AUTOMATION OF STANDBY DUTY PLANNING FOR RESCUE DRIVERS VIA A FORECASTING MODEL

Case Study Model Engineering - DLMDSME01

Name – Muskan

Matriculation Number - 321150201

Course of Study – Master's Data Science

Tutor's Name – Professor Sahar Qaadan

Date of Submission – 10/29/2023

Table of Contents

Project Aim:

- Develop a predictive model to help the HR department estimate the number of daily standby rescue drivers needed.
- The new model should have a higher percentage of standbys being activated than the current approach.
- Situations where there aren't enough standby drivers should be minimized.

Git Repository Structure:

```
project-name/
│
├── .gitignore              # List of files and folders to be ignored by Git
│
├── README.md               # Overview of the project, setup instructions, and other documentation
│
├── data/                   # Folder containing datasets
│   ├── raw/                # Raw, unprocessed data
│   ├── processed/          # Cleaned and processed data ready for modeling
│   └── external/           # External data sources, if any
│
├── notebooks/              # Jupyter notebooks for analysis, experimentation, and prototyping
│   ├── exploratory/        # Initial EDA notebooks
│   └── report/             # Finalized notebooks for reporting
│
├── src/                    # Source code for the project
│   ├── __init__.py         # Makes the folder a Python package
│   ├── data_processing/    # Scripts/modules for data cleaning and processing
│   ├── model/              # Scripts/modules for model training, evaluation, etc.
│   └── utils/              # Utility scripts/modules
│
├── models/                 # Trained model files and model architectures
│
```

```
├── reports/                # Generated analysis as HTML, PDF, PowerPoint, etc.
│   └── figures/            # Generated graphics and figures to be used in reporting
│
├── tests/                  # Unit tests and testing scripts
│   ├── __init__.py         # Makes the folder a Python package
│   └── test_sample.py      # Sample testing script
│
└── requirements.txt        # Required libraries and dependencies for the project
```

1. .gitignore: This file instructs Git to exclude certain files or directories. Typically, you'd exclude large datasets, environment folders, or other files that shouldn't be versioned.
2. README.md: Provides a project overview, setup instructions, contributions, and other essential details. This is usually the first file someone checks when they encounter a new repository.
3. data/: Organizing data into raw and processed subfolders helps differentiate between the original datasets and the cleaned or manipulated versions.
4. notebooks/: Keeping Jupyter notebooks separate allows for easy prototyping and experimentation. Splitting them into exploratory and report subfolders helps differentiate between initial analysis and final report notebooks.
5. src/: All the Python scripts and modules related to data processing, modeling, utilities, etc., are kept here.
6. models/: Any trained models, serialized objects, or model architectures can be saved in this directory.
7. reports/: Contains any generated reports or analysis outputs. This can be particularly helpful when sharing findings with stakeholders.
8. tests/: Contains unit tests to ensure the integrity and correctness of the code. It's a best practice to write tests for your functions and classes to catch any regressions or bugs.
9. requirements.txt: This file lists all the Python library dependencies required for the project. It ensures that anyone cloning the repo can easily recreate the necessary environment.

Understanding the Dataset:

Columns:

- date: The specific day for which the data is recorded.
- n_sick: The number of rescue drivers who reported sick on that day.
- calls: The number of emergency calls received on that day.
- n_duty: The total number of rescue drivers who were available and on duty for that day.
- n_sby: The number of drivers kept on standby for that day.
- sby_need: The actual number of standby drivers who were called into action on that day.
- dafted: The number of additional drivers that were needed beyond the available standby drivers because there weren't enough standbys.

After checking the info() of the dataset, following can be interpreted:

- We have 1152 rows and 7 columns of data.
- There are no null values.

Summary Statistics:

Number of Sick Drivers (n_sick): The dataset reveals an average of approximately 69 drivers reporting sick daily.

The count of sick drivers can vary, with the lowest reported being 36 and the highest reaching up to 119.

The standard deviation of around 14.29 indicates that the number of sick drivers on a given day typically deviates from the mean by about 14 drivers.

Number of Emergency Calls (calls): On average, the service receives around 7,919.53 emergency calls daily.

The volume of calls can swing between a minimum of 4,074 to a peak of 11,850.

The standard deviation stands at approximately 1,290.06, suggesting a moderate day-to-day variation in the number of calls.

Number of Drivers on Duty (n_duty): The average number of drivers on duty each day is approximately 1,820.57.

This count ranges between 1,700 to 1,900 drivers.

The standard deviation of around 80.09 indicates a relatively stable number of drivers on duty across days.

Number of Standby Drivers (n_sby): In line with the current planning approach, there's a constant of 90 standby drivers available every day.

Number of Standby Drivers Activated (sby_need): On average, about 34.72 standby drivers are activated daily.

The range is broad, with some days requiring no standby drivers, while on more demanding days, as many as 555 standby drivers might be activated.

Number of Additional Drivers Needed (dafted): The dataset reports an average requirement of approximately 16.34 additional drivers daily, beyond the available standby drivers.

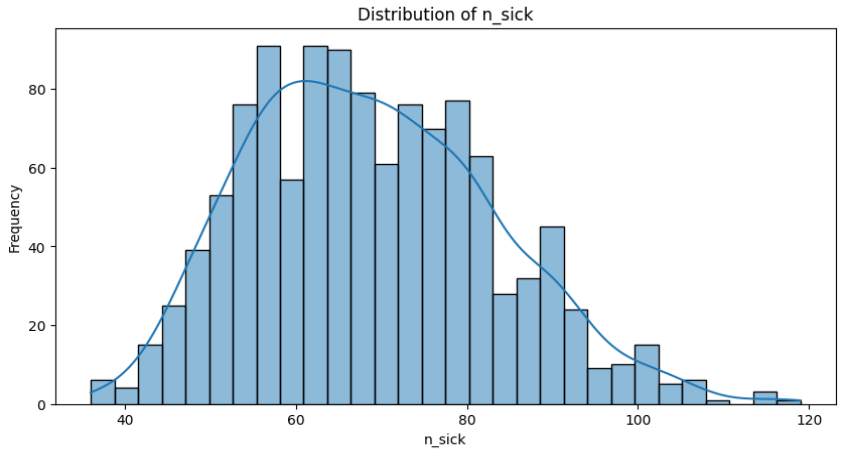
The need can vary drastically, with many days not requiring any additional drivers, while on certain challenging days, the demand can surge to as many as 465 additional drivers.

The standard deviation of 53.39 reflects the considerable variability in this measure.

Initial Data Analysis

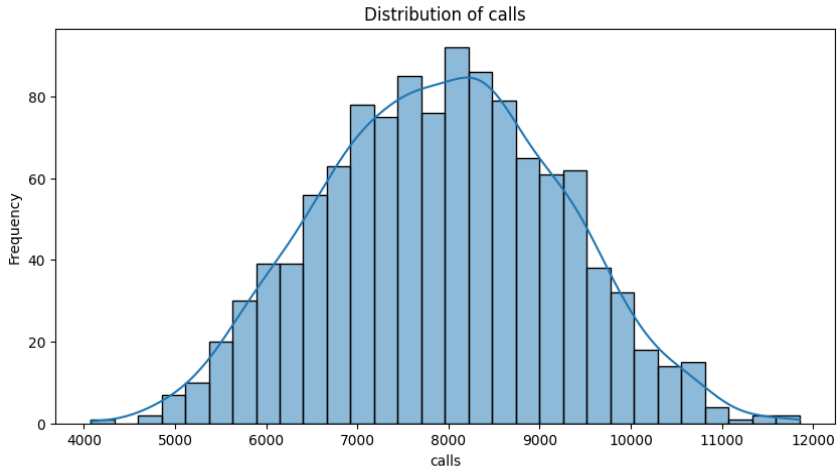
Distribution Analysis

1. **Dis**tribution of n_sick-



- The dataset unveils that the majority of days witness a count of sick drivers hovering between 60 and 80.
- The distribution of sick drivers presents a roughly normal shape. However, a closer observation hints at a slight rightward skew, indicating occasional days with a higher-than-average number of sick drivers.

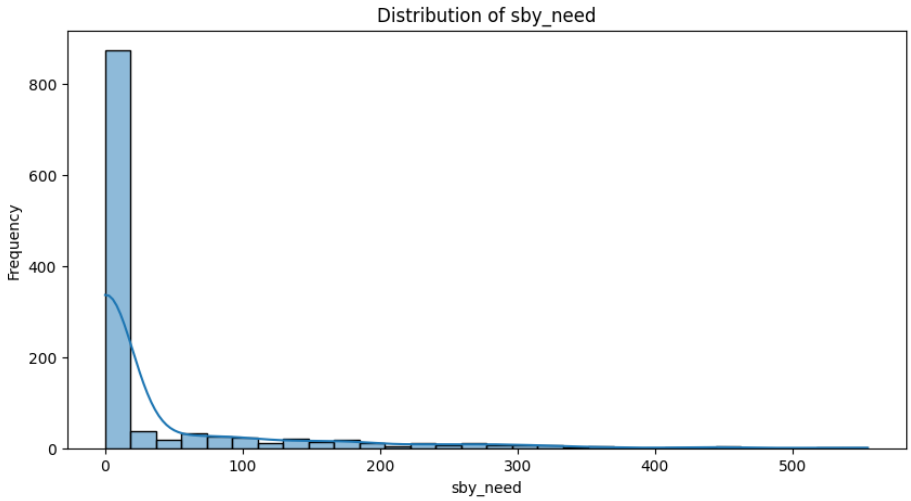
2. Distribution of calls-



The service's daily emergency call volume predominantly oscillates around 7,500 to 8,500 calls.

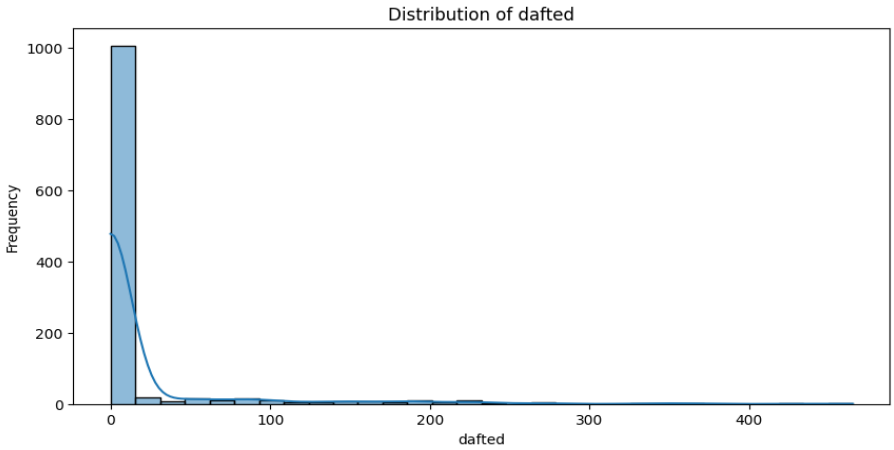
The distribution is a bell-shaped curve, characteristic of a normal distribution, underscoring a consistent pattern in emergency calls with minor day-to-day fluctuations.

3. Distribution of sby_need-



- A significant portion of days necessitates the activation of only a minimal number of standby drivers, if any. This is evident from the pronounced peak at the distribution's onset.
- Yet, the distribution unveils a long tail pointing towards days with exceptionally high standby driver activation. While these days are rarer in occurrence, their existence underscores the unpredictability of demands and the critical need for efficient HR planning.

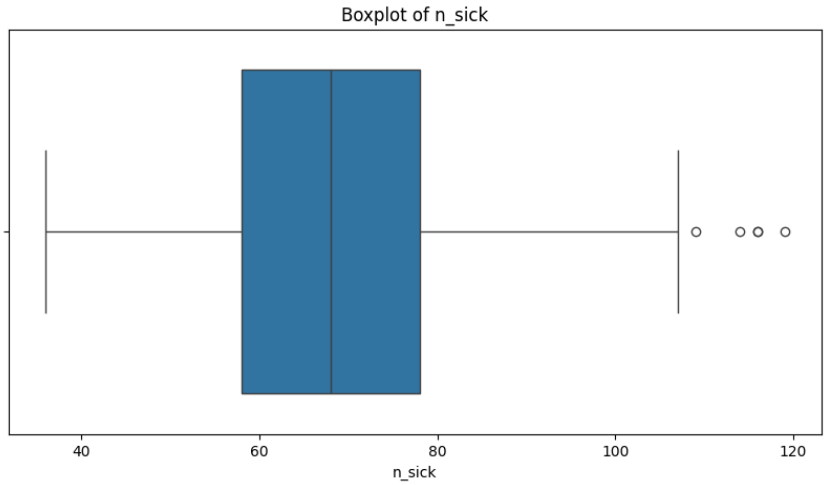
4. Distribution of dafted-



- The prevailing trend indicates that most days are well-managed with the available pool of standby drivers, negating the need for any additional drivers.
- However, the distribution also brings to light occasional days marked by a pronounced demand for additional drivers, over and above the standby pool. These outlier days, although infrequent, represent potential challenges and stresses the importance of a more dynamic and predictive allocation of resources.

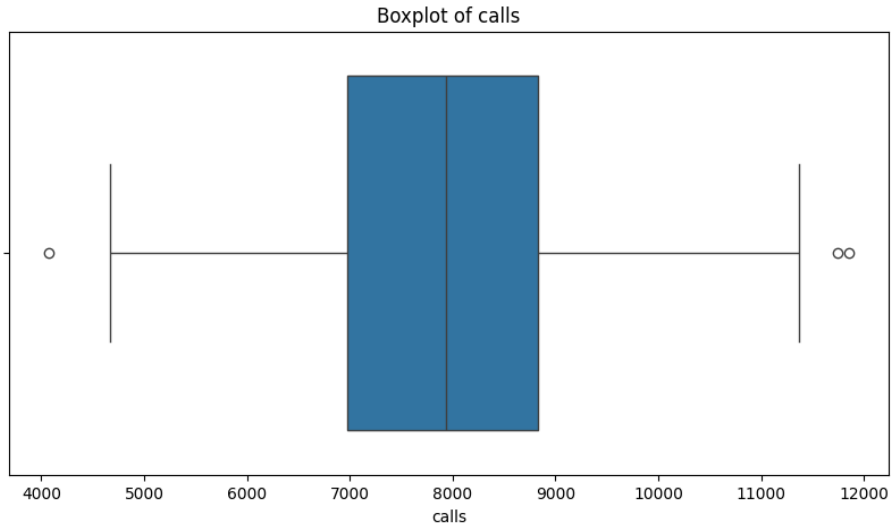
Checking for Outliers:

Outliers in Sick Drivers (n_sick) –



- Most of the data points for sick drivers are clustered around the median, with a few outliers visible on the higher end. These represent days with an unusually high number of sick drivers.

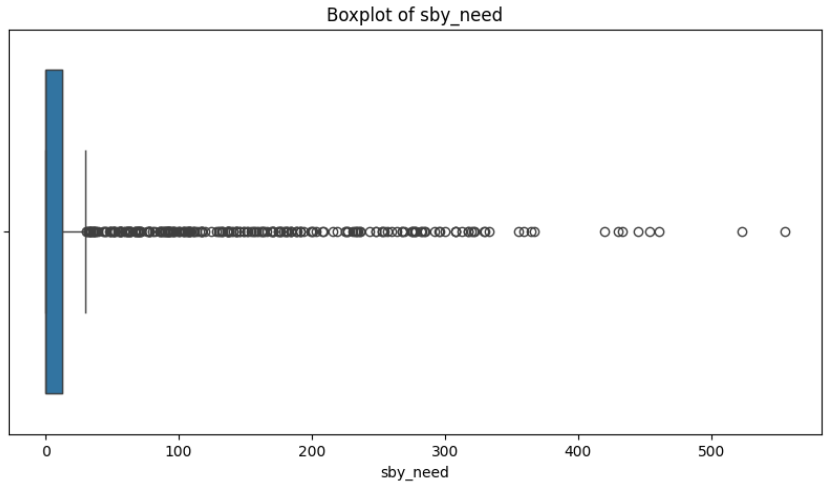
Outliers in Emergency Calls (calls) –



- 3 outliers were detected.

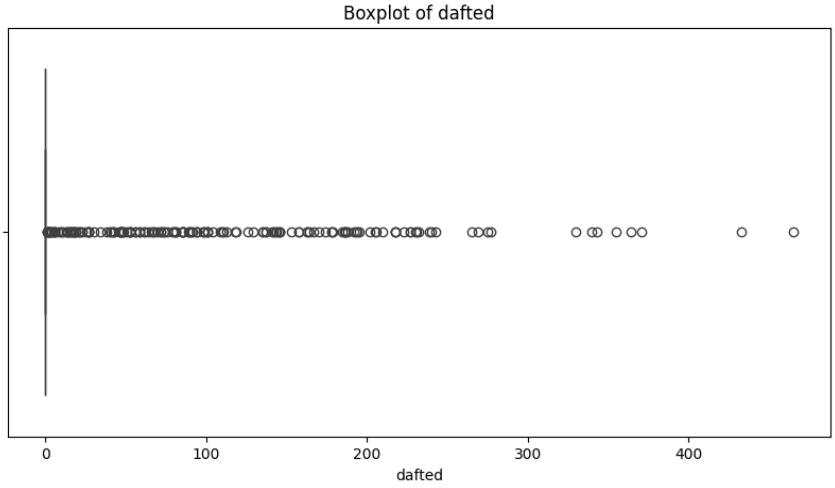
- The majority of data points for emergency calls are closely packed, with a few outliers on both the lower and higher ends.

Outliers in Standby Drivers Activated (sby_need) –



- The Number of Standby Drivers Activated has a considerable number of outliers, with 256 days falling outside the typical range. This indicates the inherent unpredictability in the demand for standby drivers.
- This distribution has a clear right-skew, with a considerable number of outliers on the higher side. These represent days when an unusually high number of standby drivers were activated.

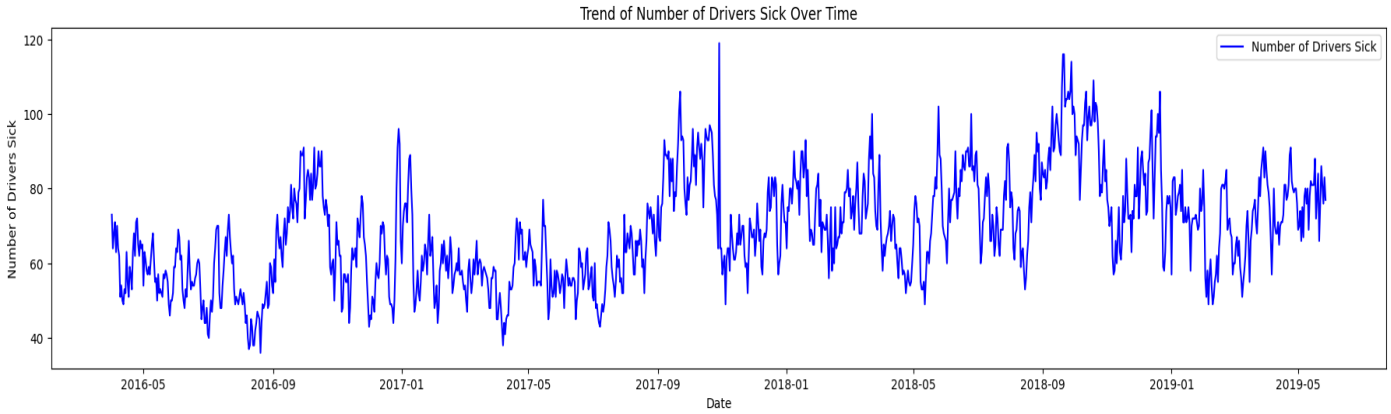
Outliers in Additional Drivers Needed (dafted) –



- 171 outlier days were identified.
- A significant portion of the data points are clustered around zero, indicating that on many days, no additional drivers were needed. However, there are several outliers on the higher end, showcasing days with a high demand for additional drivers beyond the standby pool.

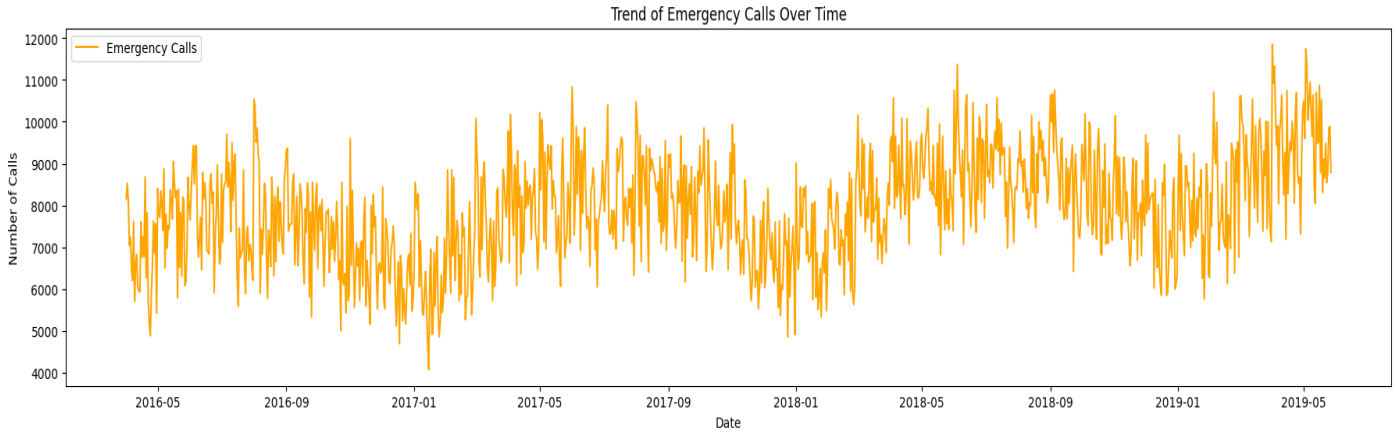
Time-Series Analysis

Number of drivers sick over time



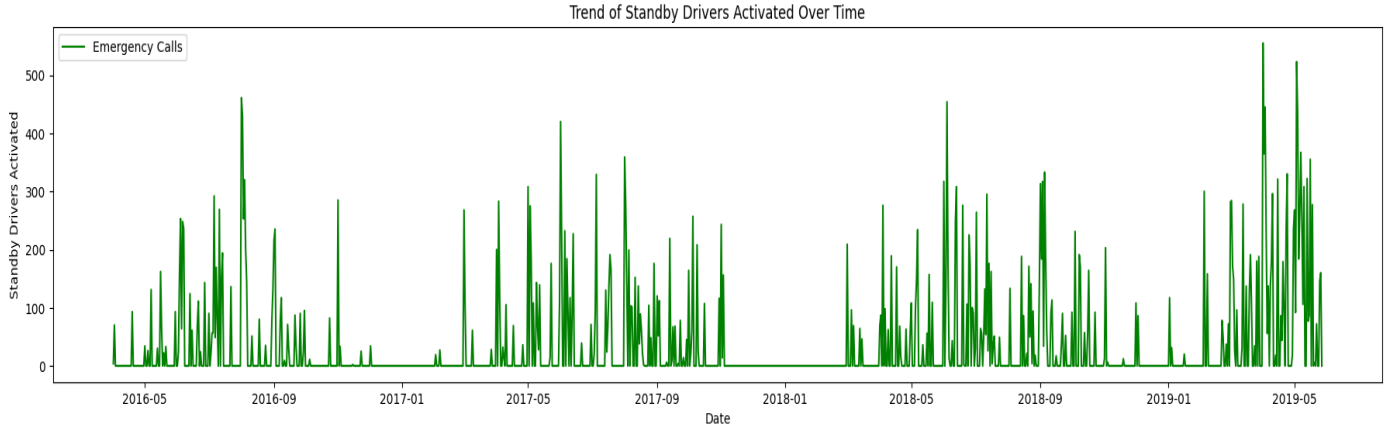
- There seems to be a recurring pattern or seasonality in the number of drivers calling in sick. This is evident from the peaks and troughs observed in the graph.
- We can see that more people are falling sick in between the months of September to January. This can be because of the cold weather so we need to keep in mind this insight while prediction.

Trend of Emergency Calls over time



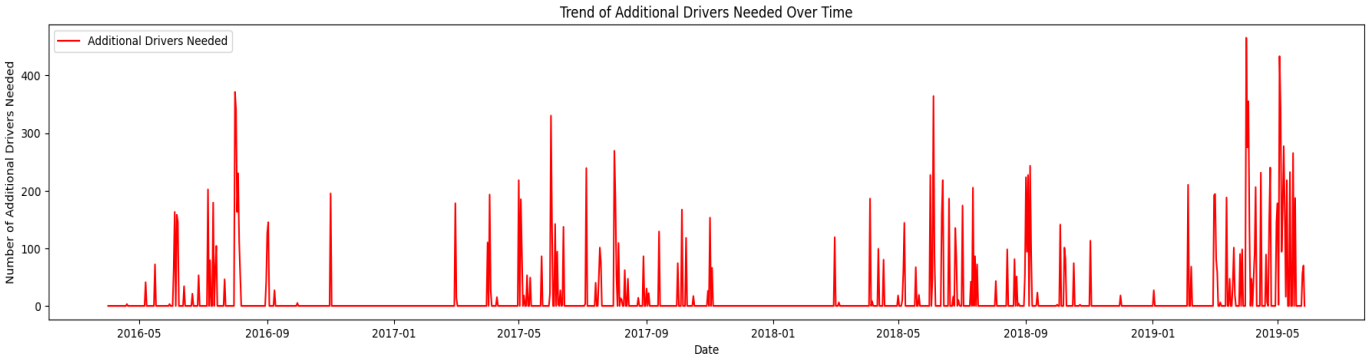
- The number of emergency calls also displays a seasonal pattern.
- Additionally, there appears to be a slight upward trend in the number of emergency calls over time, indicating that the volume of calls might be increasing. This suggests that certain times of the year may consistently require more standby drivers.
- The slight upward trend in emergency calls implies that the demand for rescue services is growing. This trend should be factored into future planning.

Trend of Standby Drivers Activated over time



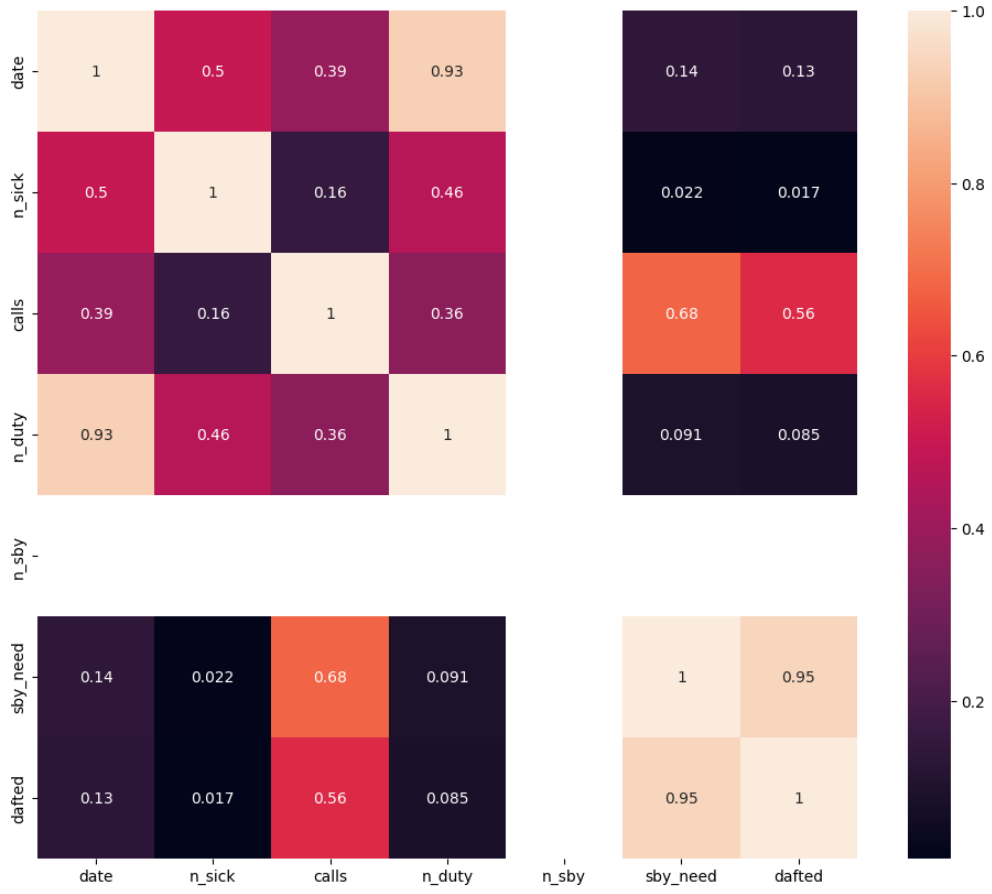
- There is a visible seasonality in the number of standbys activated in correlation with “Number of drivers sick over time”.
- It's evident that on several days, no standbys are activated, while on other days, a significant number are activated.

Trend of Additional Drivers needed over time



- While the number remains fairly low, we can see that there are certain days when additional drivers are needed due to a lack of standbys. This highlights the inefficiency in the current approach of a fixed number of standbys.

Correlation Analysis



1. n_sick (Number of Drivers Called Sick)

- Shows a moderate positive correlation with `n_duty` (0.459501). This suggests that as the number of drivers on duty increases, there's a corresponding increase in the number of drivers calling in sick.
- The correlation with `calls` is weakly positive (0.155371), indicating a slight association between the number of drivers calling in sick and the number of emergency calls.

2. calls (Number of Emergency Calls):

- It shows a strong positive correlation with `sby_need` (0.677468). On days with more emergency calls, there's a greater need to activate standby drivers.
- There's also a strong positive correlation with `dafted` (0.557340). This means that when emergency calls increase, there's often a need for additional drivers beyond the standby drivers.

3. n_duty (Number of Drivers on Duty)

- Shows a weak positive correlation with both `sby_need` and `dafted`. This suggests that even as the number of drivers on duty increases, there's still a need for standby drivers and, occasionally, additional drivers.

4. sby_need (Number of Standbys Activated)

- The extremely strong positive correlation with `dafted` (0.945168) is significant. It reveals that on days when more standby drivers are activated, there's also a higher need for additional drivers. This highlights a challenge in resource allocation and indicates that the current number of standby drivers might not be adequate.

Feature Engineering

I have implemented following feature engineering steps in order to enhance my dataset for better understanding and modeling-

Date-related Features:

Year, Month, Day, and Day of the Week: These features break down the date into its components, capturing the temporal structure. This helps the model recognize patterns like end-of-month effects, yearly trends, or weekday-specific patterns.

Is Weekend: Weekends might have different patterns compared to weekdays due to fewer working days, more recreational activities, etc. By distinguishing between them, the model can understand these patterns.

Quarter: Some patterns, like quarterly business cycles or seasonal changes, might be captured at the quarter level.

Lag and Rolling Features:

Lagged Features: Previous days' data can influence the current day's values. For instance, a spike in drivers calling in sick might have cascading effects for a few days.

Rolling Mean and Standard Deviation: These capture the local average and variability. For instance, if there's a sudden spike in the number of drivers calling in sick, comparing it to the rolling mean can give context.

Day-to-Day Difference: This captures daily fluctuations. A sudden change might indicate anomalies or specific events.

Ratios and Interactions:

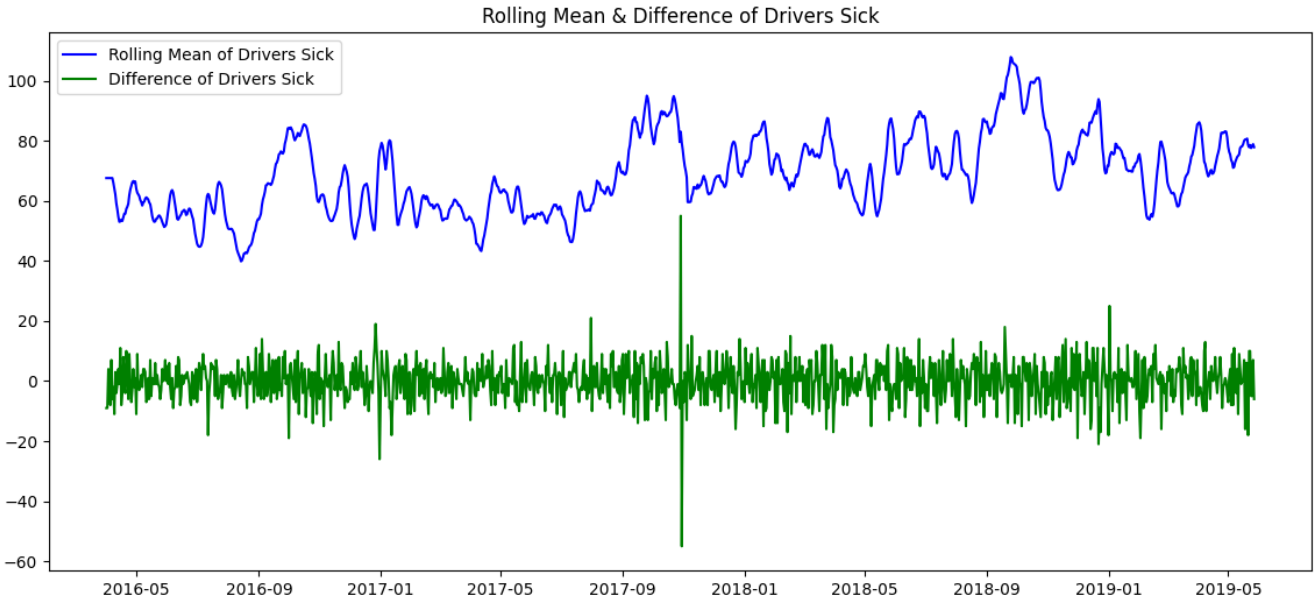
Sick to Available Ratio: By understanding the proportion of drivers who called in sick relative to available drivers, we can gauge the strain on the available resources.

Emergency Call to Driver Ratio: This ratio captures the demand (emergency calls) relative to the supply (available drivers). It can help in resource allocation and planning.

Sick Calls Interaction & Calls Driver Interaction: Interaction terms capture combined effects. For instance, days with both high sick calls and emergency calls might be particularly challenging. Similarly, understanding the relationship between demand and total resources can provide insights into operational efficiency.

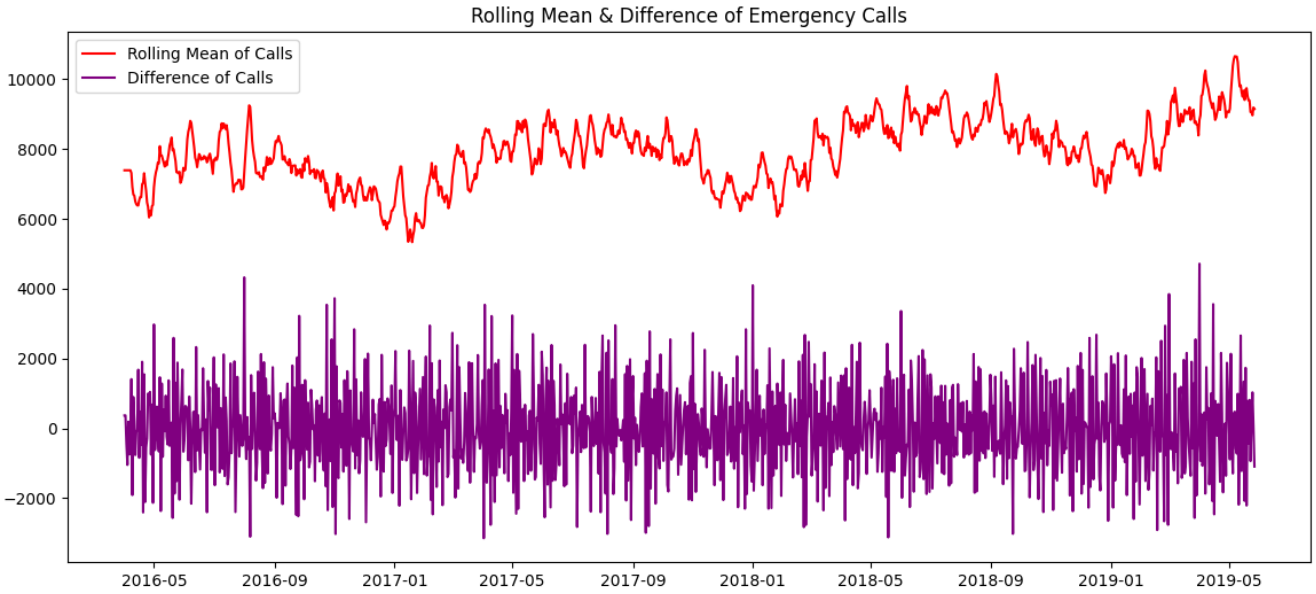
Post Feature Engineering EDA

Rolling mean and difference of sick drivers-



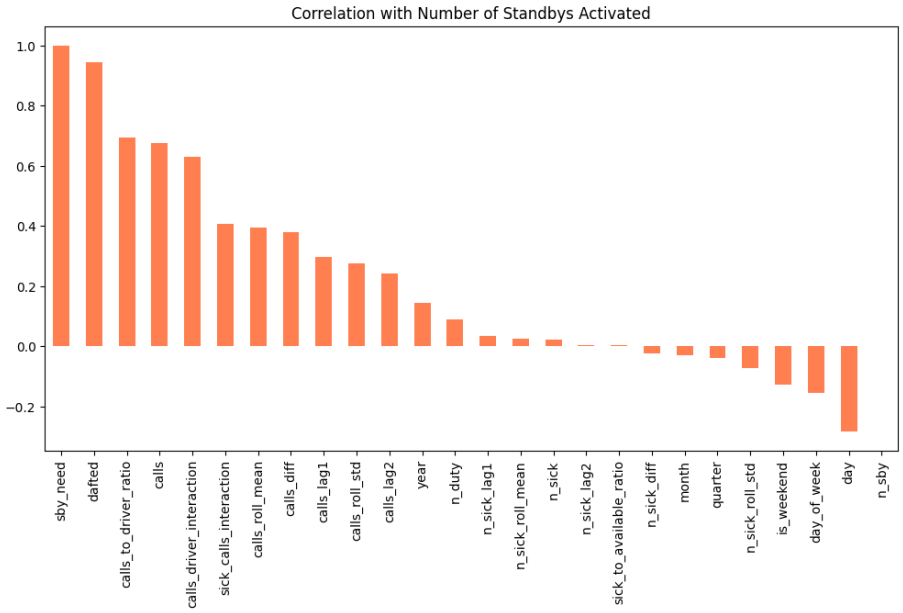
- The rolling mean of drivers calling in sick seems relatively stable, but there are noticeable peaks at specific times.
- The day-to-day difference for drivers calling in sick fluctuates around zero but does show occasional spikes and troughs. This indicates sudden increases or decreases in sick calls on certain days.

Rolling mean and difference of emergency calls –



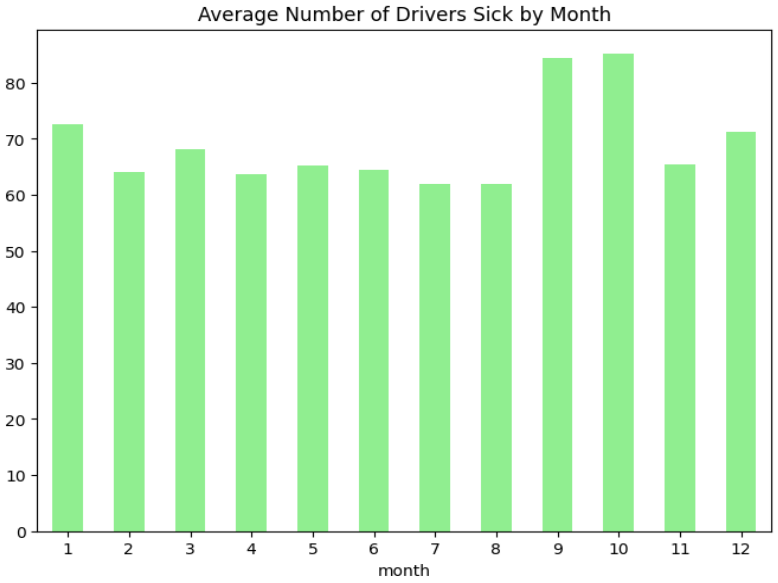
- The rolling mean for emergency calls shows a slight upward trend over time, suggesting an increase in emergency call volume.
- The day-to-day difference in emergency calls also fluctuates, with some days experiencing sharp increases or decreases in calls.

Correlation Analysis with Number of Standbys Activated –



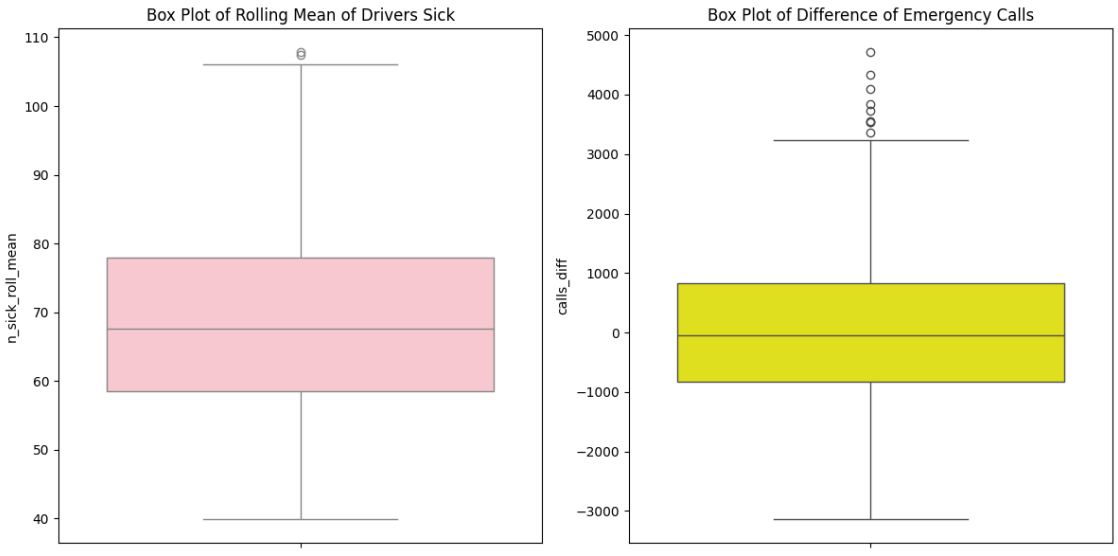
- The number of standbys activated (sby_need) has a very high correlation with the rolling mean of emergency calls, the day-to-day difference in calls, and the interaction between sick drivers and calls.

Drivers Sick by Month –



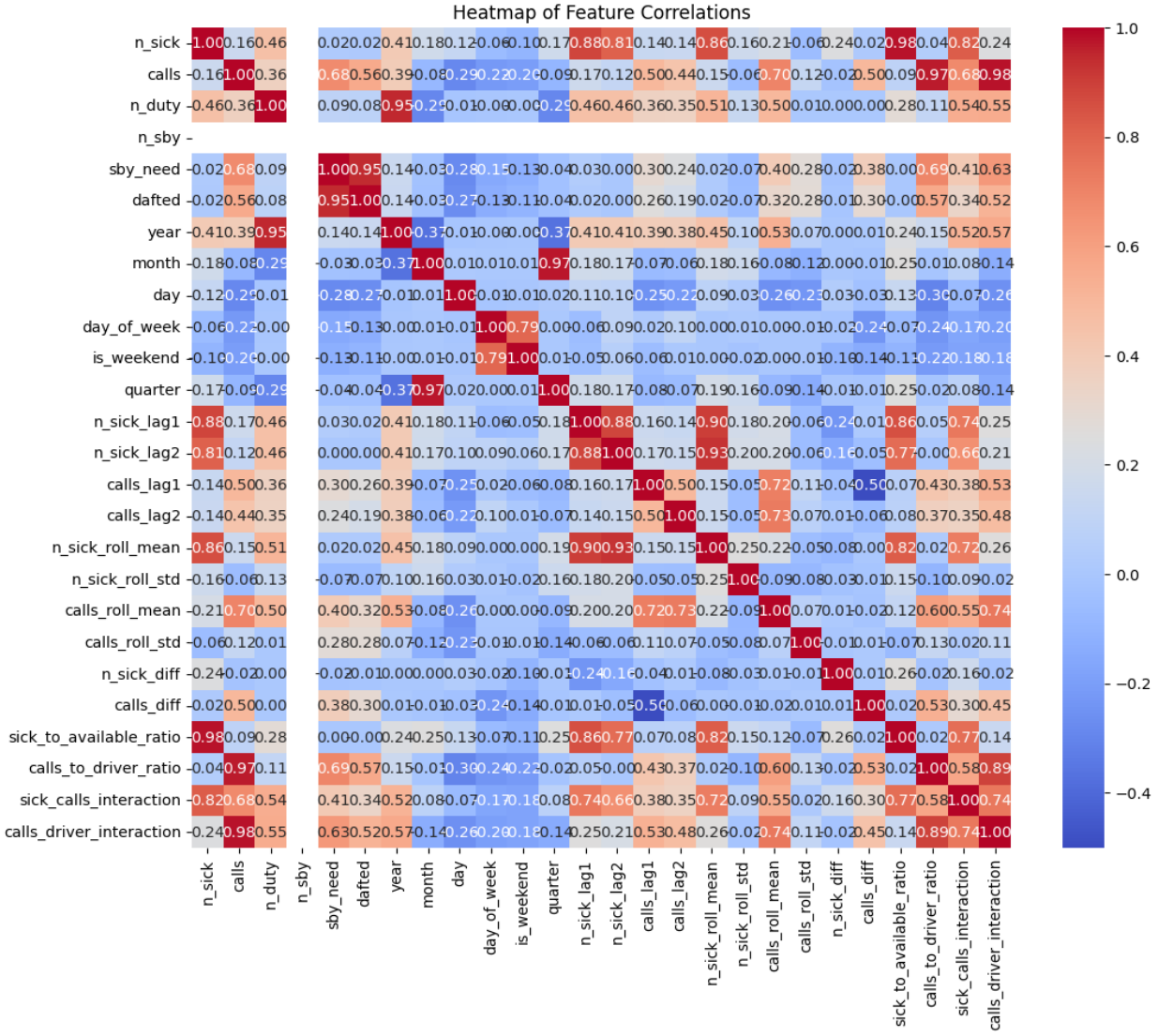
- Some months, especially in the middle and end of the year, tend to have a higher average number of drivers calling in sick. This could be indicative of seasonal factors or events in those months affecting driver availability.

Box Plots-



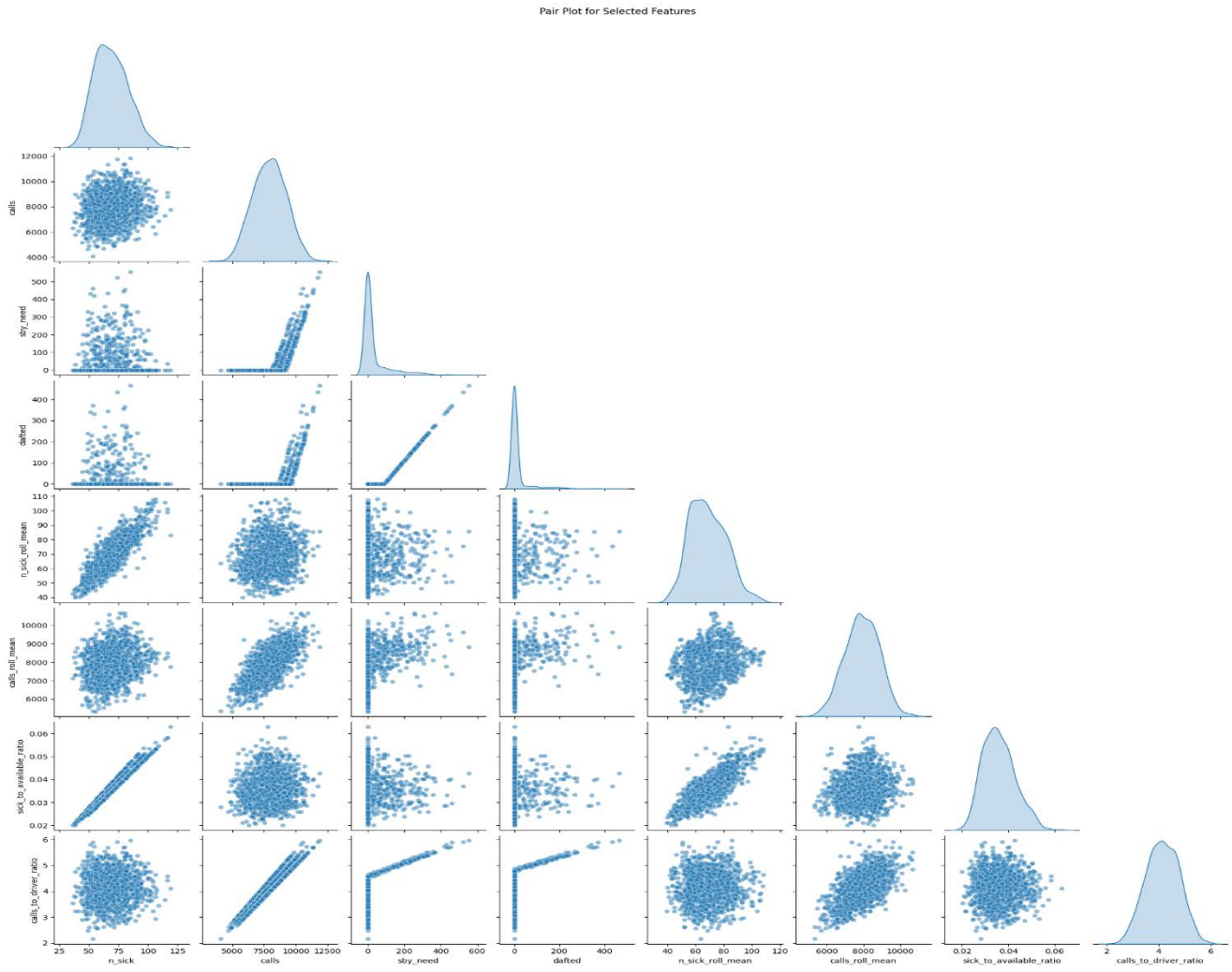
- The rolling mean of drivers calling in sick has a relatively compact interquartile range (IQR), indicating that most days have a consistent average number of sick calls. There are a few outliers suggesting unusually high or low sick calls on certain days.
- The day-to-day difference in emergency calls has a wider IQR, highlighting the variability in daily call volume changes.

Heatmap of Correlations-



- As seen in the correlation bar plot, sby_need has strong correlations with several features, emphasizing their importance in predicting the number of standbys needed.
- There's a strong positive correlation between the rolling mean of drivers calling in sick and the day-to-day difference in emergency calls. This indicates that when more drivers call in sick on average, there might be more variability in emergency calls.

Pair Plots-



'n_sick' vs. 'sby_need': There's a slight trend suggesting that as the number of sick drivers increases, the need for standby drivers might also increase.

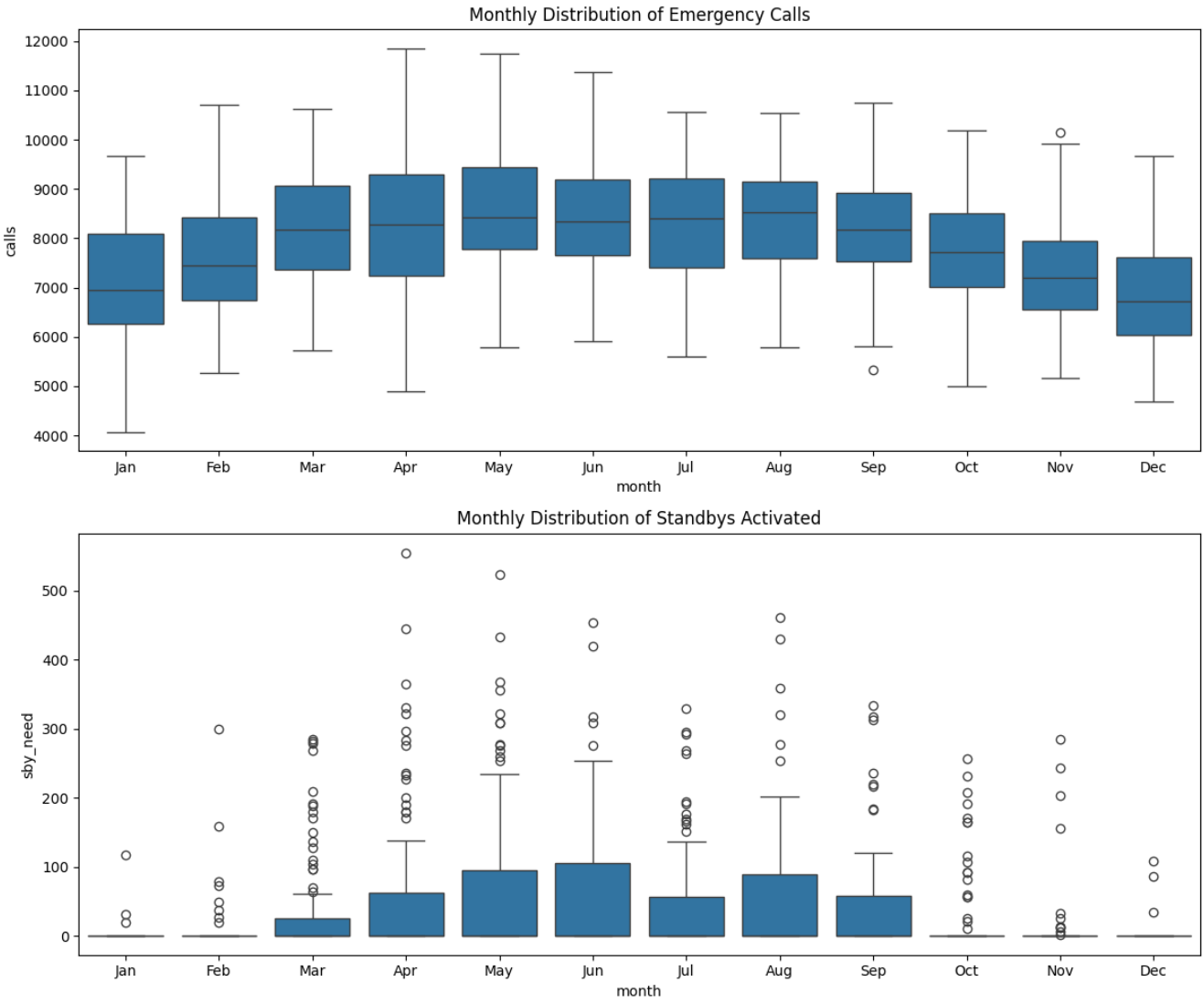
'n_sick' vs. 'n_sick_roll_mean': As expected, there's a strong positive relationship since the rolling mean is derived from the n_sick feature.

'calls' vs. 'sby_need': There seems to be a positive trend suggesting that as emergency calls increase, the need for standby drivers also goes up.

'calls' vs. 'calls_roll_mean': Strong positive relationship, which is expected since the rolling mean is derived from the calls feature.

'sby_need' vs. 'dafted': A positive relationship is evident. Days with higher standby needs also tend to have more drivers drafted from their days off.

Monthly Analysis-

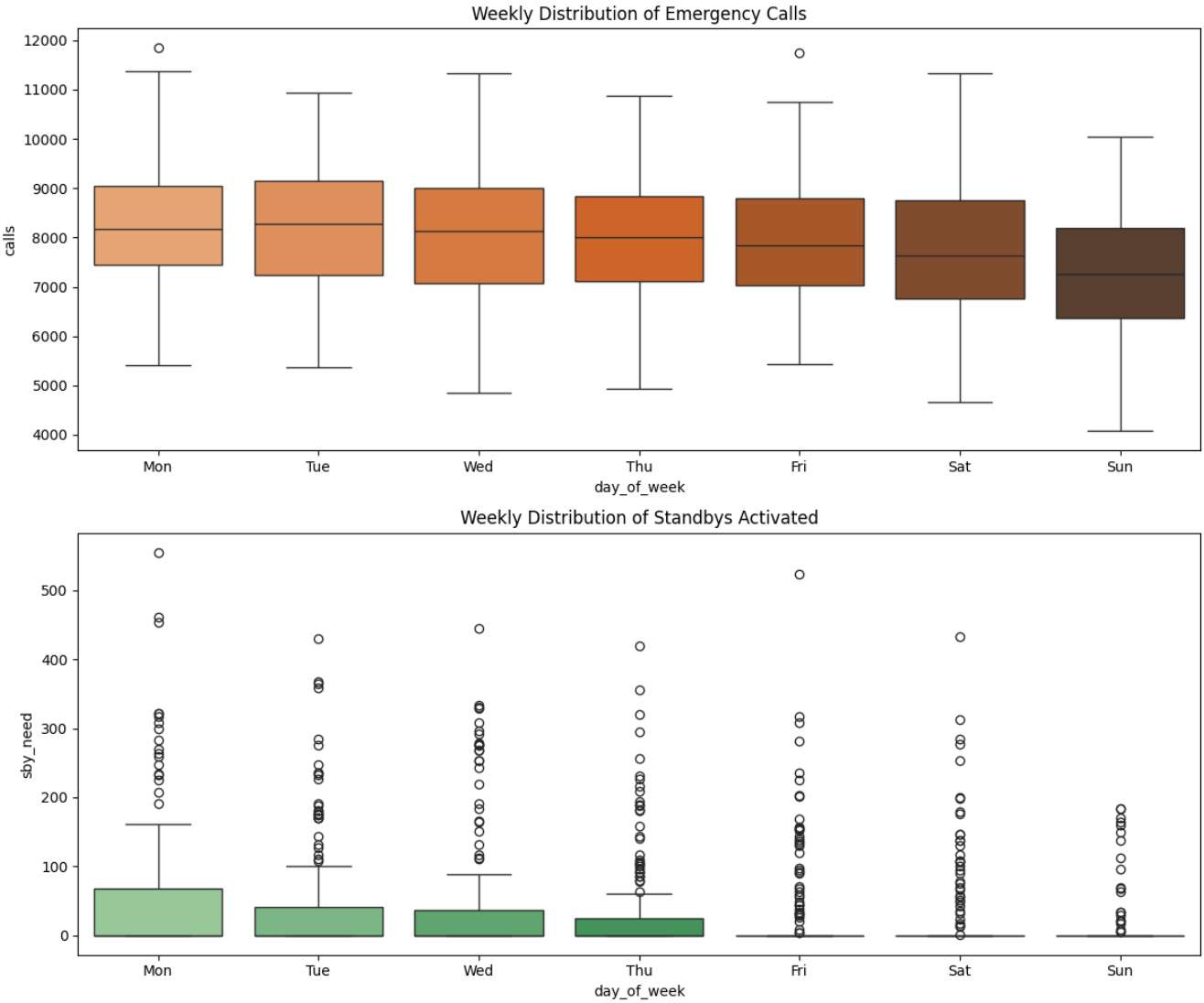


Emergency Calls: The months of January, February, November, and December seem to have a higher median number of emergency calls. This could be attributed to factors like winter-related accidents or illnesses, holidays, etc.

Standbys Activated: The requirement for standbys is higher in the months of January, February, November, and December. This aligns with the higher number of emergency calls observed in these months. The need for standbys remains relatively consistent across the other months, but there's a noticeable dip in June and September.

Outliers: There are certain outliers observed, especially in months like January, indicating days when the standby activations were unusually high.

Weekly Analysis-



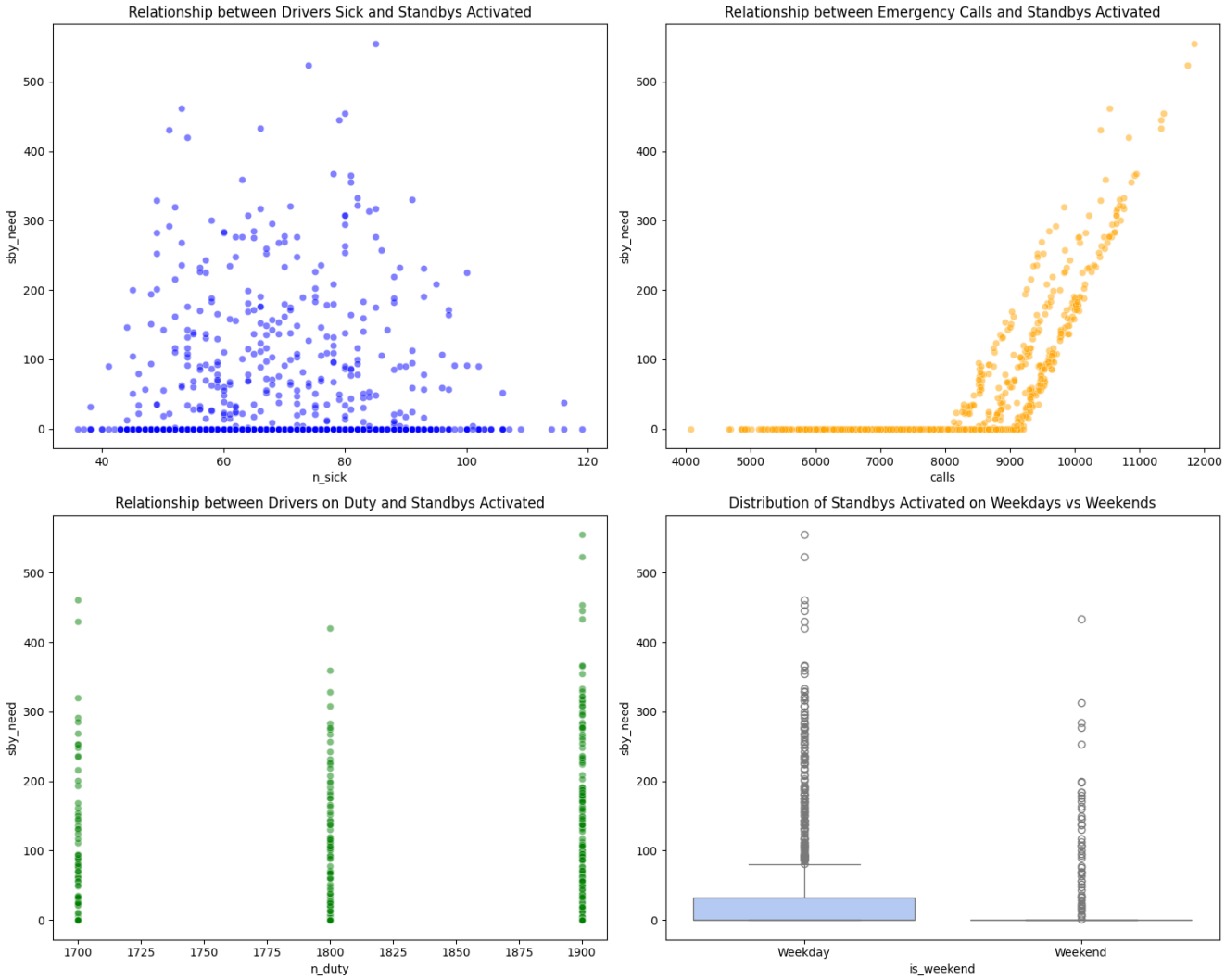
Emergency Calls: There appears to be a distinction between weekdays and weekends, with weekdays generally having a higher median number of emergency calls.

Mid-Week Surge: Wednesday seems to have a higher median and wider distribution compared to other days. This might indicate a higher variability in the number of calls received on Wednesdays. There's a noticeable drop in calls during the weekends, especially on Sundays.

Standbys Activated: Just like the emergency calls, standbys activated also show a higher median on weekdays, especially around mid-week. The number of standbys activated remains relatively consistent from Monday to Friday, with a slight dip on weekends.

Outliers: There are certain outlier days observed across the week, indicating specific days when the standby activations were unusually high or low. This could be due to special events, holidays, or other external factors.

Relationship Analysis:



Relationship between Drivers Sick and Standbys Activated (Top-left Plot - Blue): There seems to be a positive correlation between the number of drivers who called in sick and the number of standbys activated.

Insight: On days when more drivers call in sick, there's a corresponding increase in the activation of standby drivers. This relationship makes sense, as the absence of regular drivers would necessitate the use of standby resources to cover the shortfall.

Relationship between Emergency Calls and Standbys Activated (Top-right Plot - Orange): There's also a positive correlation between the number of emergency calls and the number of standbys activated.

Insight: This indicates that on days with a higher volume of emergency calls, more standby drivers are activated to handle the increased demand. This underscores the importance of having a flexible and responsive system to handle variations in emergency call volumes.

Relationship between Drivers on Duty and Standbys Activated (Bottom-left Plot - Green): The relationship here seems less direct, with a spread of points. However, there might be a slight trend indicating that as the number of drivers on duty increases, the number of standbys activated might decrease.

Insight: Having more drivers on duty can potentially reduce the need for activating standbys. However, other factors, like the actual number of emergencies or unexpected driver absences, will also play a role.

Distribution of Standbys Activated on Weekdays vs Weekends (Bottom-right Plot - Cool-warm Palette): The median number of standbys activated seems to be slightly higher on weekdays compared to weekends. There's also a broader range and more variability in the number of standbys activated on weekdays.

Insight: This might indicate a higher demand or more unpredictability during weekdays. This can be due to a variety of factors, including increased traffic, more people at work leading to potential workplace emergencies, or simply more activities taking place during weekdays.

Feature Selection and Standardization

In this step, The dataset is divided into features and the target variable.

We need to predict the sby_need, which represents the number of standby drivers that were activated on a given day.

And then we will standardize the data to ensures that all features contribute equally to the model's performance.

Model Building

In the Model Building phase, three regression models were trained and evaluated to predict the required number of standby drivers. The models chosen were Linear Regression, Random Forest Regressor, and XGBoost Regressor. Each model's performance was evaluated using the Root Mean Squared Error (RMSE), a standard metric for regression tasks that quantifies the average difference between predicted and actual values.

1. Linear Regression: RMSE: 50.7533

Analysis: The RMSE value indicates that this model's predictions deviate, on average, by approximately 50.75 drivers from the actual values. The performance of the Linear Regression model is not satisfactory for this particular task, suggesting that the relationship between the features and the target variable might be non-linear or that there might be interactions between features that the Linear Regression model is unable to capture.

2. Random Forest Regressor: RMSE: 4.6019

Analysis: The Random Forest model significantly outperformed the linear regression model. With an RMSE of approximately 4.60, this model's predictions deviate by an average of roughly 4.6 drivers from the actual values.

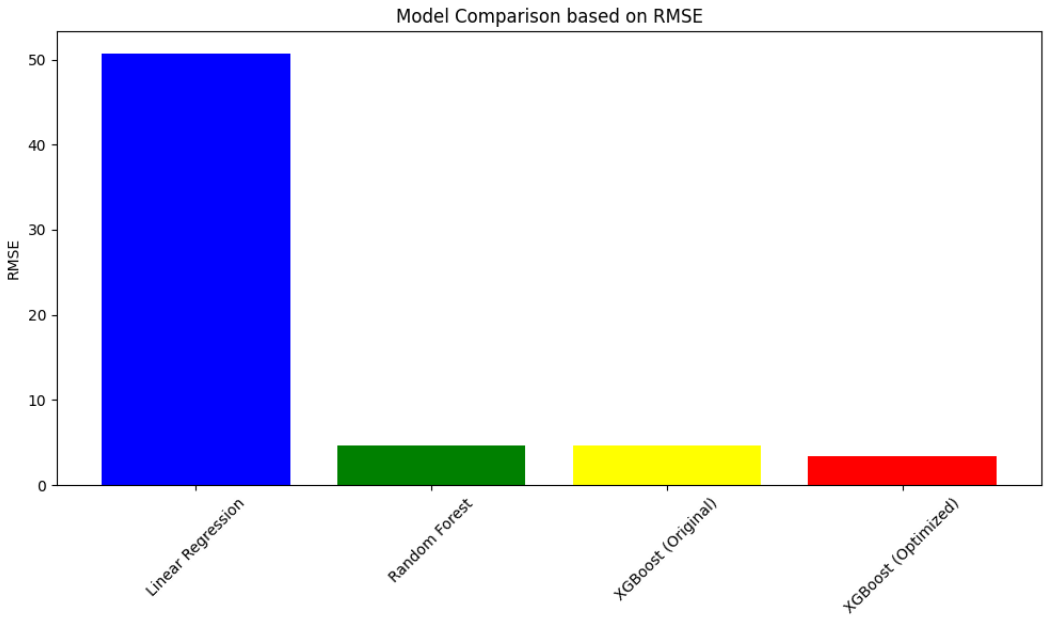
3. XGBoost Regressor: RMSE: 3.7471

Analysis: The XGBoost regressor delivered the best performance among the three models, with an RMSE of approximately 3.75. This suggests that the model's predictions deviate, on average, by about 3.75 drivers from the actual number. XGBoost is known for its efficiency and capability to handle complex datasets, making it a strong choice for this prediction task.

As we can see that XGBoost regressor gives best performance out of three. We will move forward with this model and perform some hyperparameter tuning to enhance it's performance.

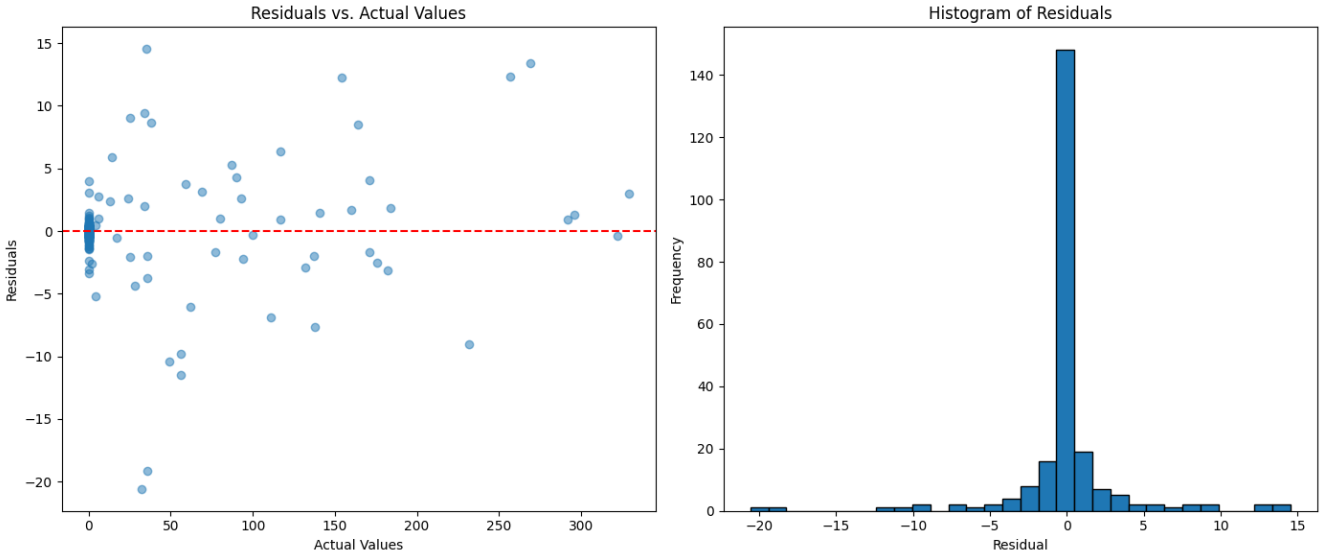
The optimized model gives an RMSE of approximately 3.47 and an MAE of 1.55. The optimized model's superior performance suggests it's well-suited for deployment and making future predictions on the required number of standby drivers.

Model Comparison



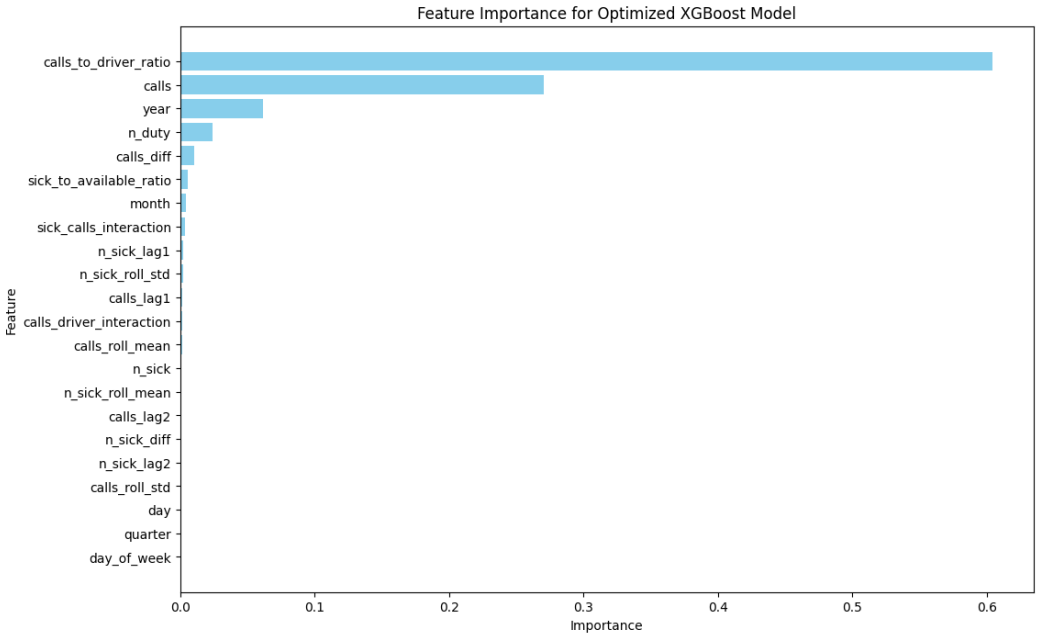
The models were compared based on their RMSE values, with the optimized XGBoost Regressor expected to perform exceptionally well given its refined hyperparameters. The bar chart visualization offers a clear depiction of each model's performance, assisting stakeholders in making informed decisions regarding model selection. Based on this comparison, we will move forward with Optimized XGBoost Model.

Error Analysis



The residual analysis for the optimized XGBoost model gives favorable results. The scatter plot suggests that the residuals, and thus the model's errors, are random across different actual values. The histogram further supports the model's effectiveness, showing a balanced distribution of residuals around zero, indicating no evident bias in predictions. Both visualizations corroborate the model's reliability and robustness in predicting the required target variable.

Feature Importance



The most critical features are displayed at the top for easy interpretation.

calls_to_driver_ratio: This feature seems to have the highest importance. This ratio likely provides insight into the balance between emergency calls and the available drivers. If this ratio is high, it suggests that there might be more calls per driver, which could be an indicator of increased need for standby drivers.

calls: The number of emergency calls on a given day. It's directly related to the demand for drivers, and it's no surprise that it's a key predictor for how many standby drivers might be required.

year: This feature's importance suggests that there might be year-over-year changes or trends affecting the number of required standby drivers. Factors such as population growth or changes in city infrastructure might influence this.

n_duty: The number of drivers available on duty. This helps determine the gap between available resources and demand, hence influencing the need for standby drivers.

calls_diff: This might represent the difference in calls from one day to the next or from a reference point. If there's a sudden spike or drop, it would certainly affect the number of standby drivers needed.

sick_to_available_ratio: This ratio gives an idea of how sickness impacts available resources. A high ratio might indicate a larger proportion of drivers calling in sick, thus stressing the system and potentially requiring more standby drivers.

month: Seasonal patterns as mentioned in the problem description. For example, more sickness during winter months.

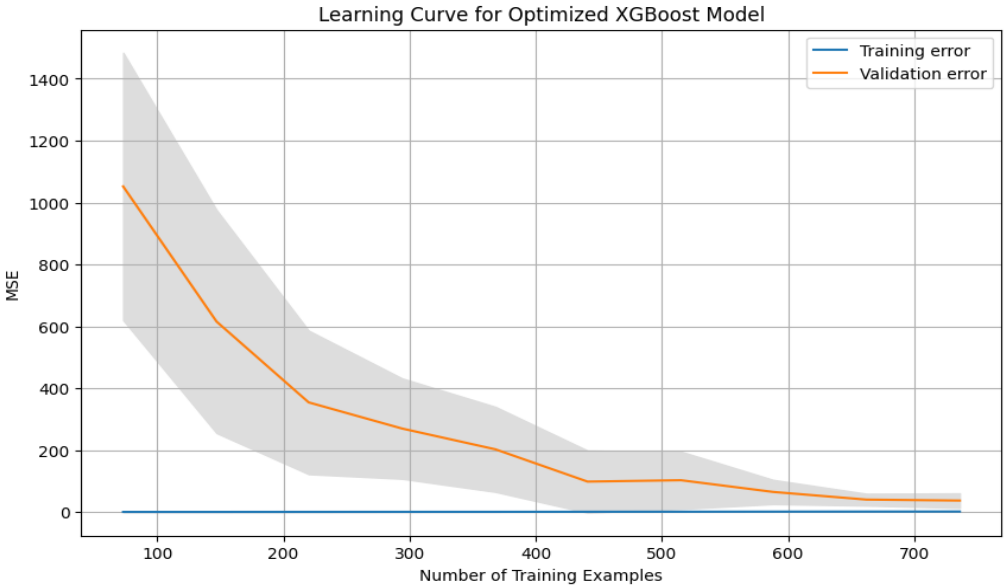
sick_calls_interaction: An interaction term between sickness and calls, which could capture how simultaneous increases in both these factors strain the system.

n_sick_lag1, calls_lag1, etc.: Lag features capture past values, which can help in identifying patterns or trends. For instance, if many drivers called in sick yesterday (n_sick_lag1), it might impact today's requirements.

n_sick_roll_std, calls_roll_mean, etc.: These are rolling statistics, which help capture short-term trends or patterns in the data. For example, if the rolling mean of calls is rising, it might suggest an increasing trend in demand.

day_of_week, quarter, etc.: These capture time-based patterns, like certain days of the week or quarters of the year having more demand or sickness.

Learning Curve Analysis



We can interpret following results from the learning curve analysis-

1. The Training error starts relatively low and then increases a bit as more data is added. This is expected behavior since it's generally easier for a model to fit to a smaller dataset perfectly. As the dataset grows, the model starts to generalize, leading to a slight increase in error.

2. The Validation error starts off quite high with a small dataset and then decreases as more data is added. This suggests that as the model is exposed to more training examples, its generalization capability improves, leading to better performance on unseen data.

3. The shaded regions represent the variability (standard deviation) of the error for both training and validation sets across the cross-validation folds. A tighter band suggests that the model's performance is consistent across different subsets of the data.

4. Over time, both curves appear to converge, but there's still a gap. This gap suggests that the model might be overfitting slightly, as it performs better on the training data compared to the validation data. However, the gap isn't very wide, which means the overfitting isn't severe.

Deployment of the Predictive Model through a Graphical User Interface

Introduction: After careful analysis, model selection, and optimization, we have developed a machine learning model that can predict the standby needs based on various features like sickness levels, call volumes, and other relevant parameters. To make the best use of this model, it's essential to make it accessible to the business users in the most seamless manner possible.

Why a GUI?

User-Friendly: A graphical interface is intuitive and requires minimal training. Users don't need to understand the underlying code or technical details to use the model.

Accessibility: A GUI can be accessed from various devices, including desktops, tablets, and even mobile phones.

Real-time Predictions: Users can input current data and get predictions on-the-fly, aiding in immediate decision-making.

Visual Feedback: Besides predictions, GUI can offer visual feedback, charts, and other visual aids to help interpret results.

Proposed GUI Layout and Features:

Data Input Section: Users can manually input data or upload a file containing the required information.

Features like n_sick, calls, n_duty, etc., would have dedicated input fields, along with date pickers for date-based fields.

Prediction Button: After inputting the data, users can click on a "Predict" button to get the standby needs prediction.

Results Display: The predicted standby needs will be displayed in a clear, prominent section.

Additionally, a confidence interval or prediction range could be shown, giving users an idea of the potential variability.

Historical Data & Predictions: A section where users can view past data alongside past predictions to understand the model's performance over time.

This could be in the form of a table or a chart.

Feature Impact Visualization: A small section that visually indicates which features had the most influence on the prediction. This can be derived from our feature importance analysis.

This helps users understand what's driving the model's decisions.

Feedback Loop: Users can provide feedback on the predictions, indicating whether it was accurate, overestimated, or underestimated. This feedback can be used to further refine and train the model.

Error Instances Analysis: A section dedicated to instances where the model's predictions had particularly large errors, allowing users to deep-dive into specific scenarios and understand potential reasons.

Technical Considerations:

- The GUI can be web-based, making it platform-independent and easily accessible.
- Backend APIs will handle the model prediction, ensuring that the GUI remains responsive.
- Data security and privacy measures will be in place, especially if the application is cloud-hosted.

Conclusion: Implementing a GUI for our predictive model will not only democratize access to the insights it provides but also ensure that decisions can be made efficiently and with a clear understanding of the influencing factors. The proposed GUI will be a valuable tool for business users, allowing them to make data-driven decisions in real-time, ultimately leading to more efficient operations and better resource management.

Please Find Attached the Code below-

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

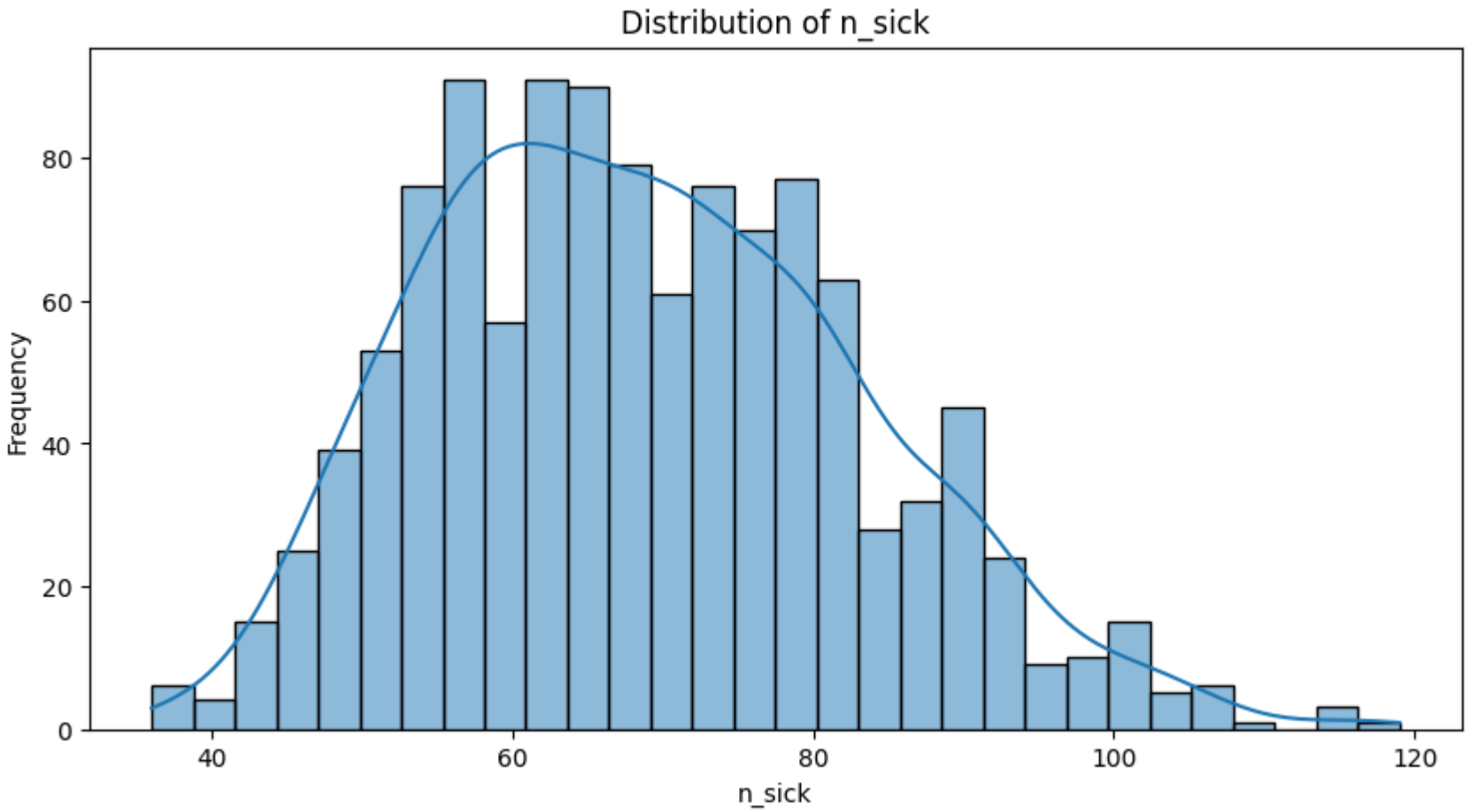
In [ ]:
```
sickness_data.isnull().sum()
```

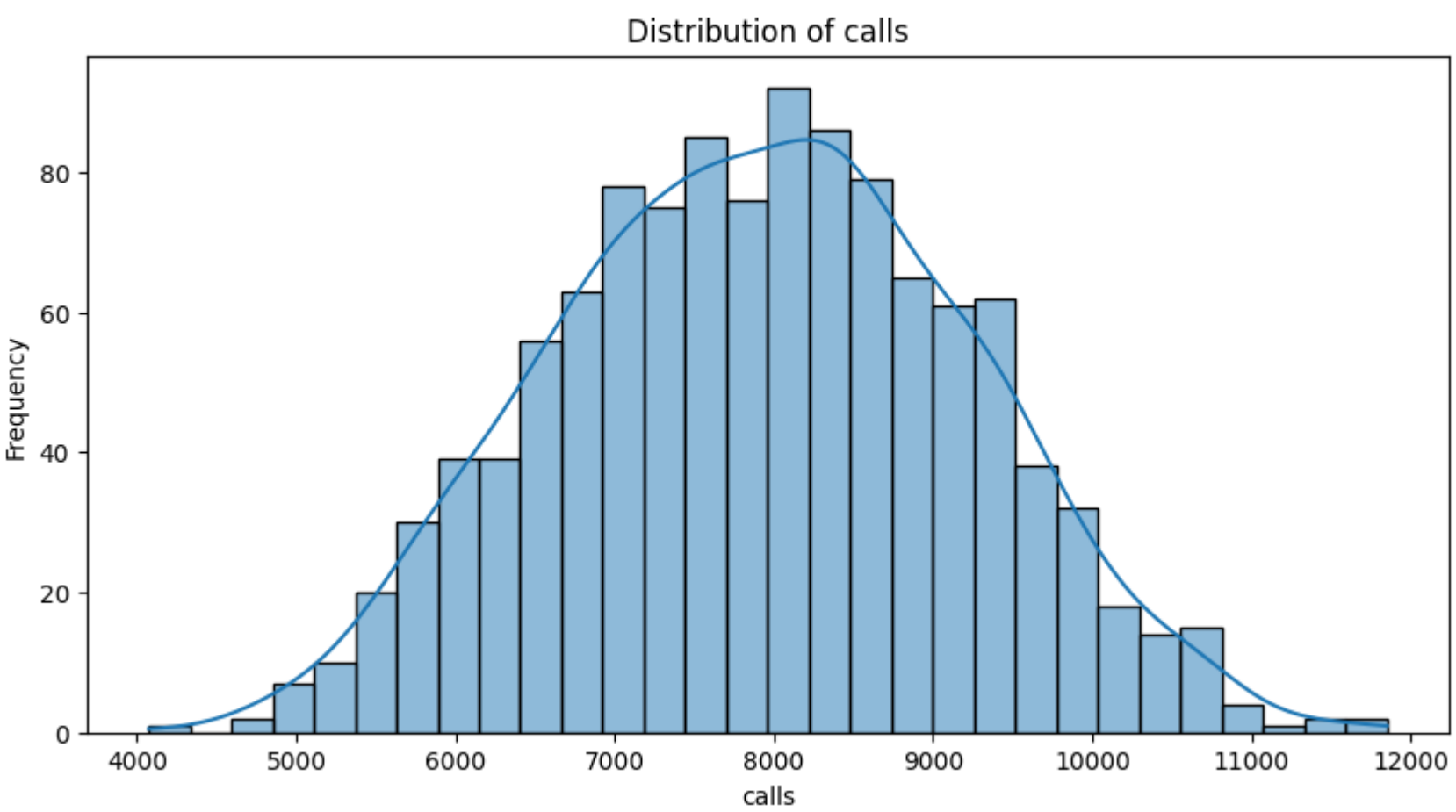
Out[ ]:
```
date        0
n_sick      0
calls       0
n_duty      0
n_sby       0
sby_need    0
dafted      0
dtype: int64
```

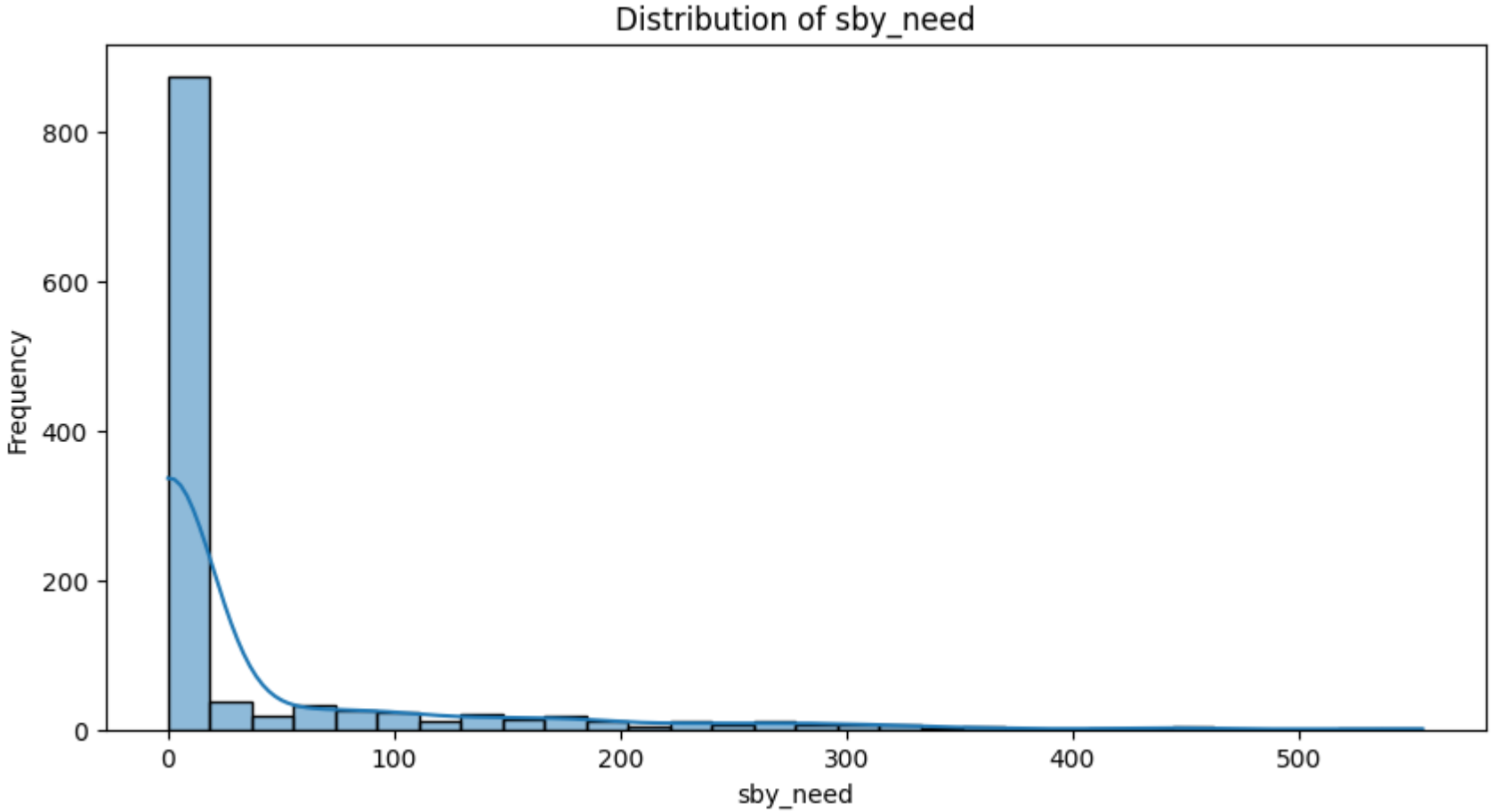
Data Exploration

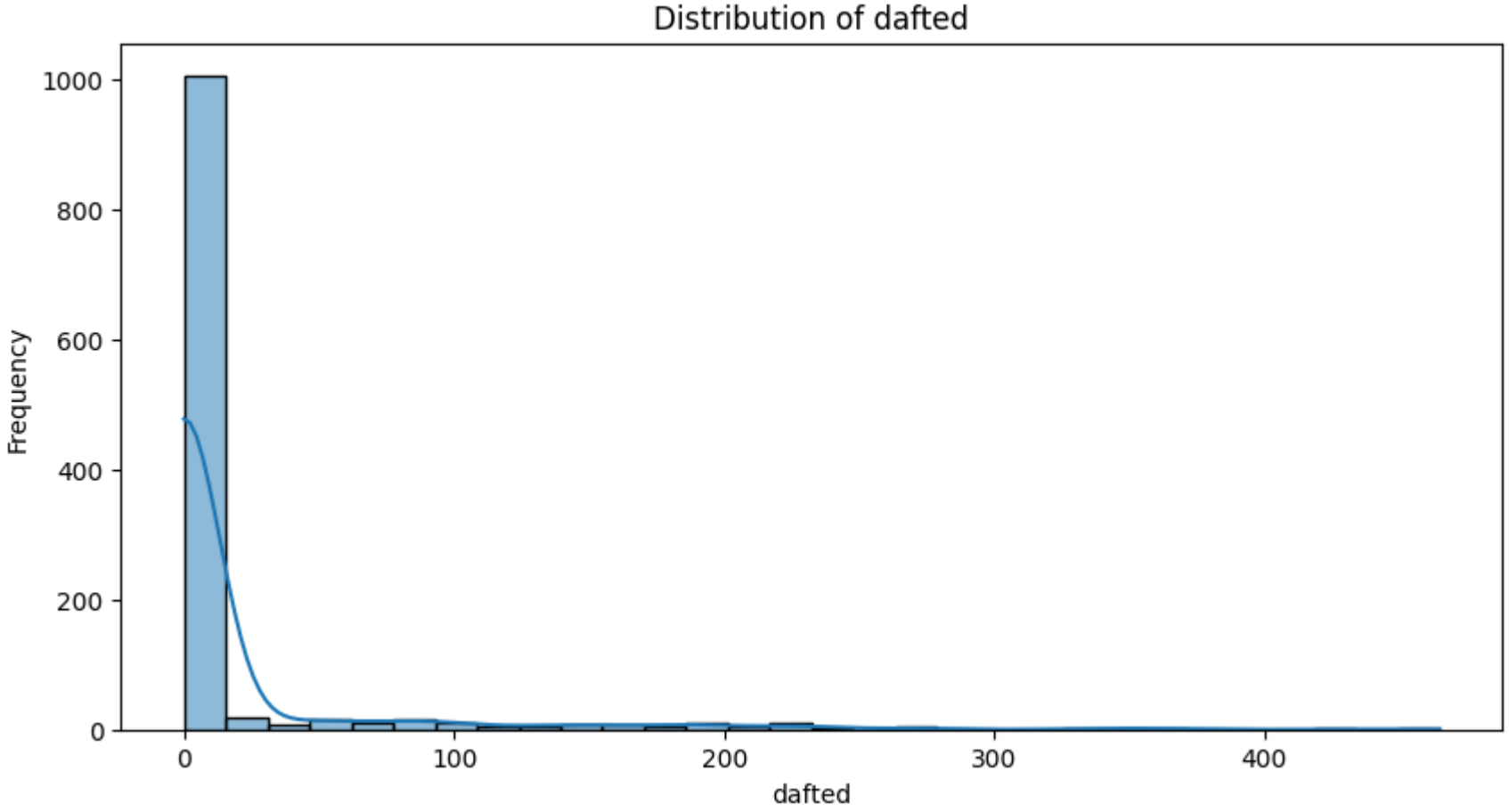
In [ ]:
```
# Columns of interest
columns_to_analyze = ['n_sick', 'calls', 'sby_need', 'dafted']

# 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)
```

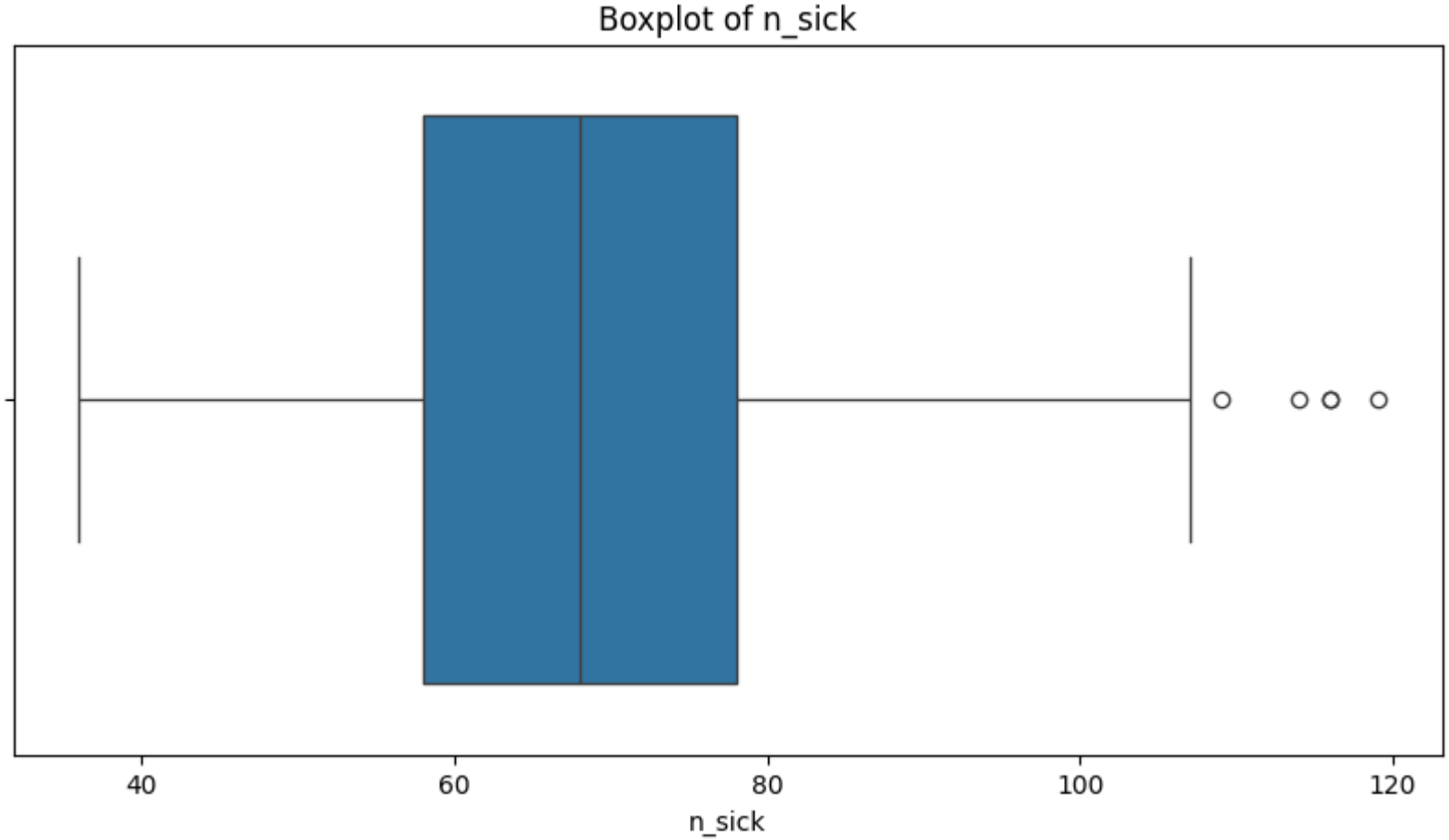
```
    plt.show()

# 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)}")
```

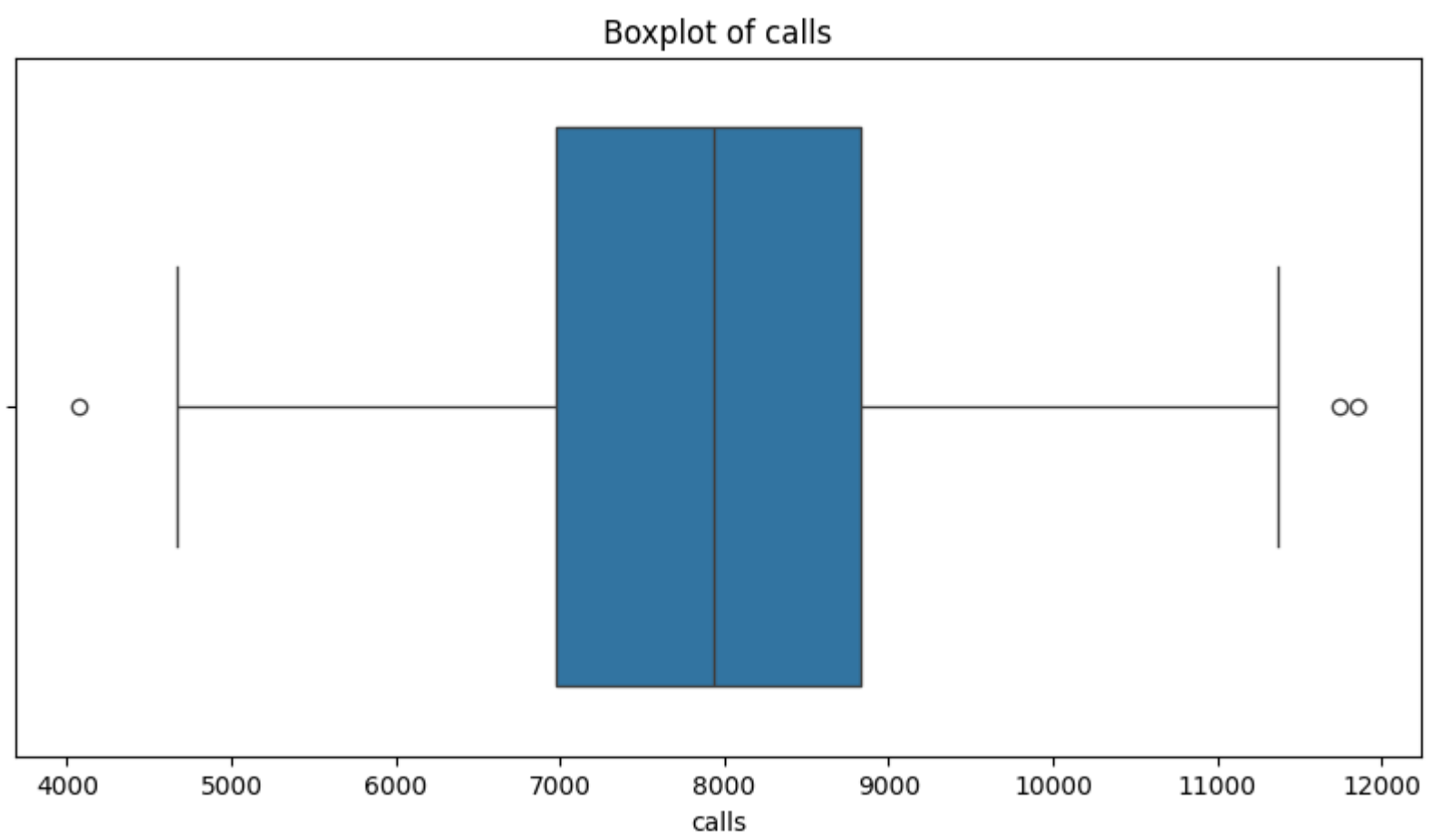


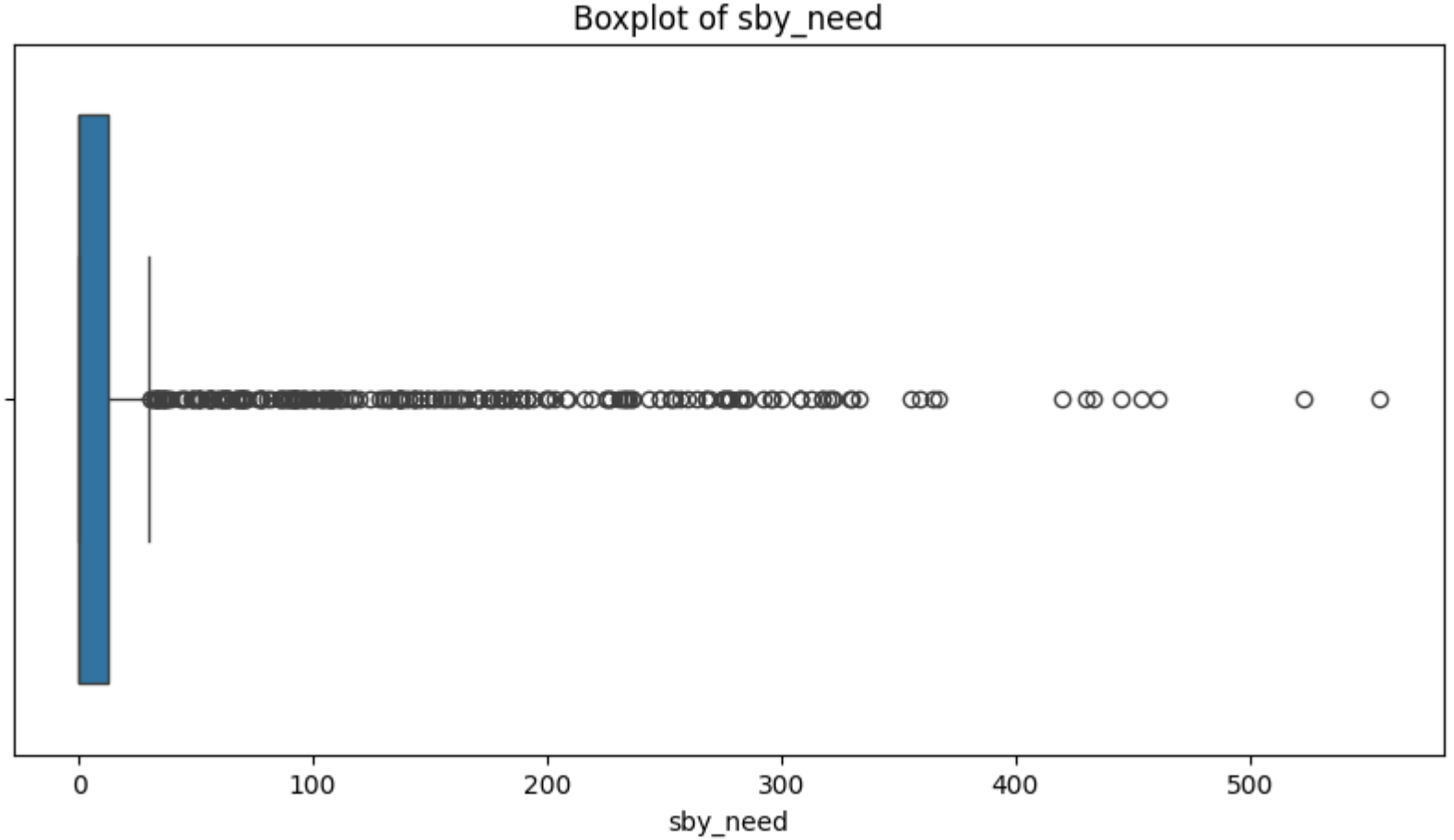

Distribution of calls
Frequency
0
20
40
60
80
4000
5000
6000
7000
8000
9000
10000
11000
12000
calls

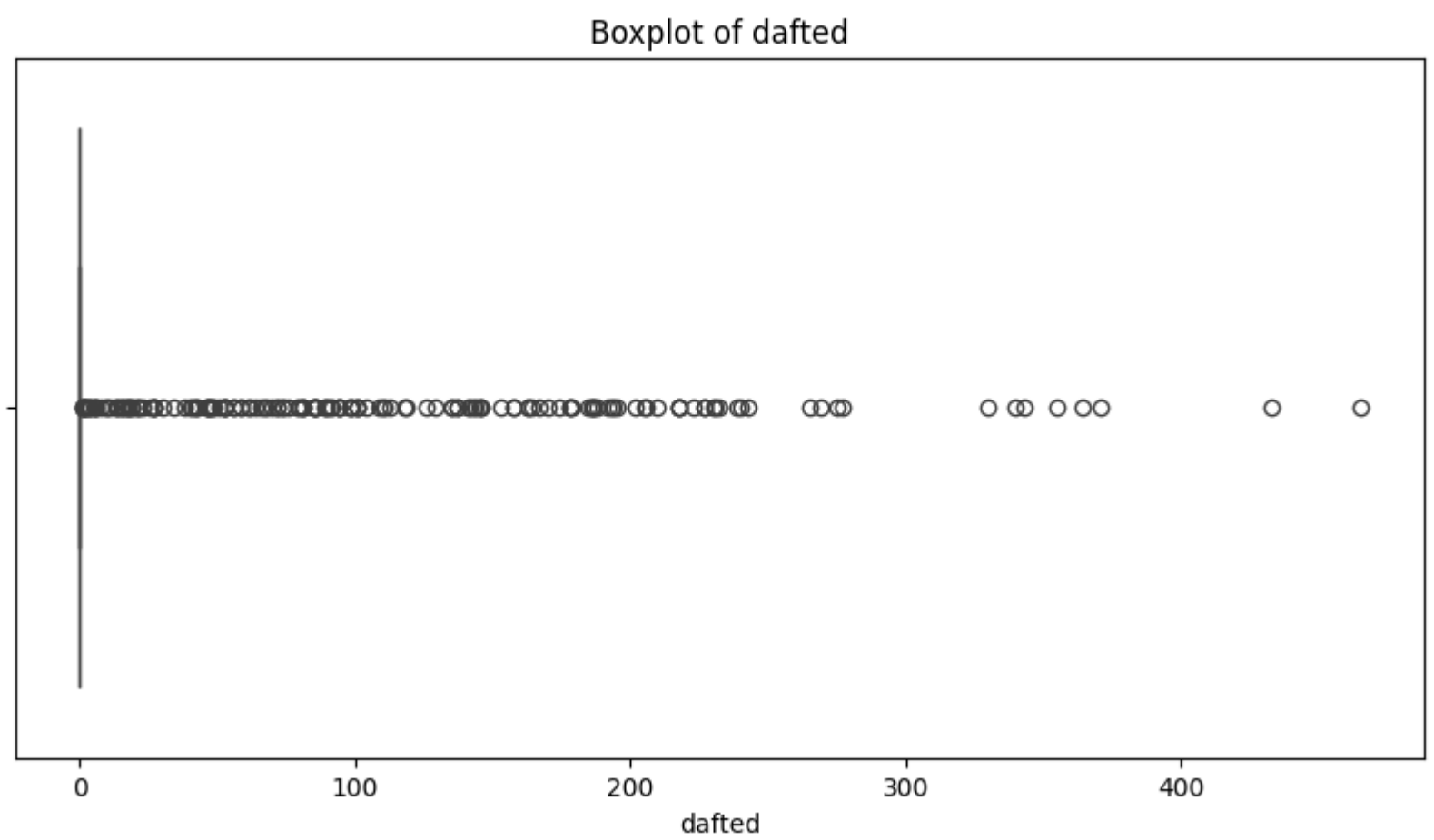

Distribution of sby_need
Frequency
sby_need
0
200
400
600
800
0
100
200
300
400
500


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.

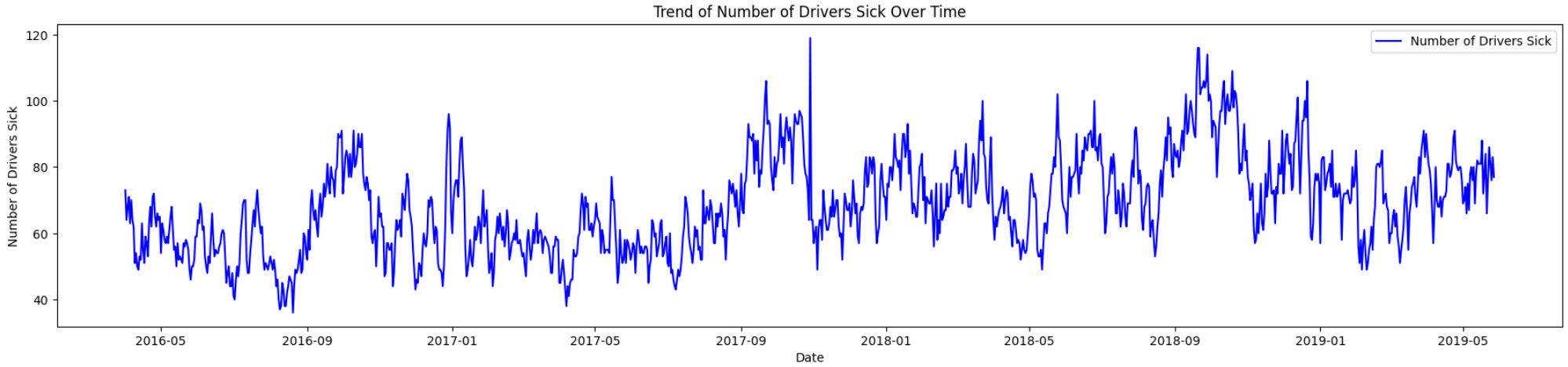
"Distribution of sby_need" :The mode, or the peak of the distribution, is at 0. This suggests that on most days, there was no reported standby need. Spread: While a significant portion of the data is clustered around 0, there are days when the standby need goes up to around 90, indicating variability 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.

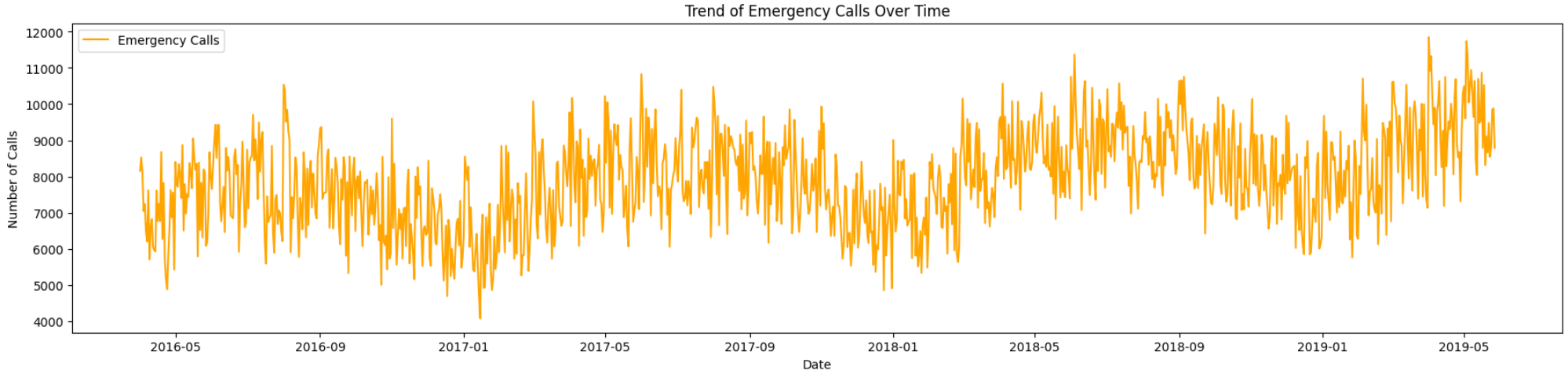
In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

# 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.

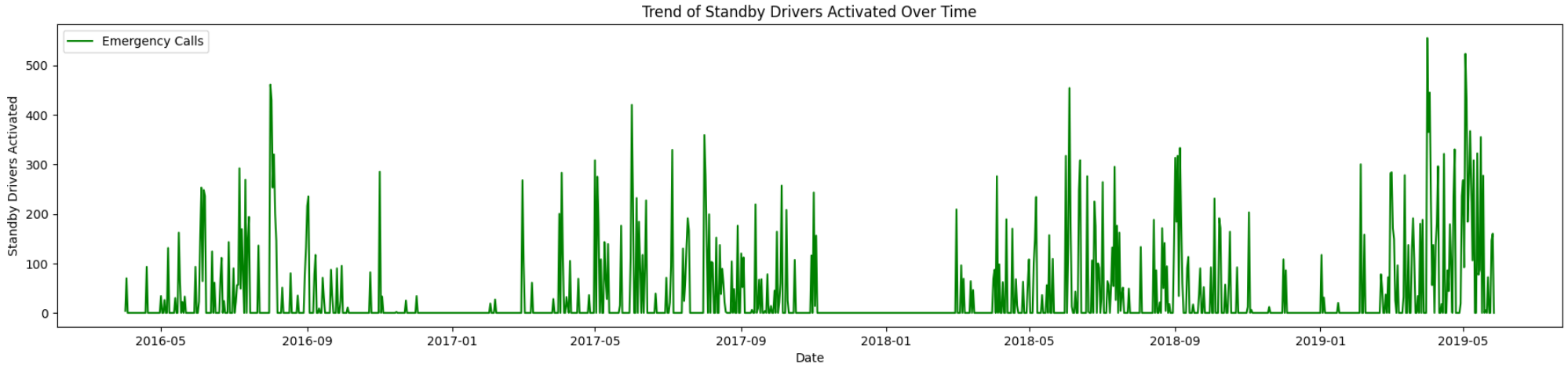
In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

# 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()
```

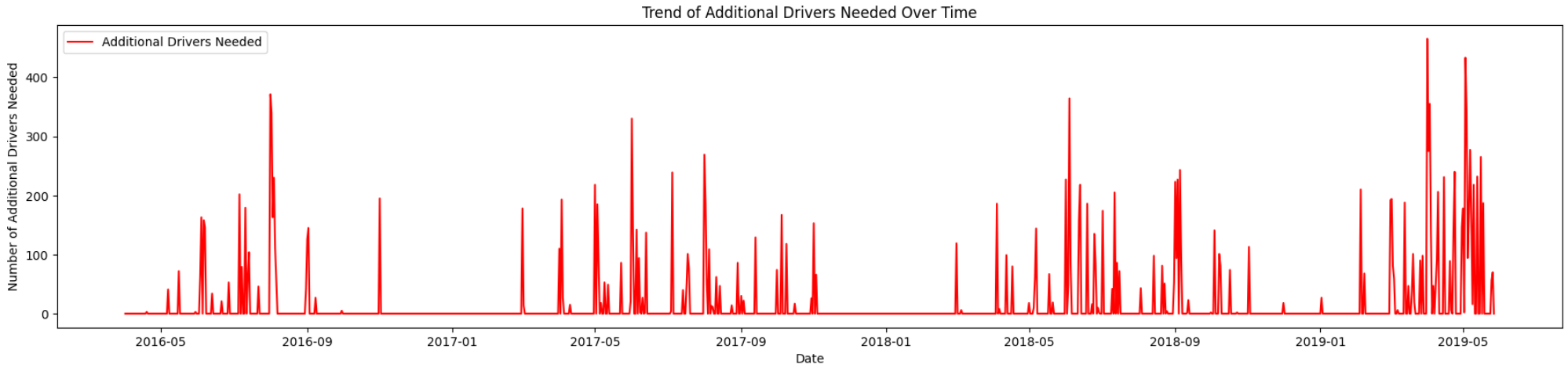
```
plt.tight_layout()
plt.show()
```



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()
```

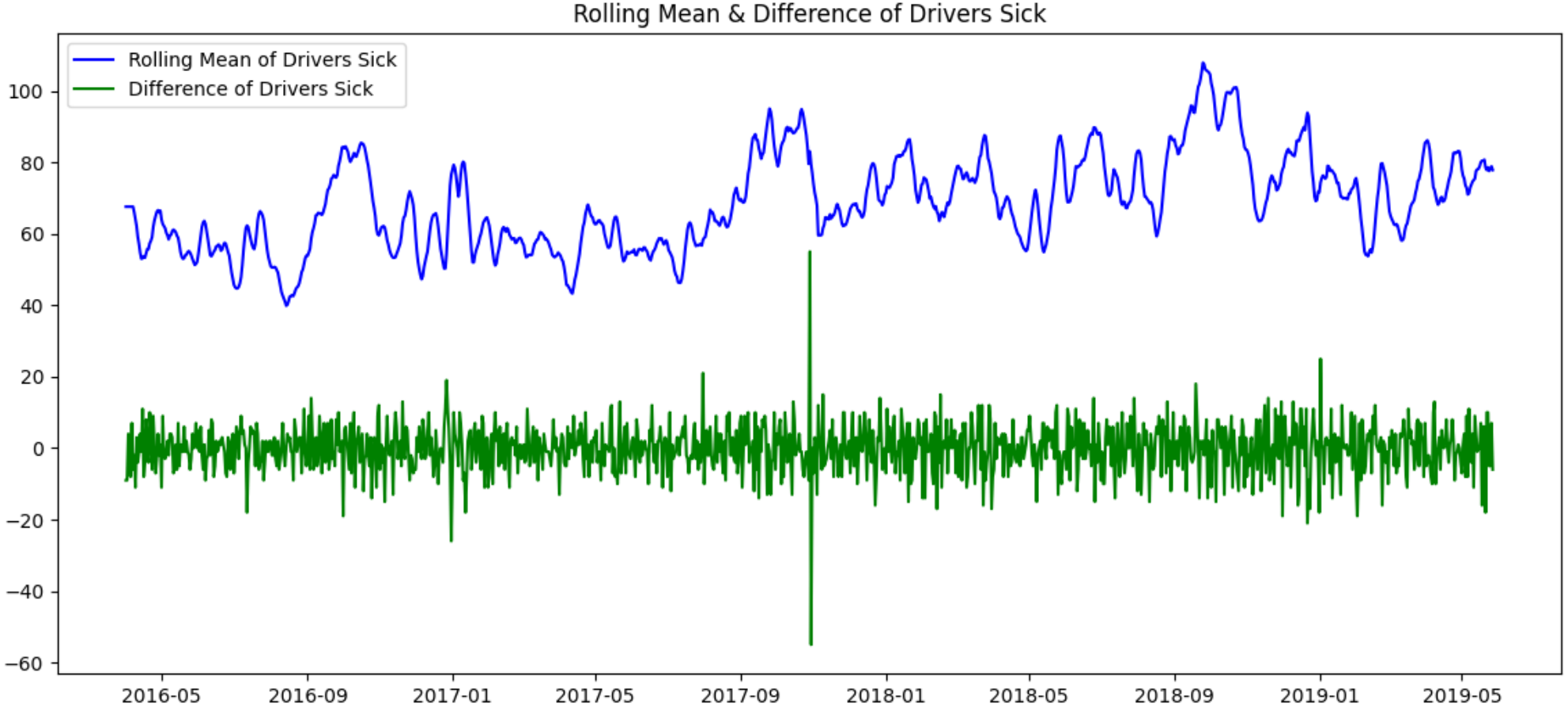
```
# Columns for which to apply backward fill
rolling_columns_to_bfill = [
    'n_sick_roll_mean', 'n_sick_roll_std', 'calls_roll_mean', 'calls_roll_std'
]

# 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)
```

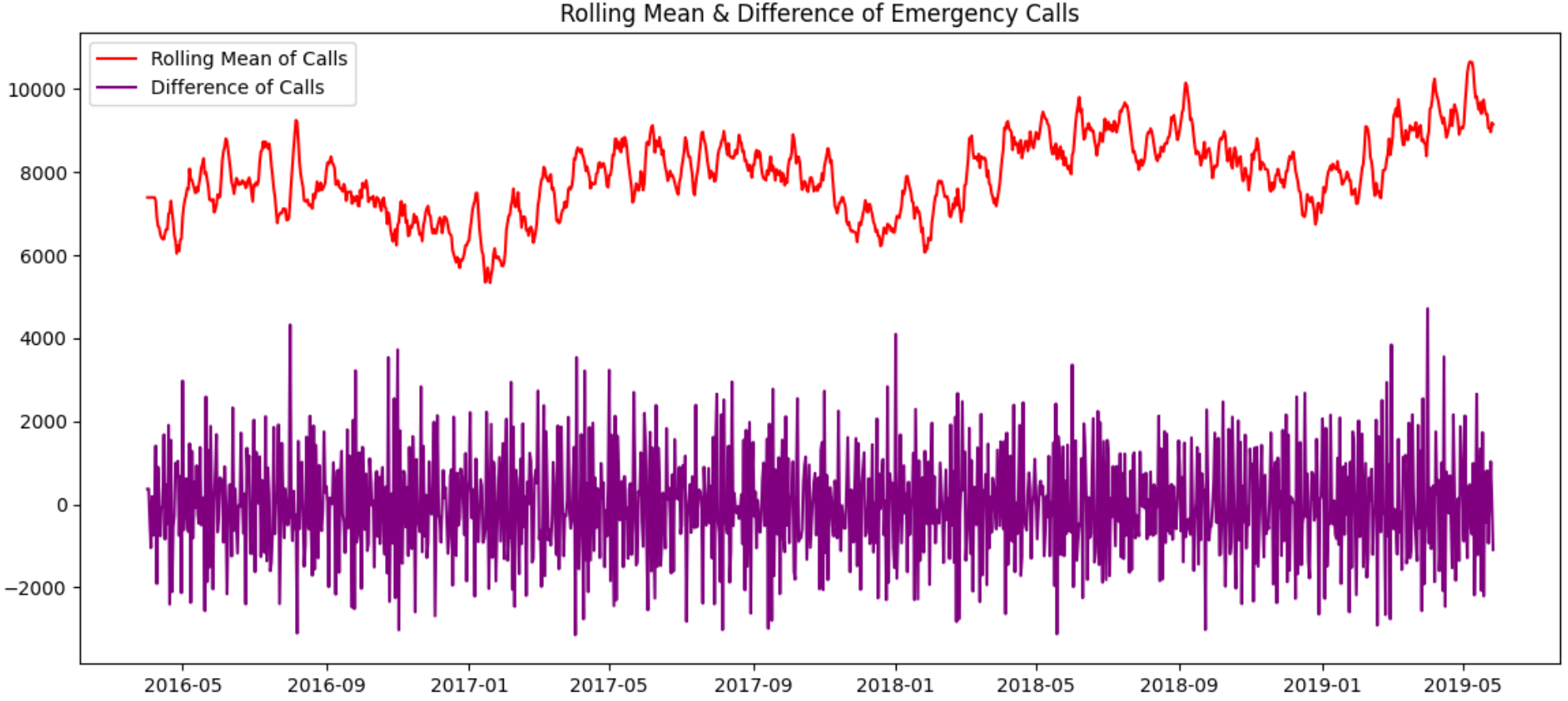
```
date                        0
n_sick                      0
calls                       0
n_duty                      0
n_sby                       0
sby_need                    0
dafted                      0
year                        0
month                       0
day                         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
```

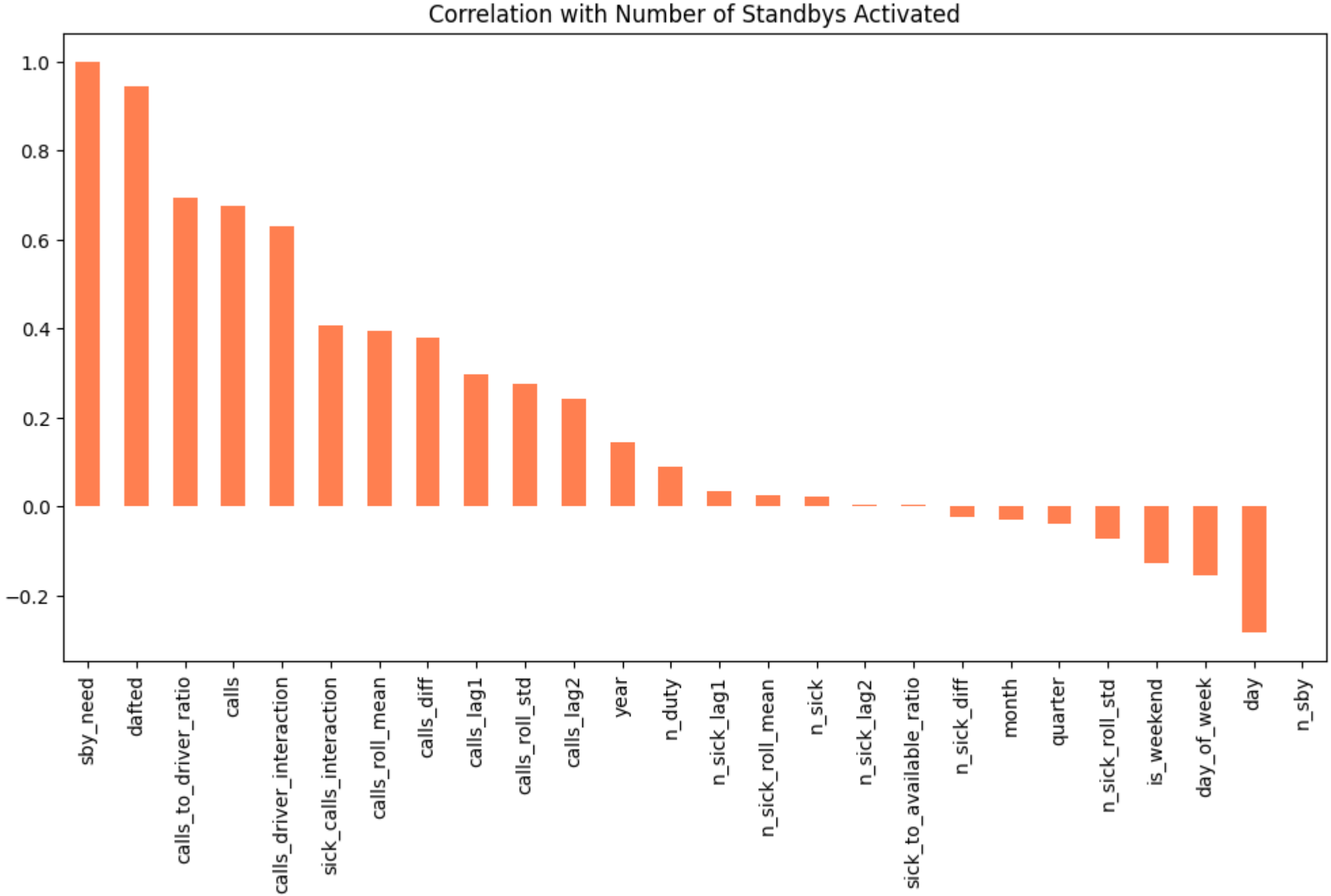
In [ ]:

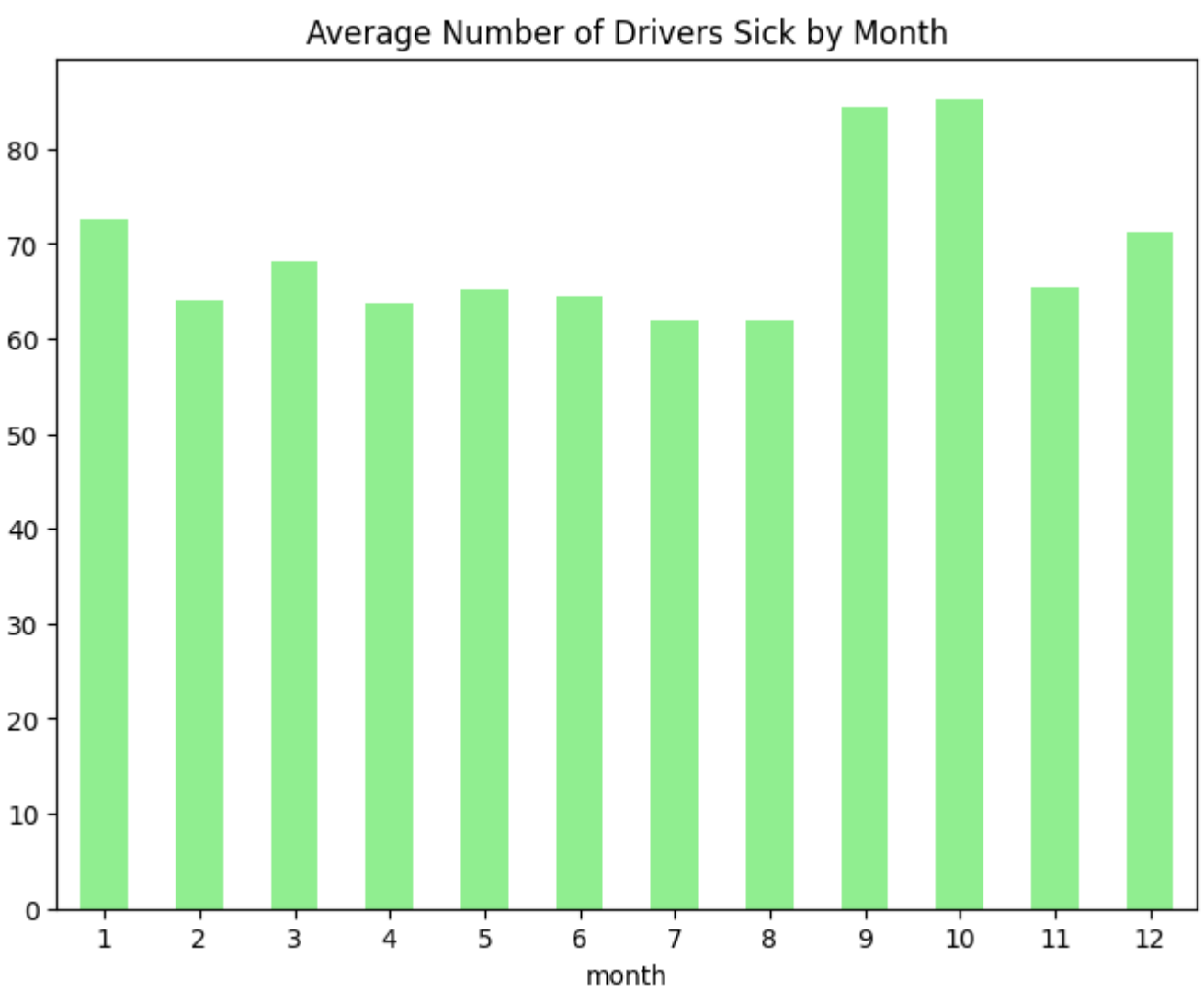
```
# 1. Temporal Visualization of New Features:
plt.figure(figsize=(14, 6))
plt.plot(sickness_data['date'], sickness_data['n_sick_roll_mean'], label='Rolling Mean of Drivers Sick', color='blue')
plt.plot(sickness_data['date'], sickness_data['n_sick_diff'], label='Difference of Drivers Sick', color='green')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

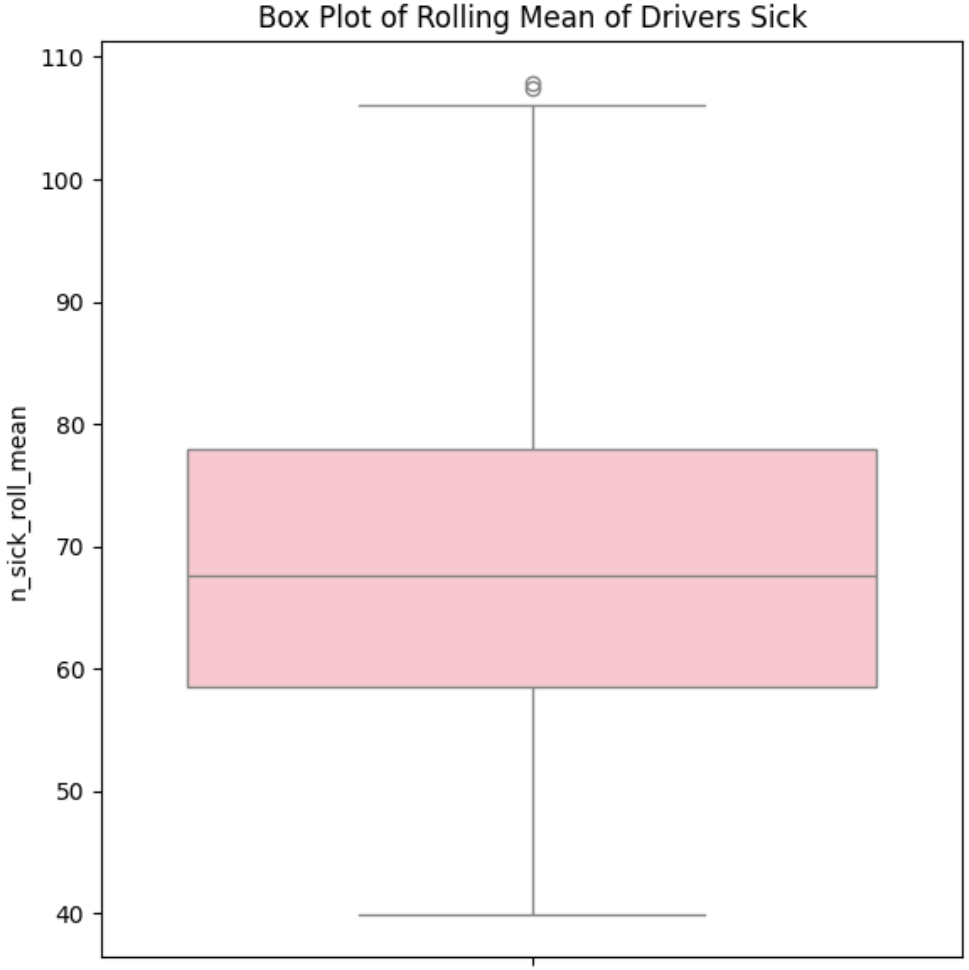
```
plt.legend()
plt.show()

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()
```

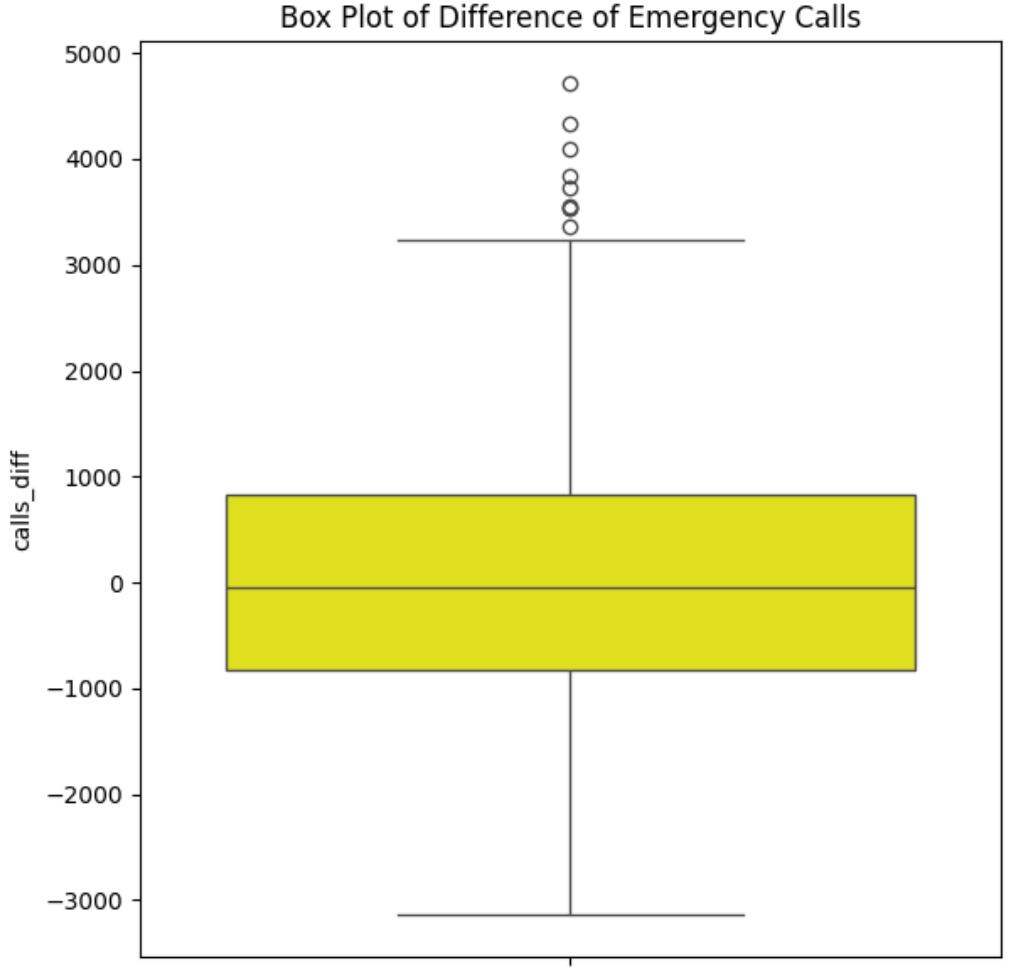

Rolling Mean & Difference of Drivers Sick
Rolling Mean of Drivers Sick
Difference of Drivers Sick
100
80
60
40
20
0
−20
−40
−60
2016-05
2016-09
2017-01
2017-05
2017-09
2018-01
2018-05
2018-09
2019-01
2019-05

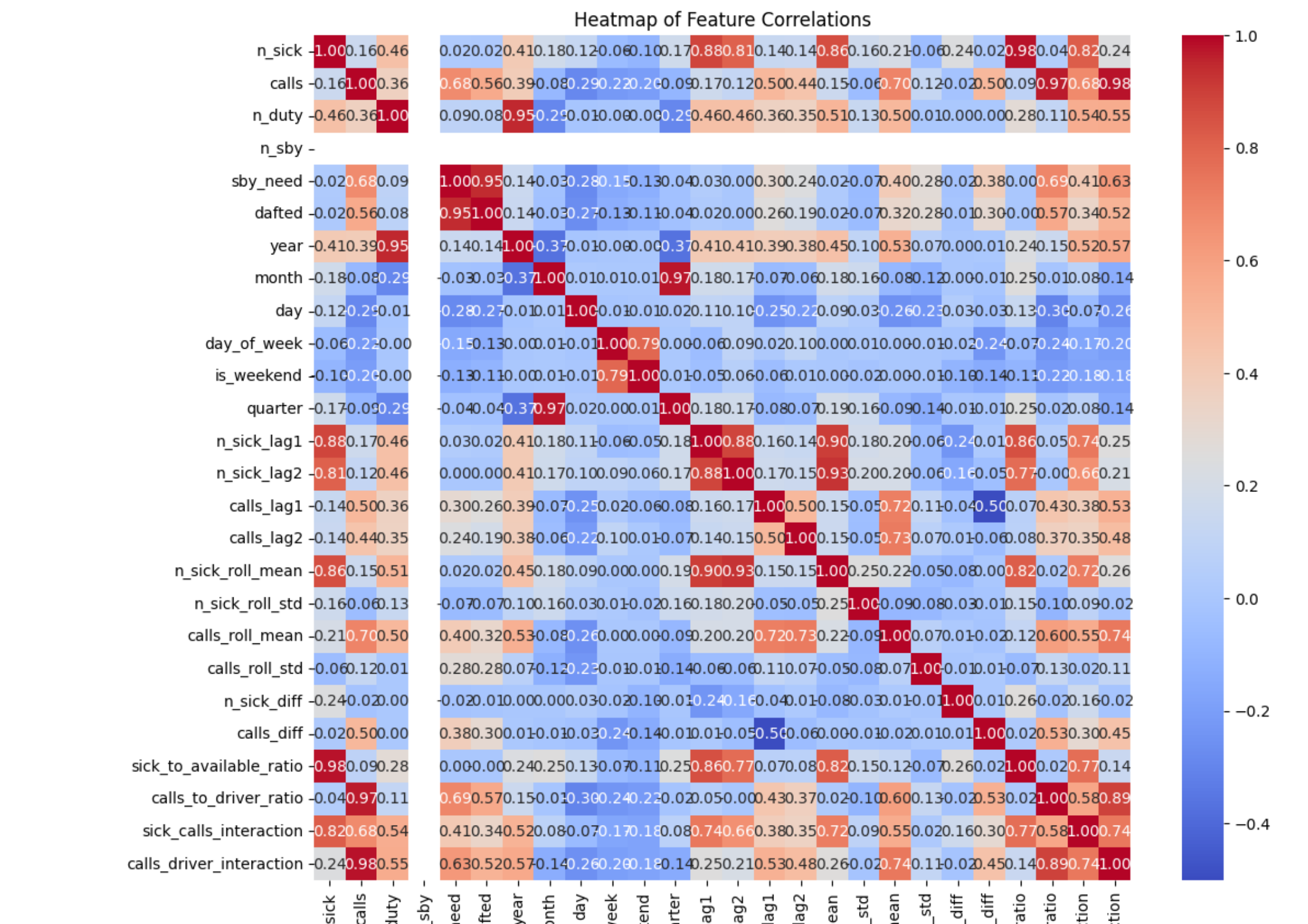

Rolling Mean & Difference of Emergency Calls
Rolling Mean of Calls
Difference of Calls
10000
8000
6000
4000
2000
0
−2000
2016-05
2016-09
2017-01
2017-05
2017-09
2018-01
2018-05
2018-09
2019-01
2019-05


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

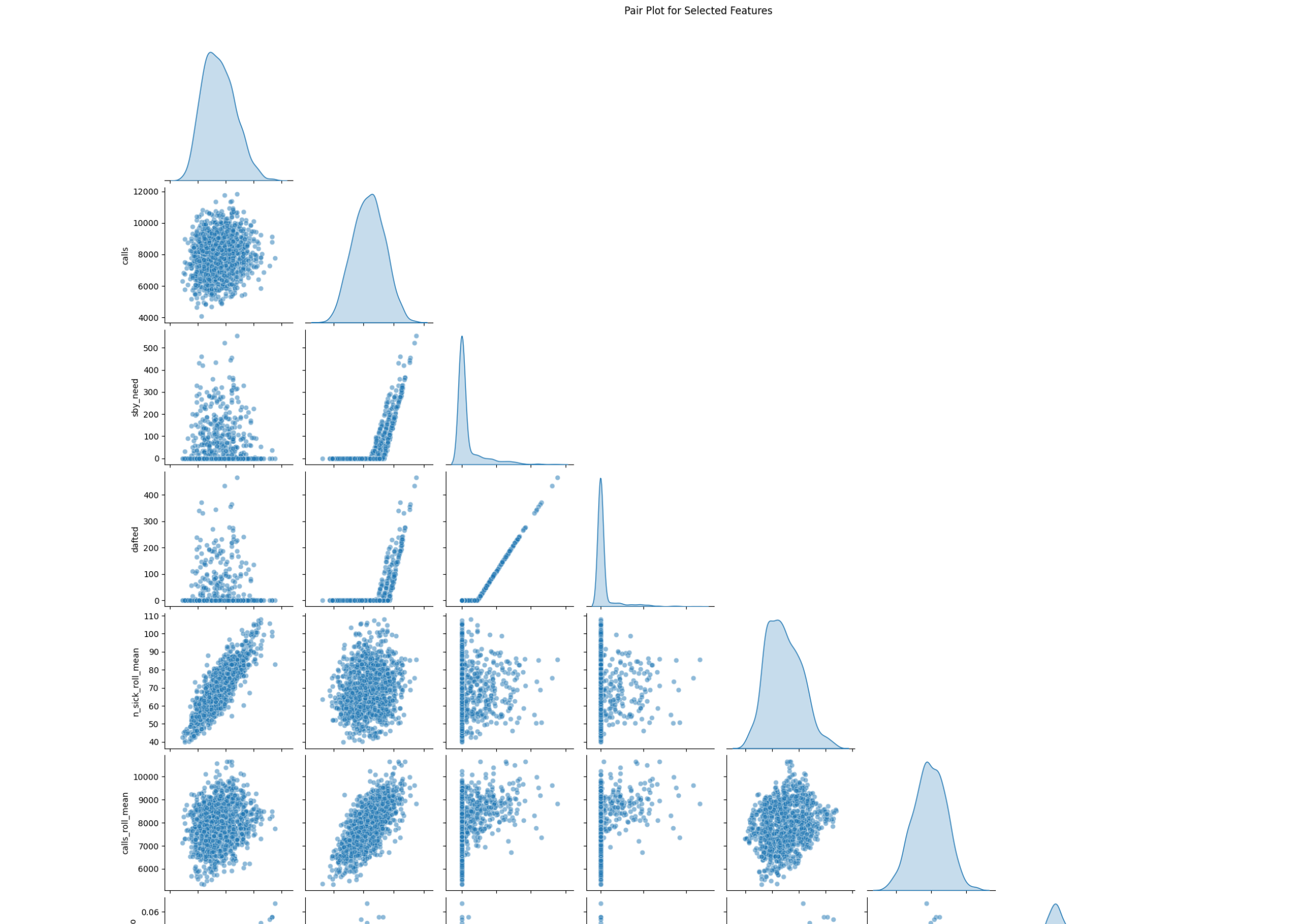

Box Plot of Rolling Mean of Drivers Sick
n_sick_roll_mean
110
100
90
80
70
60
50
40

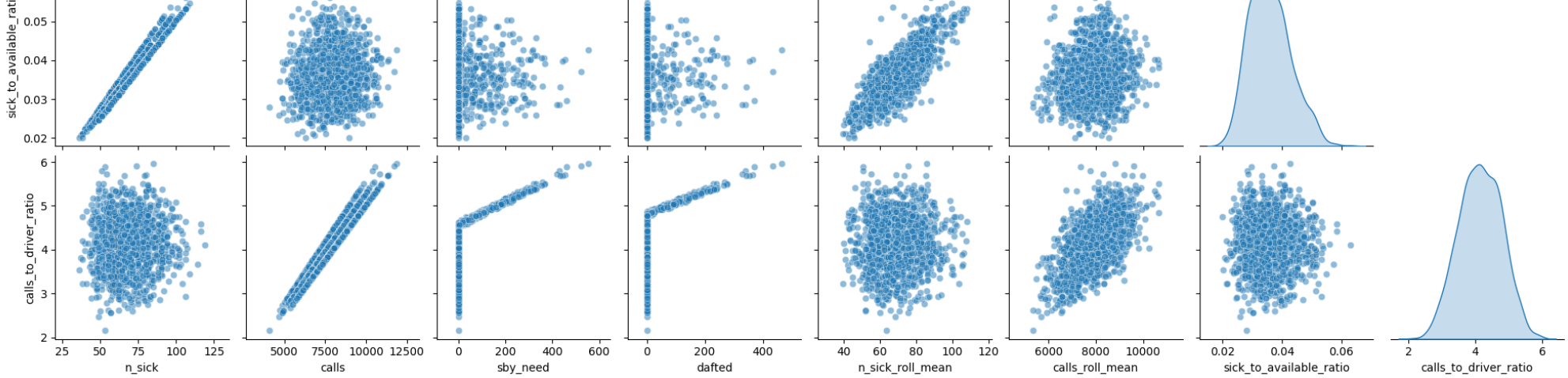

Box Plot of Difference of Emergency Calls
calls_diff
5000
4000
3000
2000
1000
0
−1000
−2000
−3000


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

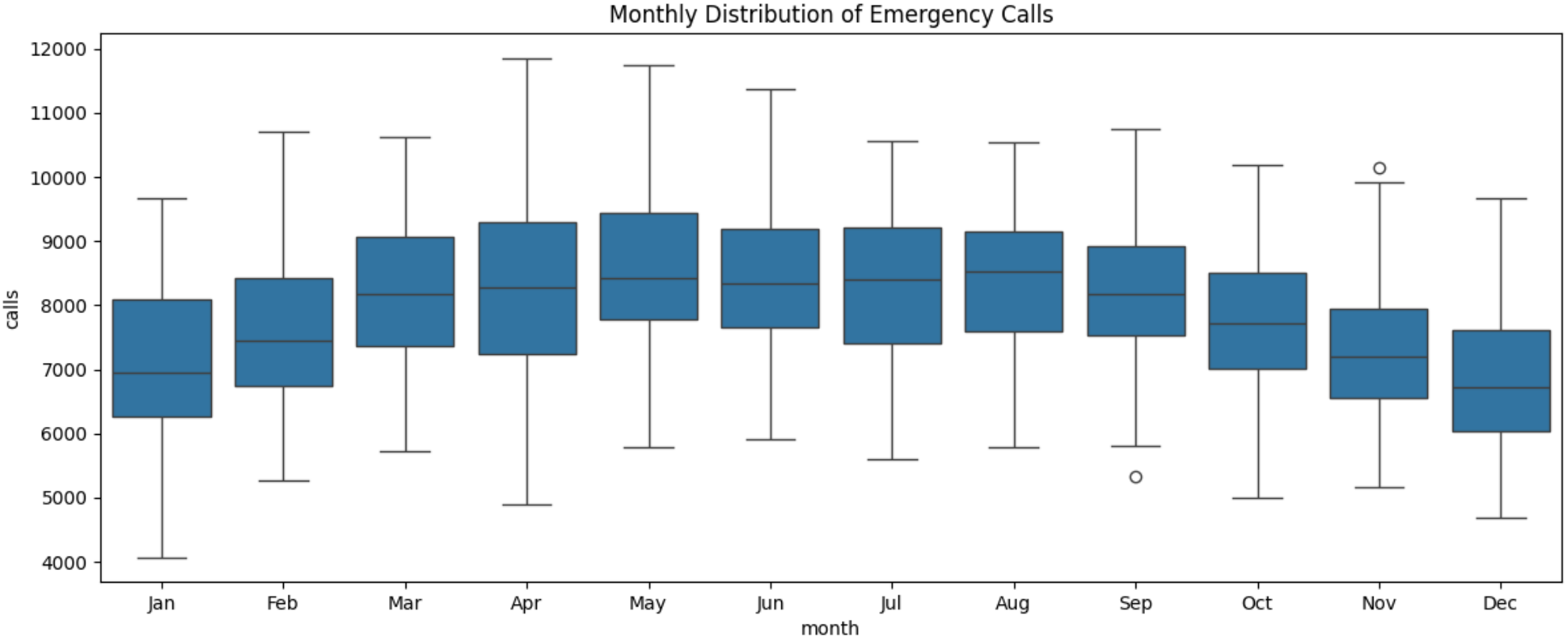


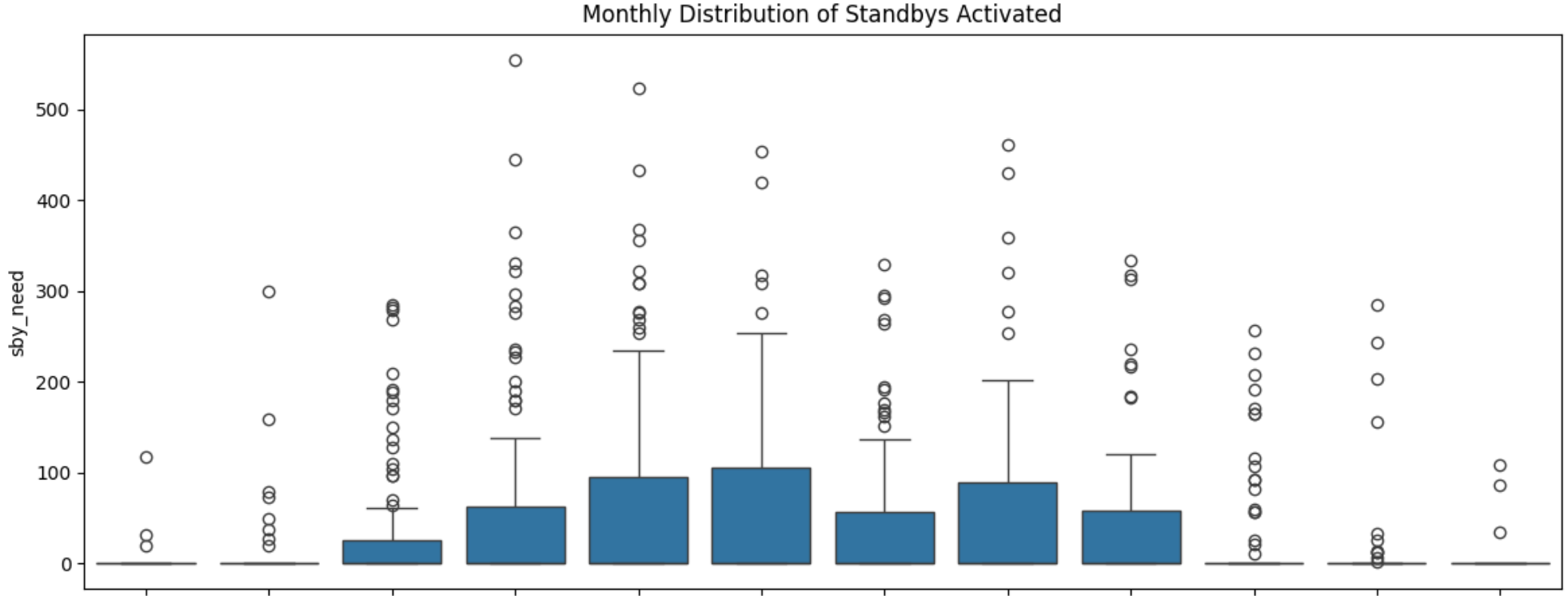
Monthly Analysis

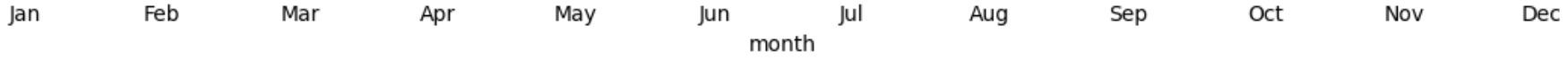
In [ ]:

```
# 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

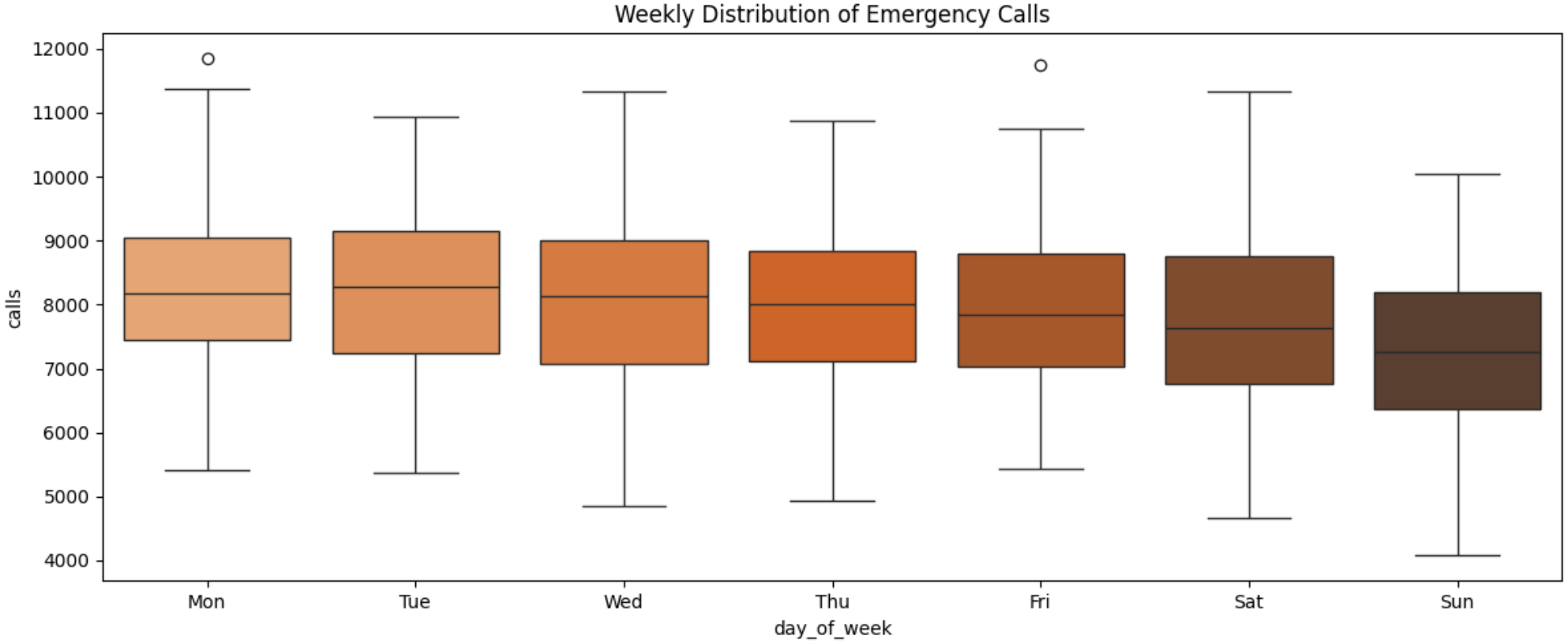

Monthly Distribution of Standbys Activated
sby_need
500
400
300
200
100
0

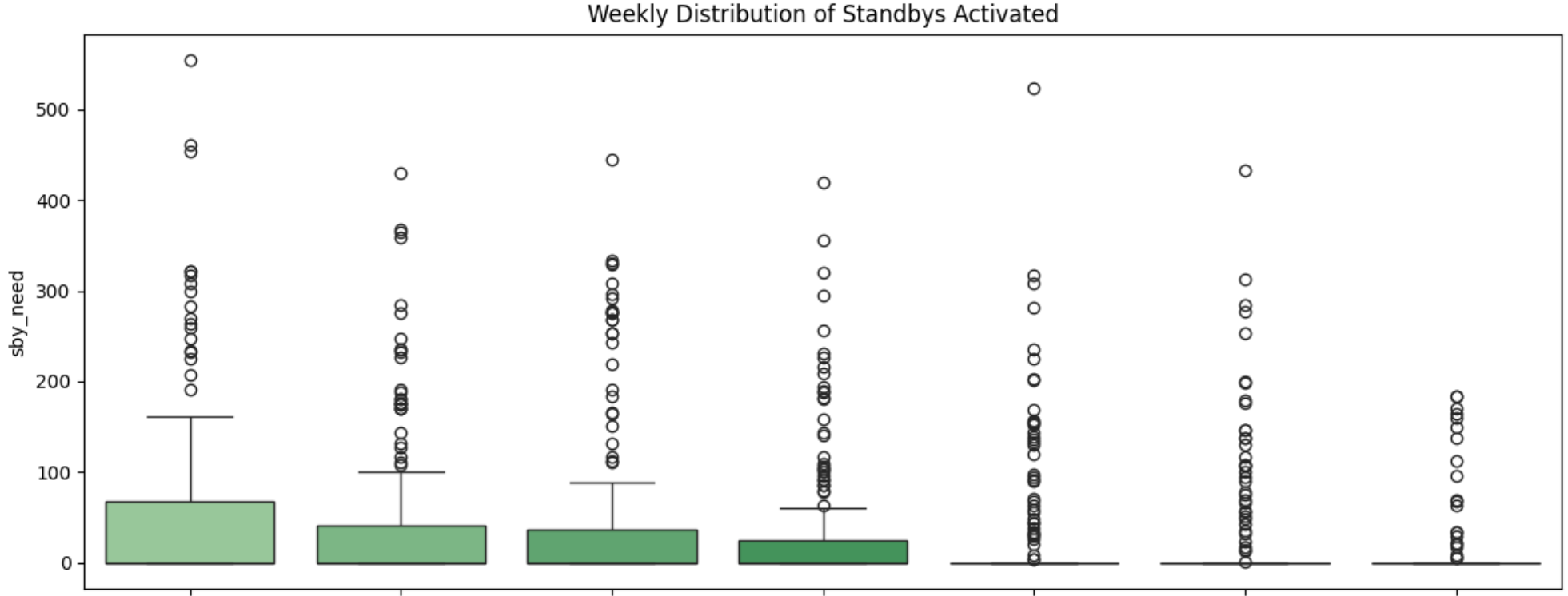


Weekly Analysis

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'])
```

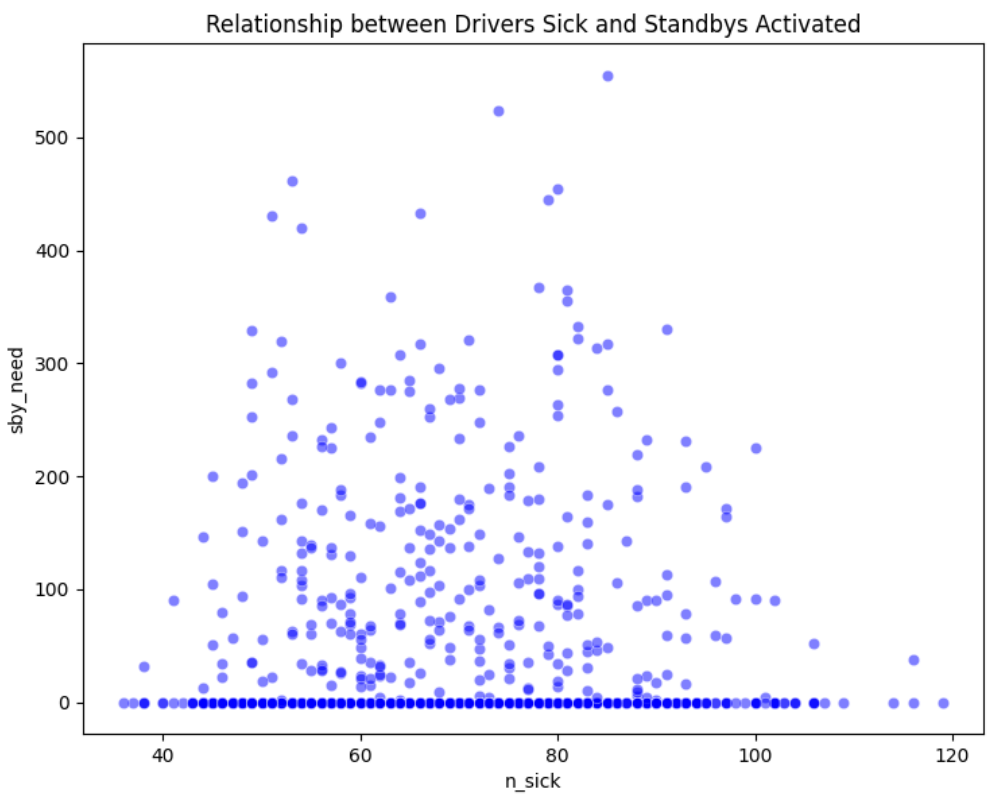

Weekly Distribution of Emergency Calls
calls
12000
11000
10000
9000
8000
7000
6000
5000
4000
Mon
Tue
Wed
Thu
Fri
Sat
Sun
day_of_week

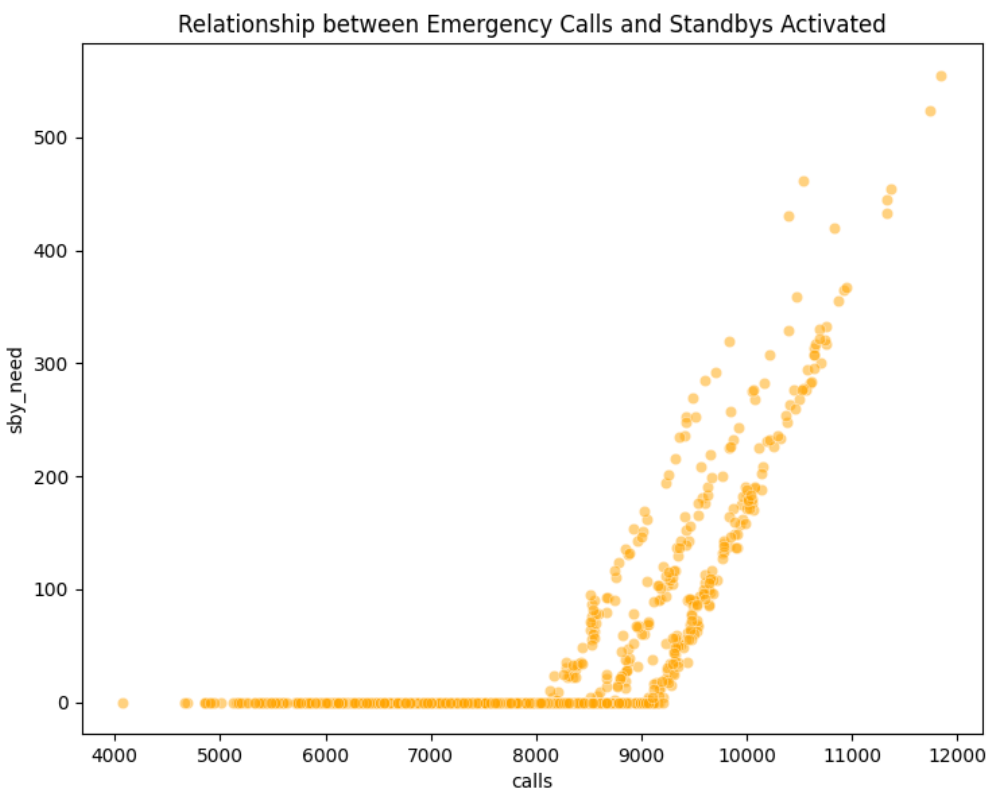

Weekly Distribution of Standbys Activated
sby_need
500
400
300
200
100
0

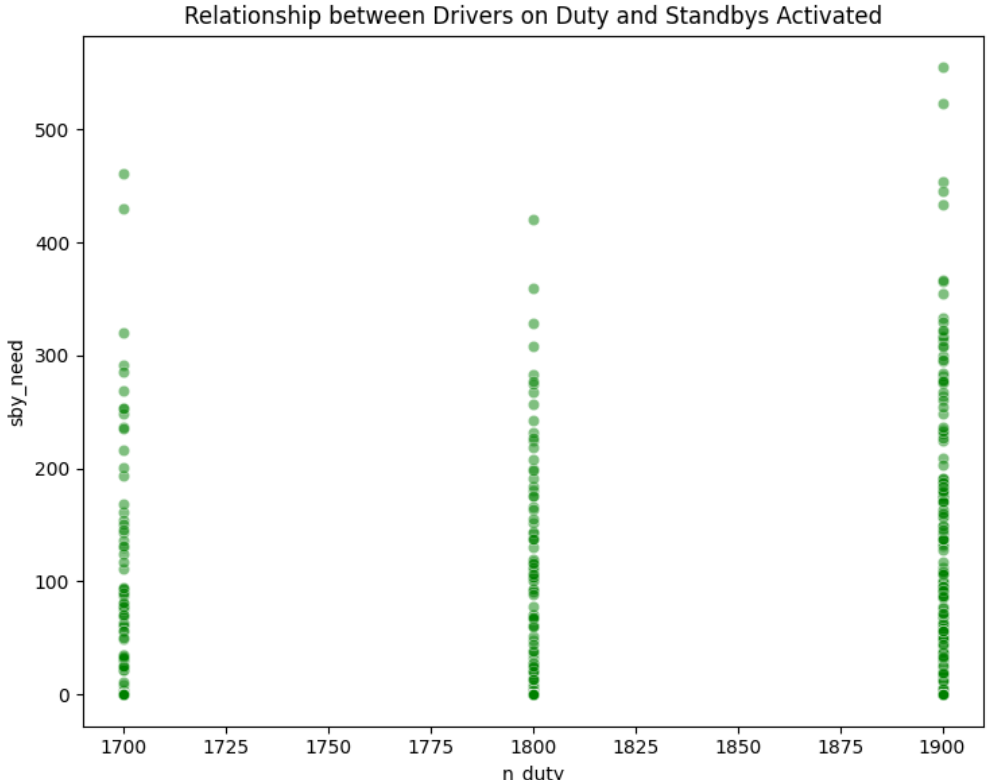
Mon Tue Wed Thu Fri Sat Sun
day_of_week

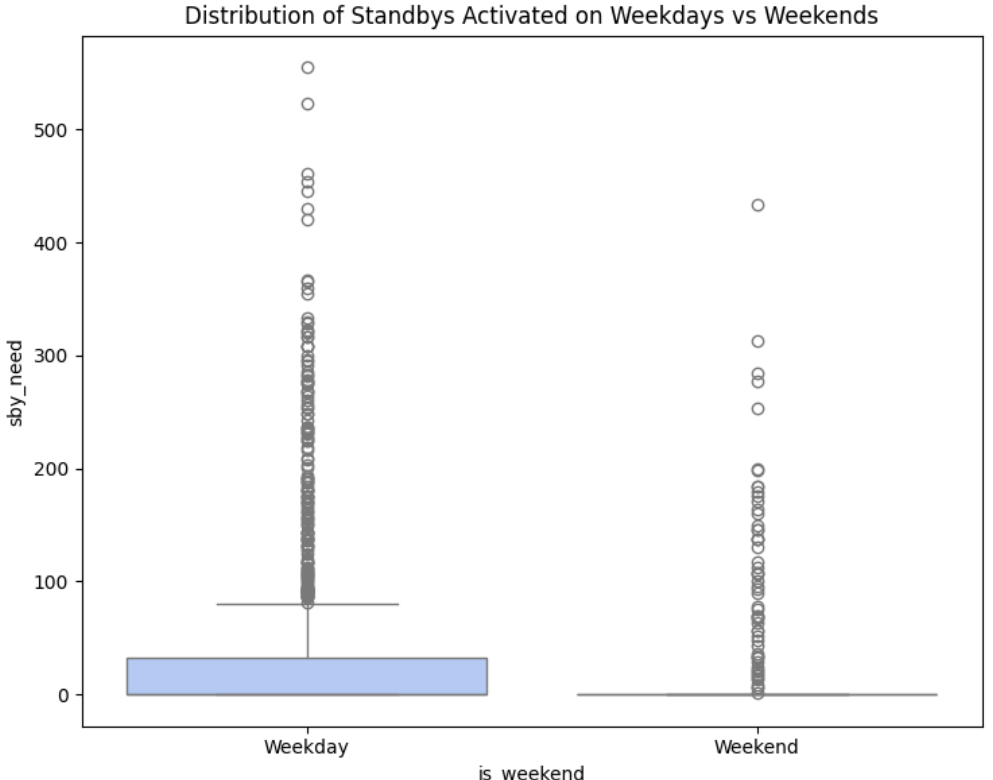
In [ ]:
```
# Set up the figure and axes
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15, 12))

# 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)
```

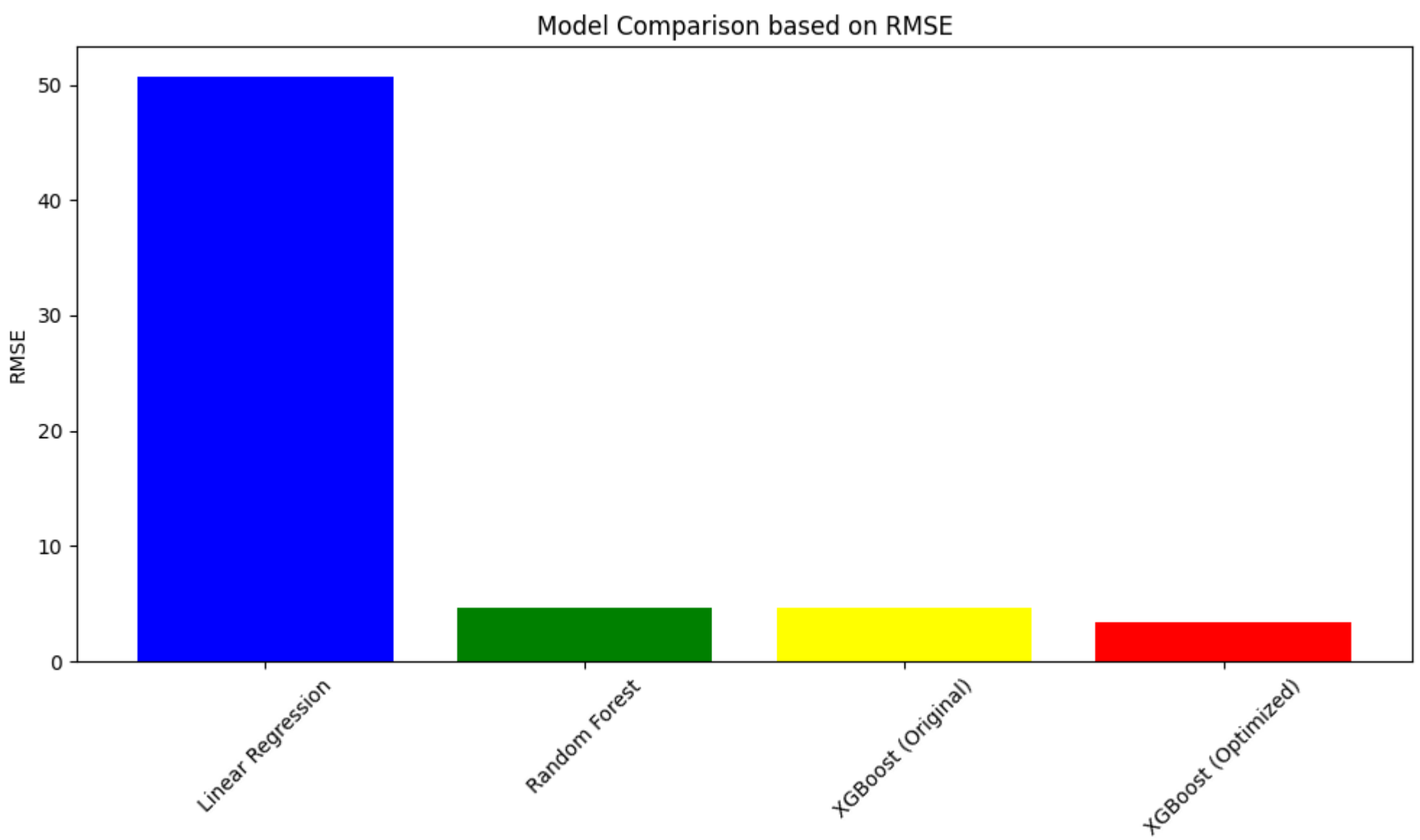
```
print(f"MAE for Optimized XGBoost Model: {mae_optimized_xgb:.4f}")
```

```
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()
```

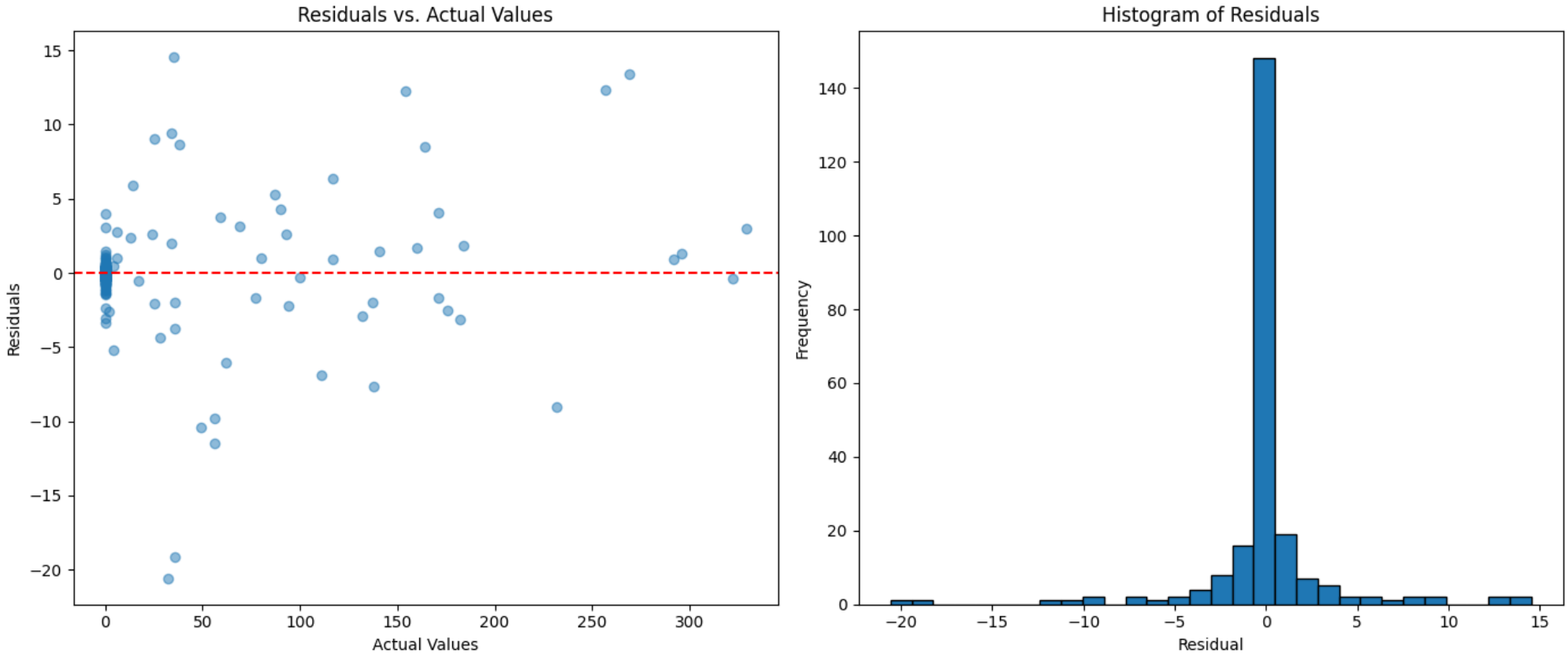


Error Analysis

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)
```

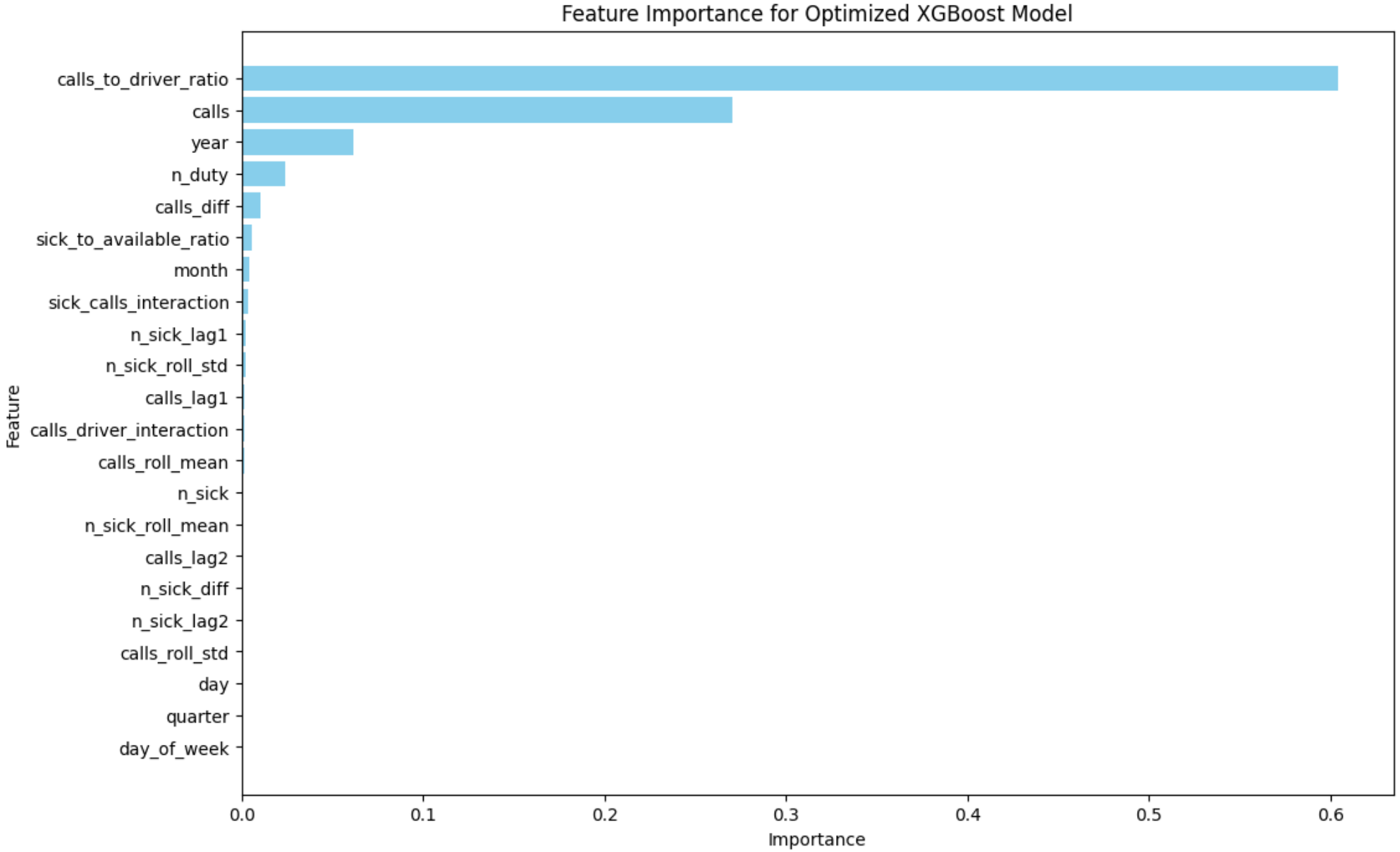
```
plt.scatter(y_test, residuals_xgb_optimized, alpha=0.5)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Actual Values')
plt.ylabel('Residuals')
plt.title('Residuals vs. Actual Values')

# 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()
```

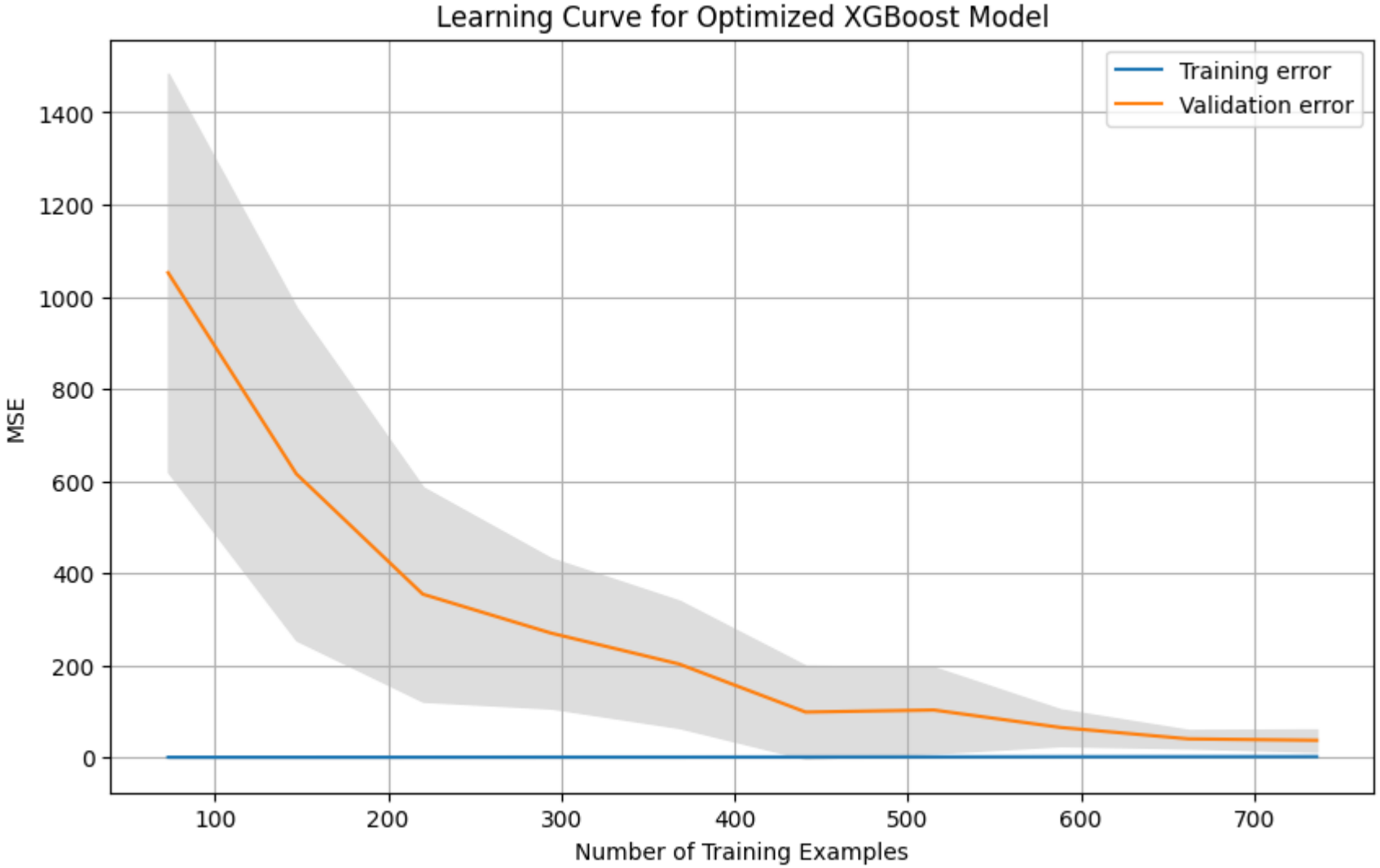


Learning Curve Analysis

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 [ ]: