Remove Redundant Columns
# Drop the 'Unnamed: 0' column (if it exists)
if 'Unnamed: 0' in data.columns:
    sickness_data = data.drop(columns=['Unnamed: 0'])
```

Convert the 'date' column to datetime format

In [ ]:
```
# Convert the 'date' column to datetime format
sickness_data['date'] = pd.to_datetime(sickness_data['date'])
```

Initial Data Analysis

In [ ]:
```
sickness_data.head()
```

Out[ ]:

| | date | n_sick | calls | n_duty | n_sby | sby_need | dafted |
|---|---|---|---|---|---|---|---|
| **0** | 2016-04-01 | 73 | 8154.0 | 1700 | 90 | 4.0 | 0.0 |
| **1** | 2016-04-02 | 64 | 8526.0 | 1700 | 90 | 70.0 | 0.0 |
| **2** | 2016-04-03 | 68 | 8088.0 | 1700 | 90 | 0.0 | 0.0 |
| **3** | 2016-04-04 | 71 | 7044.0 | 1700 | 90 | 0.0 | 0.0 |
| **4** | 2016-04-05 | 63 | 7236.0 | 1700 | 90 | 0.0 | 0.0 |

In [ ]:
```
sickness_data.shape
```

Out[ ]:
```
(1152, 7)
```

In [ ]:
```
sickness_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1152 entries, 0 to 1151
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype         
---  ------    --------------  -----         
 0   date      1152 non-null   datetime64[ns]
 1   n_sick    1152 non-null   int64         
 2   calls     1152 non-null   float64       
 3   n_duty    1152 non-null   int64         
 4   n_sby     1152 non-null   int64         
 5   sby_need  1152 non-null   float64       
 6   dafted    1152 non-null   float64       
dtypes: datetime64[ns](1), float64(3), int64(3)
memory usage: 63.1 KB
```

In [ ]:
```
sickness_data.describe()
```

Out[ ]:

| | date | n_sick | calls | n_duty | n_sby | sby_need | dafted |
|---|---|---|---|---|---|---|---|
| **count** | 1152 | 1152.000000 | 1152.000000 | 1152.000000 | 1152.0 | 1152.000000 | 1152.000000 |
| **mean** | 2017-10-28 12:00:00 | 68.808160 | 7919.531250 | 1820.572917 | 90.0 | 34.718750 | 16.335938 |
| **min** | 2016-04-01 00:00:00 | 36.000000 | 4074.000000 | 1700.000000 | 90.0 | 0.000000 | 0.000000 |
| **25%** | 2017-01-13 18:00:00 | 58.000000 | 6978.000000 | 1800.000000 | 90.0 | 0.000000 | 0.000000 |
| **50%** | 2017-10-28 12:00:00 | 68.000000 | 7932.000000 | 1800.000000 | 90.0 | 0.000000 | 0.000000 |
| **75%** | 2018-08-12 06:00:00 | 78.000000 | 8827.500000 | 1900.000000 | 90.0 | 12.250000 | 0.000000 |
| **max** | 2019-05-27 00:00:00 | 119.000000 | 11850.000000 | 1900.000000 | 90.0 | 555.000000 | 465.000000 |
| **std** | NaN | 14.293942 | 1290.063571 | 80.086953 | 0.0 | 79.694251 | 53.394089 |

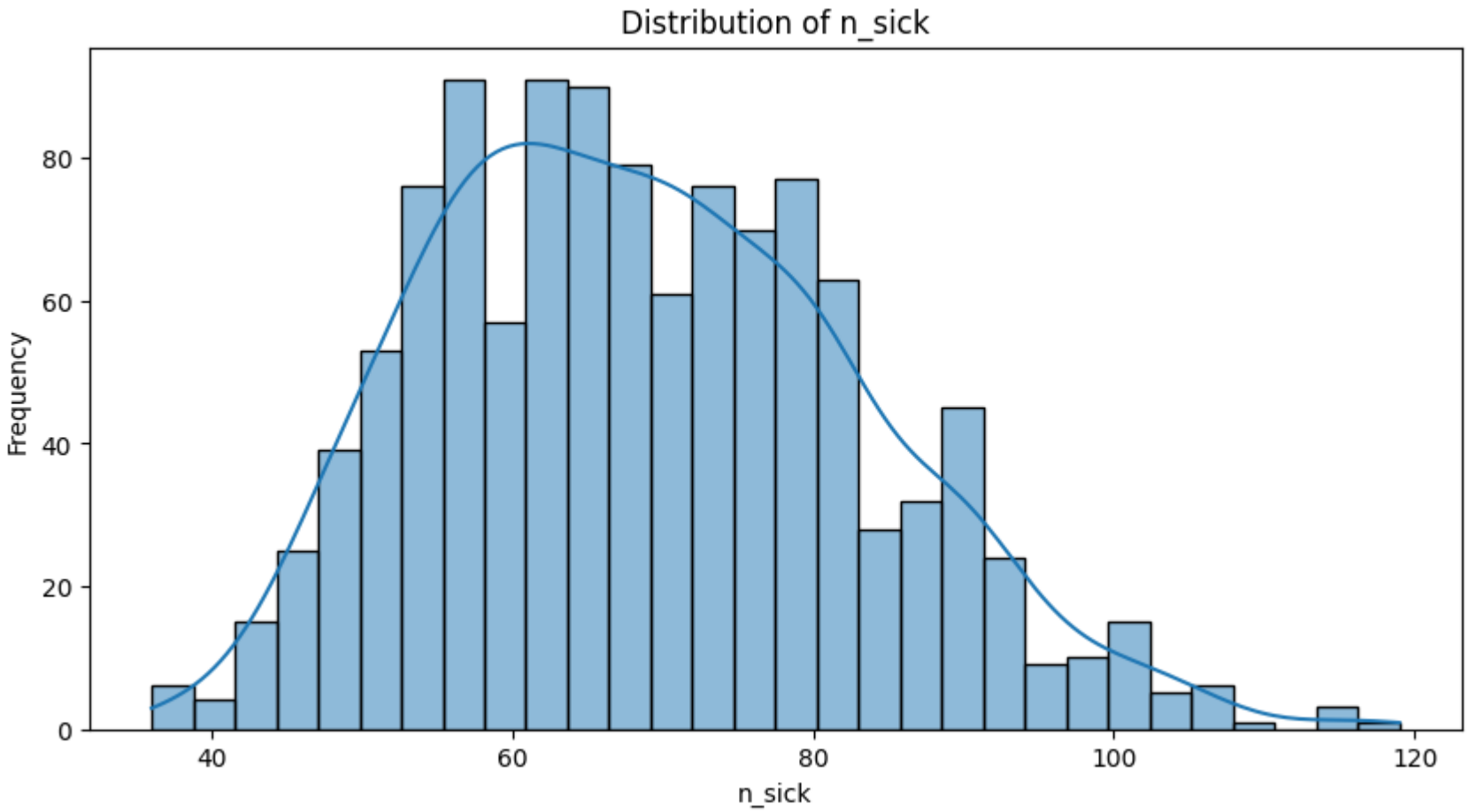
In [ ]:
```
sickness_data.isnull().sum()
```

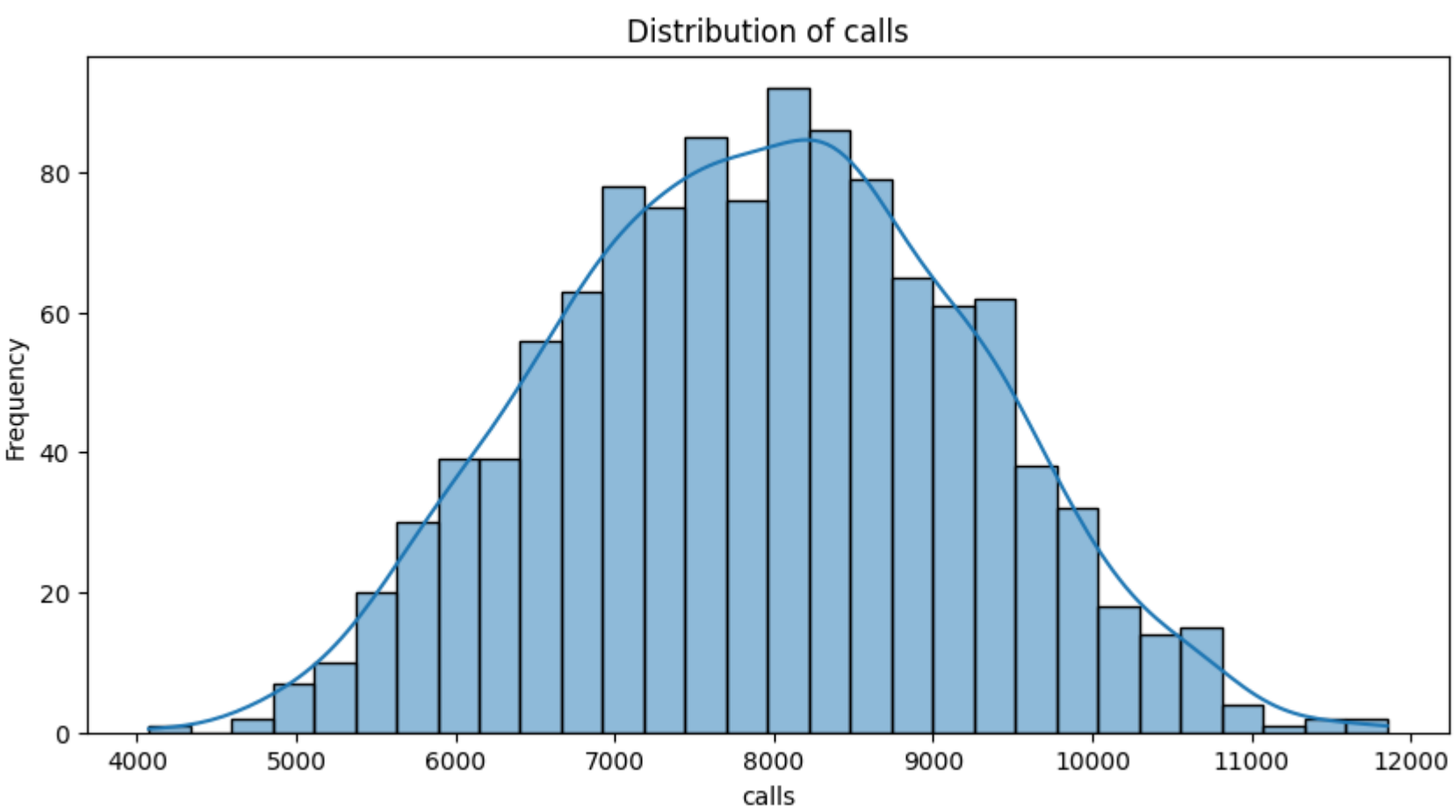
Out[ ]:
```
date        0
n_sick      0
calls       0
n_duty      0
n_sby       0
sby_need    0
dafted      0
dtype: int64
```

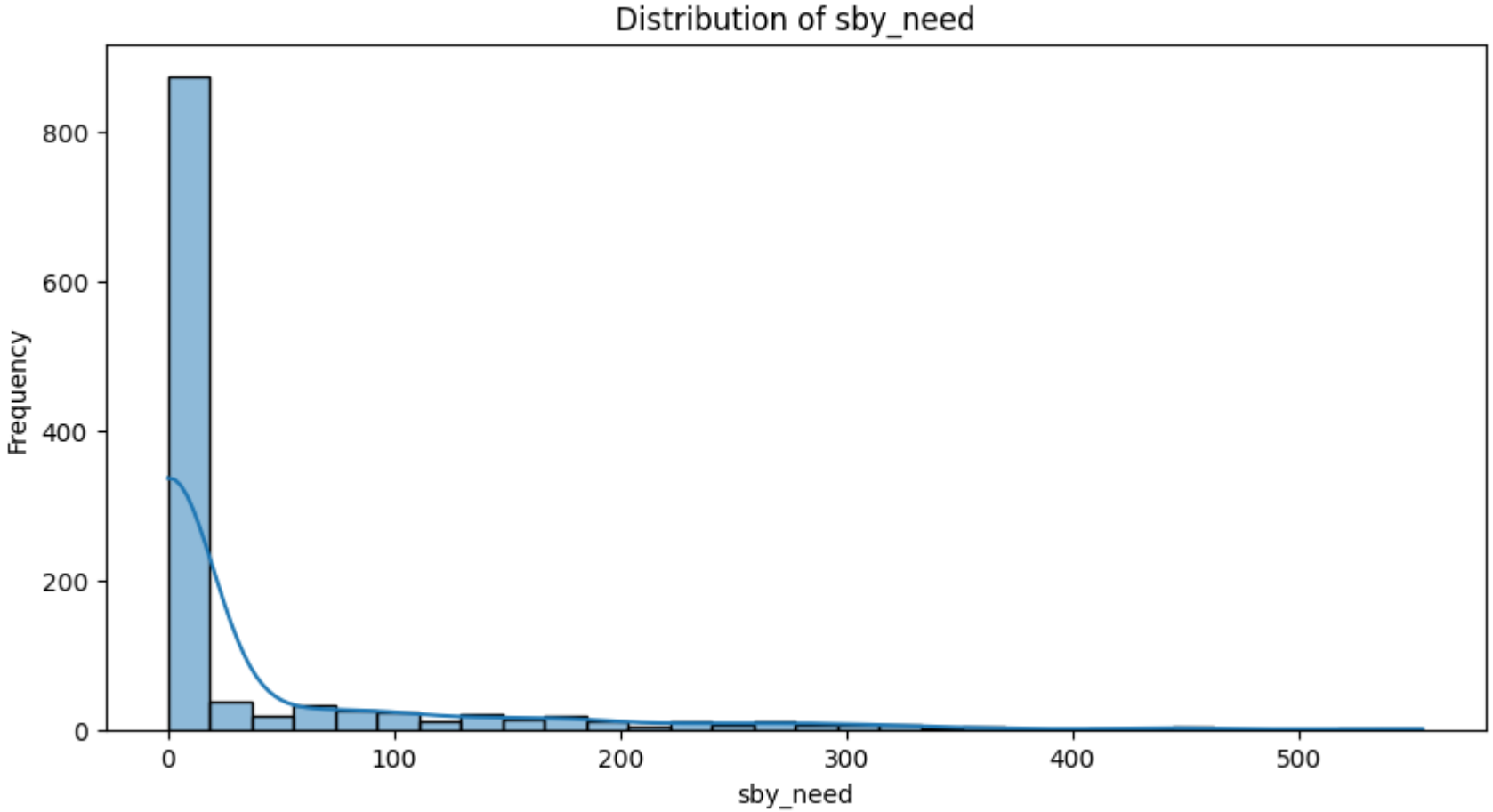
Data Exploration

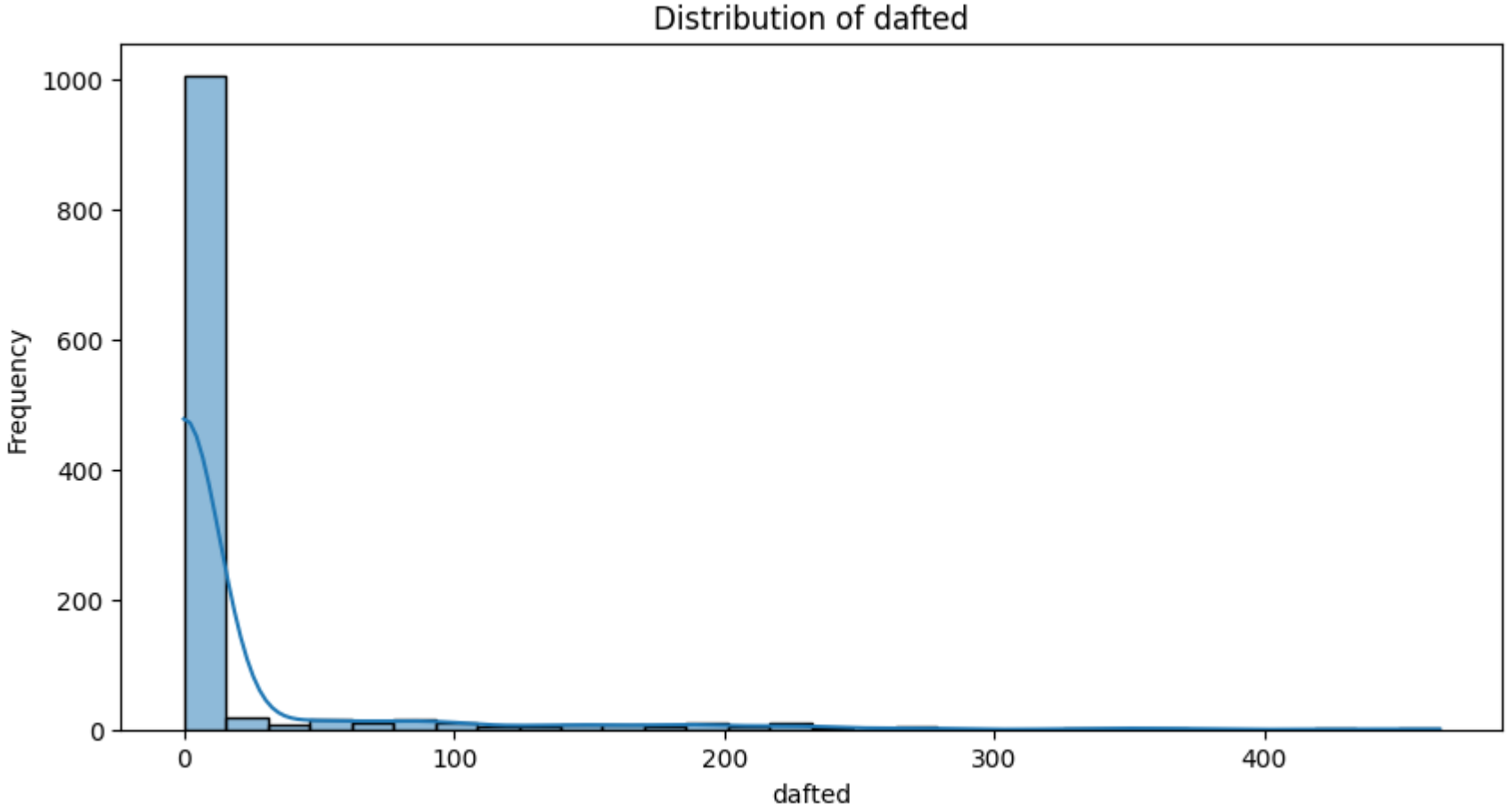
In [ ]:
```
# Columns of interest
columns_to_analyze = ['n_sick', 'calls', 'sby_need', 'dafted']

# Histograms for Distribution Analysis
for column in columns_to_analyze:
    plt.figure(figsize=(10, 5))
    sns.histplot(sickness_data[column], kde=True, bins=30)
    plt.title(f'Distribution of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.show()

# Boxplots for Outlier Analysis
for column in columns_to_analyze:
    plt.figure(figsize=(10, 5))
    sns.boxplot(x=sickness_data[column])
    plt.title(f'Boxplot of {column}')
    plt.xlabel(column)
```

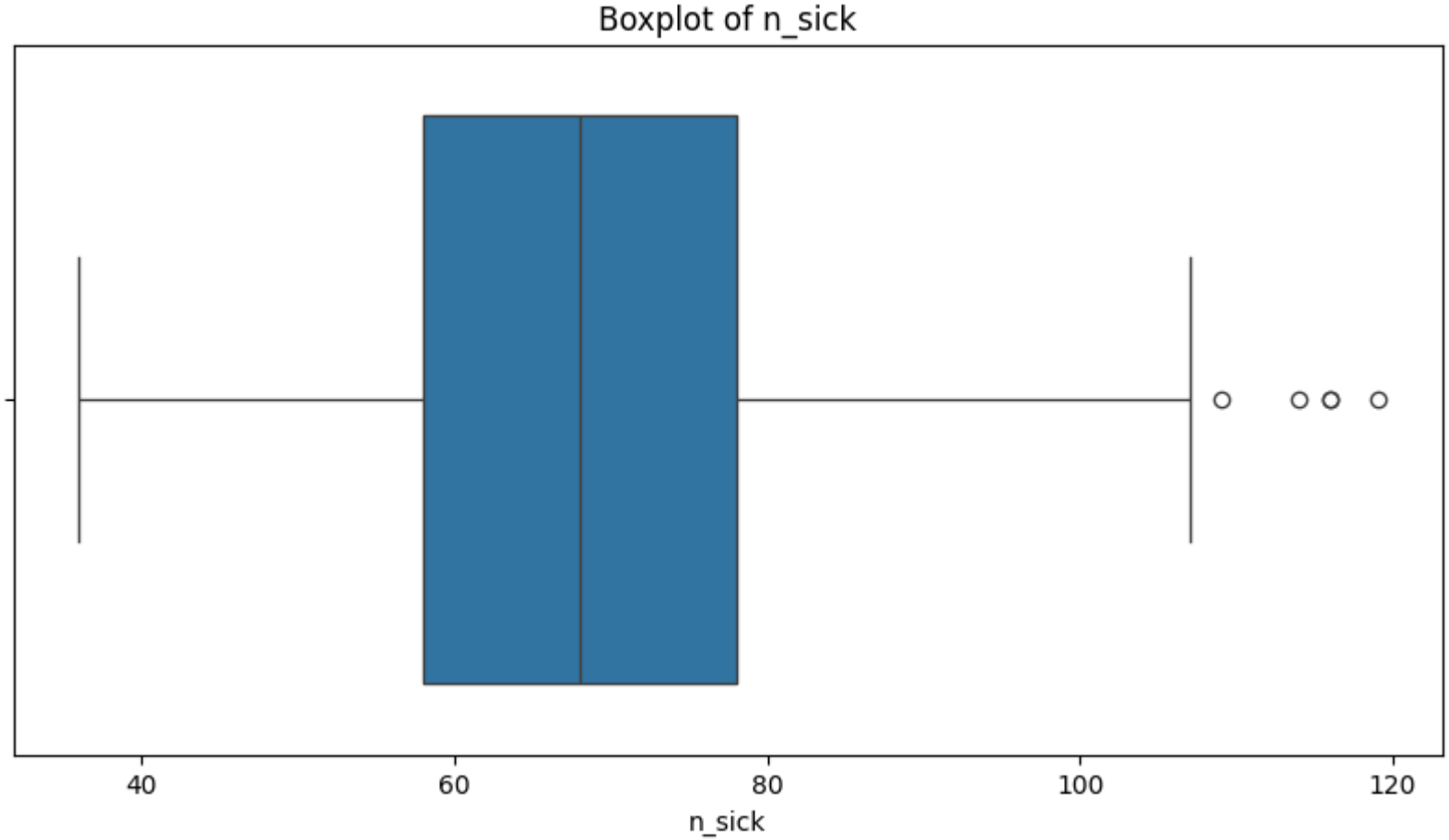
```
    plt.show()

# Summary Statistics
summary_stats = sickness_data[columns_to_analyze].describe().T[['mean', '50%', 'std']]
print(summary_stats)

# IQR for Outlier Detection
for column in columns_to_analyze:
    Q1 = sickness_data[column].quantile(0.25)
    Q3 = sickness_data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = sickness_data[(sickness_data[column] < lower_bound) | (sickness_data[column] > upper_bound)]
    print(f"Number of outliers detected in {column}: {len(outliers)}")
```

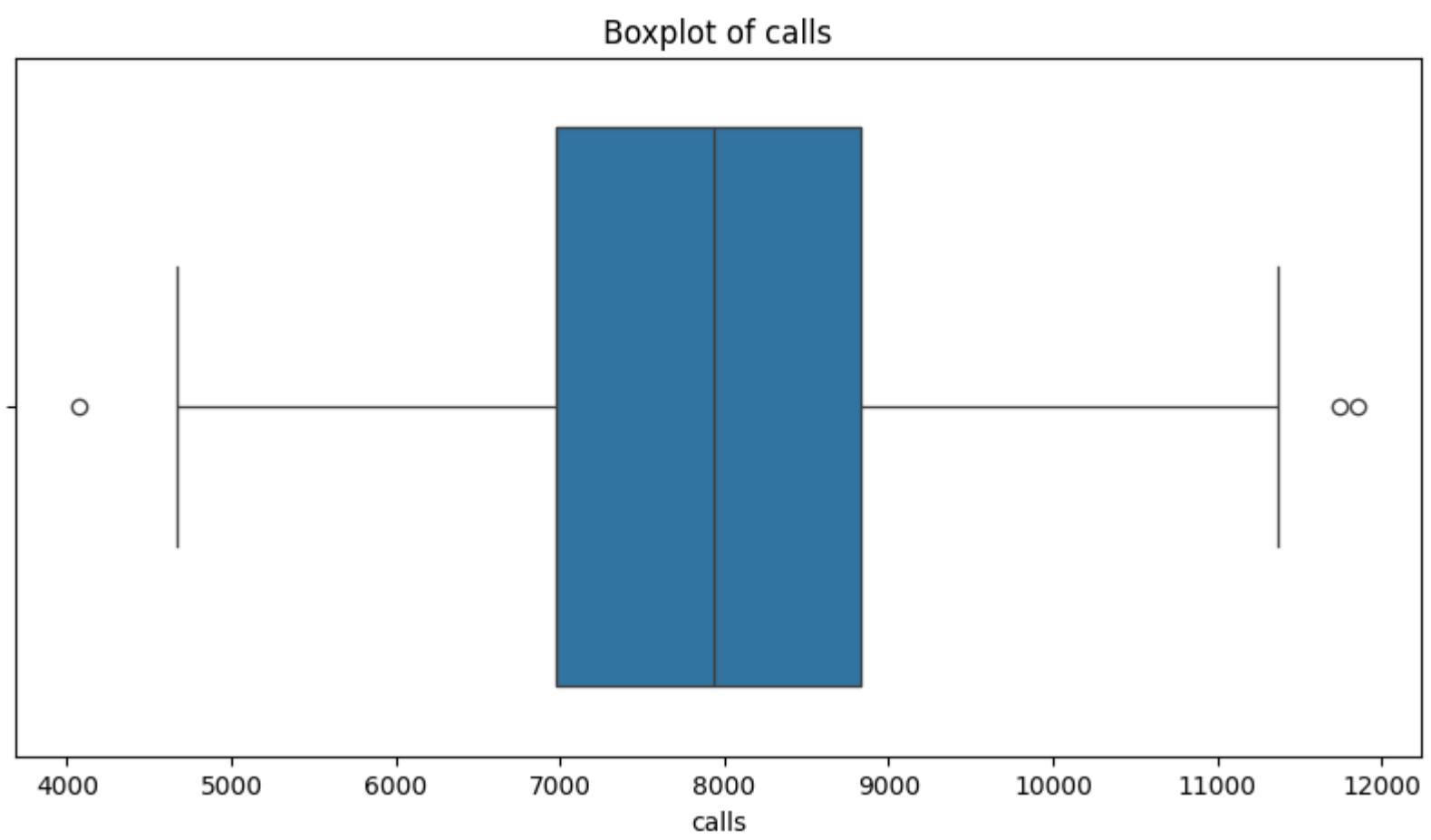


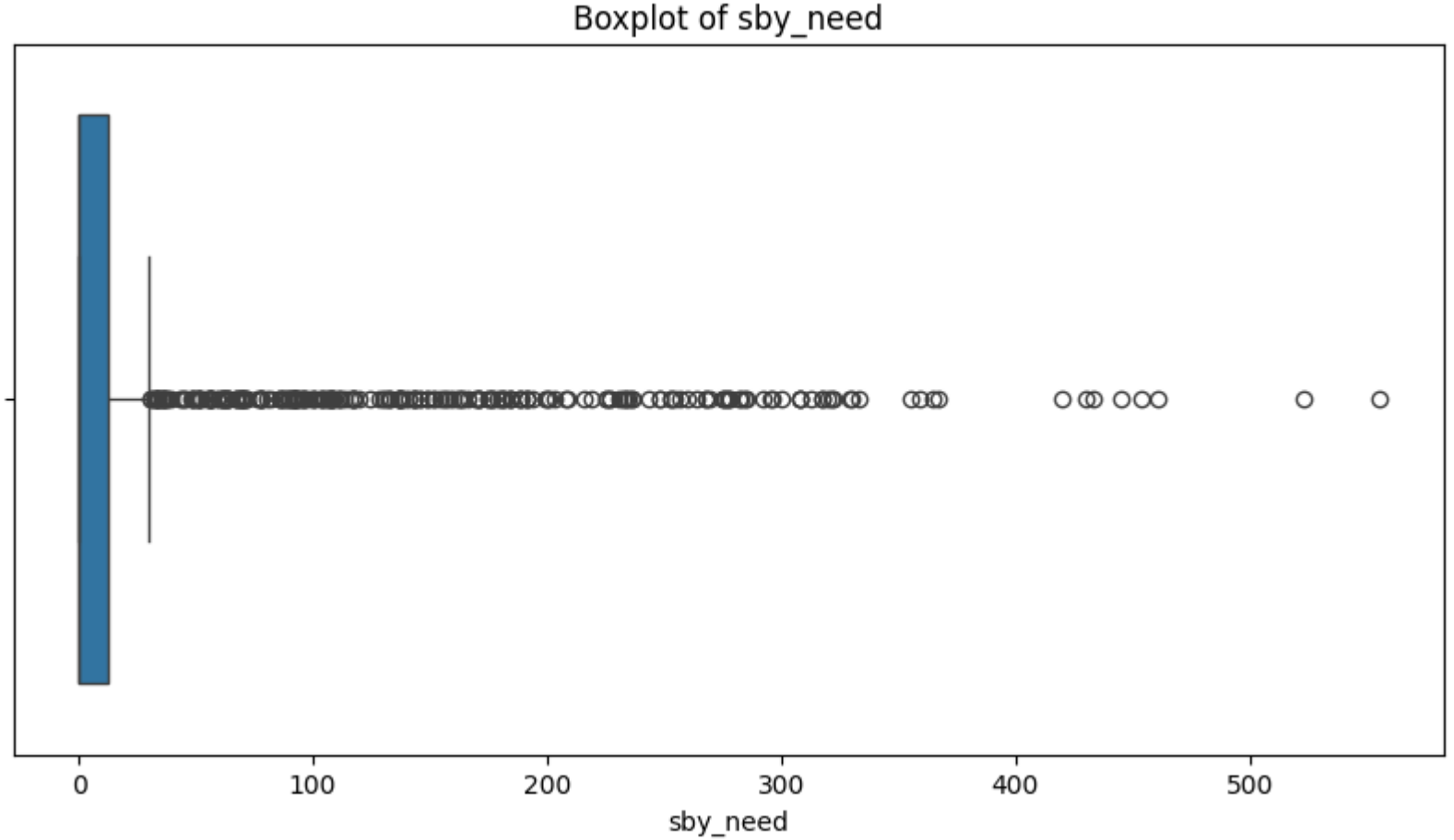

Distribution of calls
Frequency
0
20
40
60
80
4000
5000
6000
7000
8000
9000
10000
11000
12000
calls

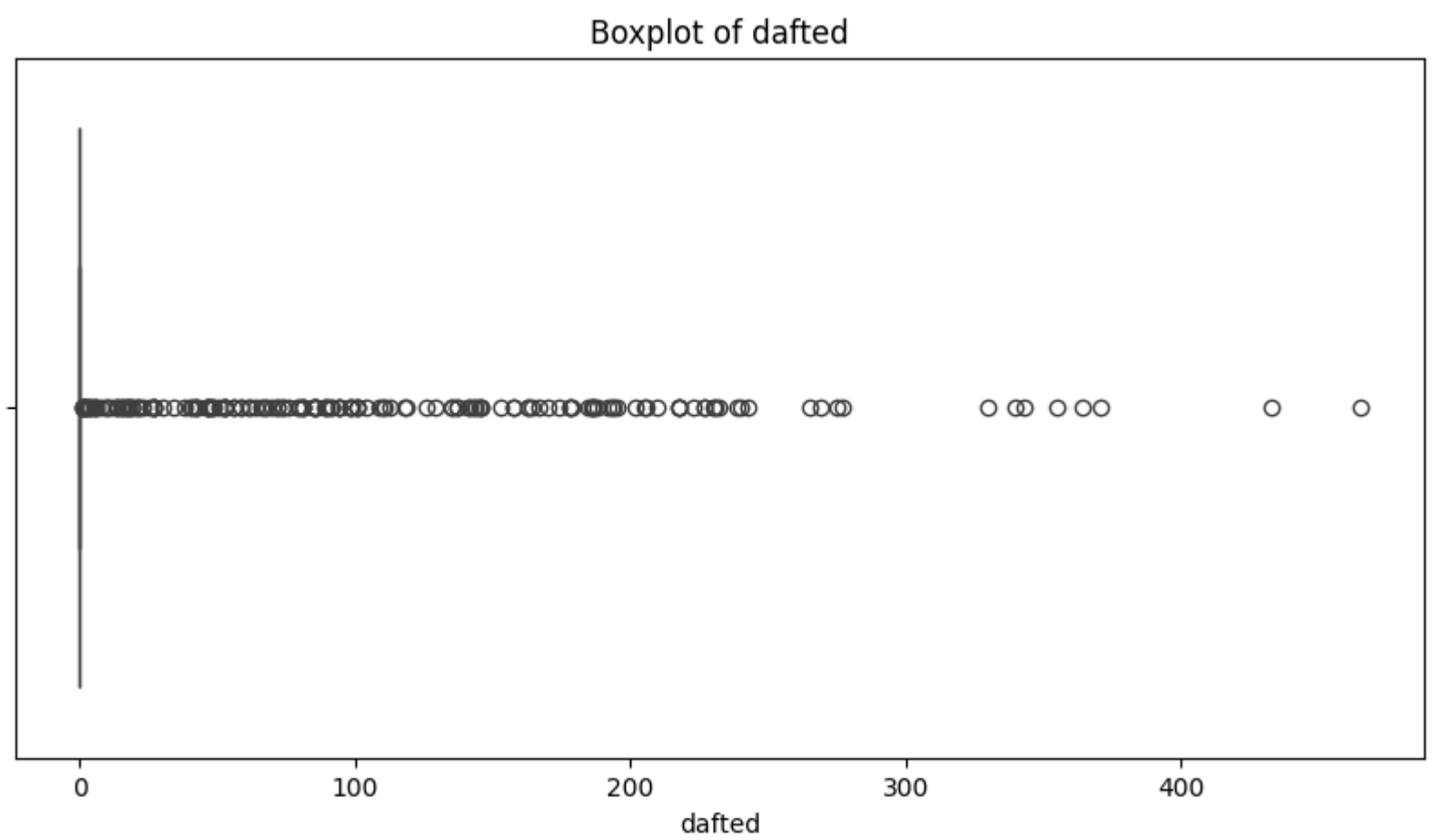

Distribution of sby_need
Frequency
sby_need
0
200
400
600
800
0
100
200
300
400
500


Distribution of dafted
Frequency
1000
800
600
400
200
0
0
100
200
300
400
dafted


Boxplot of n_sick
40
60
80
100
120
n_sick


Boxplot of calls
4000
5000
6000
7000
8000
9000
10000
11000
12000
calls


Boxplot of sby_need
0
100
200
300
400
500
sby_need




```
                mean     50%          std
n_sick      68.808160    68.0    14.293942
calls     7919.531250  7932.0  1290.063571
sby_need    34.718750     0.0    79.694251
dafted      16.335938     0.0    53.394089
Number of outliers detected in n_sick: 5
Number of outliers detected in calls: 3
Number of outliers detected in sby_need: 256
Number of outliers detected in dafted: 171
```

"Distribution of n_sick" : We can see that on an average 65-70 drivers were reported sick. The distribution is slightly right-skewed, meaning that there are a few days where the number of sick individuals is higher than the mode. There don't appear to be any extreme outliers, but there are some days where the number of sick individuals is close to 90, which are relatively rare occurrences.

"Distribution of Calls" : The distribution appears to be bimodal, with two distinct peaks. This suggests that there are two common ranges of call volumes. The first peak (mode) is observed around 7,000 to 7,500 calls. The second peak is observed around 8,500 to 9,000 calls. This means that there are two typical call volumes – one around 7,000-7,500 and another around 8,500-9,000. Skewness: The values of calls predominantly range from around 6,500 to 9,500, which

indicates the typical range of calls received. Outliers: There don't seem to be any extreme outliers, though there are fewer days with calls below 6,500 and above 9,500.

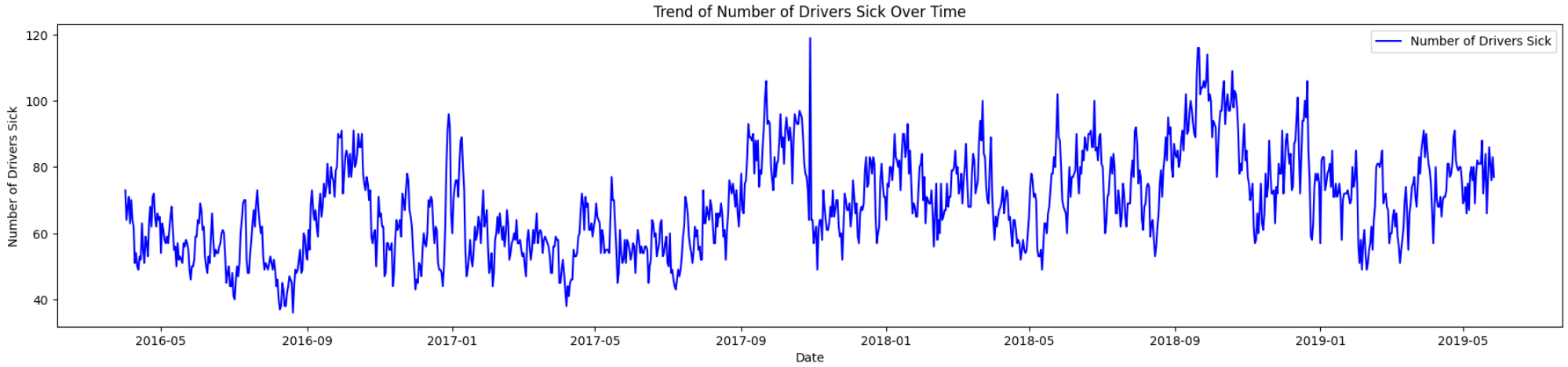
"Distribution of sby_need" :The mode, or the peak of the distribution, is at 0. This suggests that on most days, there was no reported standby need. Spread: While a significant portion of the data is clustered around 0, there are days when the standby need goes up to around 90, indicating variability in standby requirements. Skewness: The distribution is heavily right-skewed, implying that higher standby needs are rarer but still present in the dataset.

As per the statistics summary- n_sick: The number of sick individuals per day is fairly consistent, but there are a few days with unusually high or low numbers.

calls: While the call volume is generally around 7,919, there are days with significant deviations from this average.

sby_need: On most days, the standby need is low, but there are many days with unusually high requirements.

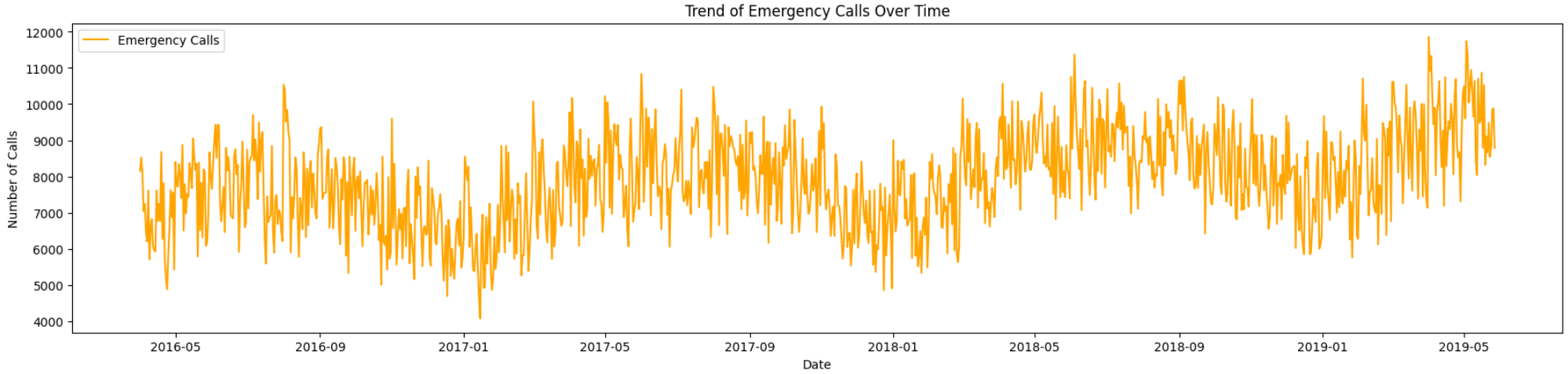
In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

# Plotting n_sick over time
plt.subplot(3, 1, 1)
plt.plot(sickness_data['date'], sickness_data['n_sick'], label='Number of Drivers Sick', color='blue')
plt.title('Trend of Number of Drivers Sick Over Time')
plt.xlabel('Date')
plt.ylabel('Number of Drivers Sick')
plt.legend()

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




Time-Series Analysis of n_sick: The presence of a consistent cyclical pattern suggests that there are predictable external factors influencing the number of sick drivers. Identifying and understanding these factors can be crucial for planning and resource allocation.

The absence of a long-term upward or downward trend indicates that, on a larger scale, the reporting patterns haven't changed drastically over the observed time span.

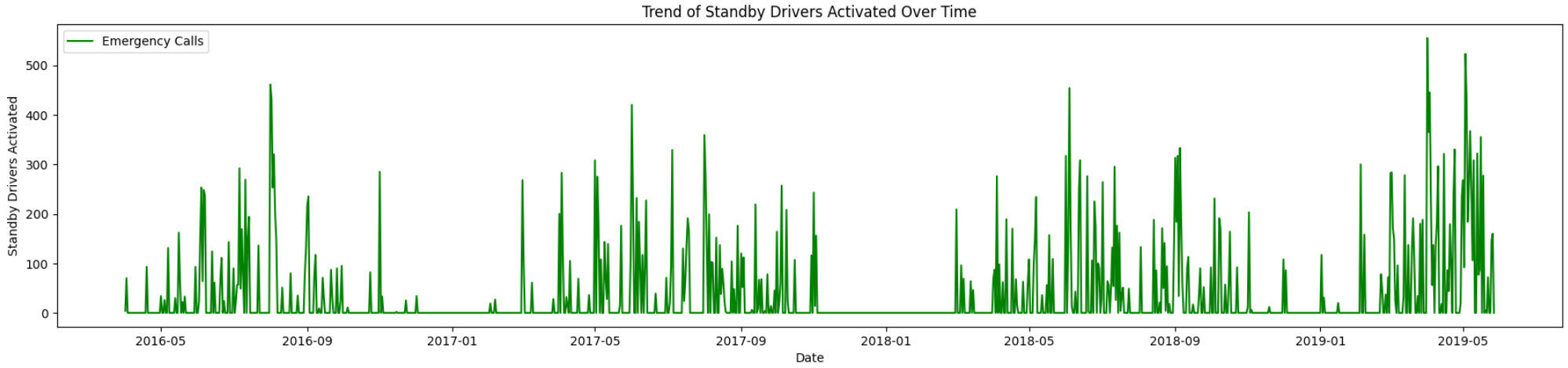
In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

# Plotting calls over time
plt.subplot(3, 1, 2)
plt.plot(sickness_data['date'], sickness_data['calls'], label='Emergency Calls', color='orange')
plt.title('Trend of Emergency Calls Over Time')
plt.xlabel('Date')
plt.ylabel('Number of Calls')
plt.legend()

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




Time Series Analysis of Calls: The strong cyclical pattern suggests that there are recurring external factors influencing the number of emergency calls. The minor anomalies, where the number of calls dips below the general trend, could be due to specific events or other factors

In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

# Plotting standby drivers activated
plt.subplot(3, 1, 1)
plt.plot(sickness_data['date'], sickness_data['sby_need'], label='Emergency Calls', color='green')
plt.title('Trend of Standby Drivers Activated Over Time')
plt.xlabel('Date')
plt.ylabel('Standby Drivers Activated')
plt.legend()
```

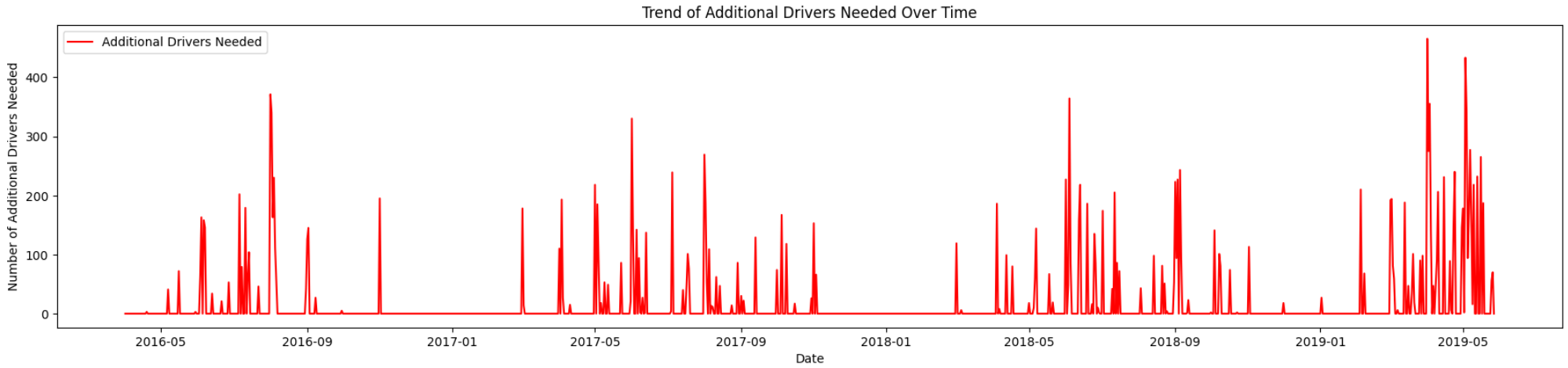
```
plt.tight_layout()
plt.show()
```




Time Series Analysis of Standby Drivers Needed: While most days require minimal to no standby drivers, there are specific days when the need surges significantly.

The recurring spikes suggest that there are predictable factors or events leading to increased standby driver requirements

In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

# Plotting dafted over time
plt.subplot(3, 1, 1)
plt.plot(sickness_data['date'], sickness_data['dafted'], label='Additional Drivers Needed', color='red')
plt.title('Trend of Additional Drivers Needed Over Time')
plt.xlabel('Date')
plt.ylabel('Number of Additional Drivers Needed')
plt.legend()

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




Time Series Analysis of Additional Drivers Needed: The consistent low values on most days suggest that the regular staffing levels are generally sufficient, but there are occasional days when the demand exceeds the supply, necessitating additional drivers. The presence of spikes, similar to the standby driver requirements, suggests that there are certain factors or events that lead to an increased need for additional drivers.

In [ ]:
```
sickness_data.corr()
```

Out[ ]:

| | date | n_sick | calls | n_duty | n_sby | sby_need | dafted |
|---|---|---|---|---|---|---|---|
| **date** | 1.000000 | 0.495959 | 0.385679 | 0.927437 | NaN | 0.137543 | 0.131938 |
| **n_sick** | 0.495959 | 1.000000 | 0.155371 | 0.459501 | NaN | 0.022321 | 0.016800 |
| **calls** | 0.385679 | 0.155371 | 1.000000 | 0.364135 | NaN | 0.677468 | 0.557340 |
| **n_duty** | 0.927437 | 0.459501 | 0.364135 | 1.000000 | NaN | 0.090654 | 0.084955 |
| **n_sby** | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **sby_need** | 0.137543 | 0.022321 | 0.677468 | 0.090654 | NaN | 1.000000 | 0.945168 |
| **dafted** | 0.131938 | 0.016800 | 0.557340 | 0.084955 | NaN | 0.945168 | 1.000000 |

In [ ]:
```
plt.figure(figsize= (12, 10))

sns.heatmap(sickness_data.corr(), annot = True)
```

Out[ ]:
```
<Axes: >
```


date
n_sick
calls
n_duty
n_sby
sby_need
dafted
1
0.5
0.39
0.93
0.14
0.13
0.5
1
0.16
0.46
0.022
0.017
0.39
0.16
1
0.36
0.68
0.56
0.93
0.46
0.36
1
0.091
0.085
0.14
0.022
0.68
0.091
1
0.95
0.13
0.017
0.56
0.085
0.95
1
1.0
0.8
0.6
0.4
0.2

date n_sick calls n_duty n_sby sby_need dafted

Correlation Analysis: The strong positive correlations between n_sick, sby_need, and dafted suggest that the number of sick individuals directly influences the need for standby and additional drivers. As more drivers are reported sick, there's a surge in the requirement for backup and additional drivers.

The call volume (calls) seems to operate independently of the other variables, suggesting that external factors might be influencing it more than internal staffing or health conditions.

Feature Engineering

Here, we are extracting the Date Features, year, month, day, day_of_week, quarter which can help capture any time-related patterns, seasonality, or trends present in the data. For instance, the number of sick drivers might be higher during certain months due to seasonal illnesses, or call volumes might differ on weekdays vs. weekends.

In [ ]:
```
# Extract year, month, day, and day of the week
sickness_data['year'] = sickness_data['date'].dt.year
sickness_data['month'] = sickness_data['date'].dt.month
sickness_data['day'] = sickness_data['date'].dt.day
sickness_data['day_of_week'] = sickness_data['date'].dt.dayofweek

# Create a binary feature indicating if the day is a weekend
sickness_data['is_weekend'] = sickness_data['day_of_week'].apply(lambda x: 1 if x >= 5 else 0)

# Extract quarter
sickness_data['quarter'] = sickness_data['date'].dt.quarter
```

Lagged features can capture the temporal dependencies in time-series data. If today's data is influenced by the past few days, these features will be valuable.

In [ ]:
```
# Create lagged features for 'n_sick' and 'calls' for 1 day and 2 days
sickness_data['n_sick_lag1'] = sickness_data['n_sick'].shift(1)
sickness_data['n_sick_lag2'] = sickness_data['n_sick'].shift(2)
sickness_data['calls_lag1'] = sickness_data['calls'].shift(1)
sickness_data['calls_lag2'] = sickness_data['calls'].shift(2)
```

Rolling statistics can provide a smoothed version of the original time series, capturing short-term trends and patterns. This can be especially useful when forecasting.

In [ ]:
```
# Create rolling mean and standard deviation for 'n_sick' and 'calls' over a 7-day window
sickness_data['n_sick_roll_mean'] = sickness_data['n_sick'].rolling(window=7).mean()
sickness_data['n_sick_roll_std'] = sickness_data['n_sick'].rolling(window=7).std()
sickness_data['calls_roll_mean'] = sickness_data['calls'].rolling(window=7).mean()
sickness_data['calls_roll_std'] = sickness_data['calls'].rolling(window=7).std()
```

These features capture the day-to-day changes, which can be helpful in understanding volatility or sudden changes in the series.

In [ ]:
```
# Calculate day-to-day difference for 'n_sick' and 'calls'
sickness_data['n_sick_diff'] = sickness_data['n_sick'].diff()
sickness_data['calls_diff'] = sickness_data['calls'].diff()
```

Ratios can provide normalized metrics, which can be more informative than raw numbers. For example, if you have an unusually high number of sick drivers but also a very high number of total drivers, the situation might not be as critical as when the total number of drivers is low.

In [ ]:
```
# Sick to Available Ratio: Ratio of drivers who called in sick to the total number of drivers available
sickness_data['sick_to_available_ratio'] = sickness_data['n_sick'] / (sickness_data['n_duty'] + sickness_data['n_sby'])

# Emergency Call to Driver Ratio: Ratio of emergency calls to the total number of drivers (both on duty and standby)
sickness_data['calls_to_driver_ratio'] = sickness_data['calls'] / (sickness_data['n_duty'] + sickness_data['n_sby'])
```

We have chosen (n_sick *calls) so we can identify days when the system is under particular stress due to these combined factors. Similarly, for (calls* (n_duty + n_sby)): This interaction captures the relationship between demand (calls) and supply (available drivers). On days with high call volumes, if there are enough drivers available (either on duty or on standby), the system can cope. However, if there's a high call volume and a lower number of available drivers, the system might be overwhelmed.

In [ ]:
```
# Interaction between the number of drivers who called in sick and the number of emergency calls
sickness_data['sick_calls_interaction'] = sickness_data['n_sick'] * sickness_data['calls']

# Interaction between the number of emergency calls and available drivers (both on duty and standby)
sickness_data['calls_driver_interaction'] = sickness_data['calls'] * (sickness_data['n_duty'] + sickness_data['n_sby'])
```

In [ ]:
```
pd.set_option('display.max_columns', 30)

sickness_data.head(10)
```

Out[ ]:

| | date | n_sick | calls | n_duty | n_sby | sby_need | dafted | year | month | day | day_of_week | is_weekend | quarter | n_sick_lag1 | n_sick_lag2 | calls_lag1 | calls_lag2 | n_si |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2016-04-01 | 73 | 8154.0 | 1700 | 90 | 4.0 | 0.0 | 2016 | 4 | 1 | 4 | 0 | 2 | NaN | NaN | NaN | NaN | |
| **1** | 2016-04-02 | 64 | 8526.0 | 1700 | 90 | 70.0 | 0.0 | 2016 | 4 | 2 | 5 | 1 | 2 | 73.0 | NaN | 8154.0 | NaN | |
| **2** | 2016-04-03 | 68 | 8088.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 3 | 6 | 1 | 2 | 64.0 | 73.0 | 8526.0 | 8154.0 | |
| **3** | 2016-04-04 | 71 | 7044.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 4 | 0 | 0 | 2 | 68.0 | 64.0 | 8088.0 | 8526.0 | |
| **4** | 2016-04-05 | 63 | 7236.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 5 | 1 | 0 | 2 | 71.0 | 68.0 | 7044.0 | 8088.0 | |
| **5** | 2016-04-06 | 70 | 6492.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 6 | 2 | 0 | 2 | 63.0 | 71.0 | 7236.0 | 7044.0 | |
| **6** | 2016-04-07 | 64 | 6204.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 7 | 3 | 0 | 2 | 70.0 | 63.0 | 6492.0 | 7236.0 | |
| **7** | 2016-04-08 | 62 | 7614.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 8 | 4 | 0 | 2 | 64.0 | 70.0 | 6204.0 | 6492.0 | |
| **8** | 2016-04-09 | 51 | 5706.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 9 | 5 | 1 | 2 | 62.0 | 64.0 | 7614.0 | 6204.0 | |
| **9** | 2016-04-10 | 54 | 6606.0 | 1700 | 90 | 0.0 | 0.0 | 2016 | 4 | 10 | 6 | 1 | 2 | 51.0 | 62.0 | 5706.0 | 7614.0 | |

In [ ]:
```
sickness_data.shape
```

Out[ ]:
```
(1152, 27)
```

Handling Missing Values

In [ ]:
```
# Columns for which to apply forward fill and then backward fill
columns_to_ffill = [
    'n_sick_lag1', 'n_sick_lag2', 'calls_lag1', 'calls_lag2',
    'n_sick_diff', 'calls_diff'
]

# Apply both forward fill and backward fill for these columns
for column in columns_to_ffill:
    sickness_data[column] = sickness_data[column].ffill().bfill()
```

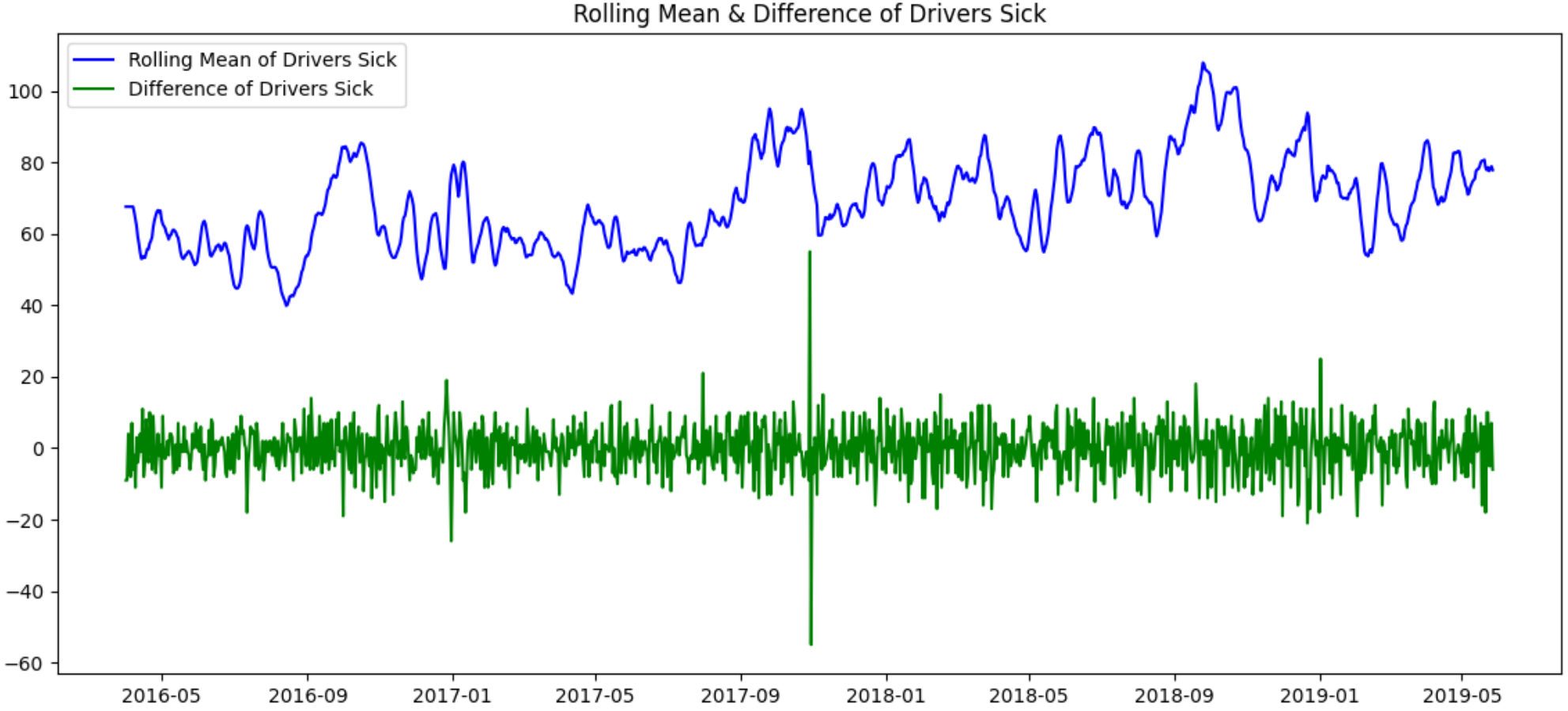
```
# Columns for which to apply backward fill
rolling_columns_to_bfill = [
    'n_sick_roll_mean', 'n_sick_roll_std', 'calls_roll_mean', 'calls_roll_std'
]

# Apply backward fill for these columns
for column in rolling_columns_to_bfill:
    sickness_data[column] = sickness_data[column].bfill()


#Verify there are no more NaN values
nan_after_handling = sickness_data.isna().sum()
print(nan_after_handling)
```

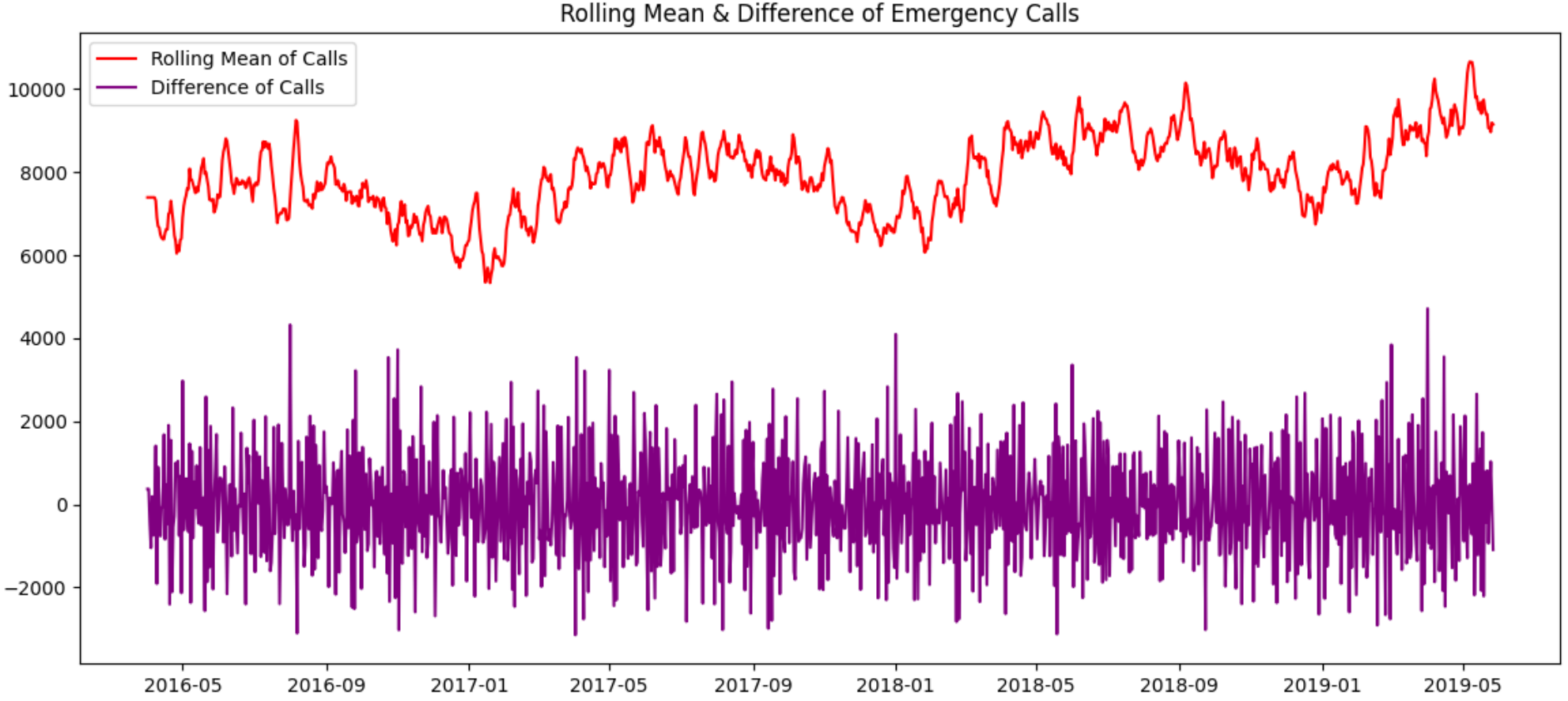
```
date                        0
n_sick                      0
calls                       0
n_duty                      0
n_sby                       0
sby_need                    0
dafted                      0
year                        0
month                       0
day                         0
day_of_week                 0
is_weekend                  0
quarter                     0
n_sick_lag1                 0
n_sick_lag2                 0
calls_lag1                  0
calls_lag2                  0
n_sick_roll_mean            0
n_sick_roll_std             0
calls_roll_mean             0
calls_roll_std              0
n_sick_diff                 0
calls_diff                  0
sick_to_available_ratio     0
calls_to_driver_ratio       0
sick_calls_interaction      0
calls_driver_interaction    0
dtype: int64
```

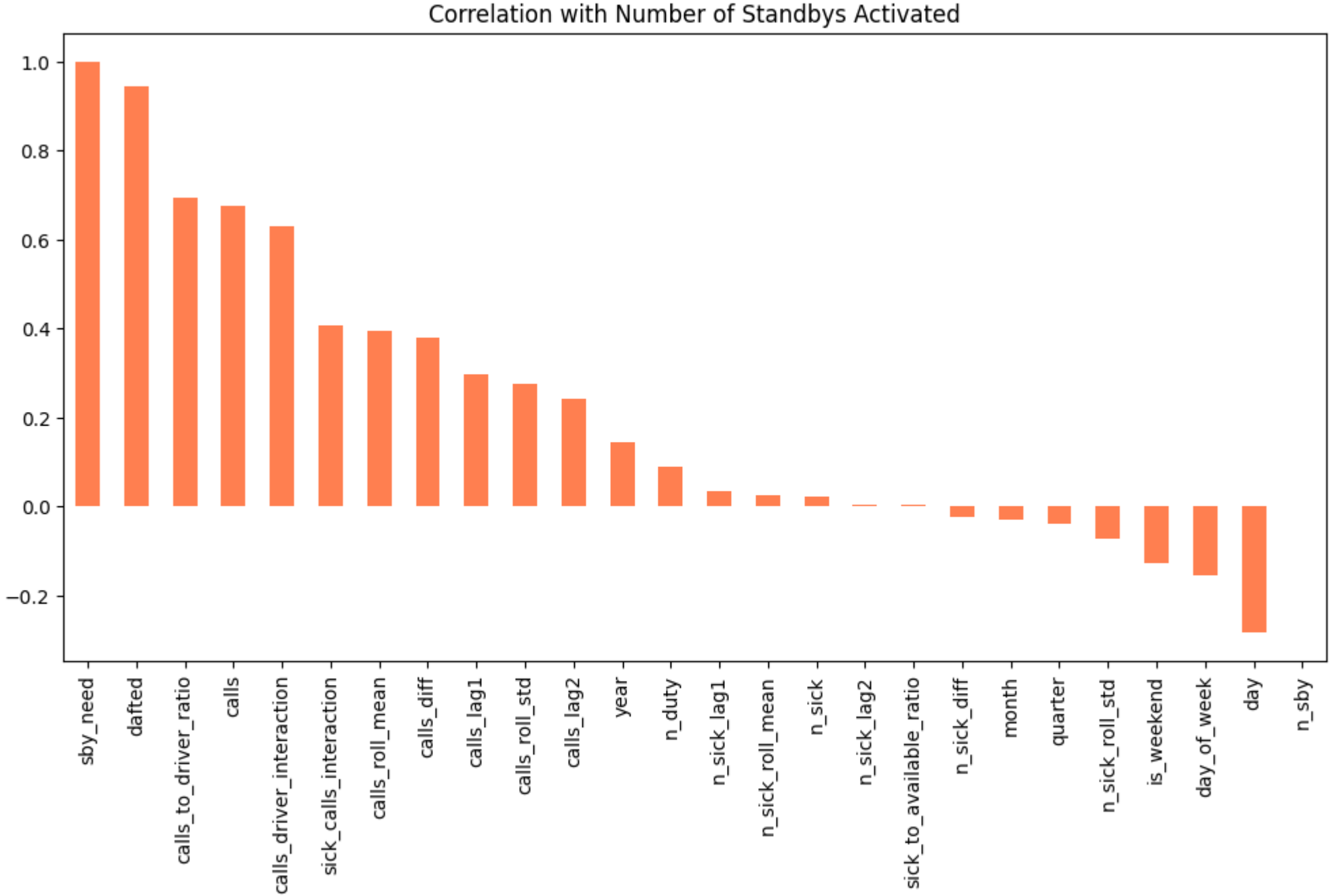
In [ ]:

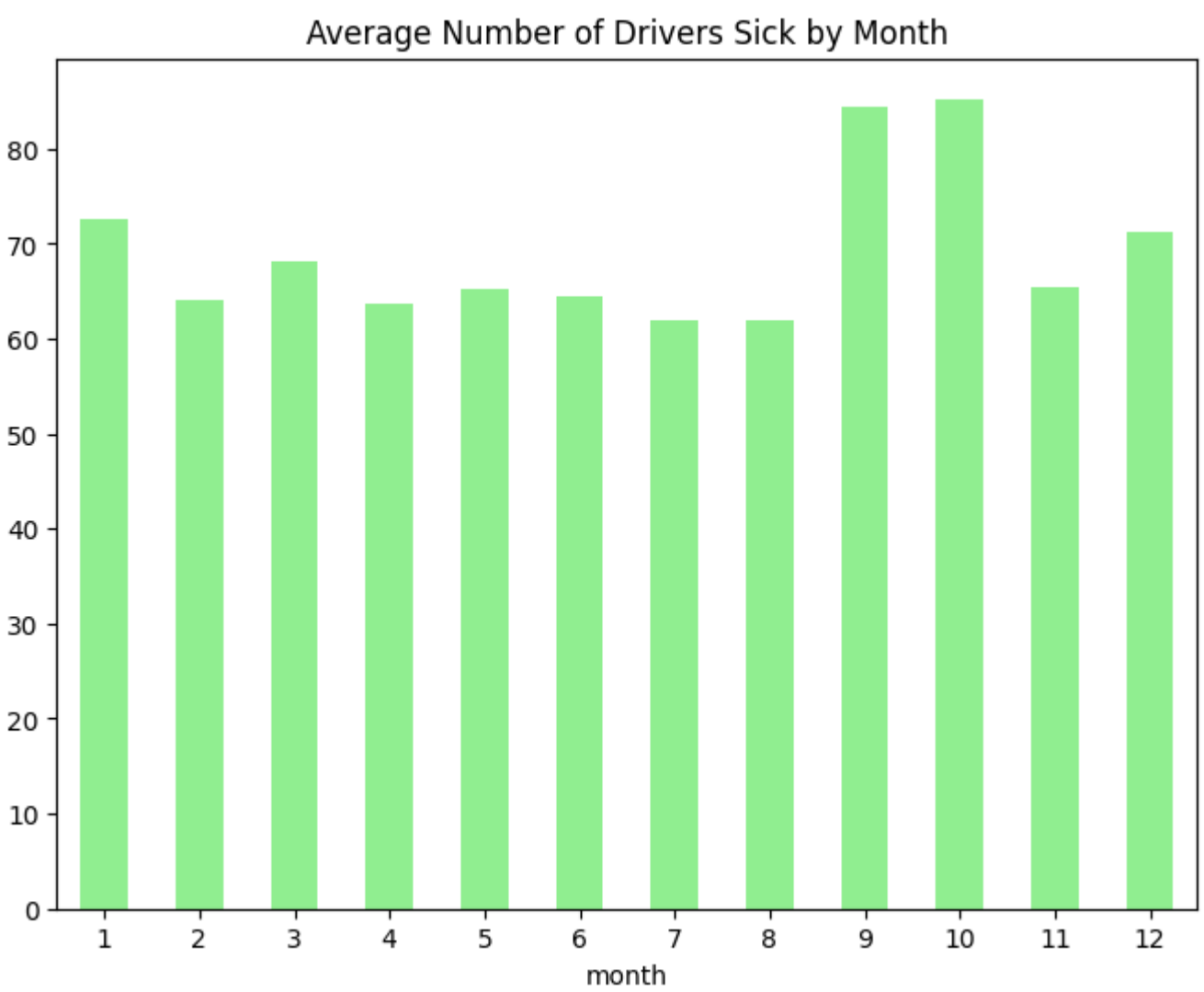
```
# 1. Temporal Visualization of New Features:
plt.figure(figsize=(14, 6))
plt.plot(sickness_data['date'], sickness_data['n_sick_roll_mean'], label='Rolling Mean of Drivers Sick', color='blue')
plt.plot(sickness_data['date'], sickness_data['n_sick_diff'], label='Difference of Drivers Sick', color='green')
plt.title('Rolling Mean & Difference of Drivers Sick')
```

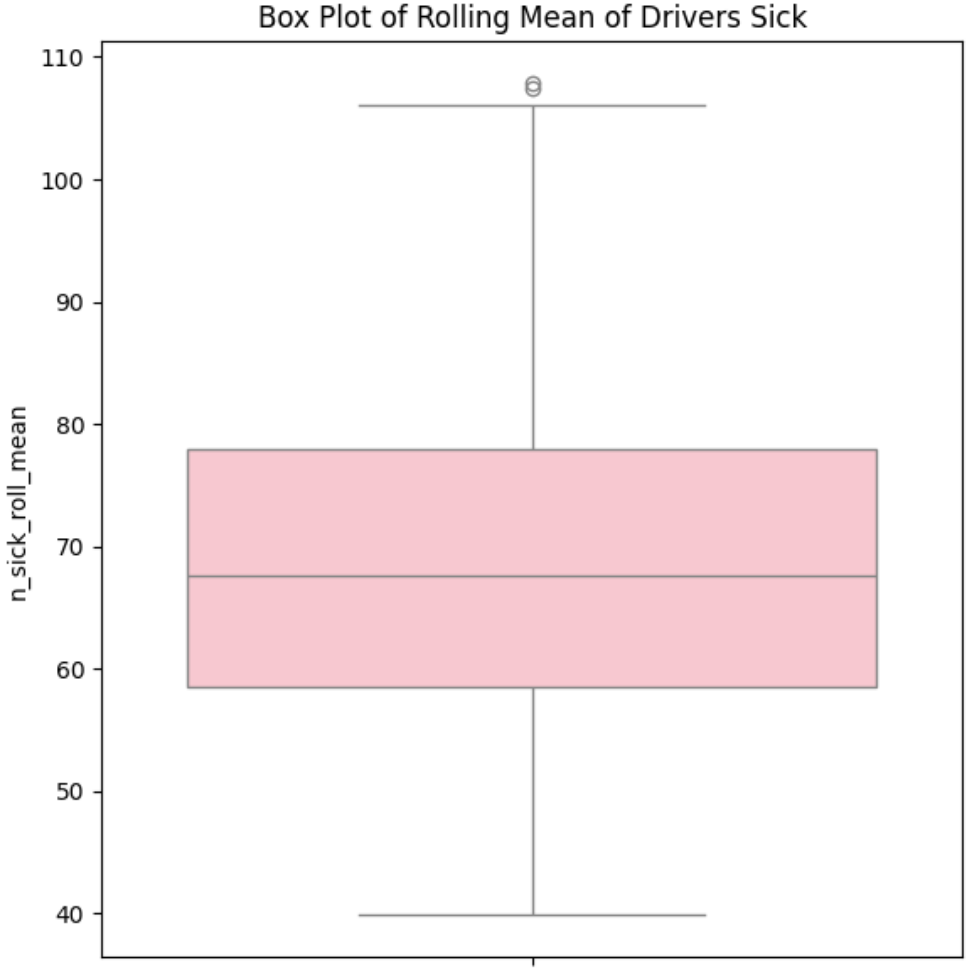
```
plt.legend()
plt.show()

plt.figure(figsize=(14, 6))
plt.plot(sickness_data['date'], sickness_data['calls_roll_mean'], label='Rolling Mean of Calls', color='red')
plt.plot(sickness_data['date'], sickness_data['calls_diff'], label='Difference of Calls', color='purple')
plt.title('Rolling Mean & Difference of Emergency Calls')
plt.legend()
plt.show()

# 2. Correlation Analysis:
correlations = sickness_data.drop(columns=['date']).corr()['sby_need'].sort_values(ascending=False)
plt.figure(figsize=(12, 6))
correlations.plot(kind='bar', color='coral')
plt.title('Correlation with Number of Standbys Activated')
plt.show()

# 3. Seasonal Patterns:
plt.figure(figsize=(8, 6))
month_avg_sick = sickness_data.groupby('month')['n_sick'].mean()
month_avg_sick.plot(kind='bar', color='lightgreen')
plt.title('Average Number of Drivers Sick by Month')
plt.xticks(ticks=range(12), labels=month_avg_sick.index, rotation=0)
plt.show()

# 4. Box Plots:
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.boxplot(data=sickness_data, y='n_sick_roll_mean', color='pink')
plt.title('Box Plot of Rolling Mean of Drivers Sick')

plt.subplot(1, 2, 2)
sns.boxplot(data=sickness_data, y='calls_diff', color='yellow')
plt.title('Box Plot of Difference of Emergency Calls')
plt.tight_layout()
plt.show()

# 5. Heatmap of Correlations:
plt.figure(figsize=(12, 10))
correlation_matrix = sickness_data.drop(columns=['date']).corr()
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=True, fmt=".2f")
plt.title('Heatmap of Feature Correlations')
plt.show()
```

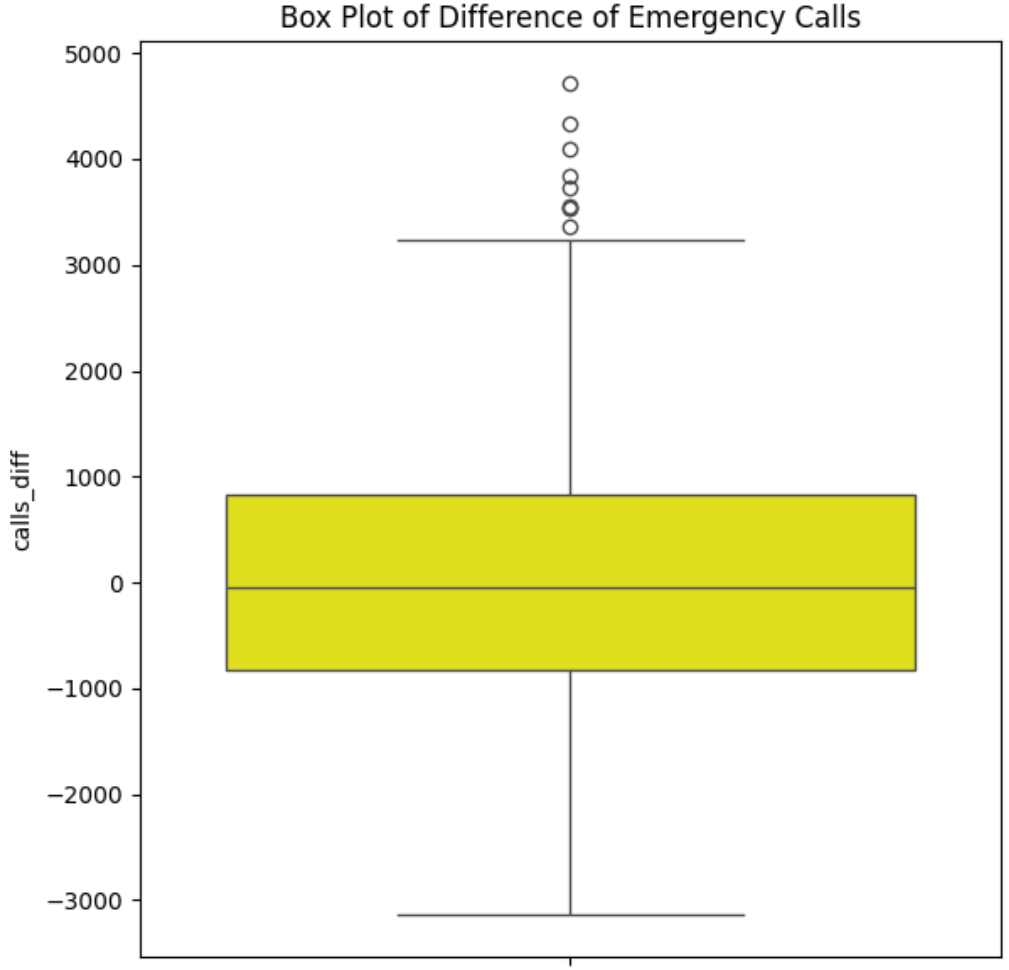

Rolling Mean & Difference of Drivers Sick
Rolling Mean of Drivers Sick
Difference of Drivers Sick
100
80
60
40
20
0
−20
−40
−60
2016-05
2016-09
2017-01
2017-05
2017-09
2018-01
2018-05
2018-09
2019-01
2019-05

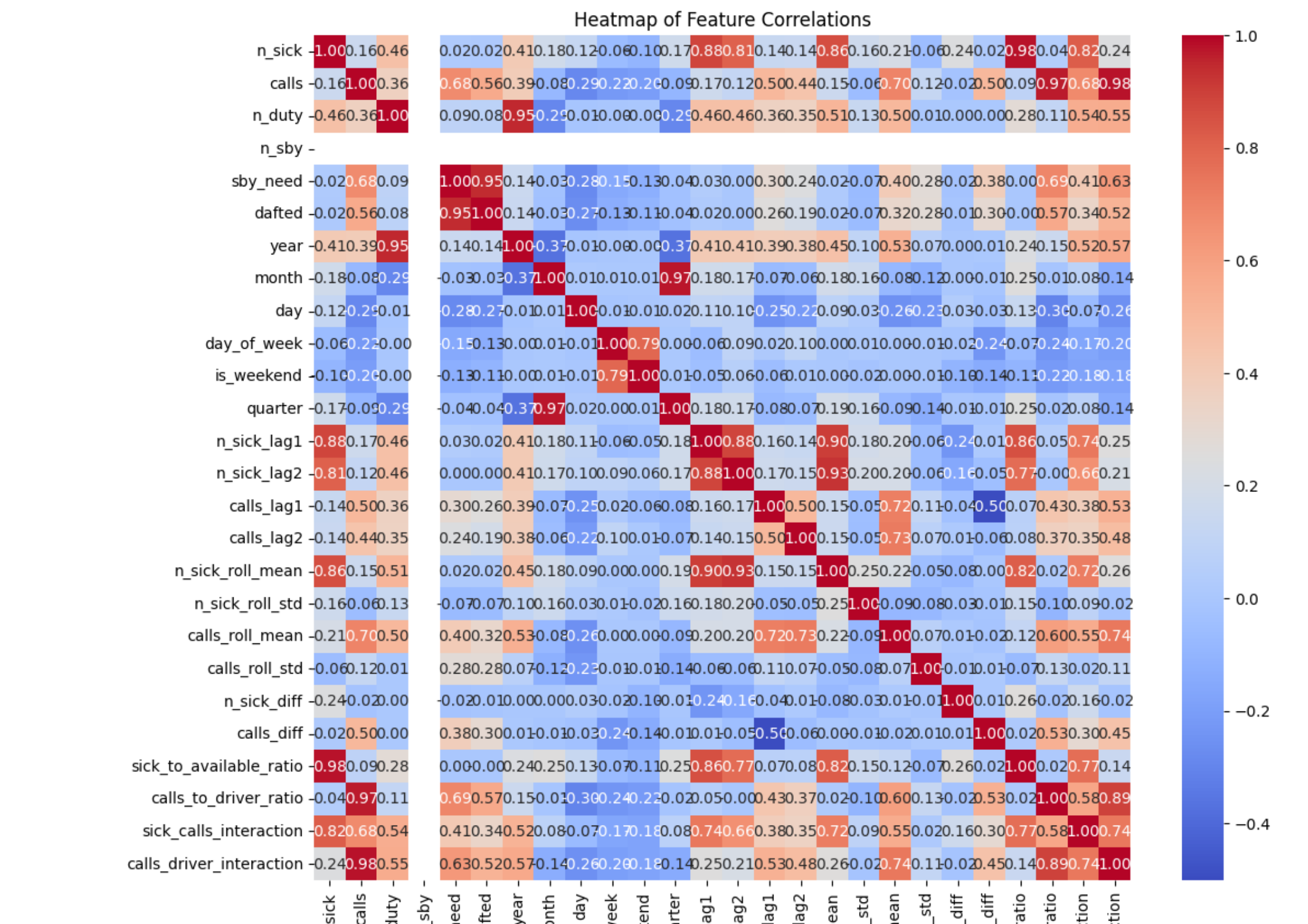

Rolling Mean & Difference of Emergency Calls
Rolling Mean of Calls
Difference of Calls
10000
8000
6000
4000
2000
0
−2000
2016-05
2016-09
2017-01
2017-05
2017-09
2018-01
2018-05
2018-09
2019-01
2019-05


Correlation with Number of Standbys Activated
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
sby_need
dafted
calls_to_driver_ratio
calls
calls_driver_interaction
sick_calls_interaction
calls_roll_mean
calls_diff
calls_lag1
calls_roll_std
calls_lag2
year
n_duty
n_sick_lag1
n_sick_roll_mean
n_sick
n_sick_lag2
sick_to_available_ratio
n_sick_diff
month
quarter
n_sick_roll_std
is_weekend
day_of_week
day
n_sby


Average Number of Drivers Sick by Month
0
10
20
30
40
50
60
70
80
1
2
3
4
5
6
7
8
9
10
11
12
month

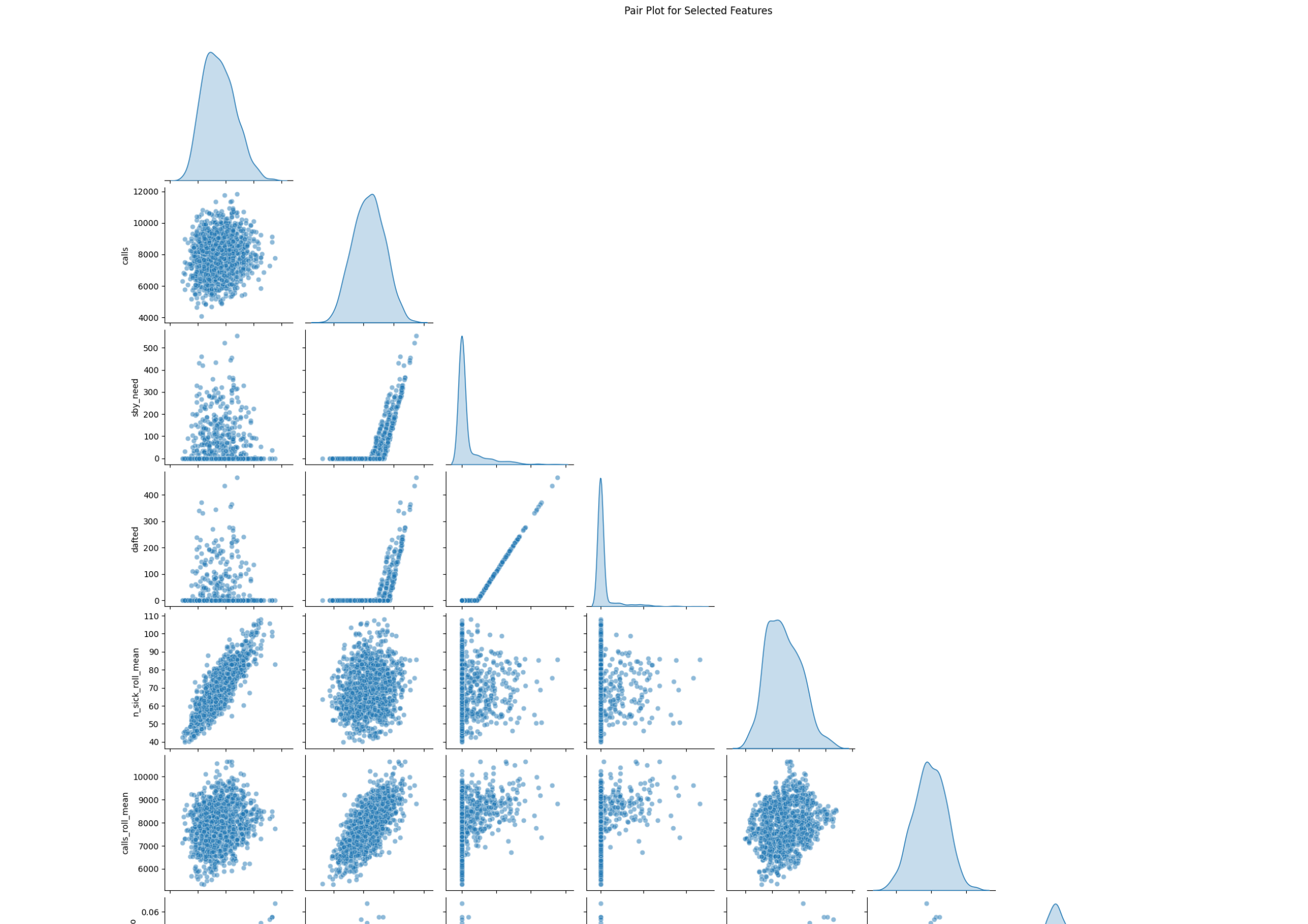

Box Plot of Rolling Mean of Drivers Sick
n_sick_roll_mean
110
100
90
80
70
60
50
40

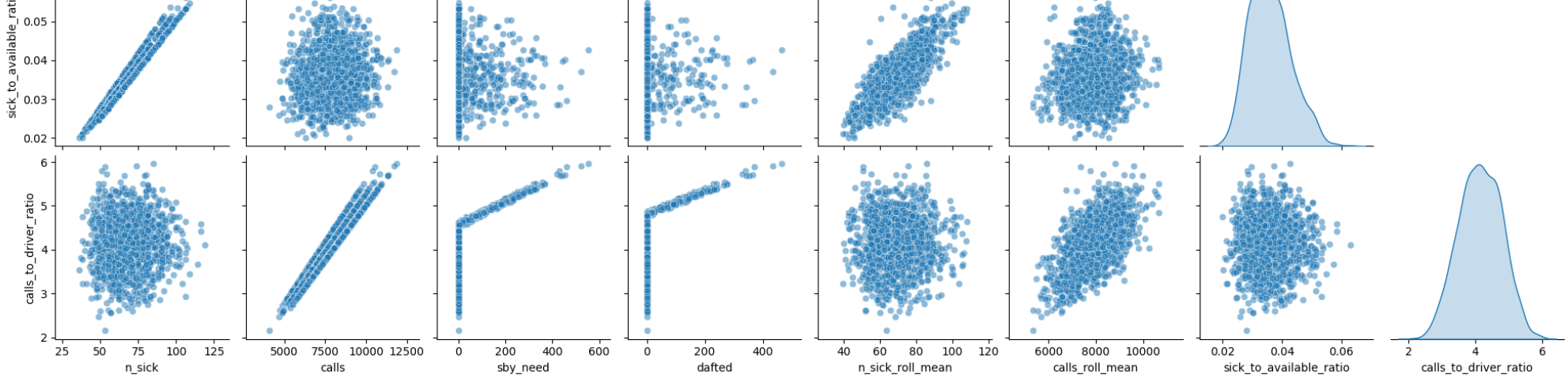

Box Plot of Difference of Emergency Calls
calls_diff
5000
4000
3000
2000
1000
0
−1000
−2000
−3000


Heatmap of Feature Correlations
n_sick
calls
n_duty
n_sby
sby_need
dafted
year
month
day
day_of_week
is_weekend
quarter
n_sick_lag1
n_sick_lag2
calls_lag1
calls_lag2
n_sick_roll_mean
n_sick_roll_std
calls_roll_mean
calls_roll_std
n_sick_diff
calls_diff
sick_to_available_ratio
calls_to_driver_ratio
sick_calls_interaction
calls_driver_interaction
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

In [ ]:

```
# Select relevant columns for pair plot
selected_columns = ['n_sick', 'calls', 'sby_need', 'dafted', 'n_sick_roll_mean',
                    'calls_roll_mean', 'sick_to_available_ratio', 'calls_to_driver_ratio']

# Generate pair plot for selected columns
pairplot_data = sickness_data[selected_columns]
sns.pairplot(pairplot_data, corner=True, diag_kind='kde', markers='o', plot_kws={'alpha': 0.5})
plt.suptitle('Pair Plot for Selected Features', y=1.02)
plt.show()
```


Pair Plot for Selected Features
calls
sby_need
dafted
n_sick_roll_mean
calls_roll_mean

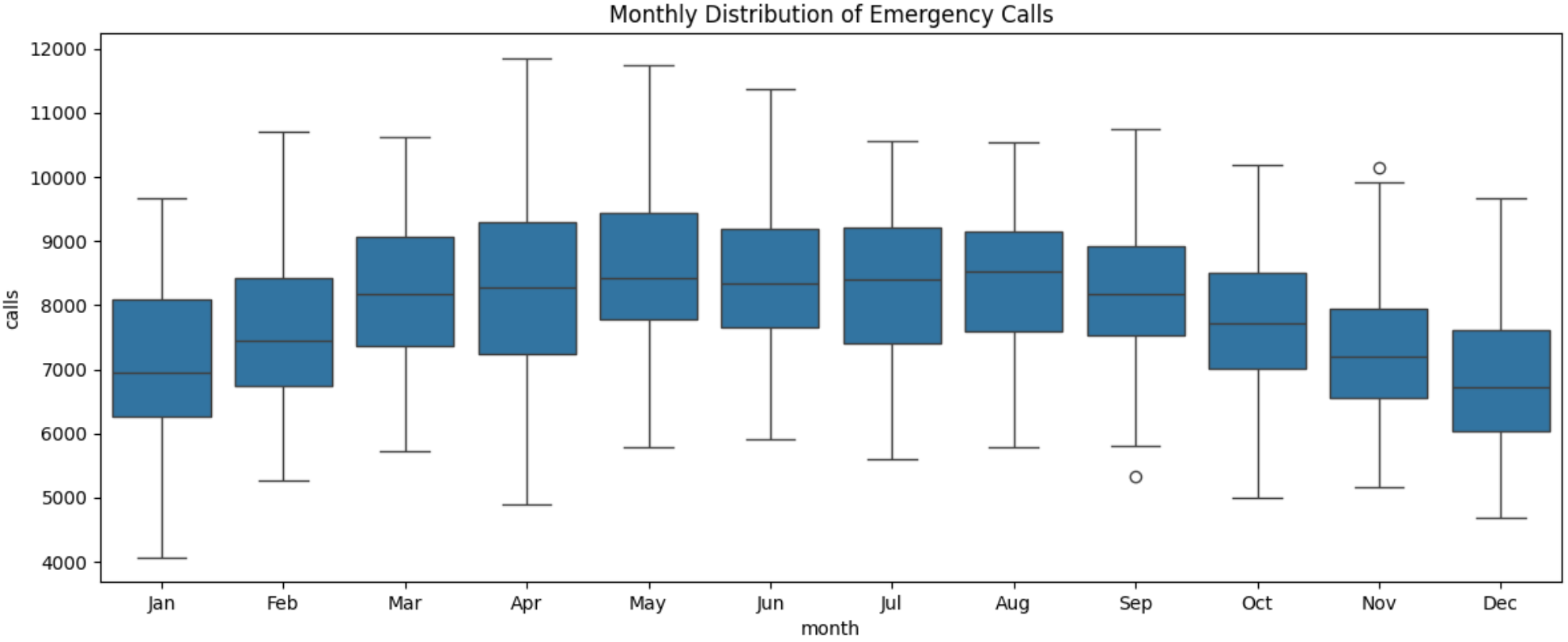


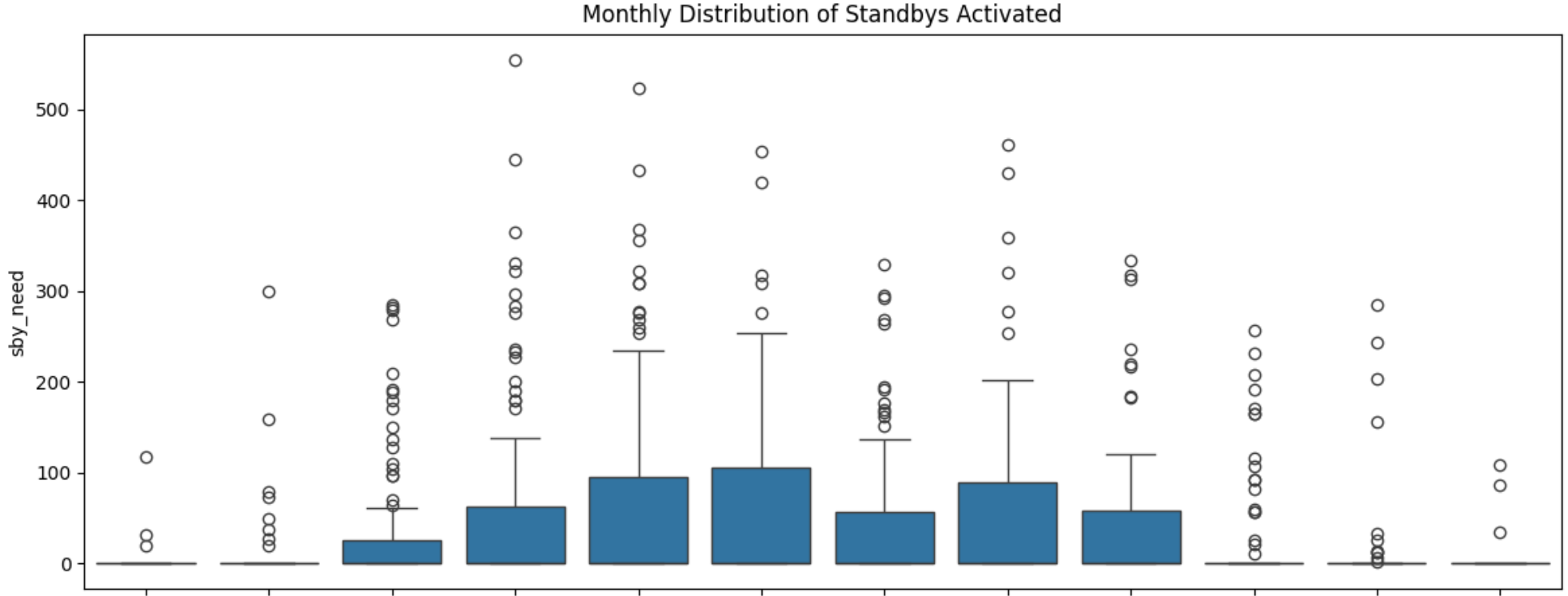

Monthly Analysis

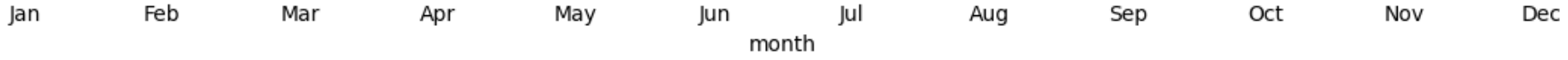
In [ ]:

```
# Set up the figure and axes
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(12, 10))

# Monthly patterns for calls
sns.boxplot(data=sickness_data, x='month', y='calls', ax=axes[0])
axes[0].set_title('Monthly Distribution of Emergency Calls')
axes[0].set_xticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])

# Monthly patterns for sby_need
sns.boxplot(data=sickness_data, x='month', y='sby_need', ax=axes[1])
axes[1].set_title('Monthly Distribution of Standbys Activated')
axes[1].set_xticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])

# Adjust the layout
plt.tight_layout()
plt.show()
```

```
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\2805215475.py:12: UserWarning: set_ticklabels() should only be used with a fixed number of t
icks, i.e. after set_ticks() or using a FixedLocator.
  axes[0].set_xticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\2805215475.py:17: UserWarning: set_ticklabels() should only be used with a fixed number of t
icks, i.e. after set_ticks() or using a FixedLocator.
  axes[1].set_xticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
```


Monthly Distribution of Emergency Calls
calls
12000
11000
10000
9000
8000
7000
6000
5000
4000
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
month

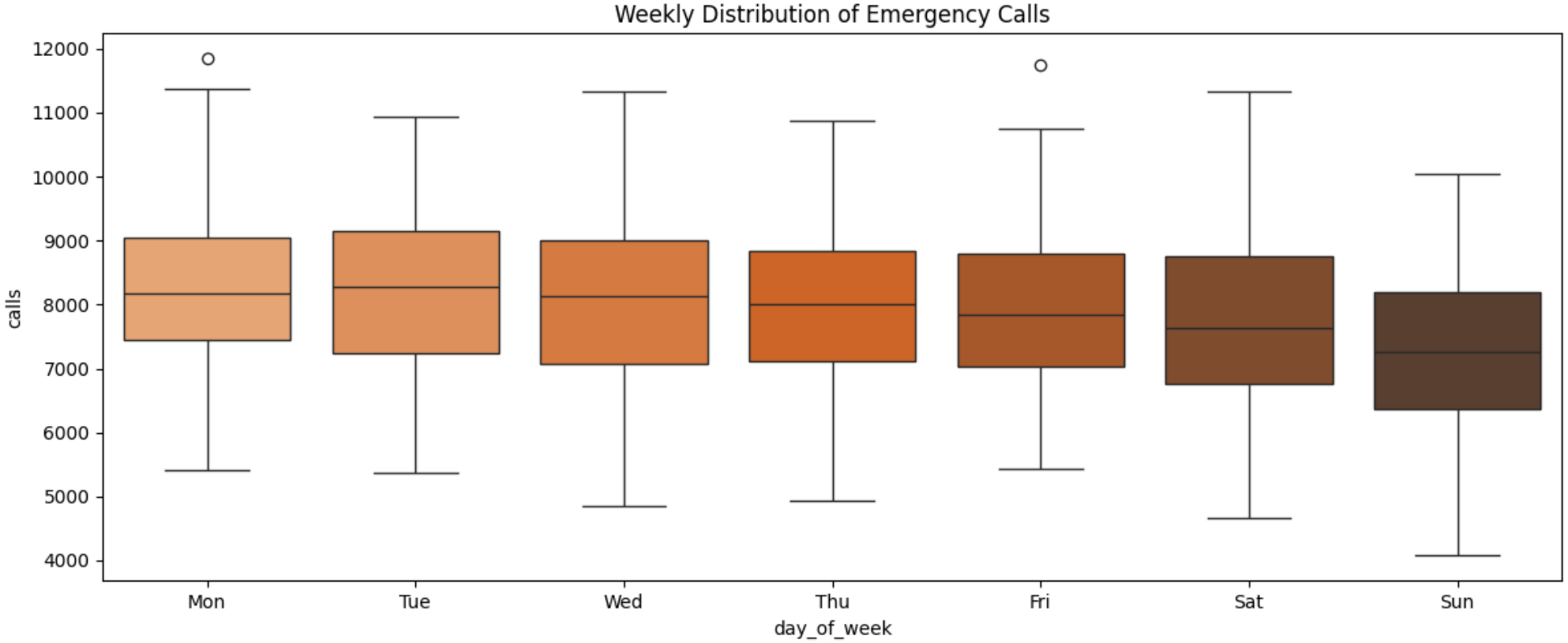

Monthly Distribution of Standbys Activated
sby_need
500
400
300
200
100
0

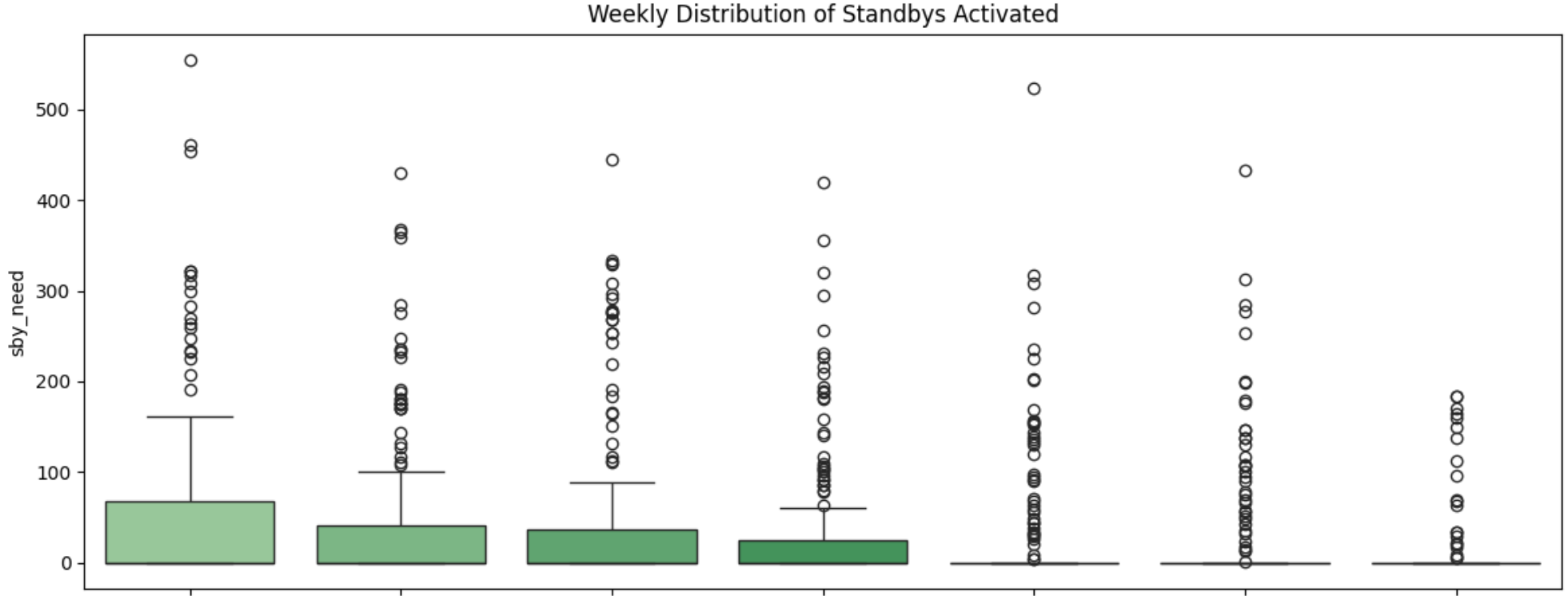



Weekly Analysis

In [ ]:
```
# Set up the figure and axes
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(12, 10))

# Weekly patterns for calls
sns.boxplot(data=sickness_data, x='day_of_week', y='calls', ax=axes[0], palette="Oranges_d")
axes[0].set_title('Weekly Distribution of Emergency Calls')
axes[0].set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])

# Weekly patterns for sby_need
sns.boxplot(data=sickness_data, x='day_of_week', y='sby_need', ax=axes[1], palette="Greens_d")
axes[1].set_title('Weekly Distribution of Standbys Activated')
axes[1].set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])

# Adjust the layout
plt.tight_layout()
plt.show()
```

```
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\1220665731.py:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=Fa
lse` for the same effect.

  sns.boxplot(data=sickness_data, x='day_of_week', y='calls', ax=axes[0], palette="Oranges_d")
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\1220665731.py:7: UserWarning: set_ticklabels() should only be used with a fixed number of ti
cks, i.e. after set_ticks() or using a FixedLocator.
  axes[0].set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\1220665731.py:10: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=Fa
lse` for the same effect.

  sns.boxplot(data=sickness_data, x='day_of_week', y='sby_need', ax=axes[1], palette="Greens_d")
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\1220665731.py:12: UserWarning: set_ticklabels() should only be used with a fixed number of t
icks, i.e. after set_ticks() or using a FixedLocator.
  axes[1].set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
```

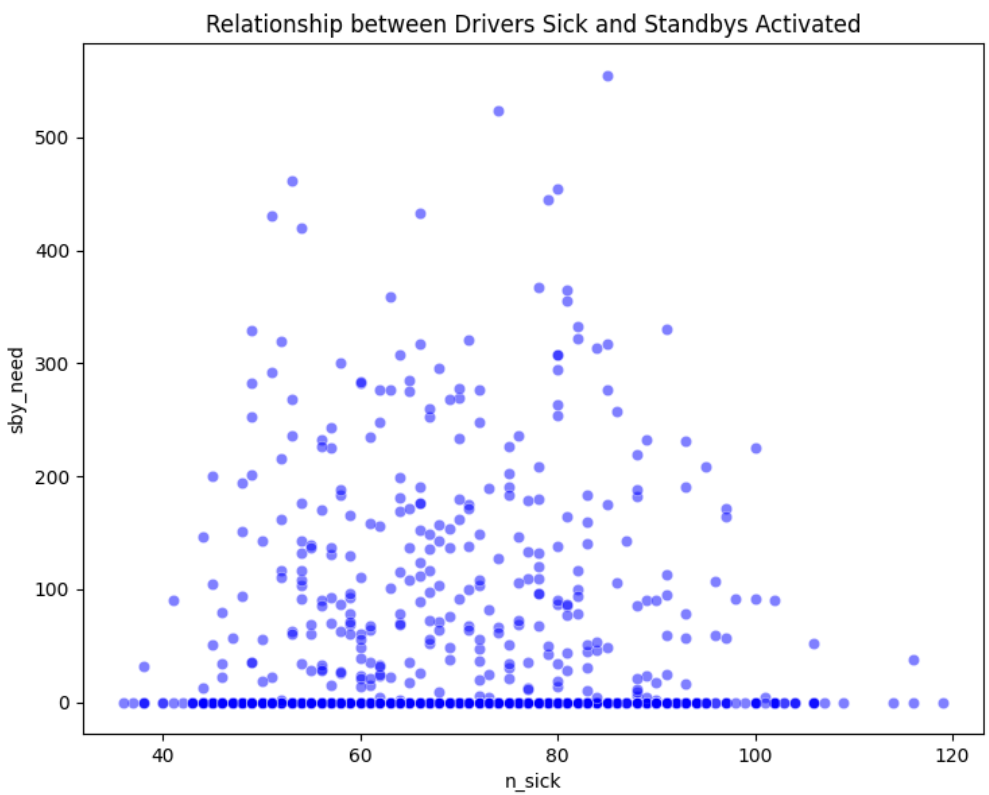

Weekly Distribution of Emergency Calls
calls
12000
11000
10000
9000
8000
7000
6000
5000
4000
Mon
Tue
Wed
Thu
Fri
Sat
Sun
day_of_week

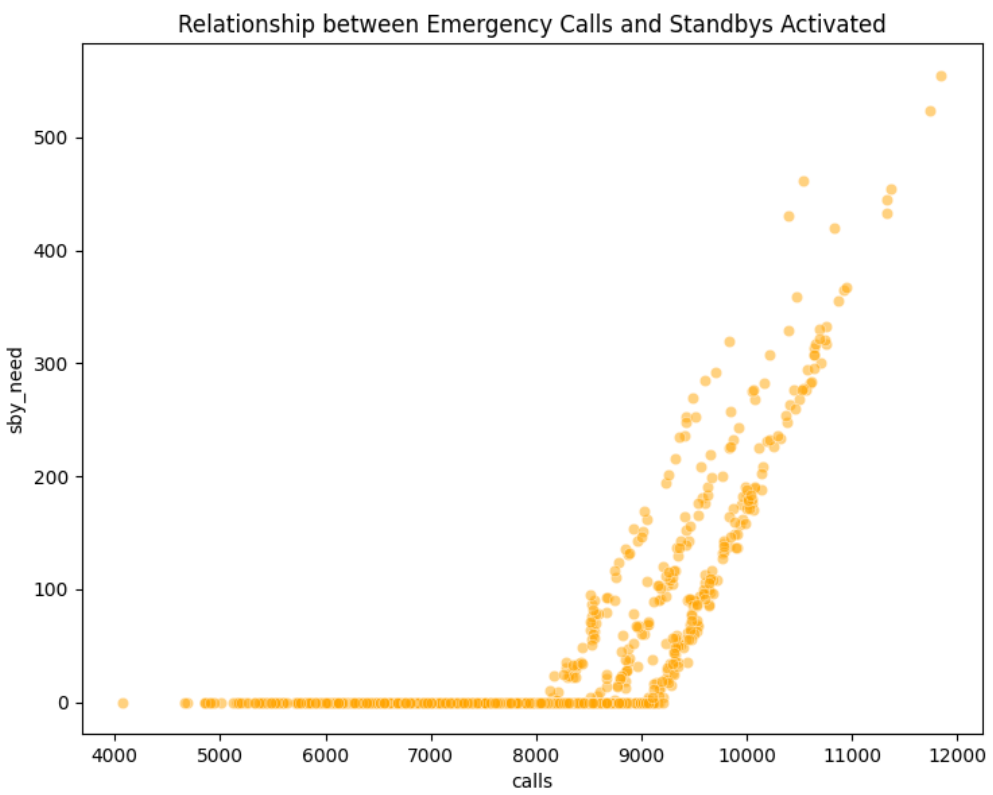

Weekly Distribution of Standbys Activated
sby_need
500
400
300
200
100
0

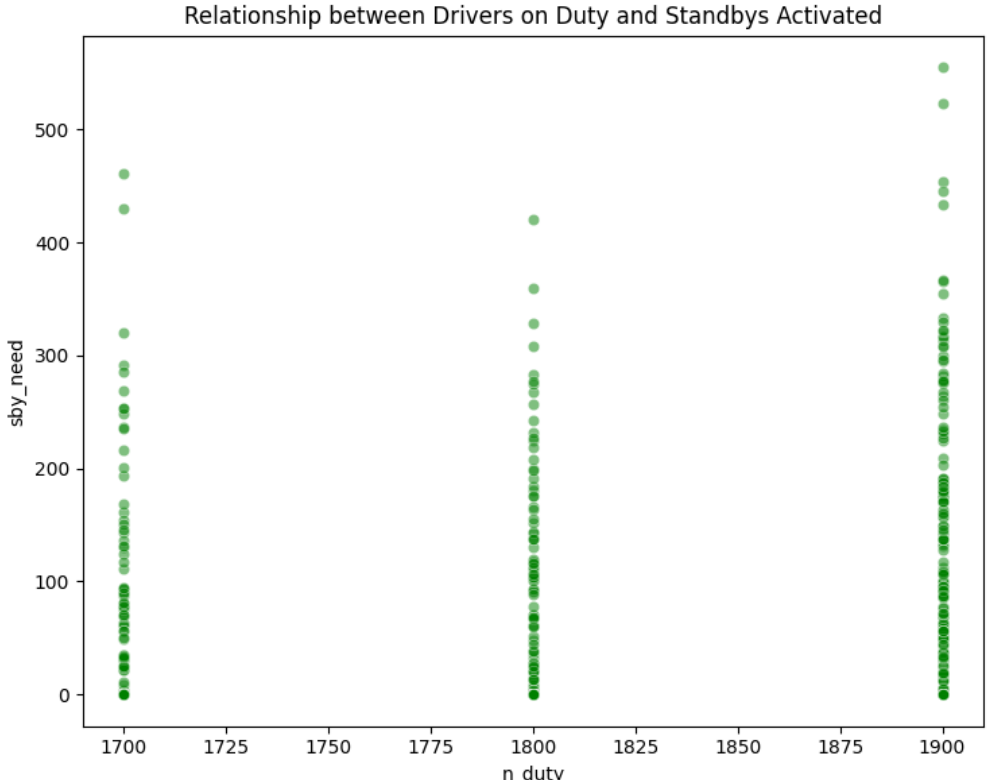
Mon Tue Wed Thu Fri Sat Sun
day_of_week

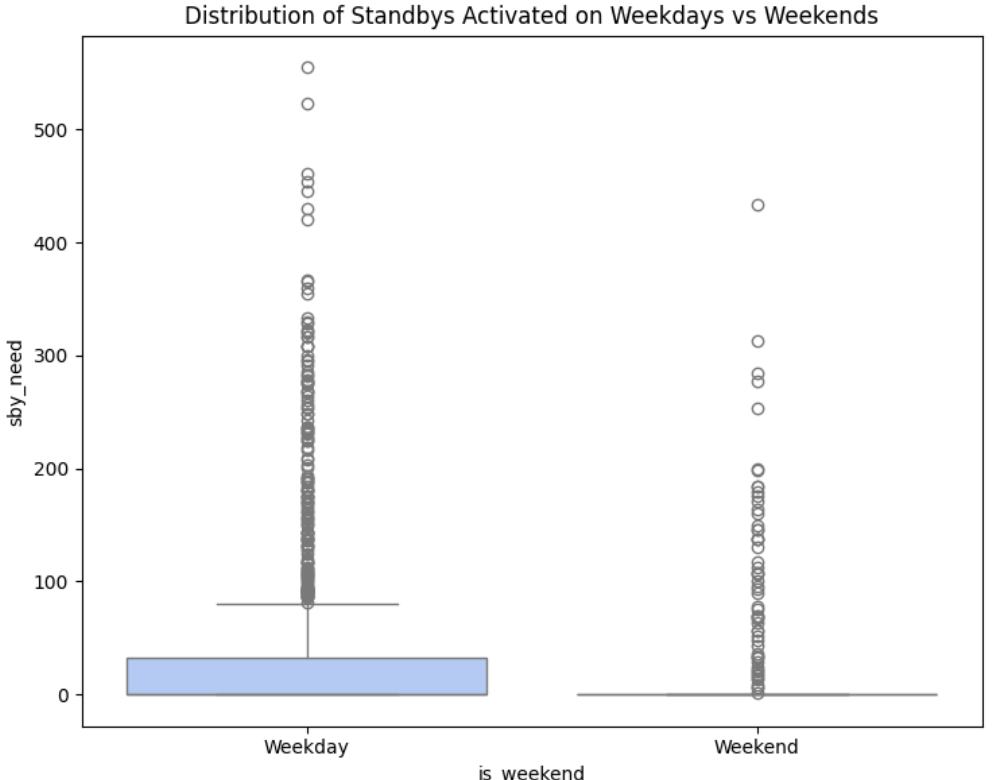
In [ ]:
```
# Set up the figure and axes
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15, 12))

# Scatter plots for relationships with 'sby_need'
sns.scatterplot(data=sickness_data, x='n_sick', y='sby_need', ax=axes[0, 0], alpha=0.5, color='blue')
axes[0, 0].set_title('Relationship between Drivers Sick and Standbys Activated')

sns.scatterplot(data=sickness_data, x='calls', y='sby_need', ax=axes[0, 1], alpha=0.5, color='orange')
axes[0, 1].set_title('Relationship between Emergency Calls and Standbys Activated')

sns.scatterplot(data=sickness_data, x='n_duty', y='sby_need', ax=axes[1, 0], alpha=0.5, color='green')
axes[1, 0].set_title('Relationship between Drivers on Duty and Standbys Activated')

# Box plots for relationship of 'is_weekend' with 'sby_need'
sns.boxplot(data=sickness_data, x='is_weekend', y='sby_need', ax=axes[1, 1], palette="coolwarm")
axes[1, 1].set_title('Distribution of Standbys Activated on Weekdays vs Weekends')
axes[1, 1].set_xticklabels(['Weekday', 'Weekend'])

# Adjust the layout
plt.tight_layout()
plt.show()
```

```
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\2660620790.py:15: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=Fa
lse` for the same effect.

  sns.boxplot(data=sickness_data, x='is_weekend', y='sby_need', ax=axes[1, 1], palette="coolwarm")
C:\Users\s9\AppData\Local\Temp\ipykernel_11968\2660620790.py:17: UserWarning: set_ticklabels() should only be used with a fixed number of t
icks, i.e. after set_ticks() or using a FixedLocator.
  axes[1, 1].set_xticklabels(['Weekday', 'Weekend'])
```


Relationship between Drivers Sick and Standbys Activated
sby_need
n_sick
0
100
200
300
400
500
40
60
80
100
120


Relationship between Emergency Calls and Standbys Activated
sby_need
calls
0
100
200
300
400
500
4000
5000
6000
7000
8000
9000
10000
11000
12000


Relationship between Drivers on Duty and Standbys Activated
sby_need
n_duty
0
100
200
300
400
500
1700
1725
1750
1775
1800
1825
1850
1875
1900


Distribution of Standbys Activated on Weekdays vs Weekends
sby_need
is_weekend
0
100
200
300
400
500
Weekday
Weekend

Selecting Features

In [ ]:
```
# Selecting features and target variable
features = sickness_data.drop(columns=['date', 'dafted', 'sby_need', 'n_sby', 'is_weekend'])
target = sickness_data['sby_need']

# Splitting the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)

# Standardizing the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # We only transform the test set based on the training set's parameters
```

Baseline Model: Linear Regression

In [ ]:
```
# Initialize the Linear Regression model
lr_model = LinearRegression()

# Train the model on the standardized training data
lr_model.fit(X_train_scaled, y_train)

# Predict on the standardized test set
y_pred = lr_model.predict(X_test_scaled)

# Evaluate the model's performance using RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(f"RMSE: {rmse}")
```

```
RMSE: 50.753284078060204
```

Random Forest Regressor

In [ ]:
```
# Initialize the Random Forest Regressor
rf_model = RandomForestRegressor(random_state=42, n_estimators=100)

# Train the model on the standardized training data
rf_model.fit(X_train_scaled, y_train)

# Predict on the standardized test set
y_pred_rf = rf_model.predict(X_test_scaled)

# Evaluate the model's performance using RMSE
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))
```

```
print(f"RMSE: {rmse_rf}")
```

```
RMSE: 4.601956952450533
```

XGBoost Regressor

In [ ]:
```
# Initialize the XGBoost regressor
xgb_regressor = xgb.XGBRegressor(objective ='reg:squarederror',
                                 n_estimators=100,
                                 max_depth=6,
                                 learning_rate=0.1,
                                 seed=42)

# Train the model on the training data
xgb_regressor.fit(X_train_scaled, y_train)

# Predict on the test data
y_pred_xgb = xgb_regressor.predict(X_test_scaled)

# Calculate the RMSE for the XGBoost model
rmse_xgb = np.sqrt(mean_squared_error(y_test, y_pred_xgb))

print(f"RMSE for XGBoost Regressor: {rmse_xgb}")
```

```
RMSE for XGBoost Regressor: 3.7470715511268384
```

Hyperparameter Tuning using RandomizedSearchCV

In [ ]:
```
# Define the hyperparameters and their possible values
param_dist_xgb = {
    'n_estimators': np.arange(50, 501, 50),
    'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.3],
    'max_depth': np.arange(3, 21, 3),
    'subsample': np.linspace(0.5, 1.0, 6),
    'colsample_bytree': np.linspace(0.5, 1.0, 6),
    'colsample_bylevel': np.linspace(0.5, 1.0, 6),
    'gamma': np.linspace(0, 0.5, 6),
    'reg_alpha': [0, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10],
    'reg_lambda': [0, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10]
}

# Initialize the XGBoost Regressor
xgb_regressor = xgb.XGBRegressor(objective='reg:squarederror', seed=42)

# Set up the RandomizedSearchCV
xgb_random_search = RandomizedSearchCV(xgb_regressor,
                                       param_distributions=param_dist_xgb,
```

```
                                   n_iter=100,
                                   cv=5,
                                   verbose=1,
                                   random_state=42,
                                   n_jobs=-1)

# Fit the model
xgb_random_search.fit(X_train_scaled, y_train)

best_hyperparameters_xgb = xgb_random_search.best_params_
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
```

In [ ]:
```
print(best_hyperparameters_xgb)
```

```
{'subsample': 1.0, 'reg_lambda': 0.001, 'reg_alpha': 0.1, 'n_estimators': 450, 'max_depth': 3, 'learning_rate': 0.3, 'gamma': 0.1, 'colsamp
le_bytree': 0.9, 'colsample_bylevel': 1.0}
```

Optimised XGBoost Model

In [ ]:
```
# Using the provided best hyperparameters to configure the XGBoost model
optimized_xgb_regressor = xgb.XGBRegressor(
    subsample=0.7,
    reg_lambda=0,
    reg_alpha=1,
    n_estimators=150,
    max_depth=3,
    learning_rate=0.1,
    gamma=0.4,
    colsample_bytree=0.8,
    colsample_bylevel=1.0,
    objective='reg:squarederror',
    seed=42
)

# Train the optimized model on the training data
optimized_xgb_regressor.fit(X_train_scaled, y_train)

# Predict on the test data
y_pred_optimized_xgb = optimized_xgb_regressor.predict(X_test_scaled)

# Calculate the RMSE for the optimized XGBoost model
rmse_optimized_xgb = np.sqrt(mean_squared_error(y_test, y_pred_optimized_xgb))

print(f"RMSE for Optimized XGBoost Regressor: {rmse_optimized_xgb}")

# MAE
mae_optimized_xgb = mean_absolute_error(y_test, y_pred_optimized_xgb)
```

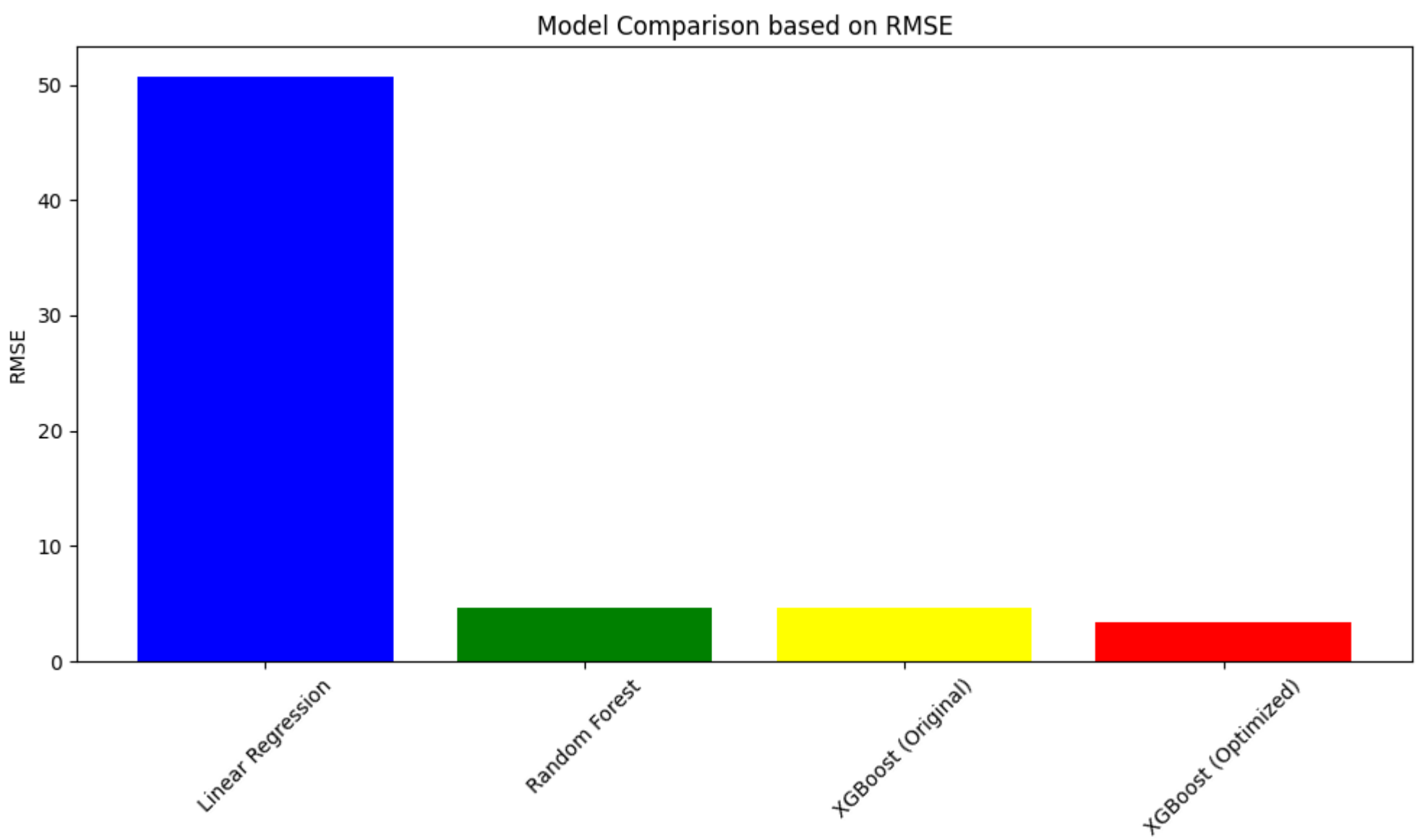
```
print(f"MAE for Optimized XGBoost Model: {mae_optimized_xgb:.4f}")
```

```
RMSE for Optimized XGBoost Regressor: 3.46577464999141
MAE for Optimized XGBoost Model: 1.5530
```

Model Comparison

In [ ]:

```
# Train the Linear Regression model
lr_model.fit(X_train_scaled, y_train)

# Train the Random Forest Regressor
rf_model.fit(X_train_scaled, y_train)

# Train the Original XGBoost Model
xgb_regressor.fit(X_train_scaled, y_train)

# Train the Optimized XGBoost Model with the provided hyperparameters
optimized_xgb_regressor.fit(X_train_scaled, y_train)

#Predict on the test set using each model
y_pred_lr = lr_model.predict(X_test_scaled)  # Baseline Model (Linear Regression)
y_pred_rf = rf_model.predict(X_test_scaled)  # Random Forest Regressor
y_pred_xgb_original = xgb_regressor.predict(X_test_scaled)  # Original XGBoost Model
y_pred_xgb_optimized = optimized_xgb_regressor.predict(X_test_scaled)  # Optimized XGBoost Model

#Calculate the RMSE for each model's predictions
rmse_values = {
    'Linear Regression': np.sqrt(mean_squared_error(y_test, y_pred_lr)),
    'Random Forest': np.sqrt(mean_squared_error(y_test, y_pred_rf)),
    'XGBoost (Original)': np.sqrt(mean_squared_error(y_test, y_pred_xgb_original)),
    'XGBoost (Optimized)': np.sqrt(mean_squared_error(y_test, y_pred_xgb_optimized))
}

#Visualize the RMSE values using a bar chart
plt.figure(figsize=(10, 6))
plt.bar(rmse_values.keys(), rmse_values.values(), color=['blue', 'green', 'yellow', 'red'])
plt.ylabel('RMSE')
plt.title('Model Comparison based on RMSE')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

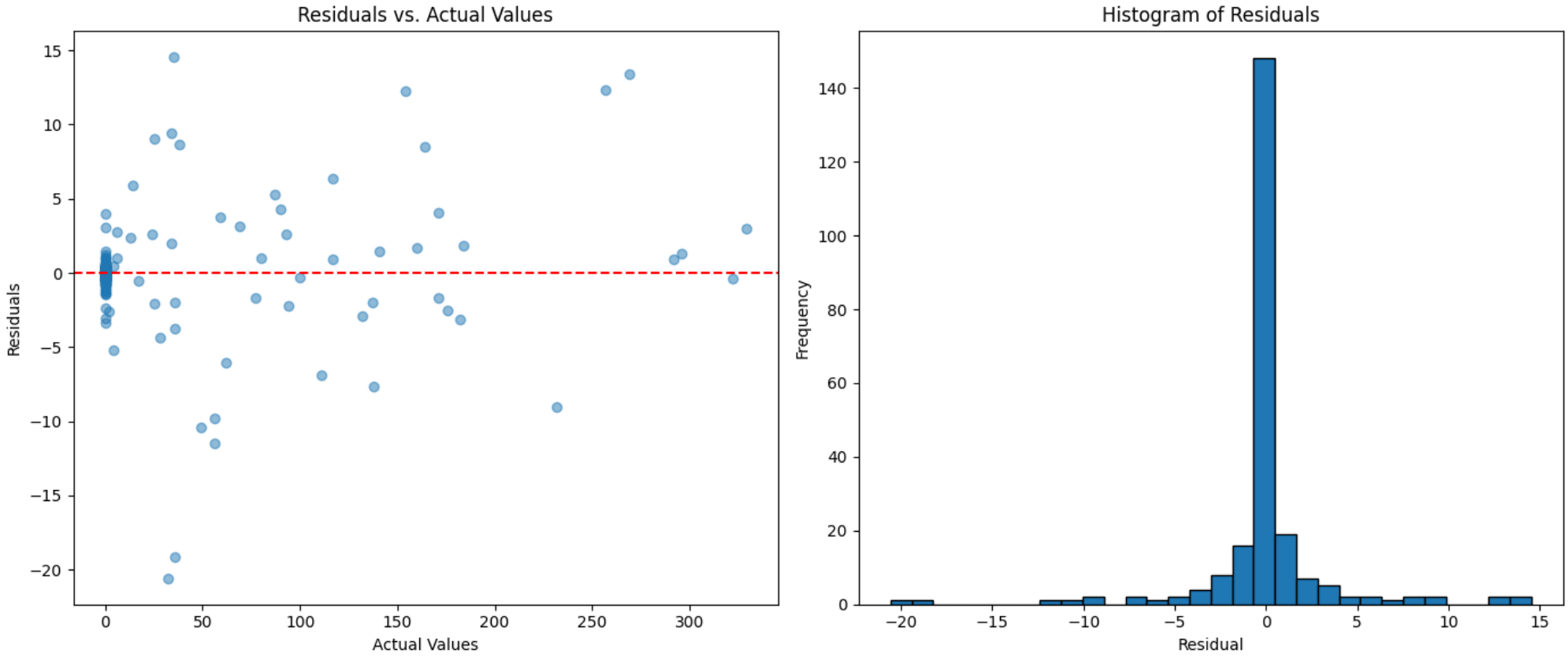



Error Analysis

In [ ]:

```
# Calculate residuals for the optimized XGBoost model
residuals_xgb_optimized = y_test - y_pred_xgb_optimized

# Plotting the residuals
plt.figure(figsize=(14, 6))

# Residual scatter plot
plt.subplot(1, 2, 1)
```

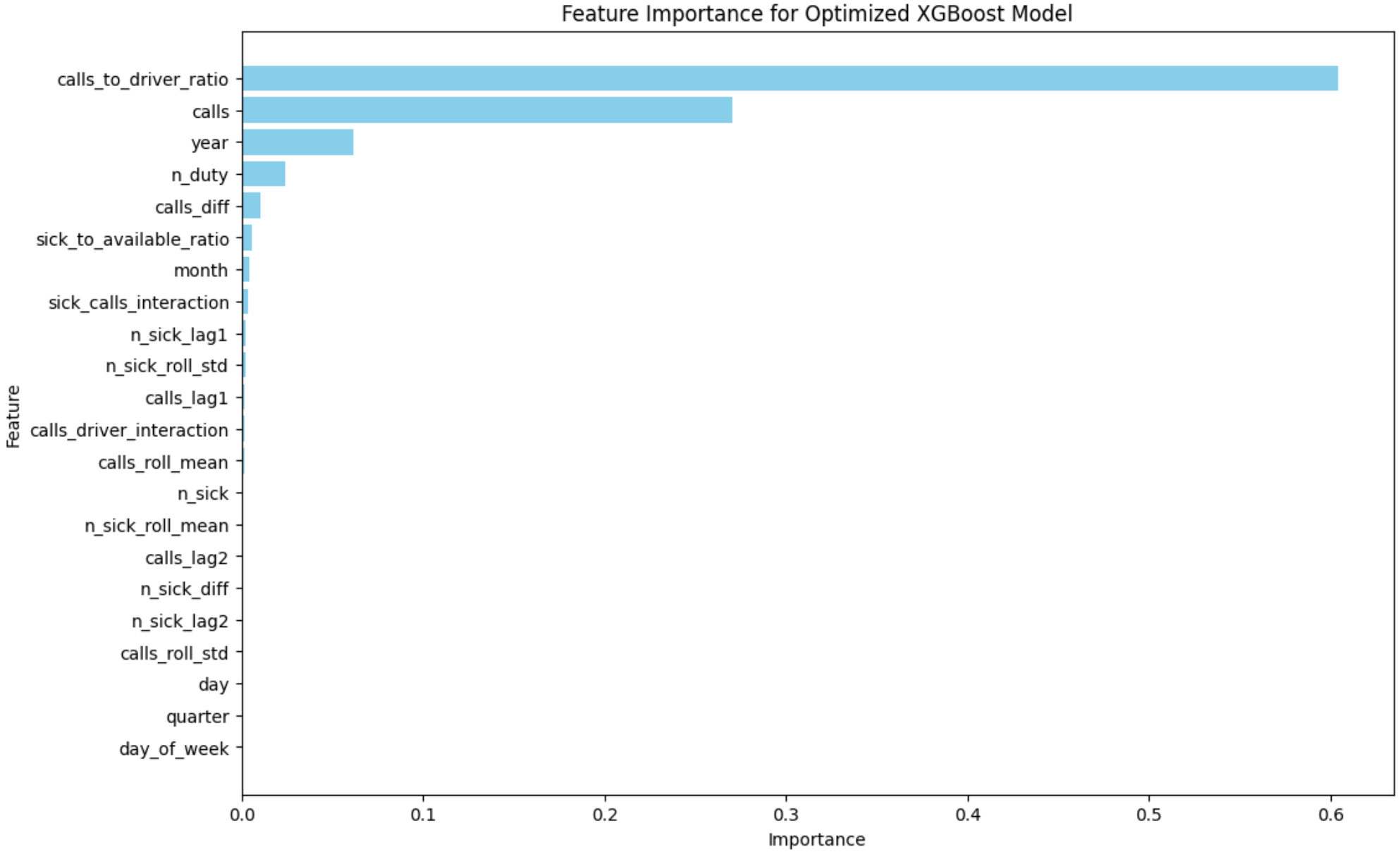
```
plt.scatter(y_test, residuals_xgb_optimized, alpha=0.5)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Actual Values')
plt.ylabel('Residuals')
plt.title('Residuals vs. Actual Values')

# Histogram of residuals
plt.subplot(1, 2, 2)
plt.hist(residuals_xgb_optimized, bins=30, edgecolor='black')
plt.xlabel('Residual')
plt.ylabel('Frequency')
plt.title('Histogram of Residuals')

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




Feature Importance

In [ ]:
```
# Extract feature importance from the optimized XGBoost model
feature_importance_optimized = optimized_xgb_regressor.feature_importances_
```

```python
# Create a DataFrame for visualization
features_df = pd.DataFrame({
    'Feature': features.columns,
    'Importance': feature_importance_optimized
})

# Sort the features based on importance
features_df = features_df.sort_values(by='Importance', ascending=False)

# Plotting the feature importance
plt.figure(figsize=(12, 8))
plt.barh(features_df['Feature'], features_df['Importance'], color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance for Optimized XGBoost Model')
plt.gca().invert_yaxis()  # To display the most important feature at the top
plt.show()
```

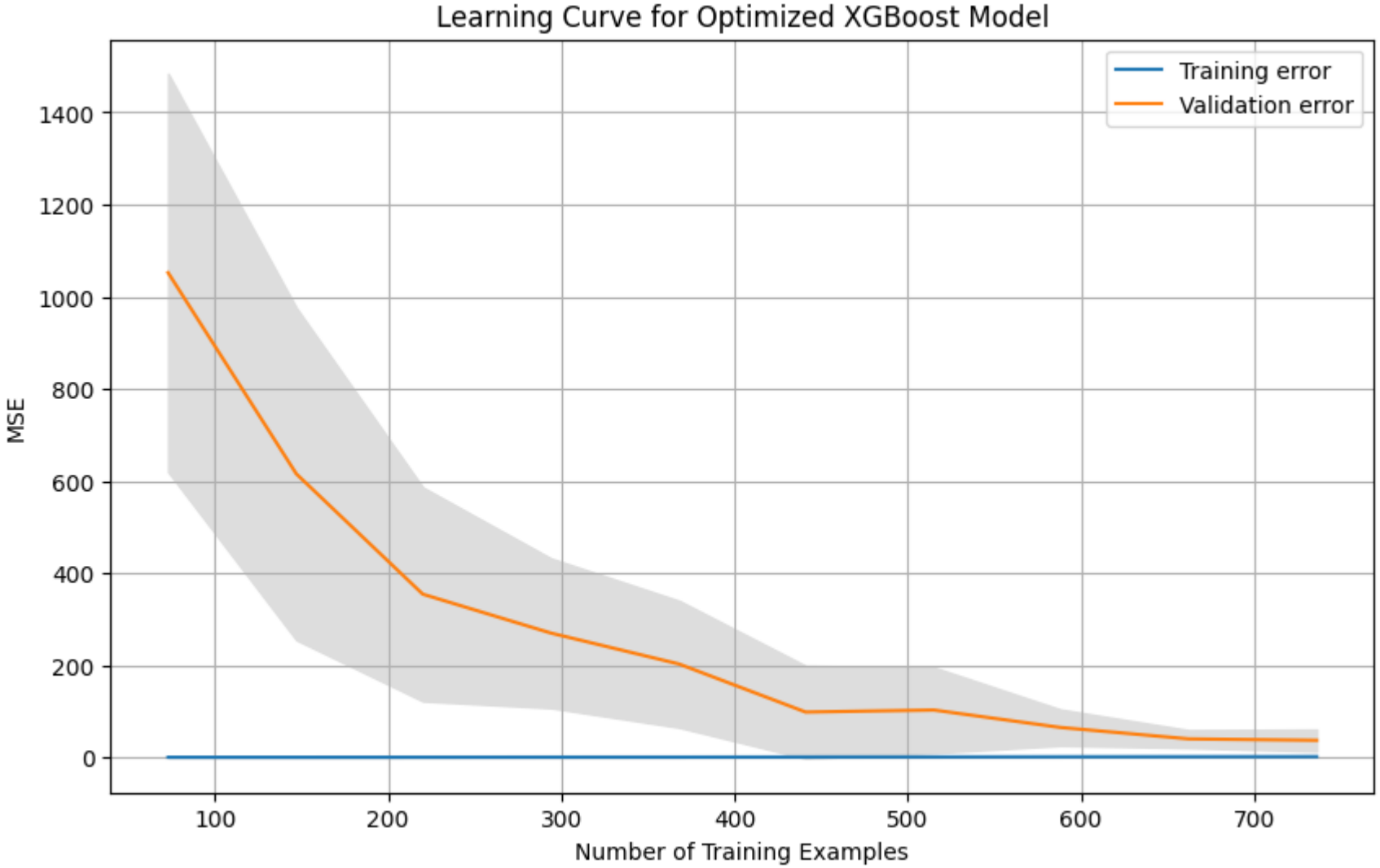



Learning Curve Analysis

In [ ]:

```
def plot_learning_curve(model, X, y):
    train_sizes, train_scores, test_scores = learning_curve(model, X, y, cv=5, scoring="neg_mean_squared_error",
                                                            train_sizes=np.linspace(0.1, 1.0, 10))

    train_mean = np.mean(-train_scores, axis=1)
    train_std = np.std(-train_scores, axis=1)
    test_mean = np.mean(-test_scores, axis=1)
```

```
    test_std = np.std(-test_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_mean, label="Training error")
    plt.plot(train_sizes, test_mean, label="Validation error")
    plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color="#DDDDDD")
    plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color="#DDDDDD")
    plt.title("Learning Curve for Optimized XGBoost Model")
    plt.xlabel("Number of Training Examples")
    plt.ylabel("MSE")
    plt.legend(loc="best")
    plt.grid()
    plt.show()

plot_learning_curve(optimized_xgb_regressor, X_train_scaled, y_train)
```


Learning Curve for Optimized XGBoost Model
Training error
Validation error
MSE
0
200
400
600
800
1000
1200
1400
100
200
300
400
500
600
700
Number of Training Examples

In [ ]: