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
```

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 Unwanted 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'])
```

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

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

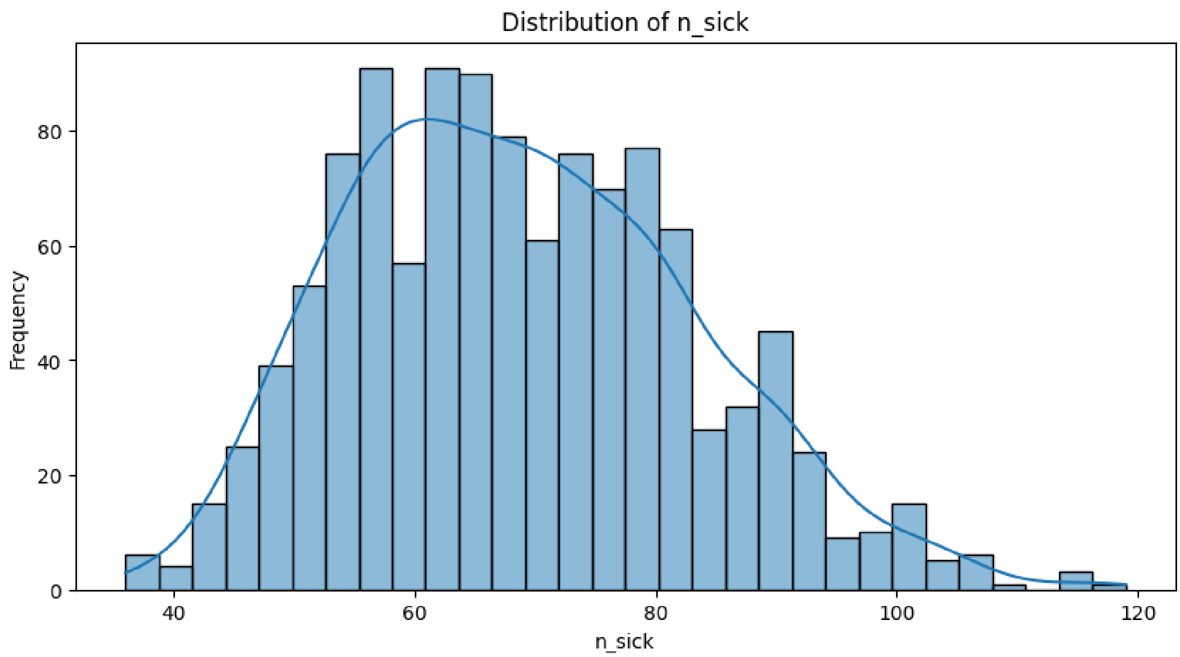
In [ ]:
```
sickness_data.isnull().sum()
```

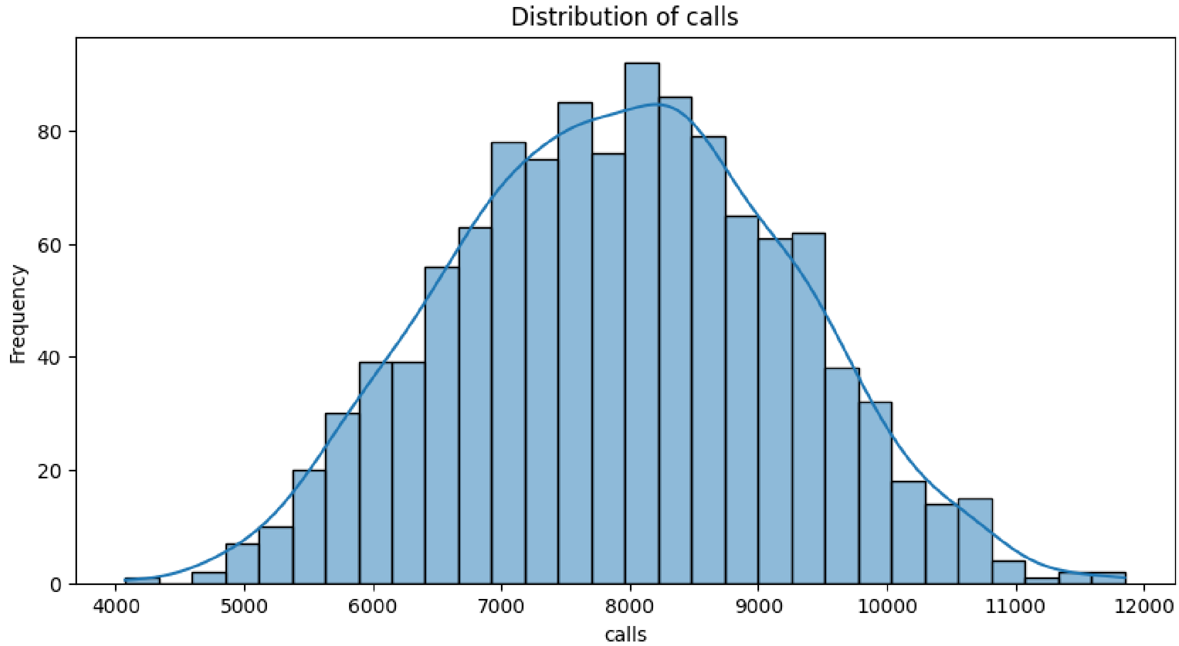
Out[ ]:
```
date        0
n_sick      0
calls       0
n_duty      0
n_sby       0
sby_need    0
dafted      0
dtype: int64
```

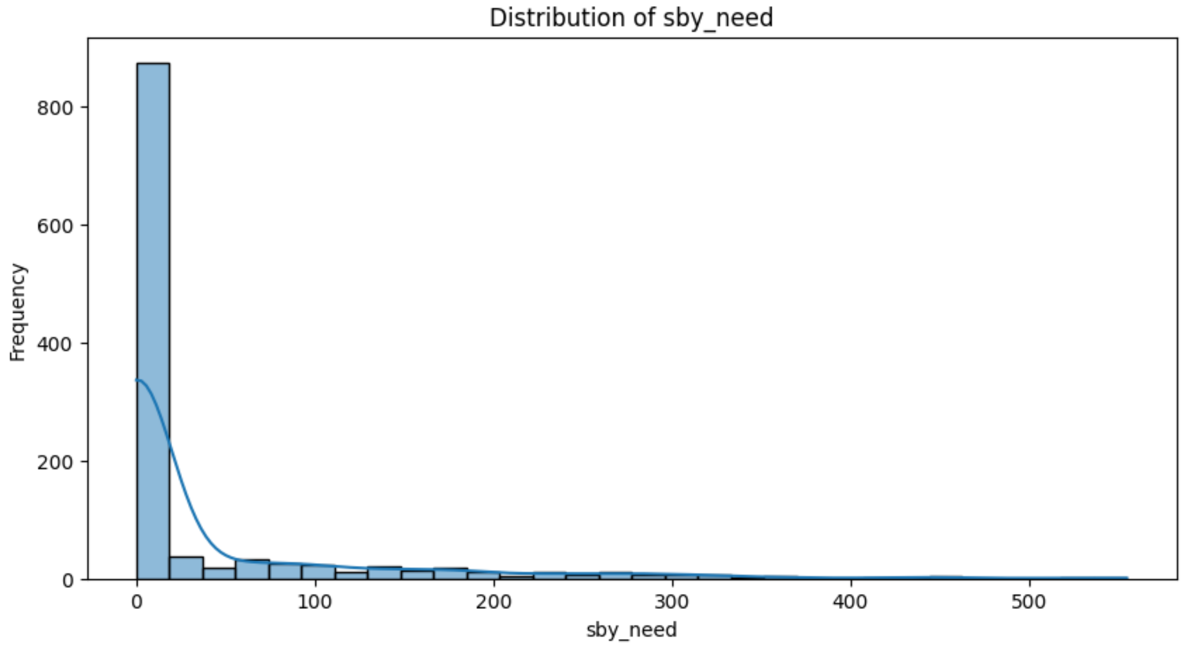
Data Exploration

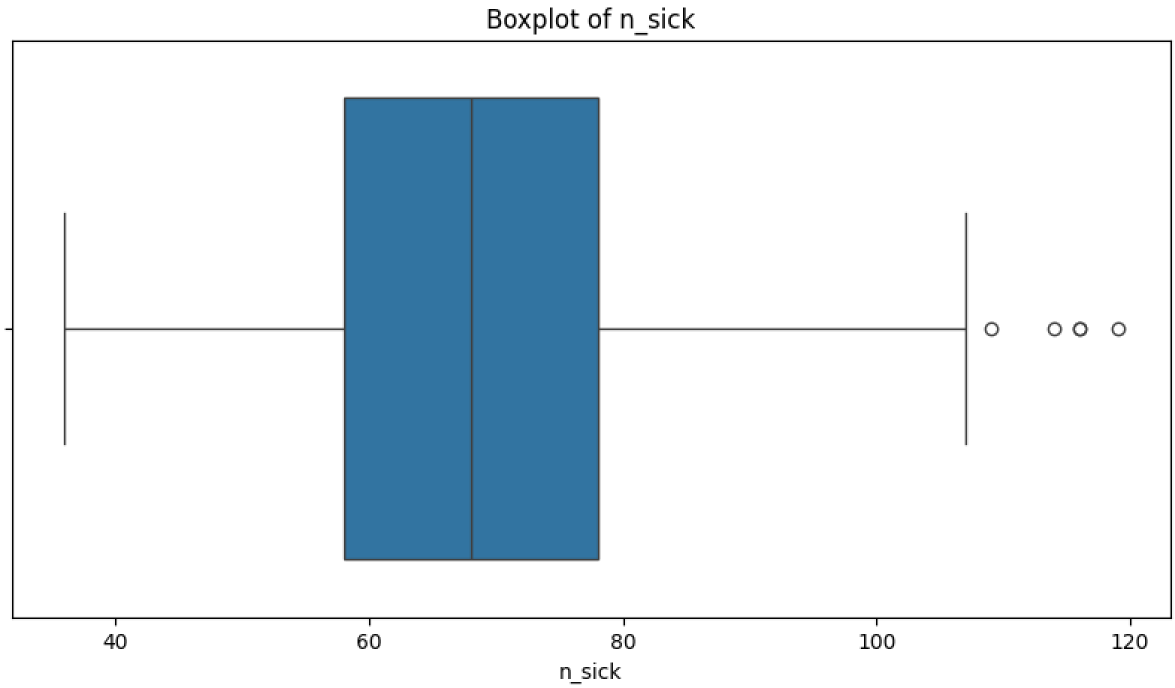
In [ ]:
```
# Columns of interest
columns_to_analyze = ['n_sick', 'calls', 'sby_need']

# Histograms for Distribution Analysis
for column in columns_to_analyze:
    plt.figure(figsize=(10, 5))
```

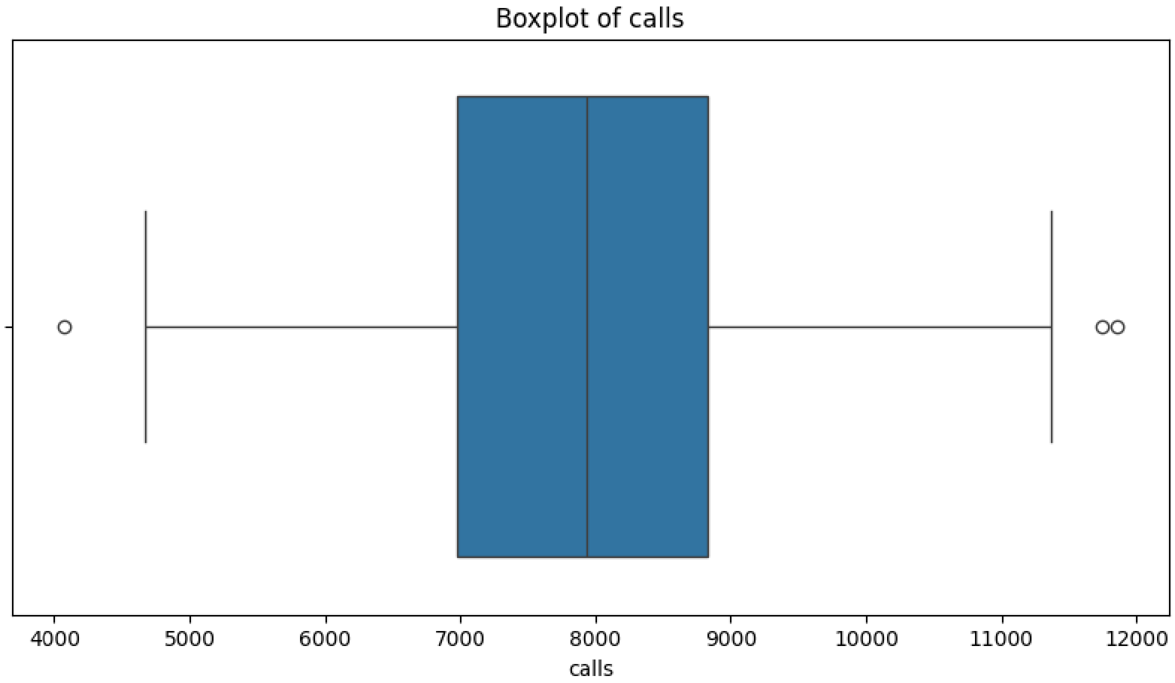
```
    sns.histplot(sickness_data[column], kde=True, bins=30)
    plt.title(f'Distribution of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.show()

# 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
    print(f"Number of outliers detected in {column}: {len(outliers)}")
```

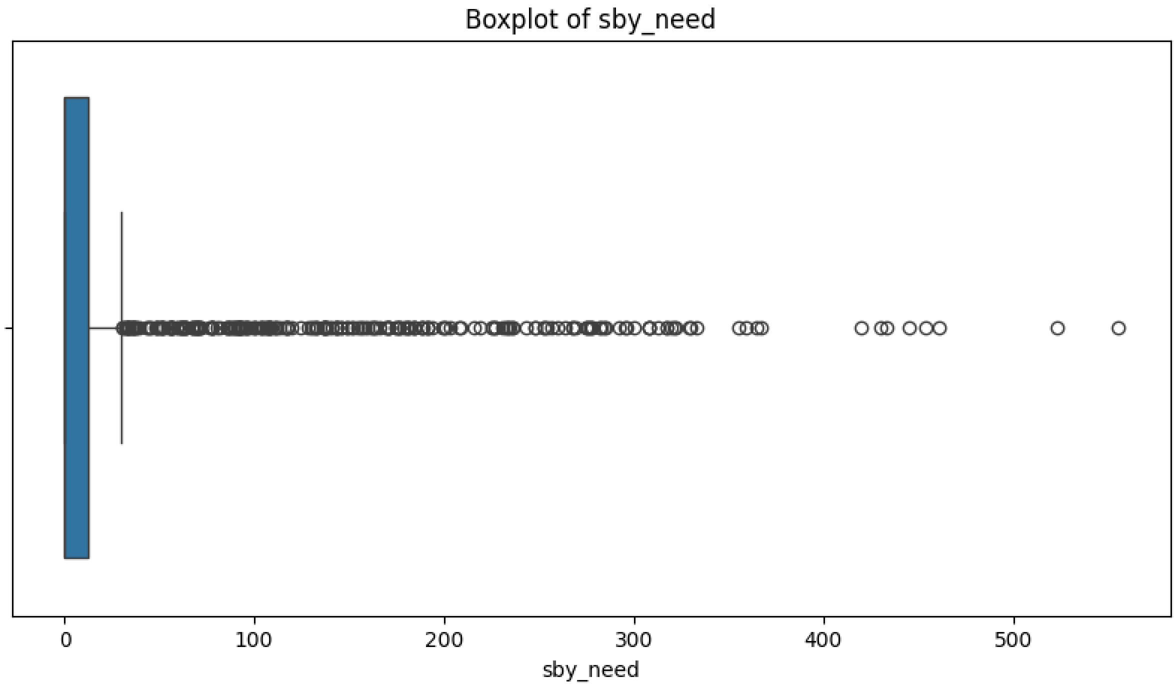



Distribution of calls
Frequency
0
20
40
60
80
4000
5000
6000
7000
8000
9000
10000
11000
12000
calls


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


Boxplot of n_sick
40
60
80
100
120
n_sick



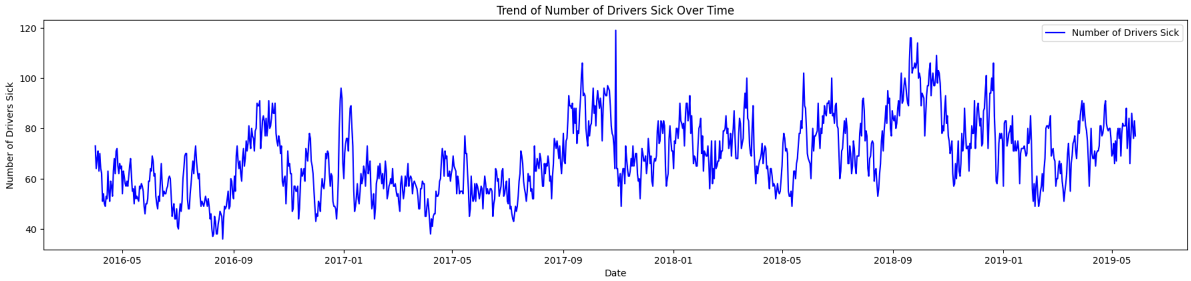



```
                 mean     50%          std
n_sick       68.80816    68.0    14.293942
calls      7919.53125  7932.0  1290.063571
sby_need     34.71875     0.0    79.694251
Number of outliers detected in n_sick: 5
Number of outliers detected in calls: 3
Number of outliers detected in sby_need: 256
```

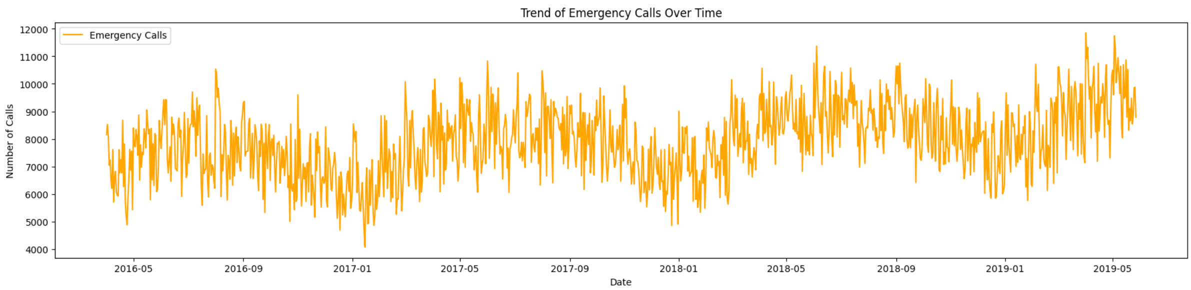
In [ ]:

```
# 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 S
plt.title('Trend of Number of Drivers Sick Over Time')
plt.xlabel('Date')
plt.ylabel('Number of Drivers Sick')
plt.legend()
```

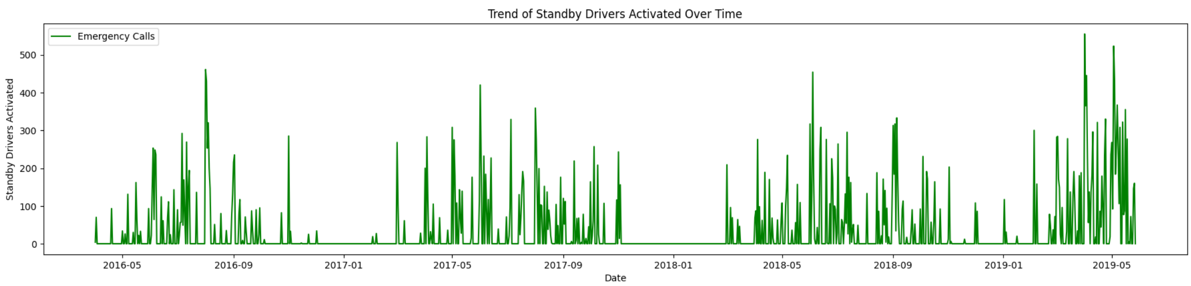
```
plt.tight_layout()
plt.show()
```



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', co
plt.title('Trend of Emergency Calls Over Time')
plt.xlabel('Date')
plt.ylabel('Number of Calls')
plt.legend()

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

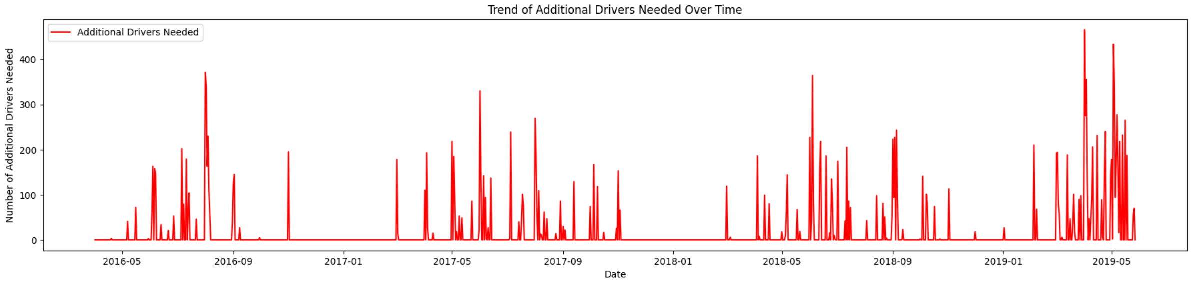


In [ ]:
```
# Time Series Analysis
plt.figure(figsize=(18, 12))

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

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



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
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()
```



In [ ]:

```
sickness_data.corr()
```

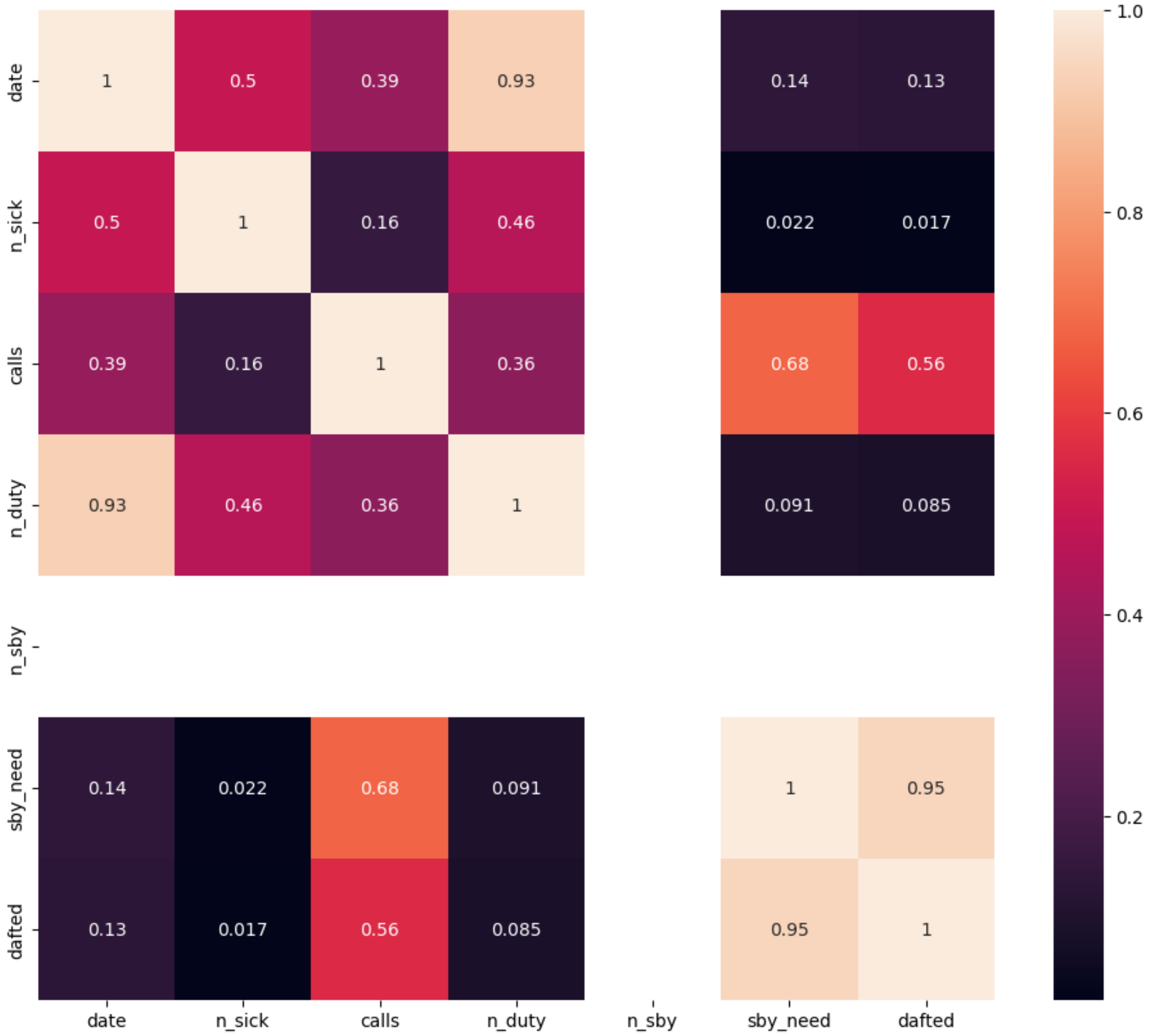
Out[ ]:

	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: >



Feature Engineering

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 >

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

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)
```

In [ ]:
```
# Create rolling mean and standard deviation for 'n_sick' and 'calls' over a 7-day
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()
```

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()
```

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

# Emergency Call to Driver Ratio: Ratio of emergency calls to the total number of d
sickness_data['calls_to_driver_ratio'] = sickness_data['calls'] / (sickness_data['r
```

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

# Interaction between the number of emergency calls and available drivers (both on
sickness_data['calls_driver_interaction'] = sickness_data['calls'] * (sickness_data
```

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_we
0	2016-04-01	73	8154.0	1700	90	4.0	0.0	2016	4	1	4	
1	2016-04-02	64	8526.0	1700	90	70.0	0.0	2016	4	2	5	
2	2016-04-03	68	8088.0	1700	90	0.0	0.0	2016	4	3	6	
3	2016-04-04	71	7044.0	1700	90	0.0	0.0	2016	4	4	0	
4	2016-04-05	63	7236.0	1700	90	0.0	0.0	2016	4	5	1	
5	2016-04-06	70	6492.0	1700	90	0.0	0.0	2016	4	6	2	
6	2016-04-07	64	6204.0	1700	90	0.0	0.0	2016	4	7	3	
7	2016-04-08	62	7614.0	1700	90	0.0	0.0	2016	4	8	4	
8	2016-04-09	51	5706.0	1700	90	0.0	0.0	2016	4	9	5	
9	2016-04-10	54	6606.0	1700	90	0.0	0.0	2016	4	10	6	

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'
]
```

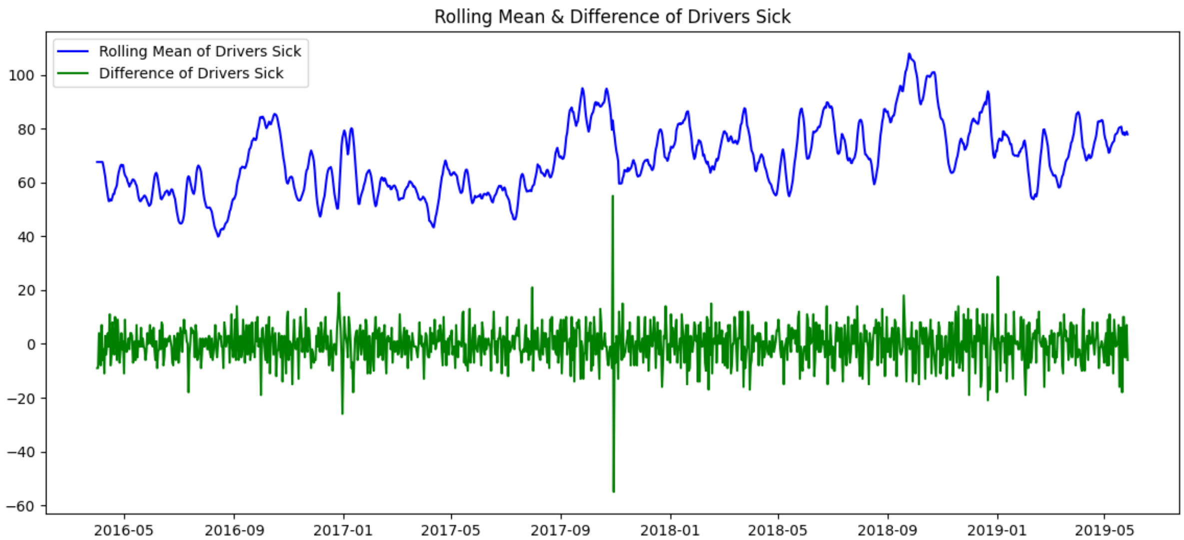
```
# Apply both forward fill and backward fill for these columns
for column in columns_to_ffill:
    sickness_data[column] = sickness_data[column].ffill().bfill()


# 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()


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

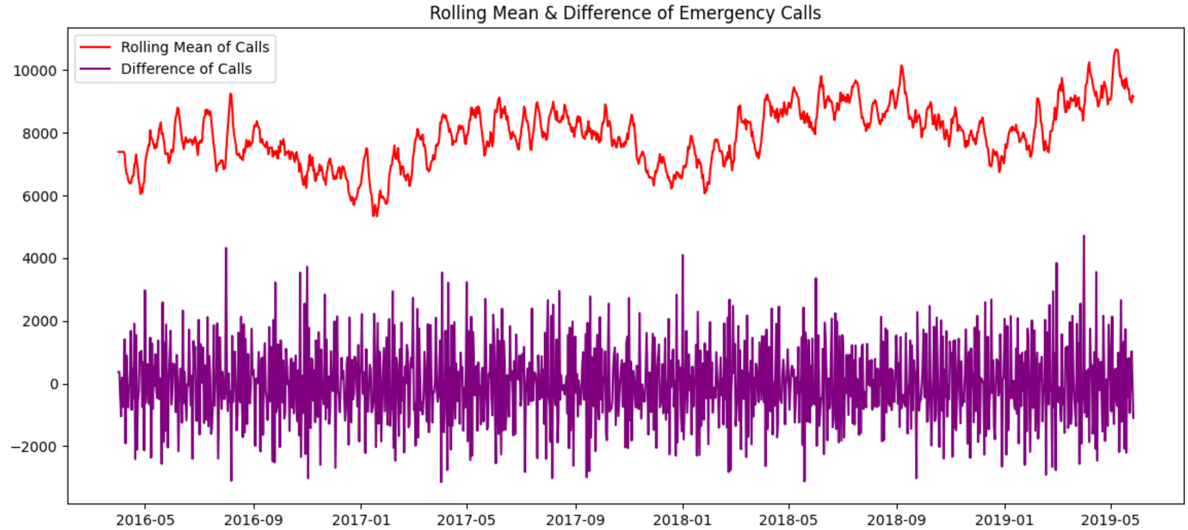
```
date                        0
n_sick                      0
calls                       0
n_duty                      0
n_sby                       0
sby_need                    0
dafted                      0
year                        0
month                       0
day                       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

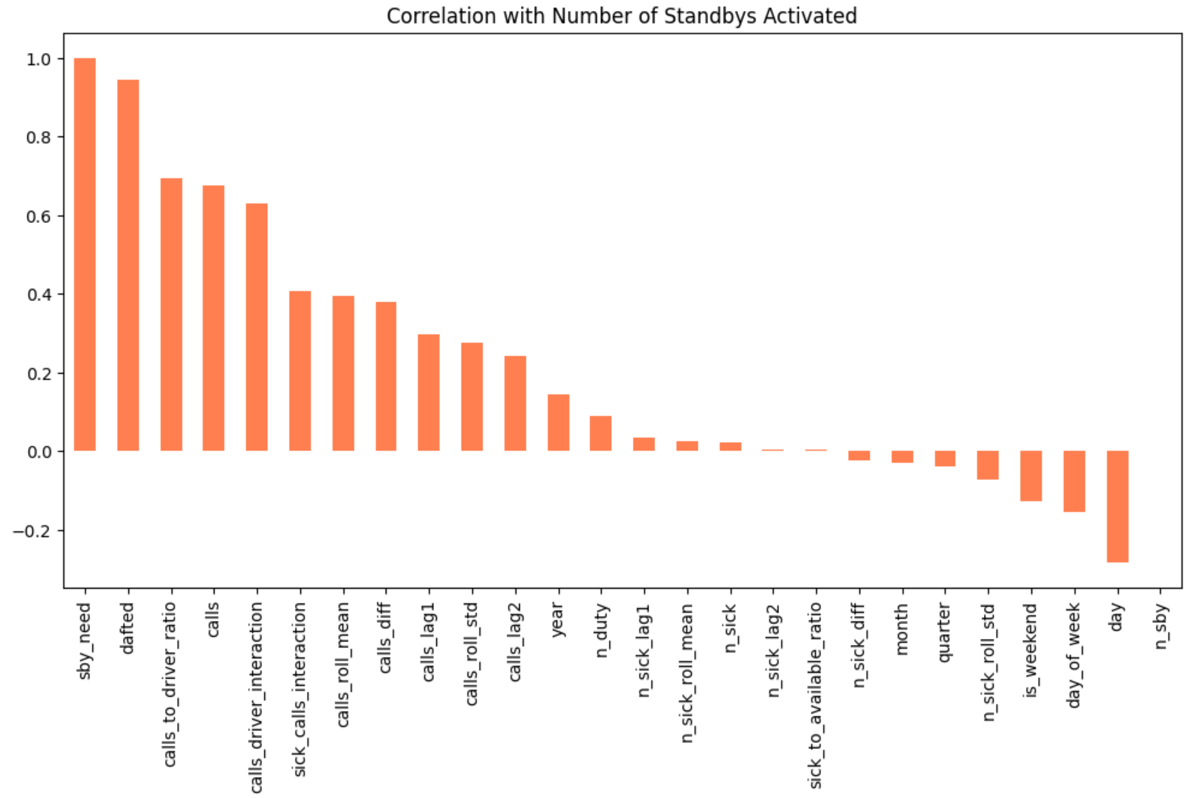
In [ ]:

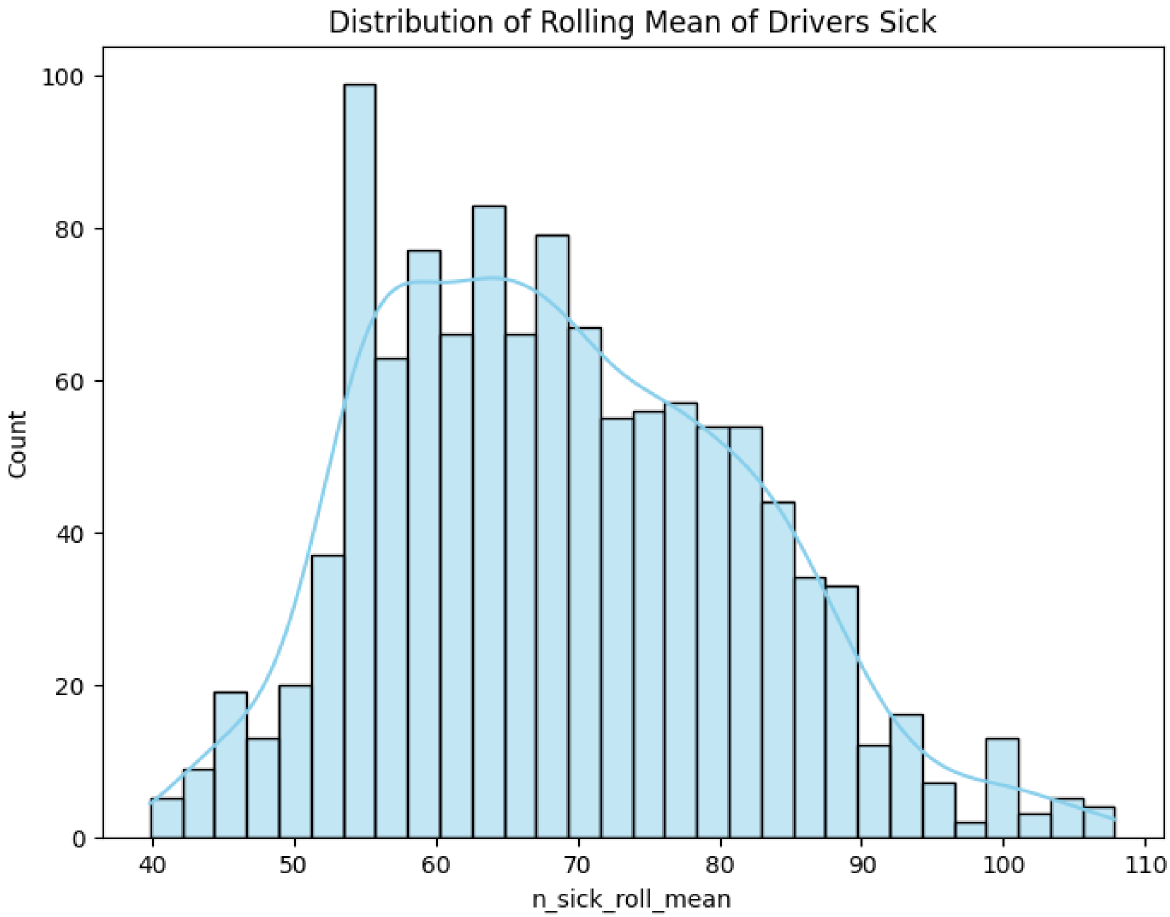
```
# 1. Temporal Visualization of New Features:
plt.figure(figsize=(14, 6))
plt.plot(sickness_data['date'], sickness_data['n_sick_roll_mean'], label='Rolling M
plt.plot(sickness_data['date'], sickness_data['n_sick_diff'], label='Difference of
plt.title('Rolling Mean & Difference of Drivers Sick')
plt.legend()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

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

# 2. Correlation Analysis:
```

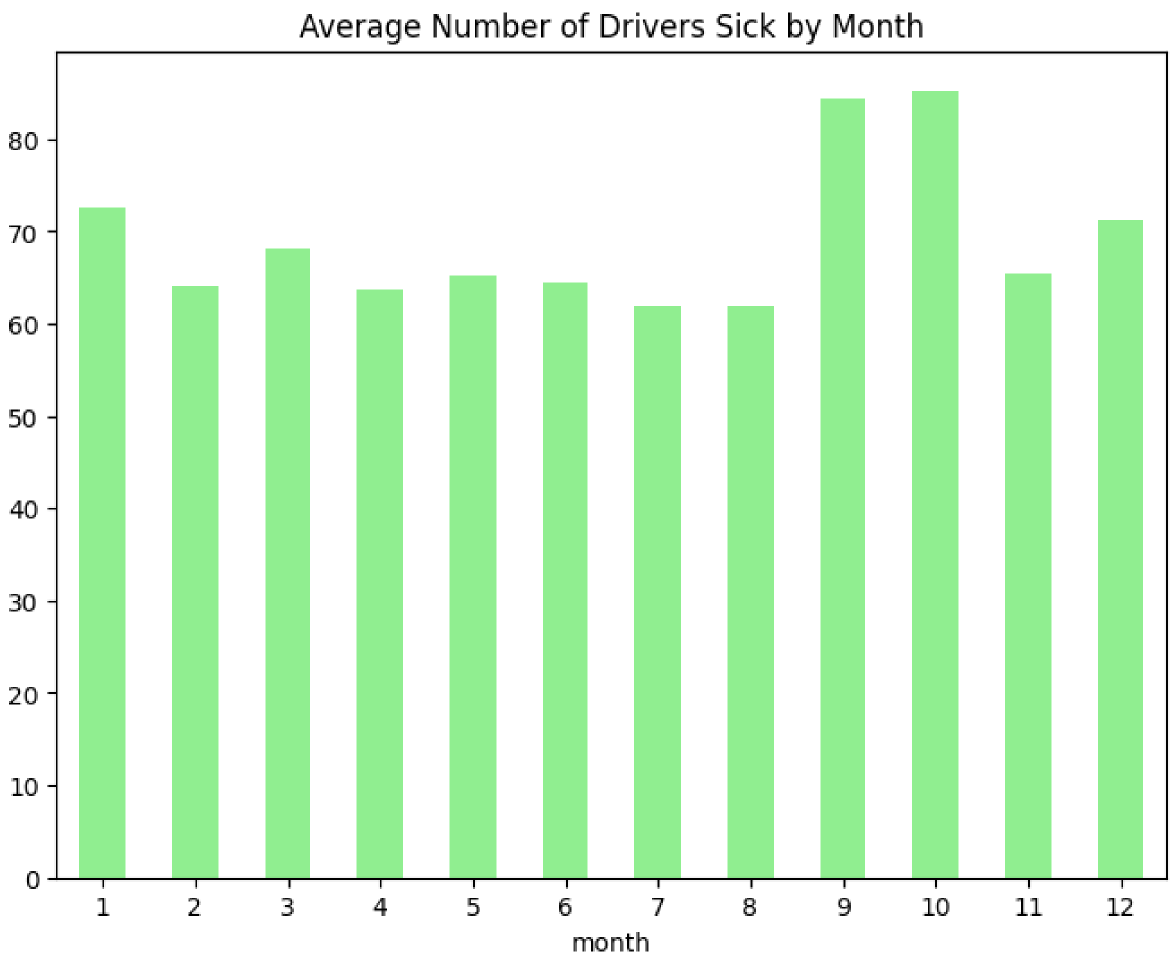
```
correlations = sickness_data.drop(columns=['date']).corr()['sby_need'].sort_values(
plt.figure(figsize=(12, 6))
correlations.plot(kind='bar', color='coral')
plt.title('Correlation with Number of Standbys Activated')
plt.show()

# 3. Distribution Analysis:
plt.figure(figsize=(8, 6))
sns.histplot(sickness_data['n_sick_roll_mean'], kde=True, color='skyblue', bins=30)
plt.title('Distribution of Rolling Mean of Drivers Sick')
plt.show()

# 4. 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()

# 5. Weekday vs. Weekend Analysis:
plt.figure(figsize=(8, 6))
weekday_avg_sick = sickness_data.groupby('day_of_week')['n_sick'].mean()
weekday_avg_sick.plot(kind='bar', color='lightblue')
plt.title('Average Number of Drivers Sick by Day of Week')
plt.xticks(ticks=range(7), labels=['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
plt.show()

# 6. 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()

# 7. 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()
```

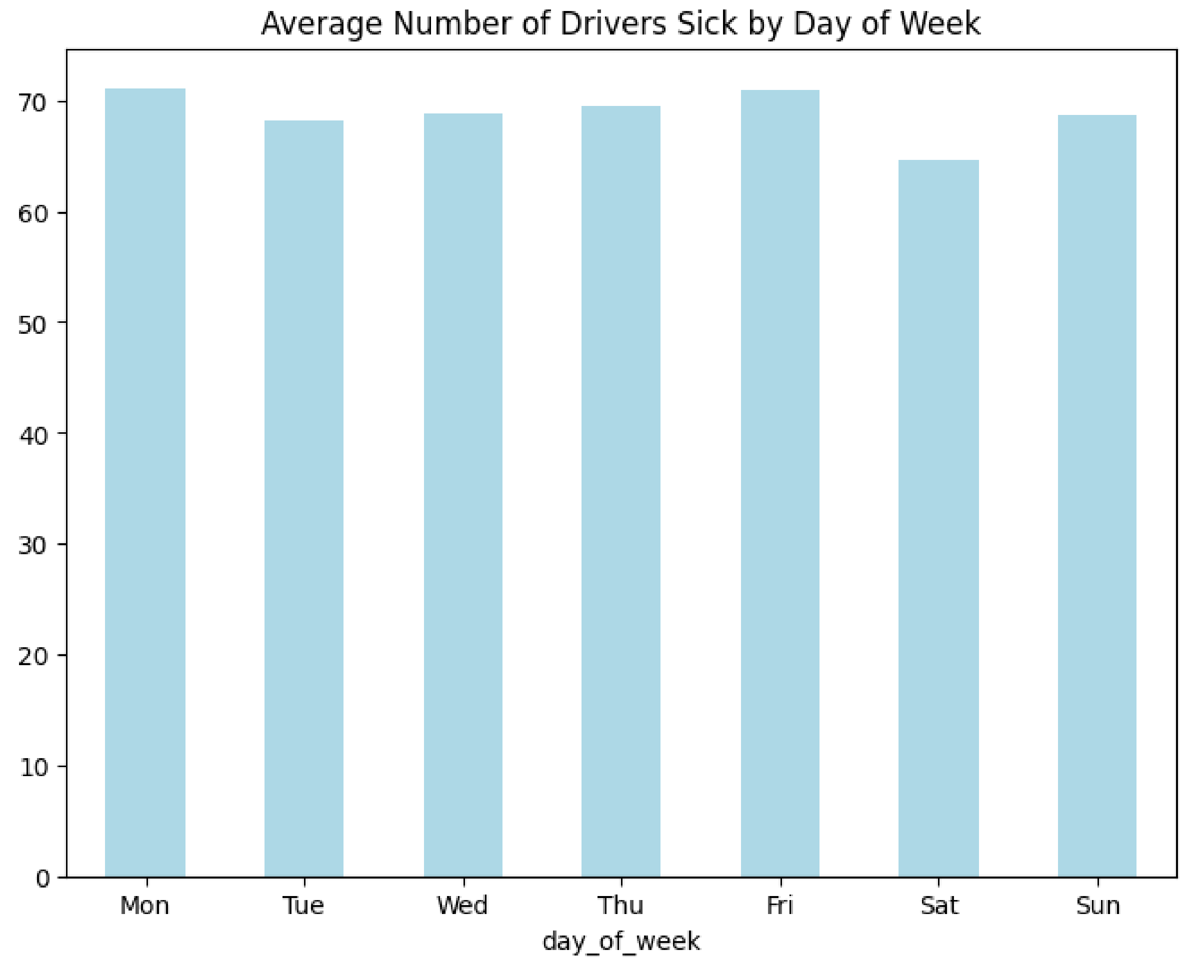


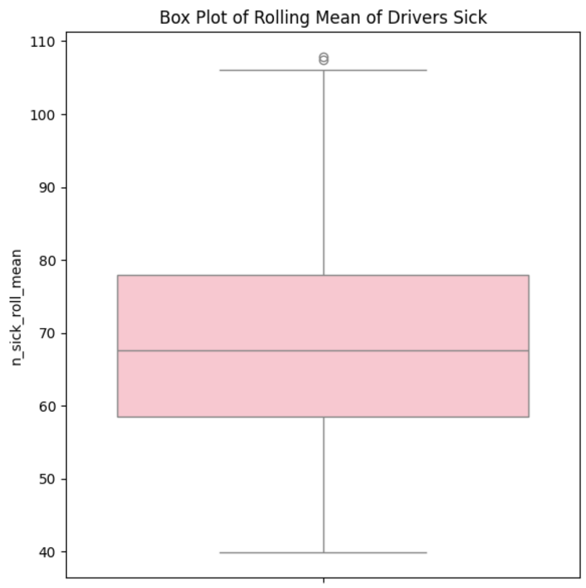

Rolling Mean & Difference of Emergency Calls
Rolling Mean of Calls
Difference of Calls
10000
8000
6000
4000
2000
0
−2000
2016-05
2016-09
2017-01
2017-05
2017-09
2018-01
2018-05
2018-09
2019-01
2019-05

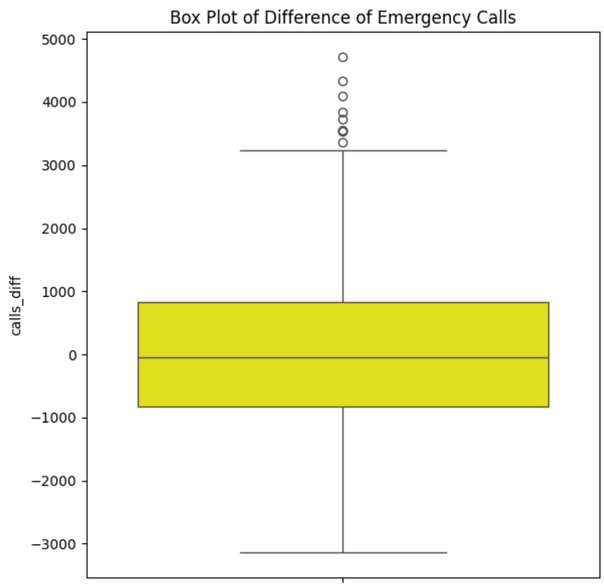

Correlation with Number of Standbys Activated
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
sby_need
dafted
calls_to_driver_ratio
calls
calls_driver_interaction
sick_calls_interaction
calls_roll_mean
calls_diff
calls_lag1
calls_roll_std
calls_lag2
year
n_duty
n_sick_lag1
n_sick_roll_mean
n_sick
n_sick_lag2
sick_to_available_ratio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

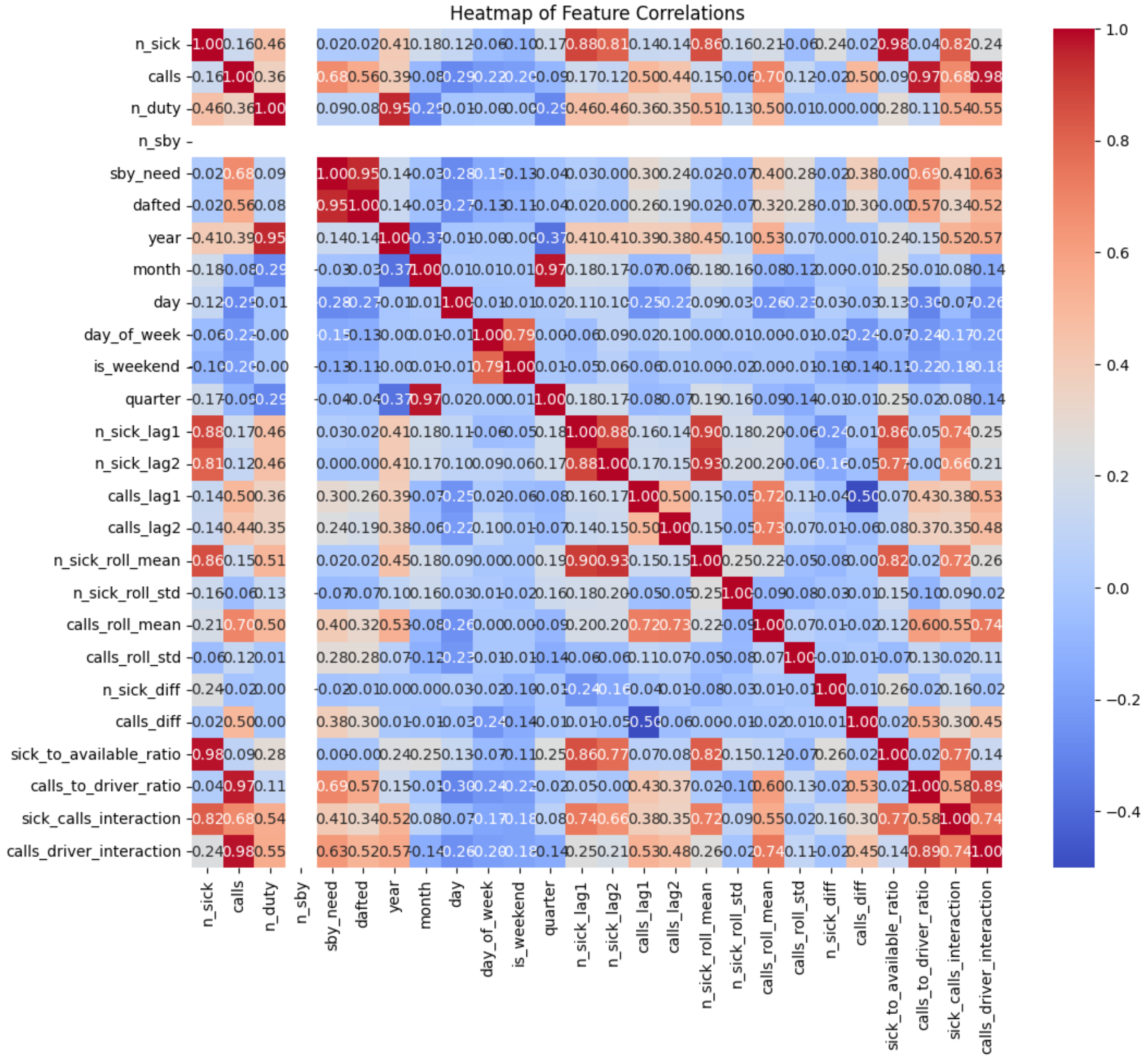

Distribution of Rolling Mean of Drivers Sick
Count
0
20
40
60
80
100
40
50
60
70
80
90
100
110
n_sick_roll_mean


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


Average Number of Drivers Sick by Day of Week
0
10
20
30
40
50
60
70
Mon
Tue
Wed
Thu
Fri
Sat
Sun
day_of_week


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

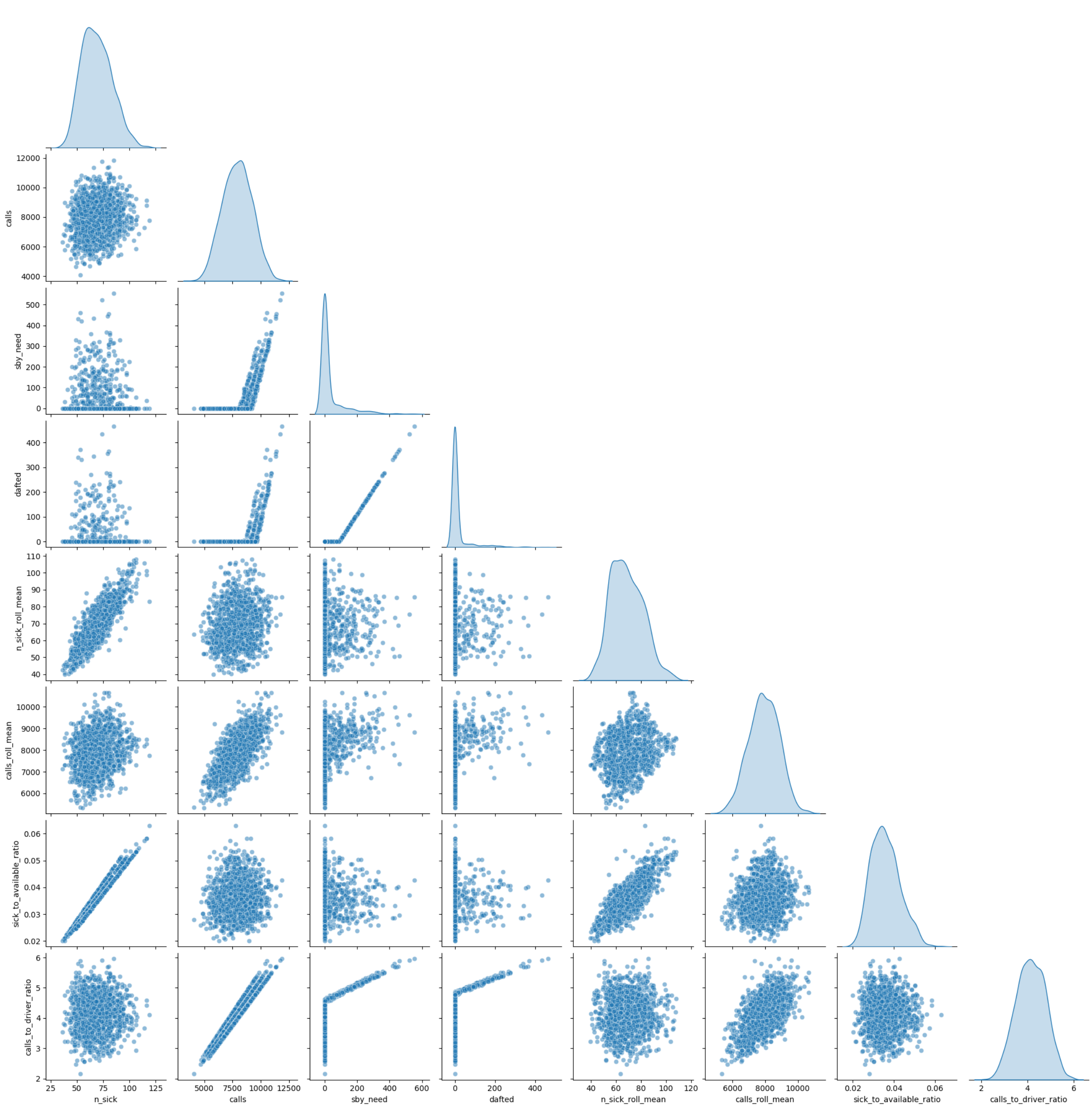

Box Plot of Difference of Emergency Calls
calls_diff
−3000
−2000
−1000
0
1000
2000
3000
4000
5000



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_

# 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={'a
plt.suptitle('Pair Plot for Selected Features', y=1.02)
plt.show()
```

Pair Plot for Selected Features



Linear Regression Model

In [ ]:

```
# Define features and target variable
X = sickness_data.drop(columns=['date', 'sby_need'])
y = sickness_data['sby_need']

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Initialize and train the baseline model (Linear Regression)
baseline_model = LinearRegression()
baseline_model.fit(X_train, y_train)

# Predict on the training set
y_pred_train = baseline_model.predict(X_train)

# Evaluate the baseline model's performance on training data
mae_train = mean_absolute_error(y_train, y_pred_train)
mse_train = mean_squared_error(y_train, y_pred_train)
rmse_train = mean_squared_error(y_train, y_pred_train, squared=False)
r2_train = r2_score(y_train, y_pred_train)


# Predict on the test set
```

```
y_pred = baseline_model.predict(X_test)

# Evaluate the baseline model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

print("Performance on Training Data:")
print(f"Mean Absolute Error (MAE): {mae_train}")
print(f"Mean Squared Error (MSE): {mse_train}")
print(f"Root Mean Squared Error (RMSE): {rmse_train}")
print(f"R^2 (Coefficient of Determination): {r2_train}")
print("\nPerformance on Test Data:")
print(f"Mean Absolute Error (MAE): {mae}")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"R^2 (Coefficient of Determination): {r2}")
```

```
Performance on Training Data:
Mean Absolute Error (MAE): 16.86249271405271
Mean Squared Error (MSE): 445.7586201349591
Root Mean Squared Error (RMSE): 21.11299647456417
R^2 (Coefficient of Determination): 0.9359094318861493

Performance on Test Data:
Mean Absolute Error (MAE): 15.959891029202183
Mean Squared Error (MSE): 397.75521259023304
Root Mean Squared Error (RMSE): 19.943801357570553
R^2 (Coefficient of Determination): 0.8949557636782547
```

Gradient Boosting Machines

In [ ]:
```
# Initialize the GBM model
gbm_model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_dept

# Train the GBM model on the training data
gbm_model.fit(X_train, y_train)

# Predict on the training set
y_pred_train_gbm = gbm_model.predict(X_train)

# Evaluate the GBM model's performance on training data
mae_gbm_train = mean_absolute_error(y_train, y_pred_train_gbm)
mse_gbm_train = mean_squared_error(y_train, y_pred_train_gbm)
rmse_gbm_train = mean_squared_error(y_train, y_pred_train_gbm, squared=False)
r2_gbm_train = r2_score(y_train, y_pred_train_gbm)



# Predict on the test set
y_pred_gbm = gbm_model.predict(X_test)

# Evaluate the GBM model's performance
mae_gbm = mean_absolute_error(y_test, y_pred_gbm)
mse_gbm = mean_squared_error(y_test, y_pred_gbm)
rmse_gbm = mean_squared_error(y_test, y_pred_gbm, squared=False)
r2_gbm = r2_score(y_test, y_pred_gbm)


print("Performance on Training Data:")
print(f"Mean Absolute Error (MAE): {mae_gbm_train}")
print(f"Mean Squared Error (MSE): {mse_gbm_train}")
print(f"Root Mean Squared Error (RMSE): {rmse_gbm_train}")
```

```
print(f"R^2 (Coefficient of Determination): {r2_gbm_train}")
print("\nPerformance on Test Data:")
print(f"Mean Absolute Error (MAE): {mae_gbm}")
print(f"Mean Squared Error (MSE): {mse_gbm}")
print(f"Root Mean Squared Error (RMSE): {rmse_gbm}")
print(f"R^2 (Coefficient of Determination): {r2_gbm}")
```

```
Performance on Training Data:
Mean Absolute Error (MAE): 0.4979518677197901
Mean Squared Error (MSE): 1.2840423087438884
Root Mean Squared Error (RMSE): 1.1331559066359265
R^2 (Coefficient of Determination): 0.9998153821433118

Performance on Test Data:
Mean Absolute Error (MAE): 1.1516967679975911
Mean Squared Error (MSE): 11.82975794674722
Root Mean Squared Error (RMSE): 3.4394415166923857
R^2 (Coefficient of Determination): 0.9968758476317762
```

In [ ]:

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingRegressor

# Simplified and adjusted hyperparameters grid
simplified_param_grid = {
    'n_estimators': [50, 100],
    'learning_rate': [0.01, 0.1],
    'max_depth': [2, 3],
    'subsample': [0.8, 1.0],
    'max_features': ['sqrt', 'log2', None]
}

# Initialize the RandomizedSearchCV object with the simplified hyperparameters
simplified_random_search = RandomizedSearchCV(GradientBoostingRegressor(random_stat
                                              param_distributions=simplified_param_g
                                              n_iter=10,  # Reduced number of iterat
                                              scoring='neg_mean_squared_error',
                                              n_jobs=-1,
                                              cv=2,  # Reduced CV folds
                                              random_state=42,
                                              error_score='raise')

# Fit to the training data
simplified_random_search.fit(X_train, y_train)

# Extract the best estimator after the search
best_gbm_simplified = simplified_random_search.best_estimator_
print(best_gbm_simplified)
```

```
GradientBoostingRegressor(max_depth=2, max_features='sqrt', random_state=42,
                          subsample=0.8)
```

In [ ]:

```
# Predict on the training data using the best GBM model
y_train_pred = best_gbm_simplified.predict(X_train)

# Evaluate the model's performance on training data
mae_train = mean_absolute_error(y_train, y_train_pred)
mse_train = mean_squared_error(y_train, y_train_pred)
rmse_train = mean_squared_error(y_train, y_train_pred, squared=False)
r2_train = r2_score(y_train, y_train_pred)

# Predict on the test data using the best GBM model
y_test_pred = best_gbm_simplified.predict(X_test)

# Evaluate the model's performance on test data
mae_test = mean_absolute_error(y_test, y_test_pred)
```

```python
mse_test = mean_squared_error(y_test, y_test_pred)
rmse_test = mean_squared_error(y_test, y_test_pred, squared=False)
r2_test = r2_score(y_test, y_test_pred)

print("Performance on Training Data:")
print(f"Mean Absolute Error (MAE): {mae_train}")
print(f"Mean Squared Error (MSE): {mse_train}")
print(f"Root Mean Squared Error (RMSE): {rmse_train}")
print(f"R^2 (Coefficient of Determination): {r2_train}")
print("\nPerformance on Test Data:")
print(f"Mean Absolute Error (MAE): {mae_test}")
print(f"Mean Squared Error (MSE): {mse_test}")
print(f"Root Mean Squared Error (RMSE): {rmse_test}")
print(f"R^2 (Coefficient of Determination): {r2_test}")
```

```
Performance on Training Data:
Mean Absolute Error (MAE): 4.306941597889101
Mean Squared Error (MSE): 41.7662257175321
Root Mean Squared Error (RMSE): 6.462679453410335
R^2 (Coefficient of Determination): 0.9939949088737813

Performance on Test Data:
Mean Absolute Error (MAE): 5.613848985003764
Mean Squared Error (MSE): 119.29142931425552
Root Mean Squared Error (RMSE): 10.922061587184698
R^2 (Coefficient of Determination): 0.9684960078575897
```

In [ ]:

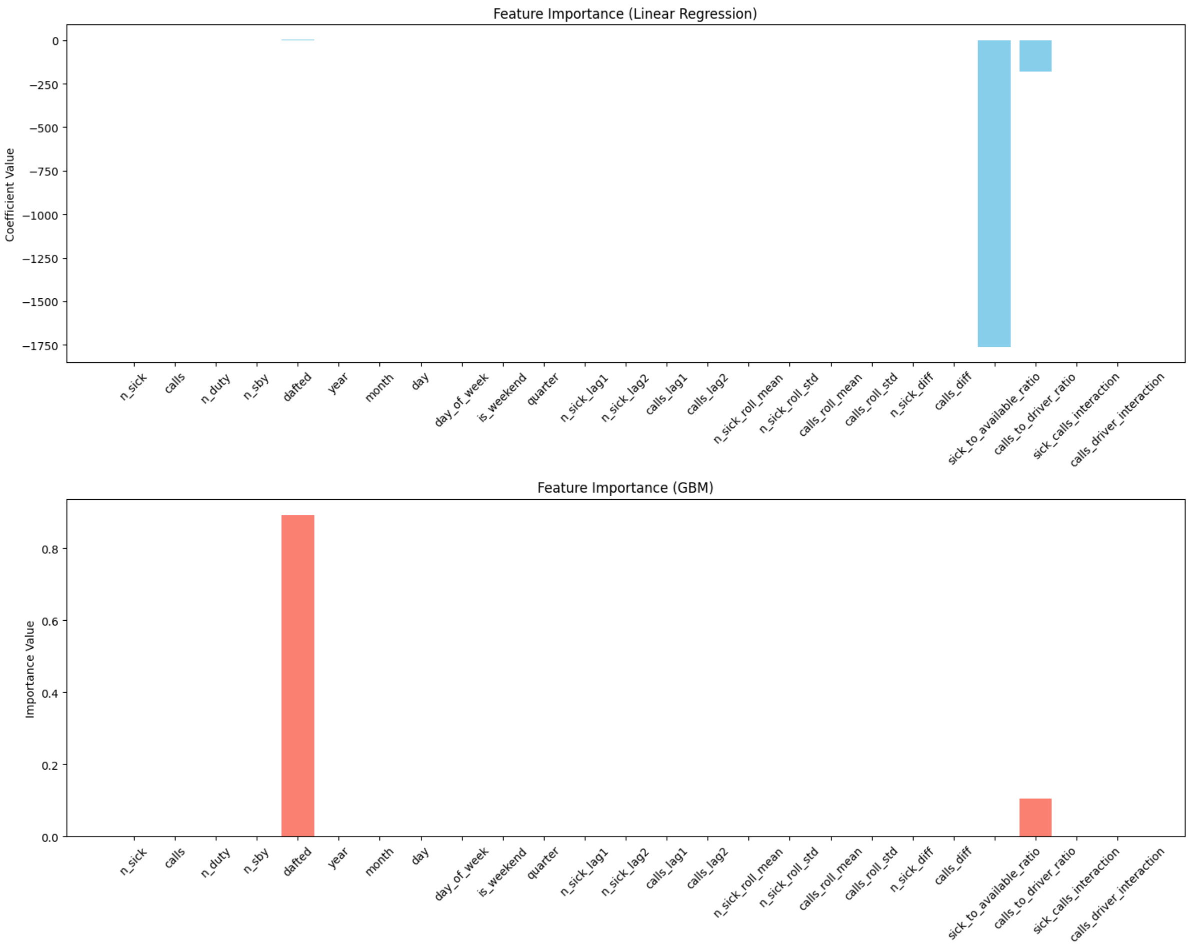
```python
# Predicted values for Linear Regression
y_pred_lr = baseline_model.predict(X_test)

# Predicted values for untuned GBM
y_pred_gbm = gbm_model.predict(X_test)

# Predicted values for hyperparameter-tuned GBM
y_pred_best_gbm = best_gbm_simplified.predict(X_test)

# Function to check insufficient standby predictions
def check_insufficient_standbys(predictions, actual_sby_need, n_sby_values):
    insufficient_standbys = sum(predictions > n_sby_values)
    actual_insufficient_standbys = sum(actual_sby_need > n_sby_values)
    return insufficient_standbys, actual_insufficient_standbys

# Check for each model
insufficient_lr, actual_insufficient = check_insufficient_standbys(y_pred_lr, y_tes
insufficient_gbm, _ = check_insufficient_standbys(y_pred_gbm, y_test, X_test['n_sby
insufficient_best_gbm, _ = check_insufficient_standbys(y_pred_best_gbm, y_test, X_t

print(f"Days with insufficient standbys (Actual): {actual_insufficient}")
print(f"Days with insufficient standbys (Linear Regression): {insufficient_lr}")
print(f"Days with insufficient standbys (Untuned GBM): {insufficient_gbm}")
print(f"Days with insufficient standbys (Tuned GBM): {insufficient_best_gbm}")
```

```
Days with insufficient standbys (Actual): 25
Days with insufficient standbys (Linear Regression): 18
Days with insufficient standbys (Untuned GBM): 25
Days with insufficient standbys (Tuned GBM): 25
```

In [ ]:

```python
import matplotlib.pyplot as plt

# Compute feature importance
# For Linear Regression: Coefficients as feature importance
lr_importance = baseline_model.coef_

# For Gradient Boosting Machines: Feature importance attribute
gbm_importance = gbm_model.feature_importances_
```

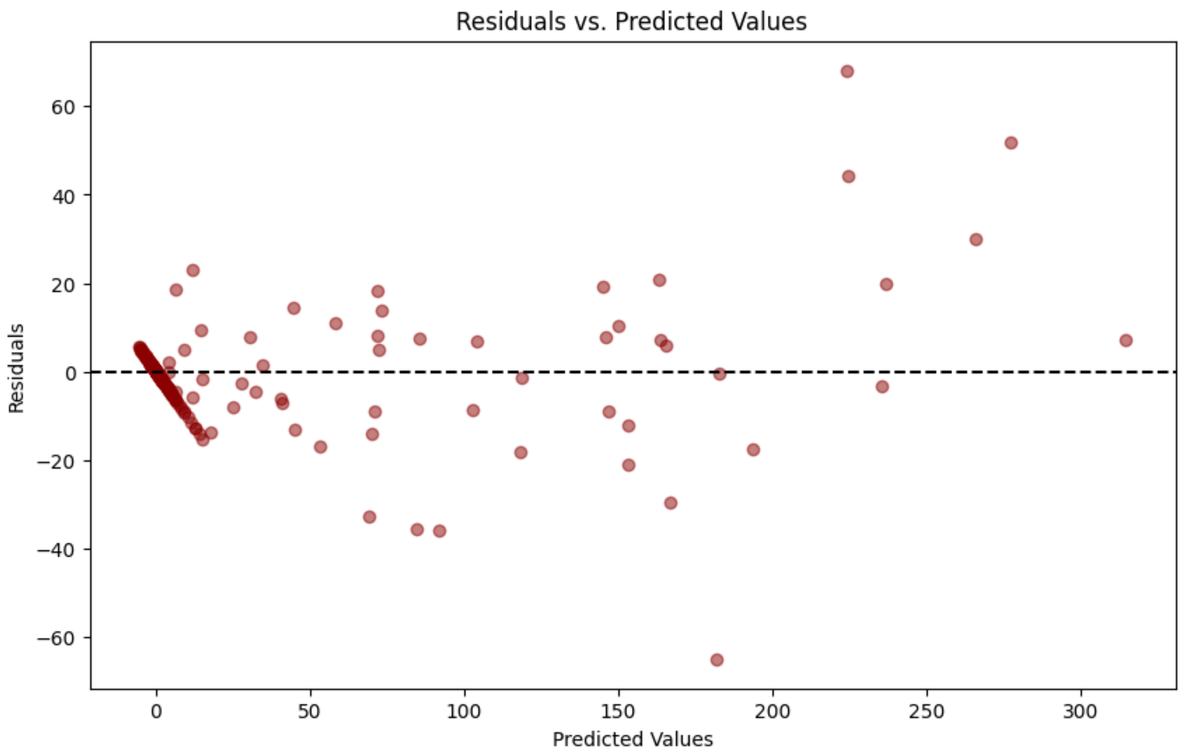
```
# Visualization
fig, ax = plt.subplots(2, 1, figsize=(15, 12))

# Plotting feature importance for Linear Regression
ax[0].bar(X.columns, lr_importance, color='skyblue')
ax[0].set_title('Feature Importance (Linear Regression)')
ax[0].tick_params(axis='x', rotation=45)
ax[0].set_ylabel('Coefficient Value')

# Plotting feature importance for GBM
ax[1].bar(X.columns, gbm_importance, color='salmon')
ax[1].set_title('Feature Importance (GBM)')
ax[1].tick_params(axis='x', rotation=45)
ax[1].set_ylabel('Importance Value')

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



Error Analysis

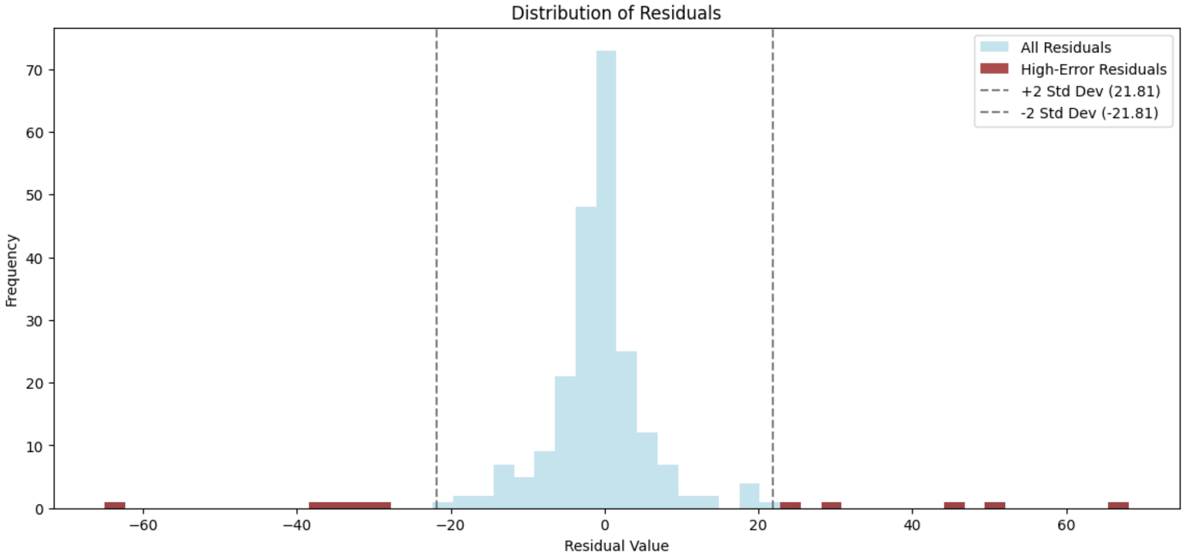
In [ ]:
```
import numpy as np

# Calculate residuals for the GBM model
residuals = y_test - y_test_pred

# Plot residuals
plt.figure(figsize=(10, 6))
plt.scatter(y_test_pred, residuals, alpha=0.5, color='darkred')
plt.axhline(y=0, color='black', linestyle='--')
plt.title('Residuals vs. Predicted Values')
plt.xlabel('Predicted Values')
```

```
plt.ylabel('Residuals')
plt.show()
```



In [ ]:

```
# Calculate residuals for the GBM model
residuals = y_test - y_test_pred

# Determine the threshold for high errors
threshold = 2 * np.std(residuals)
high_error_indices = np.where(np.abs(residuals) > threshold)[0]

# Extract high error data from the test set
high_error_data = X_test.iloc[high_error_indices]

# Visualize the distribution of residuals and highlight the high-error instances
plt.figure(figsize=(14, 6))
plt.hist(residuals, bins=50, color='lightblue', label='All Residuals', alpha=0.7)
plt.hist(residuals.iloc[high_error_indices], bins=50, color='darkred', label='High-
plt.axvline(x=threshold, color='grey', linestyle='--', label=f'+2 Std Dev ({thresho
plt.axvline(x=-threshold, color='grey', linestyle='--', label=f'-2 Std Dev (-{thres
plt.title('Distribution of Residuals')
plt.xlabel('Residual Value')
plt.ylabel('Frequency')
plt.legend()
plt.show()

# To inspect the high-error data, you can further examine the `high_error_data` Dat
# For a more detailed view, add the residuals to this data:
high_error_data['Residual'] = residuals.iloc[high_error_indices].values

# Display the first few rows of the high-error data for inspection
print(high_error_data.head())
```




```
      n_sick   calls  n_duty  n_sby  dafted  year  month  day  day_of_week  \
174       77  8286.0    1700     90     0.0  2016      9   22            3
101       70  9492.0    1700     90   179.0  2016      7   11            0
1006      82  9672.0    1900     90    27.0  2019      1    2            2
1099      67  9444.0    1900     90     0.0  2019      4    5            4
96        51  9702.0    1700     90   202.0  2016      7    6            2

      is_weekend  quarter  n_sick_lag1  n_sick_lag2  calls_lag1  calls_lag2  \
174            0        3         80.0         72.0      8532.0      7362.0
101            0        3         70.0         69.0      7374.0      8532.0
1006           0        1         57.0         75.0      8382.0      6318.0
1099           0        2         75.0         79.0     10260.0     11328.0
96             0        3         47.0         50.0      8550.0      8526.0

      n_sick_roll_mean  n_sick_roll_std  calls_roll_mean  calls_roll_std  \
174          75.714286         3.817254      7530.857143      873.224566
101          61.571429         9.396048      8730.857143      775.149755
1006         74.285714         8.056349      7638.000000     1465.646615
1099         80.000000         7.416198      9756.000000     1879.590381
96           46.142857         4.220133      8245.714286     1016.203017

      n_sick_diff  calls_diff  sick_to_available_ratio  calls_to_driver_ratio  \
174          -3.0      -246.0                 0.043017               4.629050
101           0.0      2118.0                 0.039106               5.302793
1006         25.0      1290.0                 0.041206               4.860302
1099         -8.0      -816.0                 0.033668               4.745729
96            4.0      1152.0                 0.028492               5.420112

      sick_calls_interaction  calls_driver_interaction   Residual
174                 638022.0                14831940.0  23.079963
101                 664440.0                16990680.0  44.238582
1006                793104.0                19247280.0 -64.973504
1099                632748.0                18793560.0 -35.971229
96                  494802.0                17366580.0  68.091800
```

```
C:\Users\s9\AppData\Local\Temp\ipykernel_6408\3816389916.py:25: SettingWithCopyWar
ning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  high_error_data['Residual'] = residuals.iloc[high_error_indices].values
```

In [ ]: