INTERNATIONAL UNIVERSITY OF APPLIED SCIENCES

DLMDSME01 Model Engineering

CASE STUDY

# Automation of Standby Duty Planning for Rescue Drivers via a Forecasting Model

Author: Ajaychandra Arekal Satishchandra
Enrollment Number: 321149868
Email: ajay.chandra-a-s@iu-study.org
Study Program: 120 ECTS M.Sc Data Science
Date: 11.01.2024
Place: Berlin, Germany

## Table of Contents

# Introduction

Berlin's red-cross rescue service faces challenges in planning standby duty for rescue drivers. This case study explores a data-driven approach to transform standby-duty planning for rescue drivers, optimizing the process for enhanced efficiency.

As a data science consultant, my goal is to enhance the efficiency of the current planning logic using predictive models. The existing strategy involves maintaining a daily standby roster of 90 drivers, but seasonal patterns and unexpected events introduce complexities, occasionally resulting in insufficient standby drivers and requiring on-call drivers to cover scheduled days off. Unpredictability in driver availability, caused by sudden illnesses or an increase in emergency calls, poses a significant challenge, along with dealing with seasonal variations. The current method of keeping a set group of 90 standby drivers often falls short of expectations.

The project's objective is to optimize standby driver activation and minimize instances of insufficient coverage by leveraging a robust dataset for predictive modelling. Our aim includes developing a predictive model to improve standby driver utilization and reduce instances of on-call drivers having to be called for work on their days off. Following the CRISP-DM data science methodology, we conduct a thorough business analysis, evaluating data quality, testing various modelling techniques, understanding outcomes, and implementing practical solutions. By meticulously executing this project, the goal is to instil confidence in stakeholders, enhancing the reliability, usability and efficiency of day-to-day operations.

# Main Body

## Data Science Methodology: CRISP-DM

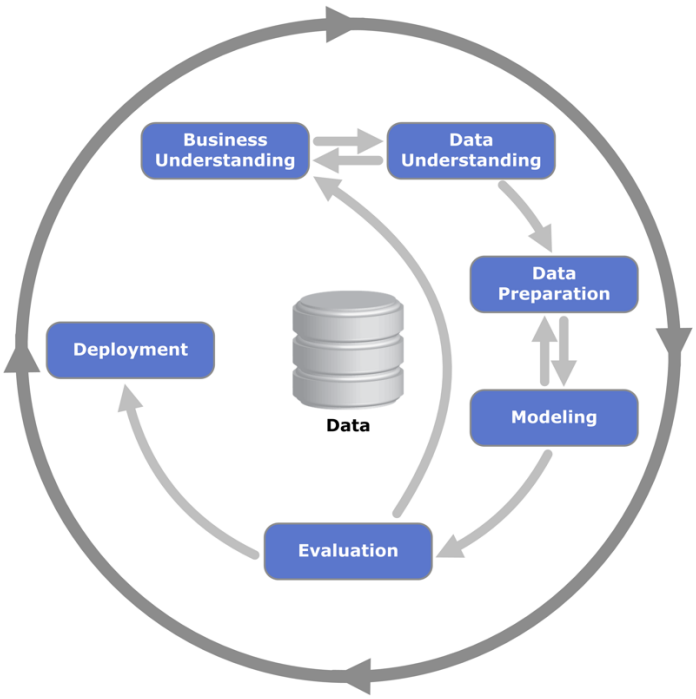



Figure: CRISP-DM

("Cross-Industry Standard Process for Data Mining," 2023)

The Cross Industry Standard Process for Data Mining (CRISP-DM) provides a foundational framework for the data science process, encompassing six consecutive phases. CRISP-DM proves adaptable and effective in various scenarios. Its structured approach ensures a right start by prioritizing business understanding and aligning technical work with objectives. Notably, its final step, Deployment, facilitates a strong finish, addressing key considerations for project closure. Its flexible nature allows for iterations, gaining deeper insights with each cycle, making it a valuable methodology for data science projects. Additionally, CRISP-DM's simplicity and ease of adoption make it a common-sense choice(Hotz, 2018).

**Below is a concise overview of the various project phases(Hotz, 2018):**

- The Business Understanding phase in our project involves comprehending objectives, assessing resource availability, defining project goals, and producing a project plan for predicting standby drivers for Berlin's Red Cross rescue service. This foundational phase is crucial for aligning technical work with business needs and ensuring project success.
- In the Data Understanding phase, we focus on identifying, collecting, and analyzing datasets relevant to predicting standby drivers. This involves acquiring initial data, describing its properties, exploring relationships, and verifying data quality.
- In the Data Preparation phase, we select, clean, construct, integrate, and format datasets to prepare them for modeling. This involves steps including correcting errors, deriving new attributes and formatting as needed for predicting standby drivers.
- In the Modeling phase, we choose modeling techniques, generate test designs, build models, and assess their performance. This involves selecting algorithms, splitting data into training and test sets, executing code for model building, and interpreting results.
- In the Evaluation phase, we assess model results against business success criteria, review the project's execution, and decide on the next steps—whether to deploy or iterate further based on the outcome. This broader perspective ensures alignment with business objectives and informed decision-making.
- In the Deployment phase, we plan the deployment, monitoring, and maintenance of the model, document a final report summarizing the results, and identify areas of improvement. This ensures a smooth transition to operational phases, emphasizing ongoing model maintenance and continuous improvement.

Furthermore, a brief explanation of an intuitive graphical user interface (GUI) will be presented, outlining the seamless integration of the model into the HR planning team's daily workflow. Emphasizing transparency, interpretability, and reliability within the CRISP-DM framework, our approach ensures that the predictive models surpass the expectations of Berlin's red-cross rescue service.

## Business Understanding

In the realm of Berlin's red-cross rescue service, the business understanding phase of our project involves a comprehensive analysis of the current standby-duty planning challenges faced by the HR planning department. The existing approach relies on maintaining a daily standby roster of 90 rescue drivers due to the unpredictability of short-term sickness and emergency call fluctuations. However, seasonal patterns contribute to an increase in sick calls, posing a challenge that is not accounted for in the current planning logic. Moreover, there are instances where the activation of all 90 standby-drivers proves insufficient, necessitating on-call drivers to step in even on their scheduled days off. To address these complexities, the aim is to develop a predictive model that efficiently estimate the daily standby rescue drivers, ensuring a higher activation percentage than the current approach while minimizing situations with insufficient standbys. This understanding sets the stage for the subsequent phases of the

CRISP-DM methodology, guiding the project toward effective solutions tailored to the specific needs of Berlin's red-cross rescue service.

### *Project Goals*

This project aims to enhance the efficiency of standby-duty planning for Berlin's red-cross rescue service. It involves creating a predictive model to accurately estimate daily standby driver requirements, with goals to optimize driver utilization, minimize shortages, and ensure timely duty plan preparation by the 15th of each month.

This project, following the CRISP-DM methodology, emphasizes key aspects such as assessing data quality, highlighting feature importance, developing predictive models for standby driver estimation, and conducting a thorough error analysis to ensure transparency and trust in the approach.

## Data Understanding

In this phase, our goal is to gain a profound comprehension of the data's characteristics, structure, and nuances. We explore patterns, distributions, and potential challenges within the dataset. Delving into the sources of data that form the basis of our predictive model for optimizing standby-duty planning for rescue drivers, the core dataset, "sickness_table.csv" provided by Berlin's red-cross rescue service, encompasses historical records of daily activities, including the number of rescue drivers on duty, emergency call volumes, and instances of driver sickness. Spanning across multiple years, the dataset provides the essential granularity required to identify patterns and trends. A comprehensive understanding of the data is pivotal to our project, necessitating exploratory data analysis for unveiling insights, trends, and potential correlations embedded within the dataset. This section provides an overview of the diverse data sources that we will harness to build an effective predictive model, laying the foundation for subsequent data preparation and modeling stages.

### Exploratory Data Analysis

Exploratory data analysis (EDA) is a statistical method used to examine and describe the main features of a dataset. This approach often involves the use of visual techniques such as statistical graphics to summarize and gain insight into the data("Exploratory Data Analysis," 2023).

In our project, we utilize a "sickness_table" file in .csv format. The data undergoes summarization and analysis, involving computations of correlations and checks for irregularities.

Berlin's red cross rescue service has already provided the necessary data, eliminating the need for additional data collection for EDA. Consequently, we will commence the analyses directly, employing various techniques to summarize the provided data and extract valuable insights. This process aids in enhancing our understanding of the data and facilitates informed decision-making based on the findings.

**Feature Description:**

**date**: Represents the entry date.

**n_sick**: Signifies the count of drivers called in sick.

**calls**: Indicates the number of emergency calls received.

**n_duty**: Represents the count of drivers on duty and available.

**n_sby**: Denotes the number of standby resources available.

**sby_need**: Reflects the count of standby drivers activated on a given day.

**dafted**: Represents the number of additional drivers needed due to an insufficient number of standbys.

We use various functions from the “pandas” library aliased as “pd” in this stage. This stage has numerous crucial steps:

Loading the dataset: In this phase, we utilize "pd.read_csv()" to read the file and subsequently employ "pd.DataFrame()" to convert the file into pandas dataframe for subsequent steps. Recognizing that the first column is unlabeled and likely serves as an ID column, we opted to exclude it as it doesn't contribute significantly to the EDA process. The initial five rows of the read file are displayed below.

| | 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 |

Next, we have employed the "describe()" method to swiftly grasp the summary statistics of the data.

| | n_sick | calls | n_duty | n_sby | sby_need | dafted |
|---|---|---|---|---|---|---|
| **count** | 1152.000000 | 1152.000000 | 1152.000000 | 1152.0 | 1152.000000 | 1152.000000 |
| **mean** | 68.808160 | 7919.531250 | 1820.572917 | 90.0 | 34.718750 | 16.335938 |
| **std** | 14.293942 | 1290.063571 | 80.086953 | 0.0 | 79.694251 | 53.394089 |
| **min** | 36.000000 | 4074.000000 | 1700.000000 | 90.0 | 0.000000 | 0.000000 |
| **25%** | 58.000000 | 6978.000000 | 1800.000000 | 90.0 | 0.000000 | 0.000000 |
| **50%** | 68.000000 | 7932.000000 | 1800.000000 | 90.0 | 0.000000 | 0.000000 |
| **75%** | 78.000000 | 8827.500000 | 1900.000000 | 90.0 | 12.250000 | 0.000000 |
| **max** | 119.000000 | 11850.000000 | 1900.000000 | 90.0 | 555.000000 | 465.000000 |

The summary statistics reveal that the dataset consists of 1,152 entries. In terms of the number of sick calls, the mean is approximately 68.81, with a minimum of 36 and a maximum of 119. The average number of emergency calls is around 7,919 with a minimum of 4,074 and a maximum of 11,850. The on-duty count has a mean of 1,820.57, while the standby count is consistently 90. The drafted column indicates a mean of 16.34, with a maximum of 465, pointing out cases where standby drivers are drafted beyond the initially planned count. These statistics provide an initial understanding of the data's distribution and key metrics.

Next, we have used the “info()” method to get a concise overview of the data.

```
<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   object 
 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: float64(3), int64(3), object(1)
memory usage: 63.1+ KB
```

The DataFrame comprises 1,152 entries and 7 columns, including "date," "n_sick," "calls," "n_duty," "n_sby," "sby_need," and "dafted." All the column contains non-null values for all entries indicating no missing data. The data types include int64 and float64 for numerical values and object for the "date" column. This concise summary provides an overview of the data structure, completeness, and data types within the DataFrame.

Next, we used the “nunique()” method to check the count of unique values in each feature.

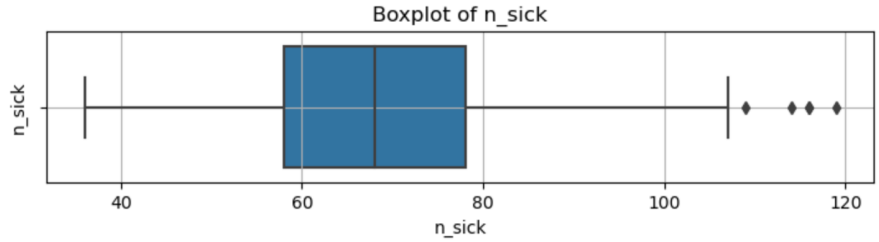
```
date        1152
n_sick        74
calls        616
n_duty         3
n_sby          1
sby_need     185
dafted       119
dtype: int64
```

The number of unique values in each column varies within the dataset. The "date" column has 1,152 unique entries, indicating a distinct date for each record. For "n_sick," there are 74 unique values, representing the different counts of drivers called in sick. The "calls" column has 616 unique values, indicating the diversity in the number of emergency calls. In contrast, "n_duty" has only 3 unique values. The "n_sby" column has 1 unique value, implying a constant count of standby resources. "sby_need" exhibits 185 unique values, reflecting the variability in the number of standbys activated on different days. Lastly, "dafted" has 119 unique values, representing the diverse count of additional drivers needed due to insufficient standbys. This overview highlights the diversity and characteristics of unique values in each column of the dataset.

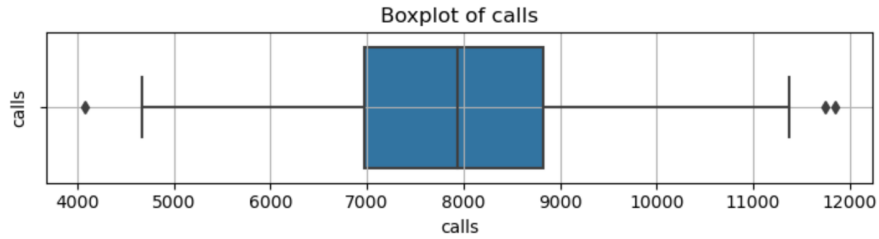
Next, we have used the "drop()" method to eliminate the "n_sby" column as this column contains only a single unique value, it lacks variability and does not contribute meaningful information to the dataset. Therefore, we have decided to exclude this column from our analysis. Removing such a column helps streamline the data and focus on variables that bring meaningful variation to the analysis. After excluding the "n_sby" column, the remaining columns are presented below.

```
['date', 'n_sick', 'calls', 'n_duty', 'sby_need', 'dafted']
```

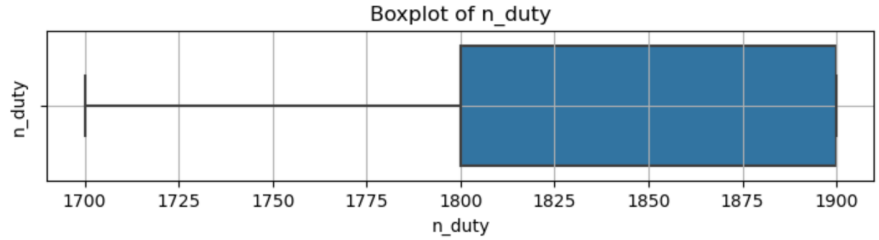
The upcoming step involves the utilization of box plots for all the remaining columns. A box plot is a descriptive statistical method used to display the distribution, variability, and skewness of numerical data. It shows quartiles of the data through a box, with whiskers extending from the box indicating variability beyond the quartiles. The plot may also display outliers as individual points beyond the whiskers. Unlike other statistical methods, box plots do not assume any specific distribution of the data ("Box Plot," 2023).



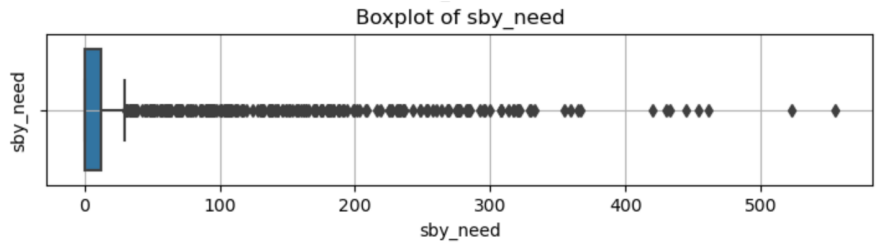

The boxplot for 'n_sick' reveals the presence of outliers, with the maximum outlier extending to approximately 120. These outliers suggest instances where the number of drivers called sick, deviates significantly from the majority of observations, highlighting potential anomalies or exceptional circumstances within the dataset.



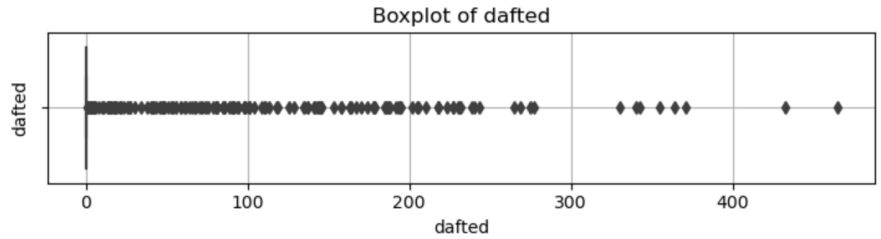

In the boxplot for 'calls,' outliers are observed, reaching a maximum value of approximately 11,900 and a minimum of around 4,100. These outliers indicate instances where the number of emergency calls significantly deviates from the majority of observations, suggesting potential exceptional circumstances or unusual events.




The boxplot for 'n_duty' indicates the absence of outliers, suggesting a consistent maintenance of the number of drivers on duty.



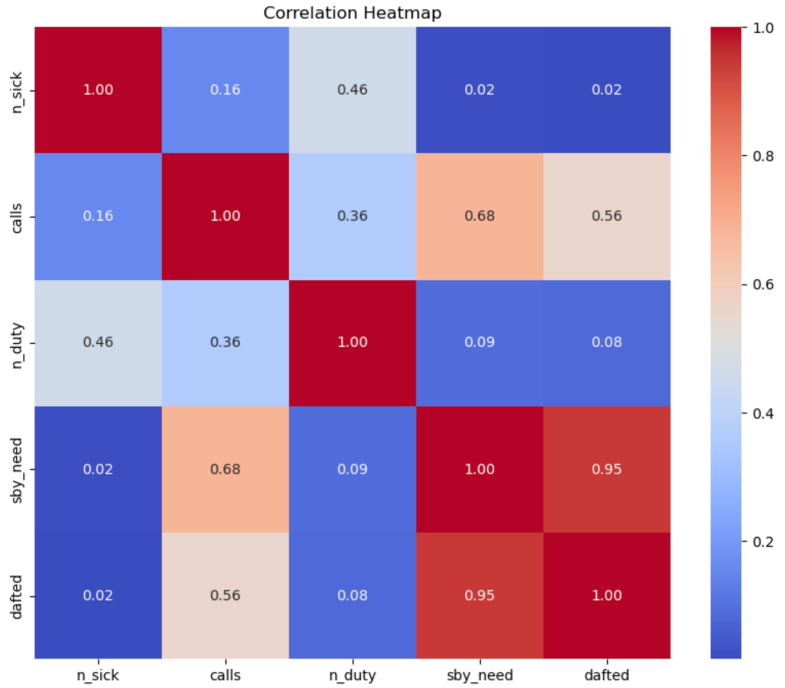

The boxplot for 'sby_need' unveils a substantial number of outliers, reaching a maximum value of approximately 580. Additionally, the median is situated outside the 75% range, indicating a significant skewness in the distribution of standby drivers needed. These findings suggest instances where the demand for standby drivers deviates noticeably, indicating potential high-demand periods or exceptional situations.




The boxplot for 'dafted' unveils a substantial number of outliers, reaching a maximum value of approximately 480. These outliers suggest instances where a considerable number of additional drivers were needed due to insufficient standby resources.

Next, we employ the "corr()" method to examine the correlation among the features and visualize it using the seaborn library.

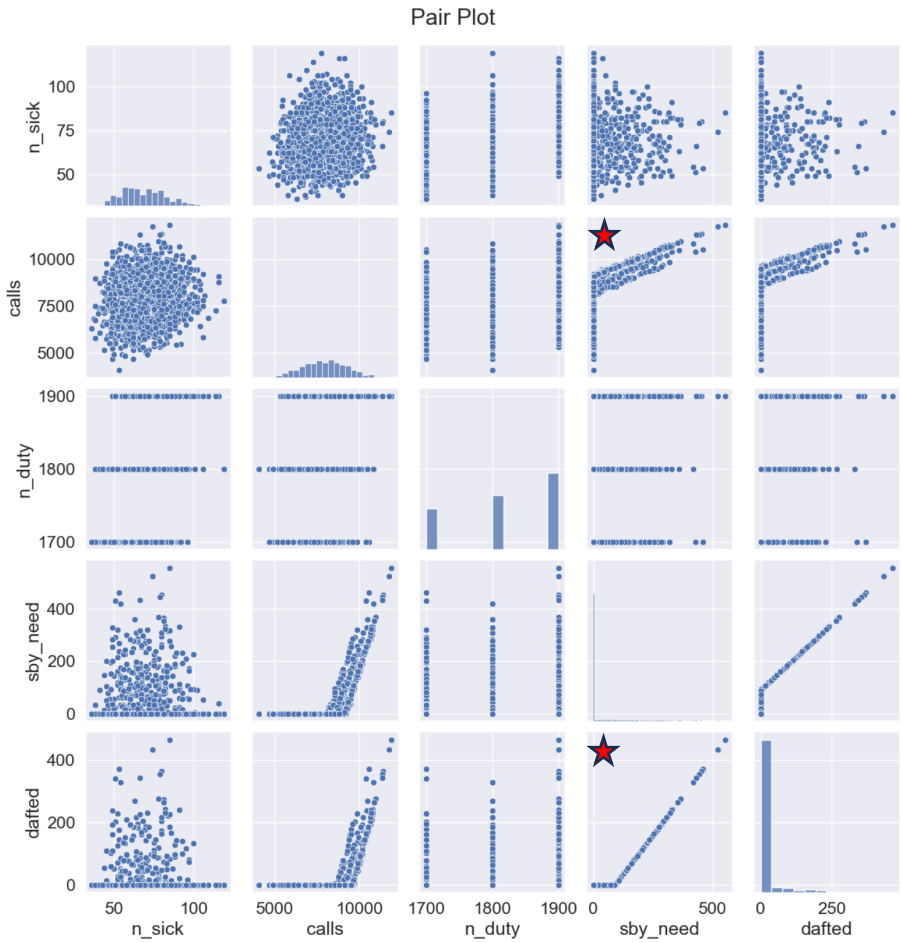
Correlation analysis serves as a tool for examining connections between variables, essentially quantifying the degree of their relationship. The exploration of these interdependencies is known as correlation analysis(*Correlation in Statistics*, n.d.). In our analysis, we will employ the Pearson correlation, which quantifies the strength of the linear relationship between two variables. This correlation coefficient ranges from -1 to 1. -1 indicates a complete negative linear correlation, 0 implies no correlation, and +1 signifies a total positive correlation(*Pearson Correlation - an Overview | ScienceDirect Topics*, n.d.).




As seen from the above Correlation Heatmap, 'dafted' and 'sby_need' exhibit a very strong positive correlation of 0.95, indicating a close relationship between the number of additional drivers needed due to insufficient standbys and the actual number of standbys activated. Additionally, 'calls' and 'sby_need' demonstrate a moderate positive correlation of 0.68, suggesting a discernible but not extremely strong relationship between the number of emergency calls and the subsequent need for activated standbys. Furthermore, 'calls' and 'dafted' show a moderate positive correlation of 0.56, indicating a discernible but not exceptionally strong relationship between the number of emergency calls and the subsequent need for additional drivers due to insufficient standbys. Based on the correlations, 'dafted' emerges as the most influential feature for estimating the amount of daily standby rescue

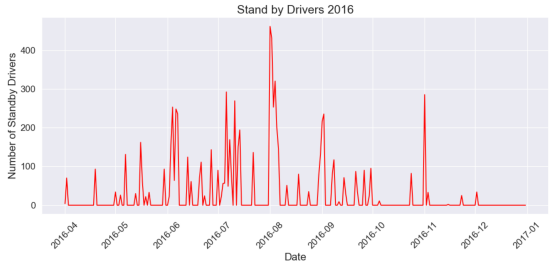
drivers, displaying a very strong positive correlation. Following closely, 'calls' stands out as the next best feature, demonstrating a moderate positive correlation.

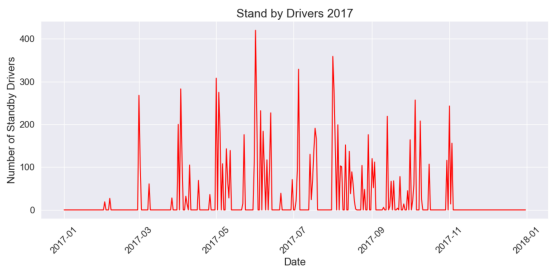
In the following step, we are utilizing a pair plot to visually represent the features using the seaborn library. A Pair plot enables the visualization of pairwise relationships between variables in a dataset, providing a comprehensive overview and aiding in understanding the data by condensing extensive information into a single figure. This proves crucial during the exploration phase of the dataset, facilitating familiarity and insights(McDonald, 2022).

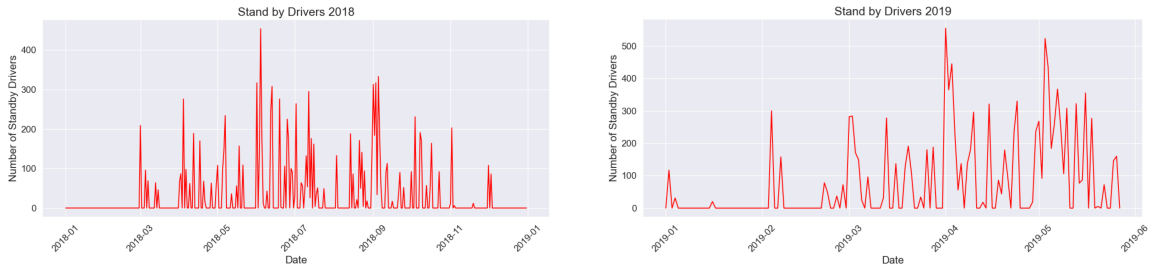



The pair plot indicates a consistent increase in 'sby_need' as 'dafted' rises, and similarly, there is an observable upward trend in 'sby_need' as 'calls' increases.

Next, we will be visualizing the yearly graphs of “date” versus “'sby_need'” to comprehend the trend of standby drivers throughout each year.

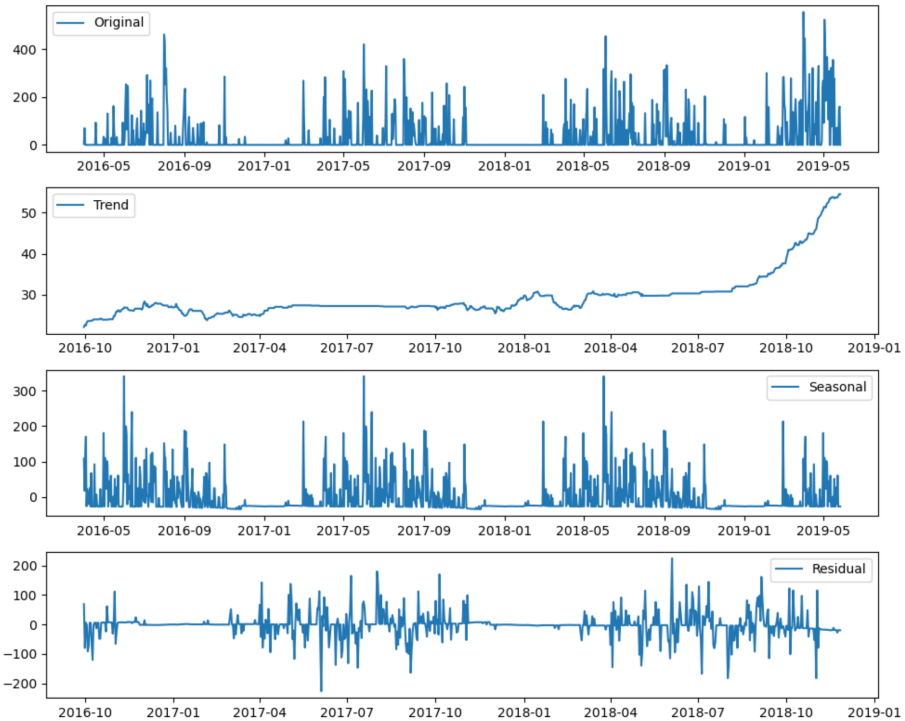





The yearly plots reveal a consistent pattern in the demand for drivers needed. Peaks in 'sby_need' consistently occur in the middle of each year, while the beginning and end of the year exhibit the lowest or no 'sby_need.' Additionally, there are similar fluctuations observed during the intervening periods of each year.

Up next, we will utilize time series decomposition to gain insights into the provided time series data. Time series decomposition is a valuable method that breaks down a series into systematic and unsystematic components. It involves identifying the trend, seasonality, and noise components. The systematic components exhibit consistency and recurrence, allowing for modeling, while the non-systematic components cannot be directly modeled. These components are crucial for understanding and forecasting time series data. The decomposition can be either additive or multiplicative, representing how the components combine. The “statsmodels” library offers a “seasonal_decompose()” function for automatic time series decomposition, allowing access to trend, seasonal, and residual(noise) components. This structured approach aids in analyzing and modeling time series data, providing insights into forecasting problems(Brownlee, 2017).




The above decomposition plots provide valuable insights. Upon examination of the original chart, it appears to exhibit seasonal patterns. However, upon analyzing the decomposition charts, a distinct trending pattern is evident in the trend chart, while the seasonal chart reveals a discernible seasonal pattern.

The trend chart indicates a gradual increase throughout the graph, with a noticeable acceleration in the rate of drivers needed towards the end. Concurrently, the seasonal chart underscores a consistent seasonal pattern observed throughout the years. In the residual chart, we identify irregularities or noise within the data, which are typically undesirable and are usually avoided.

Considering the seasonal pattern in our data, we will incorporate various time series models in the modelling phase and assess their suitability for this project.

## Data Preparation

During this phase, our focus will be on cleaning and transforming the data to ensure its suitability for upcoming phases of the project. This involves addressing any inconsistencies and subsequently structuring the data to enhance its compatibility with the modeling and analysis steps that follow. The goal is to create a refined and well-organized dataset that optimally supports the project's objectives.

In the following step, we will proceed to clip outliers from some of the chosen features. In statistical analysis, an outlier refers to a data point that deviates notably from the rest of the observations. This deviation could stem from measurement variability, represent new and valuable information, or result from experimental errors. In some cases, outliers are excluded from the dataset. While outliers can signal intriguing possibilities, they can also pose challenges in statistical analyses(“Outlier,” 2023).

As highlighted in the preceding section, box plots revealed the presence of outliers in "n_sick," "calls," "sby_need," and "dafted." We've chosen to clip outliers in the features "n_sick," "calls," and "sby_need," excluding "dafted." This decision is motivated by the fact that over 90% of the "dafted" feature comprises 0 values, and removing outliers could adversely impact the usability of this feature. Additionally, considering the wide range of outliers in 'dafted,' implementing clipping would result in a substantial number of zero values, impacting both the model input and subsequent analysis.

We will employ the IQR method to eliminate outliers from the chosen columns. The Interquartile Range (IQR) serves as a measure of statistical dispersion, indicating the range encompassing the middle 50% of the data. Calculating the IQR involves determining the difference between the 75th percentile (Q3) and the 25th percentile (Q1)(Patil, 2023).

$$IQR = Q3 - Q1$$

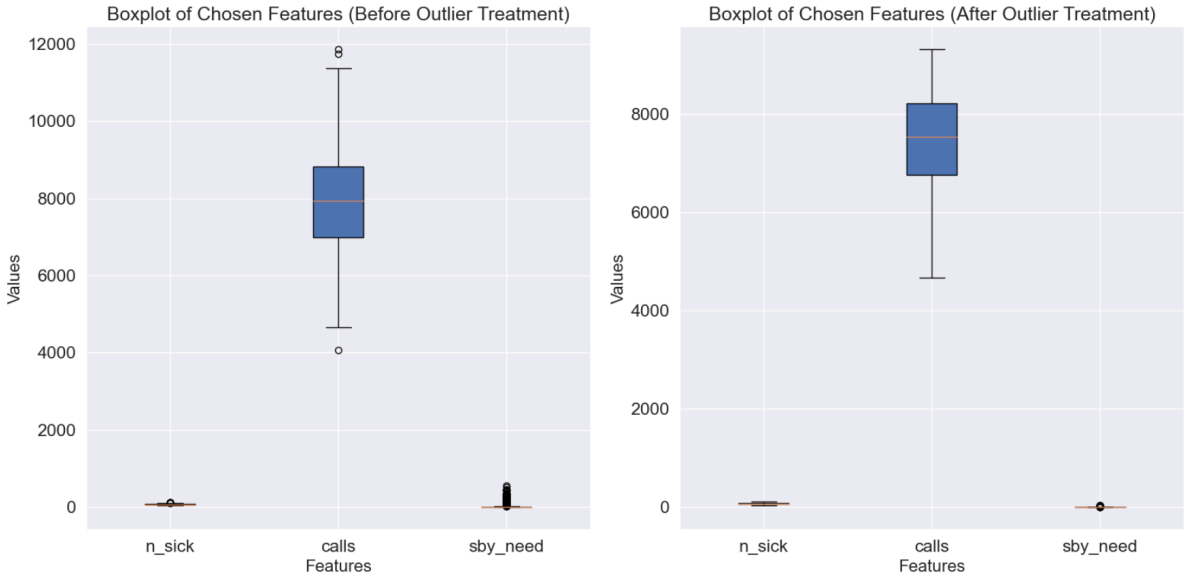
To detect outliers through the IQR method, we define two boundaries:

$$\text{Lower Bound: } Q1 - 1.5 * IQR$$

$$\text{Upper Bound: } Q3 + 1.5 * IQR$$

The boundaries aid in identifying potential outliers. Points below the lower bound are considered outliers, indicating values significantly lower than majority of the dataset. Conversely, points exceeding the upper bound are flagged as outliers, suggesting values significantly higher than most of the dataset, warranting attention. The IQR method is advantageous for its robustness to skewed data and simplicity, providing a clear range for data points while making it an effective tool in data analysis and quality control(Patil, 2023).

The box plots below illustrate the comparison between chosen unclipped features and the features with outliers clipped.




Next, we will be scaling numerical features using feature scaling techniques. Feature scaling, also known as data normalization, is a technique used to standardize the range of independent variables or features in data. This process is typically carried out during the data pre-processing step. The importance of normalization arises from the wide variability in the range of values in raw data, as certain machine learning algorithms may not function optimally without it(“Feature (Machine Learning),” 2023).

Normalization and standardization are the two widely used techniques for scaling numerical data before modelling. Normalization individually scales each input variable to the range of 0-1, optimizing precision for floating-point values. On the other hand, standardization involves scaling each input variable individually by subtracting the mean (centering) and dividing by the standard deviation. This process shifts the distribution to have a mean of zero and a standard deviation of one(Brownlee, 2020).

**Standard Scaling:** The outcomes of standard scaling for the initial five rows of the data with clipped outliers are presented in the table below.

| | date | n_sick | calls | n_duty | sby_need | dafted | year |
|---|---|---|---|---|---|---|---|
| **0** | 2016-04-01 | 0.324578 | 0.707426 | -1.466036 | 0.748064 | 0.0 | 2016 |
| **2** | 2016-04-03 | -0.030180 | 0.641216 | -1.466036 | -0.209243 | 0.0 | 2016 |
| **3** | 2016-04-04 | 0.182675 | -0.406112 | -1.466036 | -0.209243 | 0.0 | 2016 |
| **4** | 2016-04-05 | -0.384938 | -0.213500 | -1.466036 | -0.209243 | 0.0 | 2016 |
| **5** | 2016-04-06 | 0.111723 | -0.959872 | -1.466036 | -0.209243 | 0.0 | 2016 |

**Min-Max Scaling:** The outcomes of Min-Max scaling for the initial five rows of the data with clipped outliers are presented in the table below.

| | date | n_sick | calls | n_duty | sby_need | dafted | year |
|---|---|---|---|---|---|---|---|
| **0** | 2016-04-01 | 0.521127 | 0.750646 | 0.0 | 0.133333 | 0.0 | 2016 |
| **2** | 2016-04-03 | 0.450704 | 0.736434 | 0.0 | 0.000000 | 0.0 | 2016 |
| **3** | 2016-04-04 | 0.492958 | 0.511628 | 0.0 | 0.000000 | 0.0 | 2016 |
| **4** | 2016-04-05 | 0.380282 | 0.552972 | 0.0 | 0.000000 | 0.0 | 2016 |
| **5** | 2016-04-06 | 0.478873 | 0.392765 | 0.0 | 0.000000 | 0.0 | 2016 |

We are opting for Min-Max scaling as our preferred method, as standard scaling yields negative values. Additionally, Min-Max scaling ensures that all features are transformed to a consistent range, enhancing the interpretability and comparability of their contributions to the model.

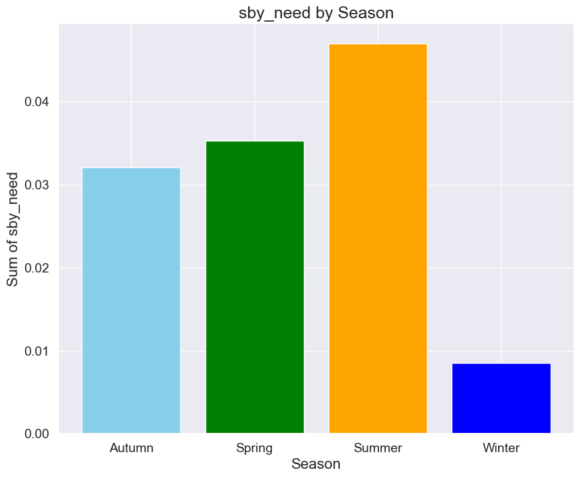
The following step involves feature engineering, where we'll enrich the outlier-clipped and Min-Max scaled dataset by incorporating categorical features such as day of the week, month, and season. Feature engineering, also known as feature extraction or discovery, refers to the process of extracting characteristics, properties, or attributes from data to facilitate the training of a subsequent statistical model("Feature Engineering," 2024).

The table below displays the initial five rows of the dataset with the newly incorporated features.

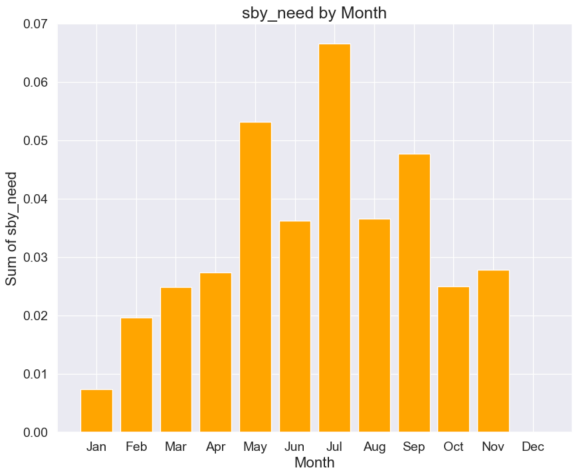
| | date | n_sick | calls | n_duty | sby_need | dafted | year | day_of_week | month | season |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2016-04-01 | 0.521127 | 0.750646 | 0.0 | 0.133333 | 0.0 | 2016 | Fri | Apr | Spring |
| **2** | 2016-04-03 | 0.450704 | 0.736434 | 0.0 | 0.000000 | 0.0 | 2016 | Sun | Apr | Spring |
| **3** | 2016-04-04 | 0.492958 | 0.511628 | 0.0 | 0.000000 | 0.0 | 2016 | Mon | Apr | Spring |
| **4** | 2016-04-05 | 0.380282 | 0.552972 | 0.0 | 0.000000 | 0.0 | 2016 | Tue | Apr | Spring |
| **5** | 2016-04-06 | 0.478873 | 0.392765 | 0.0 | 0.000000 | 0.0 | 2016 | Wed | Apr | Spring |

We've categorized the months into seasons: March, April, and May are considered Spring; June, July, and August represent Summer; September, October, and November are associated with Autumn, and the remaining months fall under Winter.

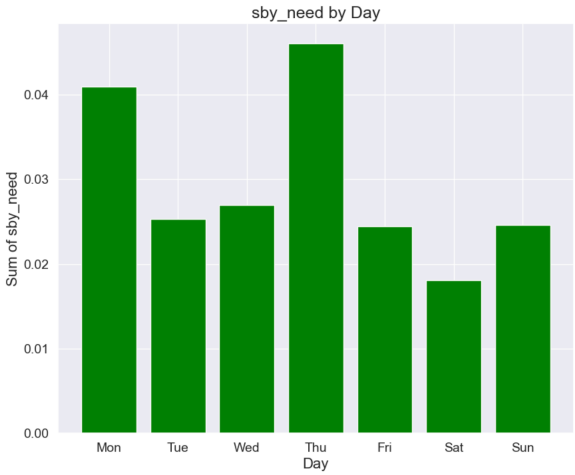
Next, we will explore the "sby_need" column alongside the newly introduced categorical features. We'll be comparing the standby need with the new features based on mean values.



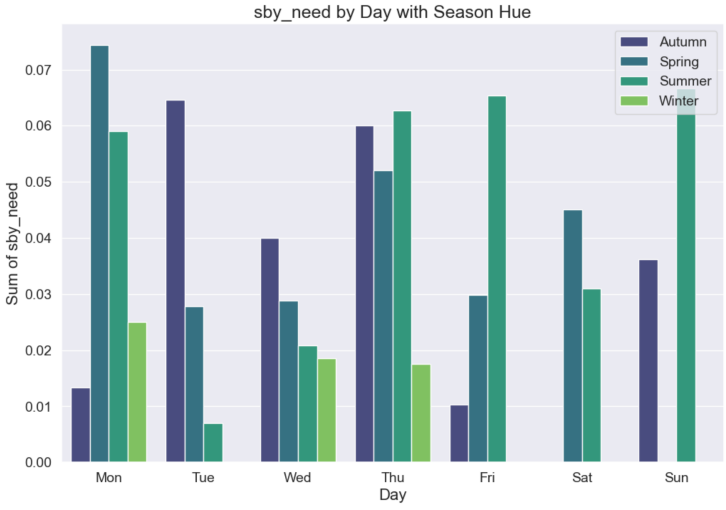

The graph depicting standby need vs. season illustrates that the Summer season demands the highest number of standby resources, followed by Spring and Autumn. In contrast, the Winter season exhibits the lowest standby demand.



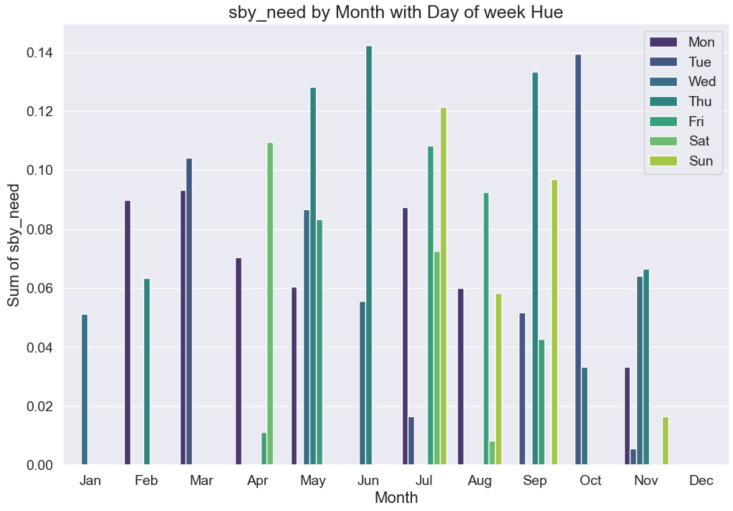

The graph depicting standby need by month highlights that July experiences the highest demand, followed by May and September. Conversely, January, February, and October exhibit relatively lower demand, while December shows no demand at all. The remaining months demonstrate average demand.




The graph depicting standby need vs. day of the week reveals that Thursday experiences the highest demand, closely followed by Monday. Saturday exhibits relatively low demand, while the rest of the days show an average level of demand.



Analyzing the graph illustrating standby need vs. day of the week with the season hue, we observe that Mondays in spring and Sundays in summer exhibit the highest demand, followed by Fridays and Thursdays in summer and Tuesdays and Thursdays in Autumn. There are days without any demand for some seasons, and a considerable amount of data reflects average demand.




Examining the graph depicting standby need vs. month with the day hue, we notice that Thursdays in June and Tuesdays in October have the highest demand, followed by Thursdays in September and May. There are days in various months with no demand, such as every day in December, and the graphs show that there is average demand in many months.

Up next, we'll apply the one-hot encoding technique to the recently introduced categorical variables. One-hot encoding is employed to convert categorical variables into numerical values for machine learning models. Its benefits include enabling the use of categorical variables in models that require numerical input and enhancing model performance by providing additional information about the categorical variable(“One Hot Encoding in Machine Learning,” 2019).

The table below displays the initial five rows, showcasing some of the columns that have undergone one-hot encoding.

| | date | n_sick | calls | n_duty | sby_need | dafted | year | day_of_week_Mon | day_of_week_Tue | day_of_week_Wed | ... | month_Jul | month_Aug | month_Sep |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2016-04-01 | 0.521127 | 0.750646 | 0.0 | 0.133333 | 0.0 | 2016 | 0 | 0 | 0 | ... | 0 | 0 | 0 |
| **2** | 2016-04-03 | 0.450704 | 0.736434 | 0.0 | 0.000000 | 0.0 | 2016 | 0 | 0 | 0 | ... | 0 | 0 | 0 |
| **3** | 2016-04-04 | 0.492958 | 0.511628 | 0.0 | 0.000000 | 0.0 | 2016 | 1 | 0 | 0 | ... | 0 | 0 | 0 |
| **4** | 2016-04-05 | 0.380282 | 0.552972 | 0.0 | 0.000000 | 0.0 | 2016 | 0 | 1 | 0 | ... | 0 | 0 | 0 |
| **5** | 2016-04-06 | 0.478873 | 0.392765 | 0.0 | 0.000000 | 0.0 | 2016 | 0 | 0 | 1 | ... | 0 | 0 | 0 |

5 rows × 30 columns

The following list includes the names of all available columns, incorporating the names of the categorical encoded columns.

```
['date', 'n_sick', 'calls', 'n_duty', 'sby_need', 'dafted', 'year',
 'day_of_week_Mon', 'day_of_week_Tue', 'day_of_week_Wed',
 'day_of_week_Thu', 'day_of_week_Fri', 'day_of_week_Sat',
 'day_of_week_Sun', 'month_Jan', 'month_Feb', 'month_Mar', 'month_Apr',
 'month_May', 'month_Jun', 'month_Jul', 'month_Aug', 'month_Sep',
 'month_Oct', 'month_Nov', 'month_Dec', 'season_Autumn', 'season_Spring',
 'season_Summer', 'season_Winter'],
```

## Modelling

During this phase, we make decisions on modelling techniques, create test designs, construct models, and evaluate their performance. This includes the selection of suitable algorithms, partitioning data into training and test sets, implementing code for model development, and interpreting the outcomes to ensure the models align with the project objectives(Hotz, 2018).

**Data Partitioning:** We will adopt a standard 70-30 split between test and train data, utilizing the outlier-clipped and scaled dataset.

### Standby Drivers Forecasting using Time Series Models

Time series analysis is a methodical approach to analyzing data points collected over time at consistent intervals. It focuses on understanding the behavior of variables at different time points and derives insights from changing features. The primary objectives include unraveling the workings of time series, exploring the impact of various factors, and predicting future values. The fundamental assumption is stationarity, ensuring the stability of the process over time. Time series analysis is crucial for prediction, forecasting, and examining time-based problem statements. It enables historical dataset analysis, pattern recognition, and understanding factors influencing variables over different periods. Utilizing time series aids in generating diverse time-based analyses and results(Pandian, 2021).

Next, we will initiate the modelling process by examining the monthly predictions of standby drivers using time series models. Monthly prediction involves forecasting the standby drivers' demand on a monthly basis. Given that the available data is recorded on a daily basis, we'll aggregate the information by date to derive monthly data. The initial five rows of the resulting monthly data are presented in the following table.

| | date | n_sick | calls | n_duty | sby_need | dafted | year | day_of_week_Mon | day_of_week_Tue | day_of_week_Wed | ... | month_Jul | month_Aug | month_ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2016-04-30 | 10.014085 | 11.923773 | 0.0 | 0.133333 | 0.0 | 56448 | 4 | 3 | 4 | ... | 0 | 0 | |
| **1** | 2016-05-31 | 7.140845 | 14.975452 | 0.0 | 2.600000 | 0.0 | 50400 | 3 | 4 | 4 | ... | 0 | 0 | |
| **2** | 2016-06-30 | 4.901408 | 11.493540 | 0.0 | 1.566667 | 0.0 | 40320 | 0 | 2 | 4 | ... | 0 | 0 | |
| **3** | 2016-07-31 | 5.943662 | 9.838501 | 0.0 | 0.733333 | 0.0 | 40320 | 2 | 3 | 2 | ... | 20 | 0 | |
| **4** | 2016-08-31 | 2.521127 | 10.802326 | 0.0 | 0.000000 | 0.0 | 40320 | 4 | 3 | 2 | ... | 0 | 20 | |

5 rows × 30 columns

In the Time Series Analysis (TSA) model preparation, it's essential to evaluate the dataset's stationarity through statistical tests. One such test is the Augmented Dickey-Fuller (ADF) test, which we'll utilize to determine the stationarity of our data(Pandian, 2021).

The ADF Test operates under the following assumptions(Pandian, 2021):

- Null Hypothesis (H0): Series is non-stationary
- Alternate Hypothesis (H1): Series is stationary

$$p - value > 0.05 \; Fail\ to\ reject\ (H0)$$

$$p - value \leq 0.05 \; Accept\ (H1)$$

Following are the outcomes of our time series analysis for the monthly aggregated data to assess the stationarity of the target variable 'sby_need', conducted through the “adfuller()” function imported from the “statsmodels.tsa.stattools” library.

```
Test Statistic                  -5.576156
p-value                          0.000001
#lags used                       0.000000
number of observations used     37.000000
dtype: float64
criticality 1% : -3.6209175221605827
criticality 5% : -2.9435394610388332
criticality 10% : -2.6104002410518627
```

The results indicate that the p-value is below 0.05, signifying that the data is stationary. Additionally, we can also see that the test statistic value is below the 1% critical threshold. With the confirmation of our data's stationarity, we can proceed to the model building process.

## Time Series Models for Monthly Prediction (ARIMA and SARIMA)

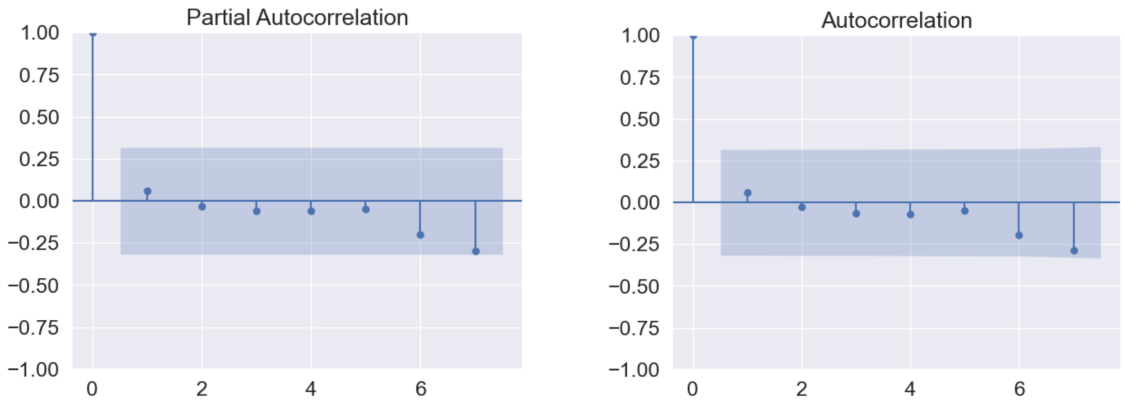
ARIMA, or Autoregressive Integrated Moving Average, is a statistical analysis model leveraging time series data for understanding the dataset or forecasting future trends. An autoregressive model predicts future values based on past values(*Autoregressive Integrated Moving Average (ARIMA) Prediction Model*, n.d.). To implement the ARIMA model for monthly predictions, we determine three key components: AutoRegressive (AR) with parameter 'p' representing the correlation between a current observation and its past observations, Moving Average (MA) with parameter 'q' capturing the correlation between a current observation and a residual error from a moving average model, and Integrated (I) with parameter 'd,' indicating the number of differencing operations required to achieve stationarity in the time series(Learnerea, 2022).

SARIMA, which stands for Seasonal Autoregressive Integrated Moving Average, is a statistical technique employed for time series data forecasting. It combines autoregressive (AR) models, moving average (MA) models, and differencing, incorporating a seasonality element(Amrullah, 2023). The SARIMA model also involves three crucial components: Seasonal AutoRegressive (SAR) denoted by 'p,' Seasonal Moving Average (SMA) denoted by 'q,' and Integrated (I) represented by 'd.' The SAR component captures the correlation between the current observation and its past observations with seasonality, while the SMA component captures the correlation between the current observation and the residual error from a seasonal moving average model. The Integrated component indicates the number of differencing operations needed to achieve stationarity in the time series(Learnerea, 2022).

For both the ARIMA and SARIMA models, the parameter 'p' is determined from the respective Partial AutoCorrelation Function (PACF) plot, while the parameter 'q' is derived from the AutoCorrelation Function (ACF) plot(Learnerea, 2022). The Auto-Correlation Function (ACF) gauges the similarity between a value in a time series and its preceding value, revealing trends and the influence of past values on the current ones. On the other hand, Partial Auto-Correlation (PACF) is akin to ACF but specifically highlights the direct correlation between a sequence and itself after removing intermediary effects, providing insights into the sequence order(Pandian, 2021).

Since the data is already stationary, there is no need to implement time-shifting methods to make the data stationary, and the differencing parameter "d" remains 0. In the SARIMA model, in addition to 'p,' 'q,' and 'd,' we introduce a seasonal component denoted by 's,' which is set to 12, reflecting the observed yearly seasonal pattern in our data.

The PACF and ACF plots, generated using the "plot_pacf()" and "plot_acf()" functions from the "statsmodels.graphics.tsaplots" library, are depicted in the following figures:




In both models, the values for 'p' and 'q' are chosen from the lines outside the shaded area of the PACF and ACF graphs(Learnerea, 2022). Based on the results of the graphs, we can conclude that both 'p' and 'q' are determined to be 0. With these results, the parameters 'p', 'd', and 'q' are all set to 0. When all three parameters are set to 0, the model essentially becomes a constant model, which does not capture any temporal patterns or trends in the data. Consequently, we can infer that these models are not best suited for time series forecasting for monthly data.

## Time Series Models for Daily Prediction

Given the unsuitability of the previous time series models for monthly predictions, we'll redirect our efforts towards implementing new models tailored for daily predictions of standby drivers. We'll commence the modeling process by exploring daily predictions, aiming to forecast the demand for standby drivers on a day-to-day basis.

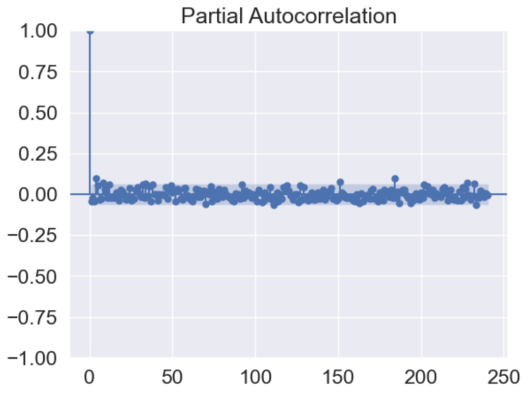
Just as we assessed the stationarity of data for monthly models, a similar examination is required for daily models. Hence, we will employ the Augmented Dickey-Fuller (ADF) test to evaluate the stationarity of the data.

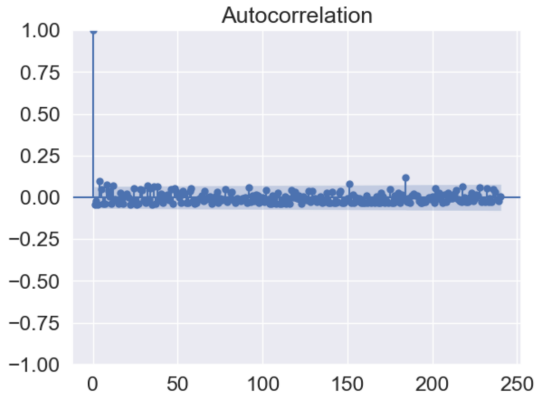
We have applied the ADF test to assess the stationarity of the target variable 'sby_need'. The outcomes of the test are as follows:

```
Test Statistic                -1.216725e+01
p-value                        1.447813e-22
#lags used                     4.000000e+00
number of observations used    8.860000e+02
dtype: float64
criticality 1% : -3.4377521975315783
criticality 5% : -2.864807640843869
criticality 10% : -2.568509921477307
```

The results indicate that the p-value is below 0.05, signifying that the data is stationary. Additionally, we can also see that the test statistic value is below the 1% critical threshold. With the confirmation of our data's stationarity, we can proceed to the model building process.

The PACF and ACF plots are illustrated in the following figures:



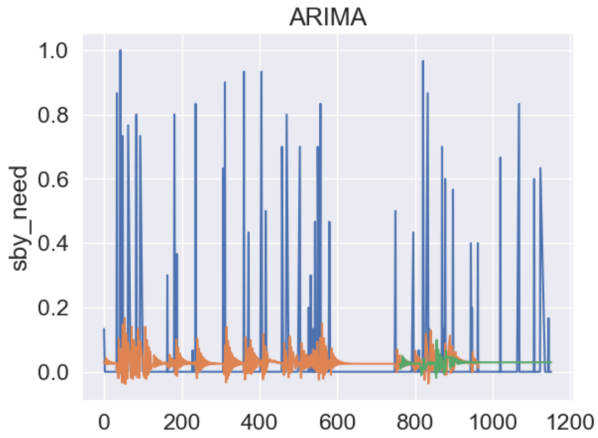




In both models, the values for 'p' and 'q' are selected from the points outside the shaded area in the PACF and ACF plots. It's important to note that we can choose any point outside the shaded area(Learnerea, 2022); in our case, we are selecting the first point at position 4 in both plots. Consequently, the parameters 'p' and 'q' are set to 4, while 'd' remains 0. We will use these values to implement ARIMA and SARIMA models. However, for the SARIMA model, as it involves a seasonal component denoted by 's,' we will set its value to '7' which captures weekly patterns.

## ARIMA

The ARIMA model is built using the "ARIMA()" function with an order of (4, 0, 4). Training set forecasting and test set forecasting are performed using the "get_prediction()" method. Predicted values are extracted and stored. The results are visualized using the Seaborn library to compare actual 'sby_need' values with the ARIMA model predictions for both training and test sets. Next, we calculated Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE) for the ARIMA model on both the training and test sets. These metrics assess the accuracy of the ARIMA model predictions.

The model outcomes are presented below:

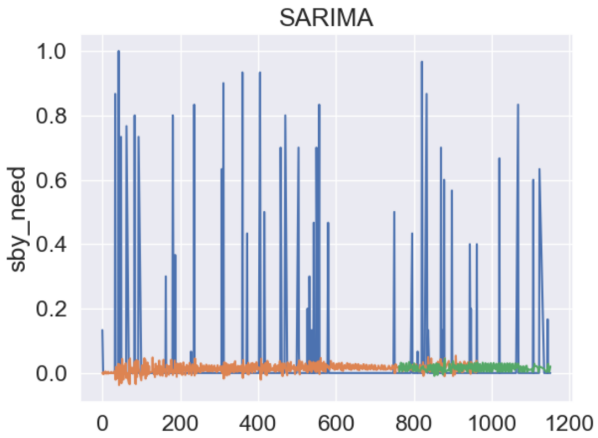



| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.00 | 0.05 | 79.29 |
| Test | 0.00 | 0.05 | 33.33 |

## SARIMA

The SARIMA model is created using the "SARIMAX()" function with an order of (4, 0, 4) and a seasonal order of (4, 0, 4, 7) to capture weekly patterns. Training set forecasting and test set forecasting are performed using the "get_prediction()" method, and the predicted values are stored. The results are visualized using the Seaborn library to compare actual 'sby_need' values with the SARIMAX model predictions for both training and test sets. Next, Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE) for the SARIMAX model on both the training and test sets are calculated to assess the accuracy of the predictions.

The model outcomes are presented below:




| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.13 | 0.04 | 84.13 |
| Test | 0.05 | 0.02 | 33.99 |

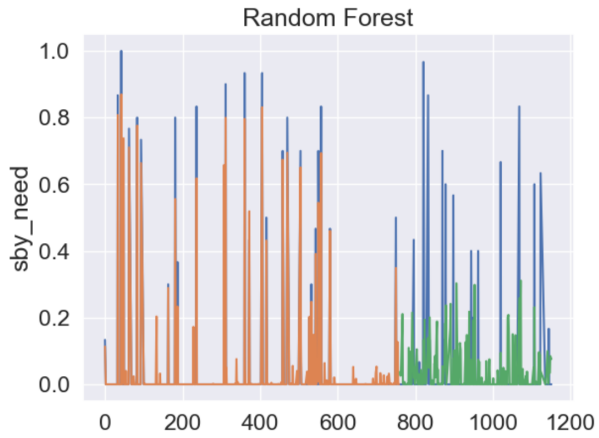
## Regression Models for Daily Prediction

## Random Forest

Random Forest Regression, a supervised learning algorithm utilizing ensemble learning for regression, constructs multiple decision trees during training and predicts the mean of the classes. Known for its power and accuracy, Random Forest Regression excels in various scenarios, particularly in handling features with non-linear relationships(Chaya, 2022).

In our efforts, the Random Forest Regression model is created using the "RandomForestRegressor" with 1000 estimators. Training and test set predictions are generated, and the results are stored. The seaborn library is utilized for visualizing the actual 'sby_need' values, training predictions, and test predictions. Subsequently, Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE) are calculated to evaluate the accuracy of the Random Forest Regression model on both the training and test sets. The metrics provide insights into the model's performance.

The model outcomes are presented below:



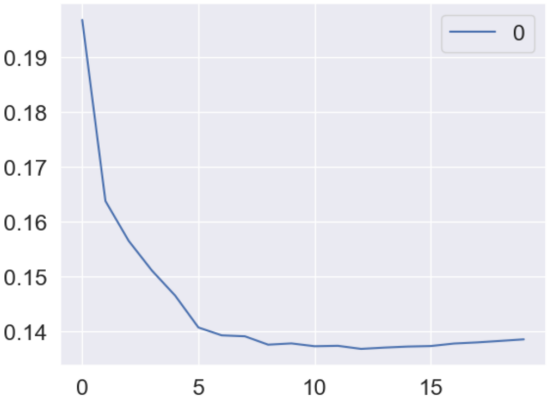

| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.00 | 0.00 | 19.27 |
| Test | 0.01 | 0.05 | 77.86 |

## K-Nearest Neighbours

K-Nearest Neighbors (KNN) is a widely used machine learning algorithm for classification and regression tasks. It operates on the principle that similar data points share common labels or values. During training, the algorithm stores the entire dataset. When predicting, it computes distances between the input data point and training examples, identifying the K nearest neighbours. For regression, it predicts the value by calculating the average or weighted average of the target values of the K neighbours. While KNN is simple and intuitive, optimal performance requires careful tuning of parameters like K and the distance metric(Srivastava, 2018).

Before implementing the KNN model, we'll compute Root Mean Square Error(RMSE) values for various 'k' values. The optimal k value for KNN is usually determined by assessing the model's performance across different k values and selecting the one that yields the lowest error. We examined k values within a range of 20, and the results are as follows:

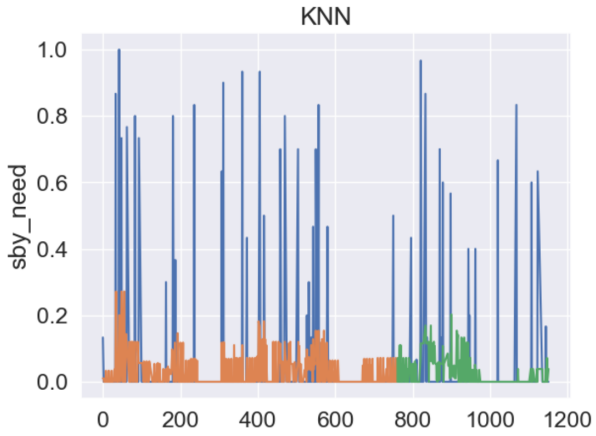
```
RMSE value for k=  1 is: 0.19672905663554072
RMSE value for k=  2 is: 0.16371452624253974
RMSE value for k=  3 is: 0.15644733715100412
RMSE value for k=  4 is: 0.15103979026111866
RMSE value for k=  5 is: 0.14646752649788267
RMSE value for k=  6 is: 0.1406615729011529
RMSE value for k=  7 is: 0.13922837313423578
RMSE value for k=  8 is: 0.13905418749728712
RMSE value for k=  9 is: 0.13752564797562233
RMSE value for k=  10 is: 0.1377563944514808
RMSE value for k=  11 is: 0.1372396356618235
RMSE value for k=  12 is: 0.13731781403991059
RMSE value for k=  13 is: 0.13675786719400687
RMSE value for k=  14 is: 0.1369934201973891
RMSE value for k=  15 is: 0.13718067263614472
RMSE value for k=  16 is: 0.1372760902087227
RMSE value for k=  17 is: 0.1377350083840671
RMSE value for k=  18 is: 0.13794315935940132
RMSE value for k=  19 is: 0.13821720319541
RMSE value for k=  20 is: 0.13849743757769933
```




The lowest RMSE value is observed for K=13. Hence, we consider K=13 as the optimal choice for our model.

In our efforts, the K-Nearest Neighbors (KNN) regression model is implemented with 13 neighbours. The model is trained on the provided training data. Subsequently, predictions are generated for both the training and test sets. The predictions are stored. A visualization is created to compare the actual 'sby_need' values with the training and test predictions using the Seaborn library. The model's performance is evaluated by calculating Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE) for both training and test sets.

The results of the model with the chosen k-value are shown below:



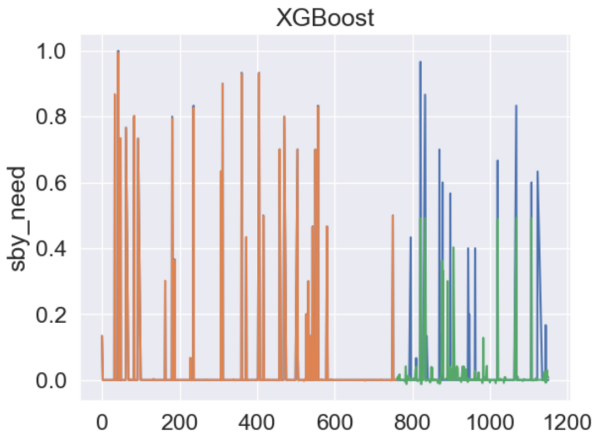

| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.01 | 0.04 | 77.33 |
| Test | 0.01 | 0.05 | 83.44 |

## XGBoost

XGBoost, short for eXtreme Gradient Boosting, is named with the goal of pushing computational resource limits for boosted tree algorithms. While the library is focused on computational speed and model performance, it offers advanced features without unnecessary complexities. XGBoost excels in two key aspects: execution speed, outperforming other gradient boosting implementations, and model performance, particularly on structured or tabular datasets for classification and regression problems. The algorithm itself implements gradient boosting decision trees, sequentially adding models to correct errors made by prior ones. This ensemble technique uses a gradient descent algorithm to minimize loss when incorporating new models, catering to both regression and classification predictive modeling(Brownlee, 2016).

In our efforts, the XGBoost model is constructed using the "XGBRegressor()" function. Training and test set predictions are generated using the "predict()" method. The predicted values are stored, and the results are visualized using the Seaborn library to compare actual 'sby_need' values with XGBoost model predictions for both training and test sets. Following this, Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE) are computed for the XGBoost model on both training and test sets, serving as metrics to assess the model's predictive accuracy.

The model outcomes are presented below:



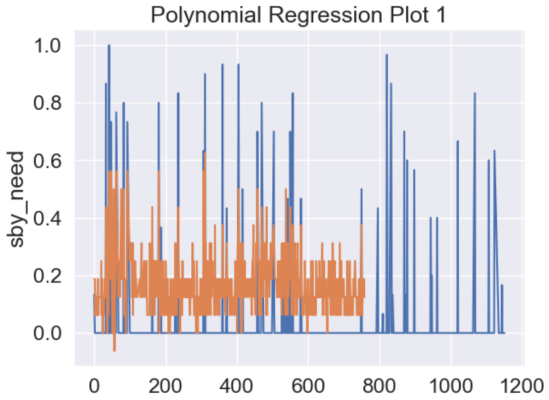

| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.00 | 0.00 | 0.30 |
| Test | 0.01 | 0.03 | 74.81 |

## Polynomial Regression

Polynomial regression, a variant of linear regression, addresses non-linear relationships between dependent and independent variables by introducing polynomial terms. It models the relationship as an nth-degree polynomial function, with quadratic models for degree 2, cubic models for degree 3, and so forth. Recognized as Polynomial Linear Regression, this technique utilizes polynomial equations to capture and represent the complex associations in the data(Agrawal, 2021).

In our efforts, we have implemented a Polynomial Regression model with a specified degree (2 in this case) using the “PolynomialFeatures” and “LinearRegression” from scikit-learn. The model is fitted to the training data, and predictions are made for both the training and test sets. The results are then visualized using the Seaborn library, comparing the actual 'sby_need' values with the Polynomial Regression model predictions. Plots are used for training and testing predictions to showcase potential variations in the results. The code also calculates and prints various evaluation metrics, such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE), assessing the accuracy of the Polynomial Regression model on both training and test sets.

The model outcomes are presented below:




Given the poor performance of the model, the graph fails to effectively capture both train and test patterns in a single plot. Hence, two separate plots are presented—one depicting model results for the training set and the other for both.

| **Metric** | **MSE** | **MAE** | **MAPE** |
|---|---|---|---|
| Train | 0.03 | 0.15 | 66.11 |
| Test | 5.81 | 1,39717E+11 | 2,92349E+13 |

## Evaluation

In this phase, we assess the outcomes, utilizing three evaluation metrics MSE, MAE, and MAPE to gauge the effectiveness of our predictive models.

The **Mean Squared Error (MSE)** serves as a measure of how well a regression line aligns with a set of data points. It represents the expected value of the squared error loss. Calculated as the sum of squared errors divided by the sample size, a higher MSE implies greater dispersion of data points around its mean, while a lower MSE indicates the opposite. A smaller MSE is preferable as it signifies that data points are closely clustered around the central moment (mean), enhancing the model's accuracy(*Mean Squared Error*, n.d.).

$$MSE = \frac{1}{n}\sum(y - \hat{y})^2$$

where:

$$n: sample\ size;\ y: actual\ data\ value;\ \hat{y}: predicted\ data\ value;$$

**MAE, or Mean Absolute Error**, gains popularity due to its unit consistency with predicted target values. Its linear nature makes changes in MAE intuitive, treating different errors equally without assigning varied weights, resulting in linearly increasing scores with error increments. Calculated as the sum of absolute errors divided by the sample size, MAE ensures positive differences between expected and predicted values by utilizing the Absolute mathematical function(*Mean Absolute Error - an Overview | ScienceDirect Topics*, n.d.).

$$MAE = \frac{1}{n}\sum|y - \hat{y}|$$

where:

$$n: sample\ size;\ y: actual\ data\ value;\ \hat{y}: predicted\ data\ value;$$

The **Mean Absolute Percentage Error (MAPE)** quantifies the accuracy of a forecasting system. It expresses this accuracy in percentage terms and is computed as the sum of absolute percent error for each time period divided by the sample size, calculated by subtracting actual values from forecasted values and then dividing by actual values. MAPE is a widely used measure for forecasting errors, particularly beneficial when dealing with scaled percentage units, simplifying comprehension. It serves effectively as a loss function in regression analysis and model evaluation, performing optimally in scenarios without extremes or zeros in the data(Tim, 2022).

$$MAPE = \frac{1}{n}\sum\left|\frac{y - \hat{y}}{y}\right|$$

where:

$$n: sample\ size;\ y: actual\ data\ value;\ \hat{y}: predicted\ data\ value;$$

Below is the evaluation results table for all our daily prediction models, considering a 70/30 split for both the training and testing datasets.

| **Model** | **MSE** | | **MAE** | | **MAPE** | |
|---|---|---|---|---|---|---|
| | Train | Test | Train | Test | Train | Test |
| ARIMA | 0.00 | 0.00 | 0.05 | 0.05 | 79.29 | 33.33 |
| SARIMA | 0.13 | 0.05 | 0.04 | 0.02 | 84.13 | 33.99 |
| Random Forest | 0.00 | 0.01 | 0.00 | 0.05 | 19.27 | 77.86 |
| KNN | 0.01 | 0.01 | 0.04 | 0.05 | 77.33 | 83.44 |
| XG Boost | 0.00 | 0.01 | 0.00 | 0.03 | 0.30 | 74.81 |
| Polynomial Regression | 0.03 | 5.81 | 0.15 | 1,39717E+11 | 66.11 | 2,92349E+13 |

Looking at the performance metrics for both the training and test sets, XG Boost appears to be the most promising model. It consistently achieves low MSE, MAE, and MAPE values for both datasets compared to other models. The low values indicate that XG Boost is effectively capturing the patterns in the data and providing accurate predictions. Therefore, based on the presented results, XG Boost emerges as the most favorable choice among the models considered.

We've decided to choose XGBoost as our model, serving as the baseline. Our next steps involve optimizing the model through various strategies, such as adjusting the train/test split and iteratively tuning its hyperparameters. Ultimately, we aim to deploy the most effective model, whether it's the baseline XGBoost or an enhanced version based on the outcome.

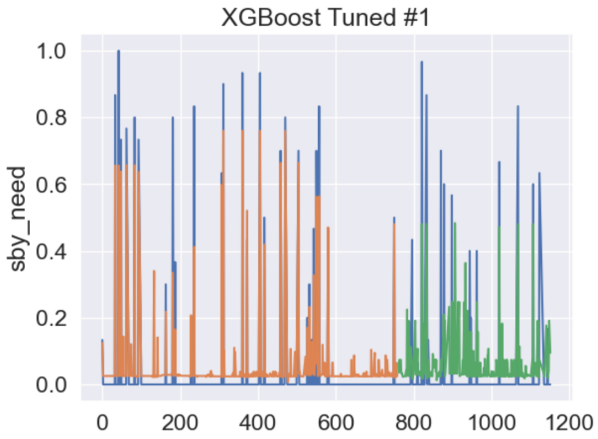
## Model Optimization

### *Tuning the baseline model with a 70/30 train-test split*

In our efforts to improve the performance of the XGBoost regressor, our baseline model, we are employing the "GridSearchCV()" function with the predefined 70-30 train-test split, conducting an exhaustive search over specified parameter values for an estimator. GridSearchCV implements "fit" and "score" methods, optimizing the parameters of the estimator through cross-validated grid-search over a parameter grid(*Sklearn.Model_selection.GridSearchCV*, n.d.). The parameters undergoing tuning include 'n_estimators' (number of trees), 'learning_rate' (step size shrinkage to prevent overfitting), and 'max_depth' (maximum depth of a tree). The potential values for these hyperparameters are specified in the “param_grid”. GridSearchCV systematically explores various combinations of hyperparameters through cross-validation to assess their performance. The optimal combination of hyperparameters is then revealed, along with its corresponding best score (negative mean squared error). The best model is extracted using

these optimal hyperparameters, and predictions are generated for both the training and test sets using this fine-tuned model. This process aims to enhance the model's predictive capabilities by identifying the most effective hyperparameter configuration.

The model outcomes are presented below:

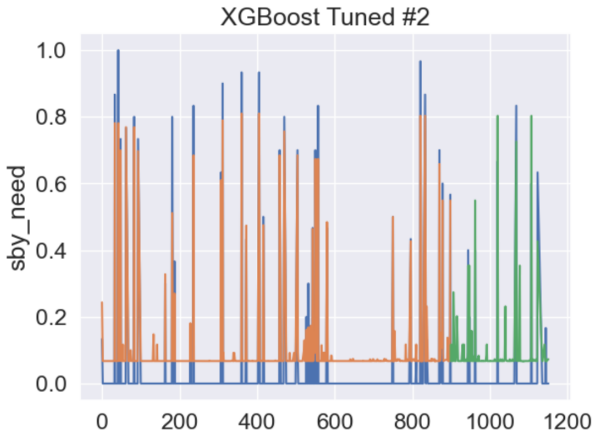



| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.00 | 0.03 | 27.47 |
| Test | 0.01 | 0.06 | 52.26 |

### *Tuning the baseline model with a 80/20 train-test split*

In this approach, we aim to optimize the hyperparameters of the baseline model using “GridSearchCV”, just like our previous attempt, but this time with an 80/20 train-test split.

The model outcomes are presented below:




| Metric | MSE | MAE | MAPE |
|---|---|---|---|
| Train | 0.00 | 0.06 | 21.13 |
| Test | 0.00 | 0.07 | 35.24 |

### Concluding Evaluation

| **Model** | **MSE** | | **MAE** | | **MAPE** | |
|---|---|---|---|---|---|---|
| | Train | Test | Train | Test | Train | Test |
| XG Boost Baseline | 0.00 | 0.01 | 0.00 | 0.03 | 0.30 | 74.81 |
| XG Boost Tuned (70-30) | 0.00 | 0.01 | 0.03 | 0.06 | 27.47 | 52.26 |
| XG Boost Tuned (80-20) | 0.00 | 0.00 | 0.06 | 0.07 | 21.13 | 35.24 |

When comparing the performance metrics of the XG Boost Baseline, XG Boost Tuned (70-30), and XG Boost Tuned (80-20), it is evident that XG Boost Tuned (80-20) outperforms the other models. The model exhibits lower Mean Squared Error (MSE) and Mean Absolute Error (MAE) values for both the training and test sets, indicating superior accuracy and precision. Moreover, the Mean Absolute Percentage Error (MAPE) values for the test set are notably lower in XG Boost Tuned (80-20), reflecting better predictive capabilities and reduced deviation from the actual values. Therefore, based on these evaluation metrics, XG Boost Tuned (80-20) emerges as the most effective and robust model for predicting standby driver availability in both training and test scenarios.

## Deployment

Based on the evaluation results, the tuned XGBoost model with an 80/20 train-test split demonstrates superior performance in predicting standby drivers for Berlin's red-cross rescue service. This superior performance suggests that deploying this model could lead to a higher percentage of standby drivers being activated. Additionally, it indicates a potential reduction in situations where there are not enough standbys available, addressing a limitation of the current approach.

The model utilizes these columns as features to make predictions. Each column represents a different aspect or characteristic related to standby drivers' demand. For instance, 'n_sick' and 'calls' provide information about sick leave and emergency call volumes, respectively. 'day_of_week' and 'month' columns capture the day and month of the observation, helping the model account for weekly and monthly patterns. Similarly, 'season' columns account for seasonal variations. By considering these features collectively, the model aims to understand the multifaceted factors influencing the demand for standby drivers and make accurate predictions.

The model is built to ensure compliance with the criterion that the duty plan must be completed by the 15th of the current month for the upcoming month by incorporating temporal information during the prediction and planning of standby drivers. By leveraging historical data, the model takes into account the relevant time frames, allowing it to meet the operational requirement of finalizing the duty plan in a timely manner.

The model's GUI provides a user-friendly interface to showcase the standby duty plan for the respective month based on the selected timeframe by the HR, and the prediction generation criteria based on the day of the month. The GUI also allows HR to manually input parameters such as current or future date, number of sick employees, and approximate emergency calls, on outlier days particularly where the model prediction might not have considered these outlier scenarios. This empowers HR to re-evaluate the predictions in case of significant variations

in the monthly forecast. After submitting these inputs, the model processes the information and provides clear predictions for the required number of standby drivers. The output enables HR to make well-informed scheduling decisions based on the model's recommendations. This intuitive design of the GUI enhances the efficiency and accessibility of the scheduling process for HR personnel.

**Here are some scenarios where the model may encounter issues:**

- **Unforeseen Events:** The model may struggle to predict accurately in the presence of unforeseen events or emergencies that were not part of the historical data. Sudden spikes in demand due to unexpected situations may lead to deviations from the predicted values.
- **Drastic Changes in Operations:** If there are significant changes in the operations or policies of the organization that were not captured in the training data, the model may fail to adapt to these changes, resulting in inaccurate predictions.
- **Data Anomalies:** Outliers or anomalies in the historical data that do not reflect the typical patterns of standby driver requirements can affect the model's ability to generalize and make accurate predictions.
- **Seasonal Variations:** If there are significant variations in seasonal patterns, the model may struggle to provide accurate predictions.

It's crucial to continuously monitor the model's performance and update it as needed to address these potential challenges and enhance its predictive capabilities.

**Here are some strategies to sustain/enhance the model performance over time:**

- Validate and monitor the collected future data to ensure its relevance and usability.
- Regularly assess the performance of the model to ensure it meets expectations.
- Update the model with new training data every quarter or half-year to sustain its effectiveness.
- Explore potential new features related to duty planning that could enhance the model's performance.

## Git Repository Proposal

The following layout illustrates the potential structure of our project's final Git repository:

```
project_repository/
│
├── Notebook/ (Notebook)
│   ├── README.md (Markdown)
│   ├── requirements.txt (Text)
│   ├── setup.py (Python)
│   └── Source/ (Folder)
│       ├── __init__.py (Python)
│
├── Data/ (Folder)
```

```
│   ├── raw_data.xlsx (Excel)
│
├── Components/ (Folder)
│   ├── data_ingestion.py (Python)
│   ├── data_preprocessing.py (Python)
│   ├── modeling.py (Python)
│   ├── deployment.py (Python)
│   ├── exception.py (Python)
│   └── logger.py (Python)
│
└── Models/ (Folder)
    └── saved_models.py (Python)
```

## Conclusion

The project commenced by adhering to the CRISP-DM methodology. Initially, we delved into Berlin's Red Cross rescue service's business workflow and identified bottlenecks. Utilizing the industry-standard Data Science methodology(CRISP-DM), we devised a project roadmap aimed at boosting standby driver activation rates, reducing overstaffing, and minimizing understaffing scenarios. Following this, we analysed the available historical data on standby driver allocation to comprehend the features thoroughly. Subsequently, we prepared the data in a manner conducive to building a predictive model that could forecast standby drivers while meeting the requirement of completing the predictions for the upcoming month by the model before the 15th of the current month.

In the modelling phase, we explored two distinct approaches. Initially, we attempted the monthly prediction of standby drivers using Time Series models, wherein the number of standby drivers for the entire month remained static. However, this approach did not yield the expected outcome, prompting us to discard it. We subsequently shifted our approach to daily prediction of standby drivers. For daily prediction, we employed Time Series models like ARIMA and SARIMA, along with regression models such as Random Forest, K-Nearest Neighbors, XGBoost, and Polynomial regression. Throughout this process, we utilized a 70/30 Train-Test split.

In the evaluation phase, we utilized various error metrics, including MSE, MAE, and MAPE, to assess the performance of our models. After evaluating each model and analyzing the scores, XGBoost emerged as the best-performing model, serving as our baseline. Subsequently, we focused on optimizing this baseline model to enhance its performance. The optimized model exhibited a notable improvement compared to the baseline, and we selected it for deployment. We also introduced an intuitive Graphical User Interface(GUI) explanation and proposed a GIT repository for the project.

The implementation of this predictive model yields significant advantages to Berlin's red-cross rescue service. Notably, it enhances overall efficiency by dynamically allocating standby drivers, thereby reducing overstaffing. This improvement contributes to enhanced emergency services quality and responsiveness by minimizing instances of understaffing. Furthermore, timely planning, achieved by meeting monthly deadlines, guarantees preparedness for the upcoming months. To sum up, our predictive model, driven by data, will transform standby duty planning, highlighting the capacity to optimize resources and enhance emergency response services for Berlin's red-cross rescue service.

# References


Agrawal, R. (2021, July 9). Master Polynomial Regression With Easy-to-Follow Tutorials. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2021/07/all-you-need-to-know-about-polynomial-regression/

Amrullah, A. (2023, June 20). SARIMA Model for Forecasting Currency Exchange Rates. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2023/06/sarima-model-for-forecasting-currency-exchange-rates/

*Autoregressive Integrated Moving Average (ARIMA) Prediction Model*. (n.d.). Investopedia. Retrieved January 8, 2024, from https://www.investopedia.com/terms/a/autoregressive-integrated-moving-average-arima.asp

Box plot. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Box_plot&oldid=1139942868

Brownlee, J. (2016, August 16). A Gentle Introduction to XGBoost for Applied Machine Learning. *MachineLearningMastery.Com*. https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

Brownlee, J. (2017, January 29). How to Decompose Time Series Data into Trend and Seasonality. *MachineLearningMastery.Com*. https://machinelearningmastery.com/decompose-time-series-data-trend-seasonality/

Brownlee, J. (2020, June 9). How to Use StandardScaler and MinMaxScaler Transforms in Python. *MachineLearningMastery.Com*. https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/

Chaya. (2022, April 14). *Random Forest Regression*. Medium. https://levelup.gitconnected.com/random-forest-regression-209c0f354c84

*Correlation in Statistics: Correlation Analysis Explained*. (n.d.). Statistics How To. Retrieved January 7, 2024, from https://www.statisticshowto.com/probability-and-statistics/correlation-analysis/

Cross-industry standard process for data mining. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Cross-industry_standard_process_for_data_mining&oldid=1191822755

Exploratory data analysis. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Exploratory_data_analysis&oldid=1145451809

Feature engineering. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Feature_engineering&oldid=1193825578

Feature (machine learning). (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Feature_(machine_learning)&oldid=1150327846

Hotz, N. (2018, September 10). What is CRISP DM? *Data Science Process Alliance*. https://www.datascience-pm.com/crisp-dm-2/

Learnerea (Director). (2022, December 8). *Time Series Analysis using Python| ARIMA & SARIMAX Model Implementation | Stationarity Handling*. https://www.youtube.com/watch?v=O5pataOw33Y

McDonald, A. (2022, July 29). *Seaborn Pairplot—Enhance your Data Understanding With a Single Plot*. Medium. https://towardsdatascience.com/seaborn-pairplot-enhance-your-data-understanding-with-a-single-plot-bf2f44524b22

*Mean Absolute Error—An overview | ScienceDirect Topics*. (n.d.). Retrieved January 9, 2024, from https://www.sciencedirect.com/topics/engineering/mean-absolute-error

*Mean Squared Error: Overview, Examples, Concepts and More | Simplilearn*. (n.d.). Simplilearn.Com. Retrieved January 9, 2024, from https://www.simplilearn.com/tutorials/statistics-tutorial/mean-squared-error

One Hot Encoding in Machine Learning. (2019, June 12). *GeeksforGeeks*. https://www.geeksforgeeks.org/ml-one-hot-encoding-of-datasets-in-python/

Outlier. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Outlier&oldid=1181881199

Pandian, S. (2021, October 23). Time Series Analysis and Forecasting | Data-Driven Insights (Updated 2023). *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-to-time-series-analysis/

Patil, P. (2023, September 24). Outlier Detection and Removal using the IQR Method. *Medium*. https://medium.com/@pp1222001/outlier-detection-and-removal-using-the-iqr-method-6fab2954315d

*Pearson Correlation—An overview | ScienceDirect Topics*. (n.d.). Retrieved January 7, 2024, from https://www.sciencedirect.com/topics/computer-science/pearson-correlation

*Sklearn.model_selection.GridSearchCV*. (n.d.). Scikit-Learn. Retrieved January 9, 2024, from https://scikit-learn/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Srivastava, T. (2018, March 25). A Complete Guide to K-Nearest Neighbors (Updated 2024). *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

Tim. (2022, May 27). *Mean Absolute Percentage Error (MAPE)*. Statistics How To. https://www.statisticshowto.com/mean-absolute-percentage-error-mape/

## Appendix

**Note:**

- Execute each code input section below (python code) individually, identified by In[number] (e.g., In[1]) along with import statements, for the corresponding outputs.
- The code follows a sequential order aligned with the assignment sections, facilitating easy reference to code snippets alongside the associated output screenshots above.

## Project Code:

**# In[1]:**

```
# Import statements

import math
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.neighbors import KNeighborsRegressor
from sklearn import neighbors
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.linear_model import LinearRegression

import xgboost as xgb
from xgboost import XGBRegressor

from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
```

**# In[2]:**

```
# Reading the dataset and converting it into a pandas dataframe
file_name= "/Users/ajaychandraas/Library/CloudStorage/OneDrive-IUInternationalUniversityofAppliedSciences/3rd Sem
Subjects/Case Study Model Engineering/sickness_table.csv"

read_file = pd.read_csv(file_name)
df= pd.DataFrame(read_file)

# Dropped the Index column from the DataFrame as it is unnecessary for our use in Pandas.
df = df.drop(df.columns[0], axis=1)
df.head(5)
```

**# In[3]:**

```
# Fetching the summary statistics
df.describe()
```

**# In[4]:**

```
# Fetching a concise overview of the dataset
df.info()
```

**# In[5]:**

```
# Checking the number of unique values in each column
unique = df.nunique()
unique
```

**# In[6]:**

```
# Dropping columns not required for further process
df.drop("n_sby", axis=1, inplace=True)
df.columns
```

**# In[7]:**

```
# Selecting only numerical columns for boxplot
numeric_data = df.select_dtypes(include='number')

# Setting up subplots
fig, axes = plt.subplots(nrows=len(numeric_data.columns), figsize=(8, 2 * len(numeric_data.columns)))
fig.tight_layout(pad=3.0)

# Plotting boxplots for each numerical column
for i, column in enumerate(numeric_data.columns):
    sns.boxplot(ax=axes[i], x=numeric_data[column])
```

```
    axes[i].set_title(f'Boxplot of {column}')
    axes[i].set_ylabel(column)
    axes[i].grid(True)

plt.show()
```

**# In[8]:**

```
# Plotting the correlation matrix for the selected columns
correlation_matrix = df[['n_sick', 'calls', 'n_duty', 'sby_need', 'dafted']].corr()

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap')
plt.show()
```

**# In[9]:**

```
# Plotting a pair plot for the selected columns
sns.set(font_scale=1.5)
pair_plot = sns.pairplot(df[['n_sick', 'calls', 'n_duty', 'sby_need', 'dafted']])
pair_plot.fig.suptitle("Pair Plot", y=1.02)
plt.show()
```

**# In[10]:**

```
# Plotting "date vs n_sby" plot for each year
df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].dt.year

selected_year =  [2016,2017,2018,2019]
for i in selected_year:
    data_yearly = df[df['year'] == i]
    plt.figure(figsize=(12, 6))
    plt.plot(data_yearly['date'], data_yearly['sby_need'], linestyle='-', color='red')
    plt.title(f'Stand by Drivers {i}', fontsize=18)
    plt.xlabel('Date', fontsize=16)
    plt.ylabel('Number of Standby Drivers', fontsize=16)
    plt.xticks(fontsize=14, rotation=45)
    plt.yticks(fontsize=14)
    plt.xticks(rotation=45)
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

**# In[11]:**

```
# Plotting Seasonal Decomposition Plots
decomposition = seasonal_decompose(df['sby_need'], period=365)  # Adjust the period as needed
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.figure(figsize=(10, 8))
plt.subplot(411)
plt.plot(df['date'], df['sby_need'], label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(df['date'], trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(df['date'], seasonal, label='Seasonal')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(df['date'], residual, label='Residual')
plt.legend(loc='best')
plt.tight_layout()
```

**# In[12]:**

```
# Clipping outliers
numeric_data = df.select_dtypes(include='number').drop(columns=['year', 'n_duty', 'dafted'])

# Detecting outliers using IQR method
Q1 = numeric_data.quantile(0.25)
Q3 = numeric_data.quantile(0.75)
IQR = Q3 - Q1

# Defining boundaries for outlier detection
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

```
# Identifying outliers
outliers = ((numeric_data < lower_bound) | (numeric_data > upper_bound)).any(axis=1)

# Removing outliers
data_cleaned = df[~outliers]

# Visualizing boxplots before and after outlier treatment for chosen features
plt.figure(figsize=(16, 8))

plt.subplot(1, 2, 1)
plt.boxplot(numeric_data.values, vert=True, patch_artist=True, labels=numeric_data.columns)
plt.title('Boxplot of Chosen Features (Before Outlier Treatment)', fontsize=18)
plt.ylabel('Values', fontsize=16)
plt.xlabel('Features', fontsize=16)
plt.tight_layout()

plt.subplot(1, 2, 2)
numeric_data_cleaned = data_cleaned.select_dtypes(include='number').drop(columns=['year', 'n_duty', 'dafted'])
plt.boxplot(numeric_data_cleaned.values, vert=True, patch_artist=True, labels=numeric_data_cleaned.columns)
plt.title('Boxplot of Chosen Features (After Outlier Treatment)', fontsize=18)
plt.ylabel('Values', fontsize=16)
plt.xlabel('Features', fontsize=16)
plt.tight_layout()

plt.show()
```

**# In[13]:**

```
# Scaling selected numerical features
data_min_max_scaled= data_cleaned.copy()
numerical_columns = ['n_sick', 'calls', 'n_duty', 'sby_need', 'dafted']

# Performing Min-Max Scaling
min_max_scaler = MinMaxScaler()
data_min_max_scaled[numerical_columns] = min_max_scaler.fit_transform(data_min_max_scaled[numerical_columns])

print("Min-Max Scaled Data:")
data_min_max_scaled.head()
```

**# In[14]:**

```
# Performing Standard Scaling
data_standard_scaled= data_cleaned.copy()
standard_scaler = StandardScaler()
data_standard_scaled[numerical_columns] = standard_scaler.fit_transform(data_standard_scaled[numerical_columns])

print("\nStandard Scaled Data:")
data_standard_scaled.head()
```

**# In[15]:**

```
# Selected Min-Max Scaled Data for further processes
new_df= data_min_max_scaled

# Adding day_of_week, month and season categorical features
new_df['date'] = pd.to_datetime(new_df['date'])

new_df['day_of_week']=new_df['date'].dt.strftime('%a')
new_df['month']=new_df['date'].dt.strftime('%b')
month_numeric = new_df['date'].dt.month

def seasons(month):
    if month in [3,4,5]:
        return "Spring"
    elif month in [6,7,8]:
        return "Summer"
    elif month in [9,10,11]:
        return "Autumn"
    else:
        return "Winter"

new_df['season'] = month_numeric.apply(seasons)
new_df.head()
```

**# In[16]:**

```
# Creating sby_need vs Season Plot
season_group = new_df.groupby('season')['sby_need'].mean().reset_index()
colors = ['skyblue', 'green', 'orange', 'blue']

# Creating a bar plot
plt.figure(figsize=(10, 8))
```

```
plt.bar(season_group['season'], season_group['sby_need'], color=colors)
plt.xlabel('Season', fontsize=16)
plt.ylabel('Sum of sby_need', fontsize=16)
plt.title('sby_need by Season', fontsize=18)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.show()
```

**# In[17]:**

```
# Creating sby_need vs Month Plot

# Define the custom order of months
month_order = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

# Convert 'month' to a categorical variable with custom order
new_df['month'] = pd.Categorical(new_df['month'], categories=month_order, ordered=True)

month_group = new_df.groupby('month')['sby_need'].mean().reset_index()

# Creating a bar plot
plt.figure(figsize=(10, 8))
plt.bar(month_group['month'], month_group['sby_need'], color='orange')
plt.xlabel('Month', fontsize=16)
plt.ylabel('Sum of sby_need', fontsize=16)
plt.title('sby_need by Month', fontsize=18)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.show()
```

**# In[18]:**

```
# Creating sby_need vs Day of Week Plot

# Define the custom order of days
day_order = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

# Convert 'day' to a categorical variable with custom order
new_df['day_of_week'] = pd.Categorical(new_df['day_of_week'], categories=day_order, ordered=True)

day_group = new_df.groupby('day_of_week')['sby_need'].mean().reset_index()
# colors = ['skyblue', 'green', 'orange', 'blue']

# # Creating a bar plot
plt.figure(figsize=(10, 8))
plt.bar(day_group['day_of_week'], day_group['sby_need'], color='green')
plt.xlabel('Day', fontsize=16)
plt.ylabel('Sum of sby_need', fontsize=16)
plt.title('sby_need by Day', fontsize=18)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.show()
```

**# In[19]:**

```
# Creating sby_need vs Day of Week Plot with Season Hue

# Define the custom order of days
day_order = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

# Convert 'day' to a categorical variable with custom order
new_df['day_of_week'] = pd.Categorical(new_df['day_of_week'], categories=day_order, ordered=True)

# Define a mapping of months to seasons
season_mapping = {
    'Jan': 'Winter', 'Feb': 'Winter', 'Mar': 'Spring',
    'Apr': 'Spring', 'May': 'Spring', 'Jun': 'Summer',
    'Jul': 'Summer', 'Aug': 'Summer', 'Sep': 'Autumn',
    'Oct': 'Autumn', 'Nov': 'Autumn', 'Dec': 'Winter'
}

# Map 'month' to 'season'
new_df['season'] = new_df['month'].map(season_mapping)

# Group by day and season, calculate the sum of 'sby_need'
day_season_group = new_df.groupby(['day_of_week', 'season'])['sby_need'].mean().reset_index()

# Creating a bar plot with seaborn
plt.figure(figsize=(12, 8))
sns.barplot(x='day_of_week', y='sby_need', hue='season', data=day_season_group, palette='viridis')
plt.xlabel('Day', fontsize=16)
plt.ylabel('Sum of sby_need', fontsize=16)
```

```
plt.title('sby_need by Day with Season Hue', fontsize=18)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.legend(fontsize=14)
plt.show()
```

**# In[20]:**

```
# Creating sby_need vs Month Plot with Day of Week Hue

# Define the custom order of days
day_order = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

# Convert 'day_of_week' to a categorical variable with custom order
new_df['day_of_week'] = pd.Categorical(new_df['day_of_week'], categories=day_order, ordered=True)

# Define a mapping of months to seasons
season_mapping = {
    'Jan': 'Winter', 'Feb': 'Winter', 'Mar': 'Spring',
    'Apr': 'Spring', 'May': 'Spring', 'Jun': 'Summer',
    'Jul': 'Summer', 'Aug': 'Summer', 'Sep': 'Autumn',
    'Oct': 'Autumn', 'Nov': 'Autumn', 'Dec': 'Winter'
}

# Map 'month' to 'season'
new_df['season'] = new_df['month'].map(season_mapping)

# Group by month and day, calculate the sum of 'sby_need'
month_day_group = new_df.groupby(['month', 'day_of_week'])['sby_need'].mean().reset_index()

# Creating a bar plot with seaborn
plt.figure(figsize=(12, 8))
sns.barplot(x='month', y='sby_need', hue='day_of_week', data=month_day_group, palette='viridis')
plt.xlabel('Month', fontsize=16)
plt.ylabel('Sum of sby_need', fontsize=16)
plt.title('sby_need by Month with Day of week Hue', fontsize=18)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.legend(fontsize=14)
plt.show()
```

**# In[21]:**

```
# Performing One-hot Encoding for Categorical columns
cat_columns= ['day_of_week', 'month', 'season']
df_transformed = pd.get_dummies(new_df, columns = cat_columns, dtype=int)
df_transformed.head()
```

**# In[22]:**

```
# Displaing all the columns in the dataset
df_transformed.columns
```

**# In[23]:**

```
# Creating a Dataset copy
final_data= df_transformed.copy()
```

**### Aggregation of Data for Monthly Prediction of Time Series Models**

**# In[24]:**

```
# Monthly Aggregated Data
mo_df= final_data.copy()

# 'M' indicates monthly aggregation (summing up values)
mo_df.set_index('date', inplace=True)
monthly_aggregated_data = mo_df.resample('M').sum()
monthly_aggregated_data.reset_index(inplace=True)
monthly_aggregated_data.head()
```

**### Preparation steps to implement Time Series Model for monthly aggregated data**

**# In[25]:**

```
# Function to check stationarity
# Testing the stationarity of the data to implement Time Series Models

def test_stationarity(dataFrame, var):
    dataFrame['rollMean']  = dataFrame[var].rolling(window=12).mean()
    dataFrame['rollStd']  = dataFrame[var].rolling(window=12).std()
```

```
    adfTest = adfuller(dataFrame[var],autolag='AIC')
    stats = pd.Series(adfTest[0:4],index=['Test Statistic','p-value','#lags used','number of observations used'])
    print(stats)

    for key, values in adfTest[4].items():
        print('criticality',key,":",values)

    sns.lineplot(data=dataFrame,x=dataFrame.index,y=var)
    sns.lineplot(data=dataFrame,x=dataFrame.index,y='rollMean')
    sns.lineplot(data=dataFrame,x=dataFrame.index,y='rollStd')
```

**### Checking Stationarity of monthly aggregated data**

**# In[26]:**

```
# check stationary of aggregated data using ADF Test
unshifted_df_mo= monthly_aggregated_data[['sby_need']].copy()

test_stationarity(unshifted_df_mo.dropna(), 'sby_need')

"""The values calculate after 10 units is because the first 12 windows is used for learning"""
```

**# In[27]:**

```
# Creating a copy of the dataset
mad_df= monthly_aggregated_data.copy()
mad_cpy= mad_df[['sby_need']].copy()
```

**### PACF Plot**

**# In[28]:**

```
"""Lag value for PACF test (system message: Can only compute partial correlations
for lags up to 50% of the sample size. Because of this, I chose lag as 7"""
plt.figure(figsize=(10, 5))
plot_pacf(mad_cpy['sby_need'].dropna(), lags=7)
plt.show()
```

**### ACF Plot**

**# In[29]:**

```
plt.figure(figsize=(10, 5))
# plot_acf(mad_cpy['sby_need'].dropna(), lags= 7)
plot_acf(mad_cpy['sby_need'].dropna(), lags=7)
plt.show()
```

**### Monthly Prediction using Time Series Models**

**### Train-Test Split of Date and Target variable for Monthly Prediction of Time Series data**

**# In[30]:**

```
# Train-Test Split for Monthly Aggregated Data for Time Series Model
train_mo= mad_cpy[:round(len(mad_cpy)*70/100)]
test_mo= mad_cpy[round(len(mad_cpy)*70/100):]
```

**# ARIMA Model Monthly Prediction**

**# In[31]:**

```
# ARIMA Model (monthly)
ar_mo_model = ARIMA(mad_cpy['sby_need'], order=(0, 0, 0))
ar_mo_model_fit = ar_mo_model.fit()

# Forecasting the training set
train_forecast_ar_mo = ar_mo_model_fit.get_prediction(start= train_mo.index[0], end= train_mo.index[-1])
predicted_train_values_ar_mo = train_forecast_ar_mo.predicted_mean
mad_cpy['arimaPred_train'] = predicted_train_values_ar_mo

# Forecasting the test set
test_forecast_ar_mo = ar_mo_model_fit.get_prediction(start= test_mo.index[0], end= test_mo.index[-1])
predicted_test_values_ar_mo = test_forecast_ar_mo.predicted_mean
mad_cpy['arimaPred_test'] = predicted_test_values_ar_mo

# ARIMA Model Plot with actual and predicted values
sns.lineplot(data=mad_cpy,x=mad_cpy.index,y='sby_need')
sns.lineplot(data=mad_cpy,x=mad_cpy.index,y='arimaPred_train')
sns.lineplot(data=mad_cpy,x=mad_cpy.index,y='arimaPred_test')

# MSE Train Arima
mse_tr_ar_mo= mean_squared_error(train_mo['sby_need'], predicted_train_values_ar_mo)
```

```
# MAE Train Arima
mae_tr_ar_mo= mean_absolute_error(train_mo['sby_need'], predicted_train_values_ar_mo)
# MAPE Train Arima
mape_tr_ar_mo= mean_absolute_error(train_mo['sby_need'], predicted_train_values_ar_mo) / abs(train_mo['sby_need'].mean())
* 100 / train_mo.shape[0]

# MSE Test Arima
mse_tst_ar_mo= mean_squared_error(test_mo['sby_need'], predicted_test_values_ar_mo)
# MAE Test Arima
mae_tst_ar_mo= mean_absolute_error(test_mo['sby_need'], predicted_test_values_ar_mo)
# MAPE Test Arima
mape_tst_ar_mo= mean_absolute_error(test_mo['sby_need'], predicted_test_values_ar_mo) / abs(test_mo['sby_need'].mean()) *
100 / test_mo.shape[0]

print("MSE Train: {}".format(mse_tr_ar_mo))
print("MAE Train: {}".format(mae_tr_ar_mo))
print("MAPE Train: {}".format(mape_tr_ar_mo))
print("MSE Test: {}".format(mse_tst_ar_mo))
print("MAE Test: {}".format(mae_tst_ar_mo))
print("MAPE Test: {}".format(mape_tst_ar_mo))
```

**# SARIMA Model Monthly Prediction**

**# In[32]:**

```
# SARIMAX Model (monthly)
sr_mo_model = SARIMAX(mad_cpy['sby_need'], order=(0, 0, 0), seasonal_order=(0, 0, 0, 12))
sr_mo_model_fit = sr_mo_model.fit()

# Forecasting the training set
train_forecast_sr_mo = sr_mo_model_fit.get_prediction(start= train_mo.index[0], end= train_mo.index[-1])
predicted_train_values_sr_mo = train_forecast_sr_mo.predicted_mean
mad_cpy['sarimaxPred_train'] = predicted_train_values_sr_mo

# Forecasting the test set
test_forecast_sr_mo = sr_mo_model_fit.get_prediction(start= test_mo.index[0], end= test_mo.index[-1])
predicted_test_values_sr_mo = test_forecast_sr_mo.predicted_mean
mad_cpy['sarimaxPred_test'] = predicted_test_values_sr_mo

# SARIMAX Model Plot with actual and predicted values
sns.lineplot(data= mad_cpy, x= mad_cpy.index, y='sby_need')
sns.lineplot(data= mad_cpy, x= mad_cpy.index, y='sarimaxPred_train')
sns.lineplot(data= mad_cpy, x= mad_cpy.index, y='sarimaxPred_test')

# MSE Train SARIMAX
mse_tr_sr_mo= np.sqrt(mean_squared_error(train_mo['sby_need'], predicted_train_values_sr_mo))
# MAE Train SARIMAX
mae_tr_sr_mo= mean_absolute_error(train_mo['sby_need'], predicted_train_values_sr_mo)
# MAPE Train SARIMAX
mape_tr_sr_mo= mean_absolute_error(train_mo['sby_need'], predicted_train_values_sr_mo) / abs(train_mo['sby_need'].mean())
* 100 / train_mo.shape[0]
# MSE Test SARIMAX
mse_tst_sr_mo= np.sqrt(mean_squared_error(test_mo['sby_need'], predicted_test_values_sr_mo))
# MAE Test SARIMAX
mae_tst_sr_mo= mean_absolute_error(test_mo['sby_need'], predicted_test_values_sr_mo)
# MAPE Test SARIMAX
mape_tst_sr_mo= mean_absolute_error(test_mo['sby_need'], predicted_test_values_sr_mo) / abs(test_mo['sby_need'].mean()) *
100 / test_mo.shape[0]

print("MSE Train: {}".format(mse_tr_sr_mo))
print("MAE Train: {}".format(mae_tr_sr_mo))
print("MAPE Train: {}".format(mape_tr_sr_mo))
print("MSE Test: {}".format(mse_tst_sr_mo))
print("MAE Test: {}".format(mae_tst_sr_mo))
print("MAPE Test: {}".format(mape_tst_sr_mo))
```

**### Preparation steps to implement Time Series Model for daily prediction**

**# In[33]:**

```
# Creating a copy of the dataset
mad_df_dly= final_data.copy()
```

**### Checking stationarity**

**# In[34]:**

```
# Checking stationarity using aggregated data
unshifted_df_daily= final_data[['sby_need']].copy()
test_stationarity(unshifted_df_daily.dropna(), 'sby_need')
```

**# In[35]:**

```
# Creating a copy of the dataset
mad_dly_cpy= mad_df_dly[['sby_need']].copy()
```

**### PACF Plot**

**# In[36]:**

```
# Lag is 240 because of 7 months seasonality
plt.figure(figsize=(10, 5))
plot_pacf(mad_dly_cpy['sby_need'].dropna(), lags=240)
plt.show()
```

**### ACF Plot**

**# In[37]:**

```
plt.figure(figsize=(10, 5))
plot_acf(mad_dly_cpy['sby_need'].dropna(), lags= 240)
plt.show()
```

**### 70-30 Split for Daily Prediction of Time Series Models**

**# In[38]:**

```
# We split date and target variable data for implementing the Time Series Model
train_dly= mad_dly_cpy[:round(len(mad_dly_cpy)*70/100)]
test_dly= mad_dly_cpy[round(len(mad_dly_cpy)*70/100):]
train_dly.shape[0]
```

**# ARIMA Model for Daily Prediction**

**# In[39]:**

```
# Fit the ARIMA model on the entire dataset
arima_model = ARIMA(mad_dly_cpy['sby_need'], order=(4, 0, 4))
arima_model_fit = arima_model.fit()

# Forecasting the training set
train_forecast_ar = arima_model_fit.get_prediction(start= train_dly.index[0], end= train_dly.index[-1])
predicted_train_values_ar = train_forecast_ar.predicted_mean
mad_dly_cpy['arimaPred_train'] = predicted_train_values_ar

# Forecasting the test set
test_forecast_ar = arima_model_fit.get_prediction(start= test_dly.index[0], end= test_dly.index[-1])
predicted_test_values_ar = test_forecast_ar.predicted_mean
mad_dly_cpy['arimaPred_test'] = predicted_test_values_ar

# Plotting results of ARIMA Model with actual and predicted values
sns.lineplot(data=mad_dly_cpy,x=mad_dly_cpy.index,y='sby_need')
sns.lineplot(data=mad_dly_cpy,x=mad_dly_cpy.index,y='arimaPred_train')
sns.lineplot(data=mad_dly_cpy,x=mad_dly_cpy.index,y='arimaPred_test')
plt.title('ARIMA')
plt.show()

# ARIMA Model Evaluation
# MSE Train Arima
mse_tr_ar= ((train_dly['sby_need'] - predicted_train_values_ar.dropna()).sum())**2 / train_dly.shape[0]
# MAE Train Arima
mae_tr_ar= abs(train_dly['sby_need'] - predicted_train_values_ar.dropna()).sum() / train_dly.shape[0]
# MAPE Train Arima
mape_df_ar_tr = mad_dly_cpy[['sby_need', 'arimaPred_train']]
mape_df_ar_tr = mape_df_ar_tr[mape_df_ar_tr['sby_need'] != 0]
mape_tr_ar= (1/mape_df_ar_tr.shape[0]) * abs((mape_df_ar_tr['sby_need'] -
mape_df_ar_tr['arimaPred_train'].dropna())/mape_df_ar_tr['sby_need']).sum() * 100
# MSE Test Arima
mse_tst_ar= ((test_dly['sby_need'] - predicted_test_values_ar.dropna()).sum())**2 / test_dly.shape[0]
# MAE Test Arima
mae_tst_ar= abs(test_dly['sby_need'] - predicted_test_values_ar.dropna()).sum() / test_dly.shape[0]
# MAPE Test Arima
mape_df_ar_tst = mad_dly_cpy[['sby_need', 'arimaPred_test']]
mape_df_ar_tst = mape_df_ar_tst[mape_df_ar_tst['sby_need'] != 0]
mape_tst_ar= (1/mape_df_ar_tst.shape[0]) * abs((mape_df_ar_tst['sby_need'] -
mape_df_ar_tst['arimaPred_test'].dropna())/mape_df_ar_tst['sby_need']).sum() * 100

print("MSE Train: {}".format(mse_tr_ar))
print("MAE Train: {}".format(mae_tr_ar))
print("MAPE Train: {}".format(mape_tr_ar))
print("MSE Test: {}".format(mse_tst_ar))
print("MAE Test: {}".format(mae_tst_ar))
print("MAPE Test: {}".format(mape_tst_ar))
```

**# SARIMAX Model for Daily Prediction**

**# In[40]:**

```
# Fit the SARIMAX model on the entire dataset
# weekly seasonal pattern
sarimax_model = SARIMAX(mad_dly_cpy['sby_need'], order=(4, 0, 4), seasonal_order=(4, 0, 4, 7))
sarimax_model_fit = sarimax_model.fit()

# Forecasting the training set
train_forecast_sr = sarimax_model_fit.get_prediction(start= train_dly.index[0], end= train_dly.index[-1])
predicted_train_values_sr = train_forecast_sr.predicted_mean
mad_dly_cpy['sarimaxPred_train'] = predicted_train_values_sr

# Forecasting the test set
test_forecast_sr = sarimax_model_fit.get_prediction(start= test_dly.index[0], end= test_dly.index[-1])
predicted_test_values_sr = test_forecast_sr.predicted_mean
mad_dly_cpy['sarimaxPred_test'] = predicted_test_values_sr

# Plotting results of SARIMAX Model with actual and predicted values
sns.lineplot(data= mad_dly_cpy, x= mad_dly_cpy.index, y='sby_need')
sns.lineplot(data= mad_dly_cpy, x= mad_dly_cpy.index, y='sarimaxPred_train')
sns.lineplot(data= mad_dly_cpy, x= mad_dly_cpy.index, y='sarimaxPred_test')
plt.title('SARIMA')
plt.show()

# SARIMAX Model Evaluation
# MSE Train SARIMAX
mse_tr_sr= ((train_dly['sby_need'] - predicted_train_values_sr.dropna()).sum())**2 / train_dly.shape[0]
# MAE Train SARIMAX
mae_tr_sr= abs(train_dly['sby_need'] - predicted_train_values_sr.dropna()).sum() / train_dly.shape[0]
# MAPE Train SARIMAX
mape_df_sr_tr = mad_dly_cpy[['sby_need', 'sarimaxPred_train']]
mape_df_sr_tr = mape_df_sr_tr[mape_df_sr_tr['sby_need'] != 0]
mape_tr_sr= (1/mape_df_sr_tr.shape[0]) * abs((mape_df_sr_tr['sby_need'] -
mape_df_sr_tr['sarimaxPred_train'].dropna())/mape_df_sr_tr['sby_need']).sum() * 100
# MSE Test SARIMAX
mse_tst_sr= ((test_dly['sby_need'] - predicted_test_values_sr.dropna()).sum())**2 / test_dly.shape[0]
# MAE Test SARIMAX
mae_tst_sr= abs(test_dly['sby_need'] - predicted_train_values_sr.dropna()).sum() / test_dly.shape[0]
# MAPE Test SARIMAX
mape_df_sr_tst = mad_dly_cpy[['sby_need', 'sarimaxPred_test']]
mape_df_sr_tst = mape_df_sr_tst[mape_df_sr_tst['sby_need'] != 0]
mape_tst_sr= (1/mape_df_sr_tst.shape[0]) * abs((mape_df_sr_tst['sby_need'] -
mape_df_sr_tst['sarimaxPred_test'].dropna())/mape_df_sr_tst['sby_need']).sum() * 100

print("MSE Train: {}".format(mse_tr_sr))
print("MAE Train: {}".format(mae_tr_sr))
print("MAPE Train: {}".format(mape_tr_sr))
print("MSE Test: {}".format(mse_tst_sr))
print("MAE Test: {}".format(mae_tst_sr))
print("MAPE Test: {}".format(mape_tst_sr))


# """Mention you also tried SARIMA model for monthly aggregated data for month wise
# predictions because the data was seasonal but not happy with the results"""
```

**# Regression models for Daily Prediction**

**# In[41]:**

```
# Train-Test Split of data for Daily Prediction 70-30
x = final_data.drop(columns=['sby_need', 'date'])
y = final_data['sby_need']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,shuffle=False, random_state=42)
```

**# In[42]:**

```
# Random Forest Model

rf_df_all= final_data[['sby_need']]
rf_df_tr = rf_df_all[:math.floor(len(rf_df_all) * 70/100)]
rf_df_tst= rf_df_all[math.floor(len(rf_df_all)* 70/100):]

# Fitting the model
rf_reg = RandomForestRegressor(n_estimators=1000)
rf_reg.fit(x_train, y_train)

# Making predictions
train_predictions_rf = rf_reg.predict(x_train)
test_predictions_rf = rf_reg.predict(x_test)
```

```
# Storing the results
rf_df_tr['train_pred']= train_predictions_rf
rf_df_tst['test_pred']= test_predictions_rf

rf_result_df = rf_df_all[['sby_need']]
rf_result_df['train_pred']= rf_df_tr['train_pred']
rf_result_df['test_pred']= rf_df_tst['test_pred']

# Random Forest Results Plot with actual and predicted values
sns.lineplot(data= rf_result_df, x= rf_result_df.index, y='sby_need')
sns.lineplot(data= rf_result_df, x= rf_result_df.index, y='train_pred')
sns.lineplot(data= rf_result_df, x= rf_result_df.index, y='test_pred')
plt.title('Random Forest')
plt.show()

# Evaluation
# MSE Train
train_mse_rf = mean_squared_error(y_train, train_predictions_rf)
# MAE Train
train_mae_rf = mean_absolute_error(y_train, train_predictions_rf)
# MAPE Train
mape_df_fet_tr_rf = rf_df_tr[['sby_need', 'train_pred']]
mape_df_tr_rf = mape_df_fet_tr_rf[mape_df_fet_tr_rf['sby_need'] != 0]
train_mape_rf= (1/mape_df_tr_rf.shape[0]) * abs((mape_df_tr_rf['sby_need'] -
mape_df_tr_rf['train_pred'].dropna())/mape_df_tr_rf['sby_need']).sum() * 100

# MSE Test
test_mse_rf = mean_squared_error(y_test, test_predictions_rf)
# MAE Test
test_mae_rf = mean_absolute_error(y_test, test_predictions_rf)
# MAPE Test
mape_df_fet_tst_rf = rf_df_tst[['sby_need', 'test_pred']]
mape_df_tst_rf = mape_df_fet_tst_rf[mape_df_fet_tst_rf['sby_need'] != 0]
test_mape_rf= (1/mape_df_tst_rf.shape[0]) * abs((mape_df_tst_rf['sby_need'] -
mape_df_tst_rf['test_pred'].dropna())/mape_df_tst_rf['sby_need']).sum() * 100

print(f'Training MSE: {train_mse_rf}')
print(f'Training MAE: {train_mae_rf}')
print(f'Training MAPE: {train_mape_rf}')
print(f'Test MSE: {test_mse_rf}')
print(f'Test MAE: {test_mae_rf}')
print(f'Test MAPE: {test_mape_rf}')
```

**# K-nearest neighbors Model**

**# In[43]:**

```
# KNN model calculating RMSE scores for different K values
rmse_val = [] #to store rmse values for different k
for K in range(20):
    K = K+1
    k_chk_model = KNeighborsRegressor(n_neighbors = K)

    k_chk_model.fit(x_train, y_train)  #fit the model
    k_chk_pred=k_chk_model.predict(x_test) #make prediction on test set
    knn_error = math.sqrt(mean_squared_error(y_test,k_chk_pred)) #calculate rmse
    rmse_val.append(knn_error) #store rmse values
    print('RMSE value for k= ' , K , 'is:', knn_error)
```

**# In[44]:**

```
# Plotting the rmse values against k values
curve = pd.DataFrame(rmse_val) #elbow curve
curve.plot()
```

**# In[45]:**

```
# Implementing KNN Model

knn_df_all= final_data[['sby_need']]
knn_df_tr = rf_df_all[:math.floor(len(knn_df_all) * 70/100)]
knn_df_tst= rf_df_all[math.floor(len(knn_df_all)* 70/100):]

# Fitting the model with k=13
knn_model = KNeighborsRegressor(n_neighbors = 13)
knn_model.fit(x_train, y_train)

# Making predictions
train_predictions_knn= knn_model.predict(x_train)
test_predictions_knn= knn_model.predict(x_test)
```

```
# Storing the results
knn_df_tr['train_pred']= train_predictions_knn
knn_df_tst['test_pred']= test_predictions_knn

knn_result_df = knn_df_all[['sby_need']]
knn_result_df['train_pred']= knn_df_tr['train_pred']
knn_result_df['test_pred']= knn_df_tst['test_pred']

# KNN Results Plot with actual and predicted values
sns.lineplot(data= knn_result_df, x= knn_result_df.index, y='sby_need')
sns.lineplot(data= knn_result_df, x= knn_result_df.index, y='train_pred')
sns.lineplot(data= knn_result_df, x= knn_result_df.index, y='test_pred')
plt.title('KNN')
plt.show()

# KNN Evaluation
# MSE Train
train_mse_knn = mean_squared_error(y_train, train_predictions_knn)
# MAE Train
train_mae_knn = mean_absolute_error(y_train, train_predictions_knn)
# MAPE Train
mape_df_fet_tr_knn = knn_df_tr[['sby_need', 'train_pred']]
mape_df_tr_knn = mape_df_fet_tr_knn[mape_df_fet_tr_knn['sby_need'] != 0]
train_mape_knn= (1/mape_df_tr_knn.shape[0]) * abs((mape_df_tr_knn['sby_need'] -
mape_df_tr_knn['train_pred'].dropna())/mape_df_tr_knn['sby_need']).sum() * 100

# MSE Test
test_mse_knn = mean_squared_error(y_test, test_predictions_knn)
# MAE Test
test_mae_knn = mean_absolute_error(y_test, test_predictions_knn)
# MAPE Test
mape_df_fet_tst_knn = knn_df_tst[['sby_need', 'test_pred']]
mape_df_tst_knn = mape_df_fet_tst_knn[mape_df_fet_tst_knn['sby_need'] != 0]
test_mape_knn= (1/mape_df_tst_knn.shape[0]) * abs((mape_df_tst_knn['sby_need'] -
mape_df_tst_knn['test_pred'].dropna())/mape_df_tst_knn['sby_need']).sum() * 100

print(f'Training MSE: {train_mse_knn}')
print(f'Training MAE: {train_mae_knn}')
print(f'Training MAPE: {train_mape_knn}')
print(f'Test MSE: {test_mse_knn}')
print(f'Test MAE: {test_mae_knn}')
print(f'Test MAPE: {test_mape_knn}')
```

**# XGBoost Model**

**# In[46]:**

```
# Implementing XGBoost Model

xgb_df_all= final_data[['sby_need']]
xgb_df_tr = xgb_df_all[:math.floor(len(xgb_df_all) * 70/100)]
xgb_df_tst= xgb_df_all[math.floor(len(xgb_df_all)* 70/100):]

# Fitting the model
xgb_model = xgb.XGBRegressor(objective='reg:squarederror', random_state=42)
xgb_model.fit(x_train, y_train)

# Making predictions
train_predictions_xgb = xgb_model.predict(x_train)
test_predictions_xgb = xgb_model.predict(x_test)

# Storing the results
xgb_df_tr['train_pred']= train_predictions_xgb
xgb_df_tst['test_pred']= test_predictions_xgb

xgb_result_df = xgb_df_all[['sby_need']]
xgb_result_df['train_pred']= xgb_df_tr['train_pred']
xgb_result_df['test_pred']= xgb_df_tst['test_pred']

# XGBoost Results Plot with actual and predicted values
sns.lineplot(data= xgb_result_df, x= xgb_result_df.index, y='sby_need')
sns.lineplot(data= xgb_result_df, x= xgb_result_df.index, y='train_pred')
sns.lineplot(data= xgb_result_df, x= xgb_result_df.index, y='test_pred')
plt.title('XGBoost')
plt.show()

# XGB Evaluation
# MSE Train
train_mse_xgb = mean_squared_error(y_train, train_predictions_xgb)
# MAE Train
train_mae_xgb = mean_absolute_error(y_train, train_predictions_xgb)
# MAPE Train
```

```
mape_df_fet_tr_xgb = xgb_df_tr[['sby_need', 'train_pred']]
mape_df_tr_xgb = mape_df_fet_tr_xgb[mape_df_fet_tr_xgb['sby_need'] != 0]
train_mape_xgb= (1/mape_df_tr_xgb.shape[0]) * abs((mape_df_tr_xgb['sby_need'] -
mape_df_tr_xgb['train_pred'].dropna())/mape_df_tr_xgb['sby_need']).sum() * 100

# MSE Test
test_mse_xgb = mean_squared_error(y_test, test_predictions_xgb)
# MAE Test
test_mae_xgb = mean_absolute_error(y_test, test_predictions_xgb)
# MAPE Test
mape_df_fet_tst_xgb = xgb_df_tst[['sby_need', 'test_pred']]
mape_df_tst_xgb = mape_df_fet_tst_xgb[mape_df_fet_tst_xgb['sby_need'] != 0]
test_mape_xgb= (1/mape_df_tst_xgb.shape[0]) * abs((mape_df_tst_xgb['sby_need'] -
mape_df_tst_xgb['test_pred'].dropna())/mape_df_tst_xgb['sby_need']).sum() * 100

print(f'Training MSE: {train_mse_xgb}')
print(f'Training MAE: {train_mae_xgb}')
print(f'Training MAPE: {train_mape_xgb}')
print(f'Test MSE: {test_mse_xgb}')
print(f'Test MAE: {test_mae_xgb}')
print(f'Test MAPE: {test_mape_xgb}')

# Polynomial Regression Model

# In[47]:

# Implementing Polynomial regression model

poly_df_all= final_data[['sby_need']]
poly_df_tr = poly_df_all[:math.floor(len(poly_df_all) * 70/100)]
poly_df_tst= poly_df_all[math.floor(len(poly_df_all)* 70/100):]

# Choose the degree of the polynomial
poly_degree = 2  # You can adjust this based on your needs

# Create a polynomial regression model
poly_model = make_pipeline(PolynomialFeatures(poly_degree), LinearRegression())

# Fit the model on the training data
poly_model.fit(x_train, y_train)

# Predictions on the training set
train_predictions_poly = poly_model.predict(x_train)

# Predictions on the test set
test_predictions_poly = poly_model.predict(x_test)

# Storing the results
poly_df_tr['train_pred']= train_predictions_poly
poly_df_tst['test_pred']= test_predictions_poly

# Polynomial result dataframe
poly_result_df = poly_df_all[['sby_need']]
poly_result_df['train_pred'] = poly_df_tr['train_pred']
poly_result_df['test_pred'] = poly_df_tst['test_pred']

# In[48]:

# Polynomial Train Plot with actual and training predictions
sns.lineplot(data= poly_result_df, x= poly_result_df.index, y='sby_need')
sns.lineplot(data= poly_result_df, x= poly_result_df.index, y='train_pred')
plt.title('Polynomial Regression Plot 1')
plt.show()

# In[49]:

# Polynomial regression plot with actual and predicted values
sns.lineplot(data= poly_result_df, x= poly_result_df.index, y='sby_need')
sns.lineplot(data= poly_result_df, x= poly_result_df.index, y='train_pred')
sns.lineplot(data= poly_result_df, x= poly_result_df.index, y='test_pred')
plt.title('Polynomial Regression Plot 2')
plt.show()

# In[50]:

# Evaluate polynomial model

# MSE Train
train_mse_poly = mean_squared_error(y_train, train_predictions_poly)
# MAE Train
train_mae_poly = mean_absolute_error(y_train, train_predictions_poly)
# MAPE Train
```

```
mape_df_fet_tr_poly = poly_df_tr[['sby_need', 'train_pred']]
mape_df_tr_poly = mape_df_fet_tr_poly[mape_df_fet_tr_poly['sby_need'] != 0]
train_mape_poly= (1/mape_df_tr_poly.shape[0]) * abs((mape_df_tr_poly['sby_need'] -
mape_df_tr_poly['train_pred'].dropna())/mape_df_tr_poly['sby_need']).sum() * 100

# MSE Test
test_mse_poly = mean_squared_error(y_test, test_predictions_poly)
# MAE Test
test_mae_poly = mean_absolute_error(y_test, test_predictions_poly)
# MAPE Test
mape_df_fet_tst_poly = poly_df_tst[['sby_need', 'test_pred']]
mape_df_tst_poly = mape_df_fet_tst_poly[mape_df_fet_tst_poly['sby_need'] != 0]
test_mape_poly= (1/mape_df_tst_poly.shape[0]) * abs((mape_df_tst_poly['sby_need'] -
mape_df_tst_poly['test_pred'].dropna())/mape_df_tst_poly['sby_need']).sum() * 100

print(f'Training MSE: {train_mse_poly}')
print(f'Training MAE: {train_mae_poly}')
print(f'Training MAPE: {train_mape_poly}')
print(f'Test MSE: {test_mse_poly}')
print(f'Test MAE: {test_mae_poly}')
print(f'Test MAPE: {test_mape_poly}')
```

**# XGBoost Model with Hyperparameter Tuning 70-30 Split**

**# In[51]:**

```
# XG Boost Hyperparameter Tuned Model

xgb_tuned_df_all= final_data[['sby_need']]
xgb_tuned_df_tr = xgb_tuned_df_all[:math.floor(len(xgb_tuned_df_all) * 70/100)]
xgb_tuned_df_tst= xgb_tuned_df_all[math.floor(len(xgb_tuned_df_all)* 70/100):]

# Initialize XGBoost regressor
xgb_model_tuned = xgb.XGBRegressor(objective='reg:squarederror', random_state=42)

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7]
}

# GridSearchCV to find the best hyperparameters
grid_search_xgb = GridSearchCV(estimator=xgb_model_tuned, param_grid=param_grid, cv=3, scoring='neg_mean_squared_error',
n_jobs=1)
grid_result_xgb = grid_search_xgb.fit(x_train, y_train)

# Print best parameters and best score
print("Best Parameters: ", grid_result_xgb.best_params_)
print("Best Score: ", np.sqrt(-grid_result_xgb.best_score_))

# Get the best model
best_xgb = grid_result_xgb.best_estimator_

# Make predictions using the best model
train_pred_tuned_xgb= best_xgb.predict(x_train)
test_pred_tuned_xgb= best_xgb.predict(x_test)

# Storing the results
xgb_tuned_df_tr['train_pred']= train_pred_tuned_xgb
xgb_tuned_df_tst['test_pred']= test_pred_tuned_xgb

xgb_tuned_result_df = xgb_tuned_df_all[['sby_need']]
xgb_tuned_result_df['train_pred']= xgb_tuned_df_tr['train_pred']
xgb_tuned_result_df['test_pred']= xgb_tuned_df_tst['test_pred']

# XGBoost Tuned Results Plot with actual and predicted values
sns.lineplot(data= xgb_tuned_result_df, x= xgb_tuned_result_df.index, y='sby_need')
sns.lineplot(data= xgb_tuned_result_df, x= xgb_tuned_result_df.index, y='train_pred')
sns.lineplot(data= xgb_tuned_result_df, x= xgb_tuned_result_df.index, y='test_pred')
plt.title('XGBoost Tuned #1')
plt.show()

# Evaluate XGBoost Tuned model 70-30 Split
# MSE Train
train_mse_xgb_tuned = mean_squared_error(y_train, train_pred_tuned_xgb)
# MAE Train
train_mae_xgb_tuned = mean_absolute_error(y_train, train_pred_tuned_xgb)
# MAPE Train
mape_df_fet_tr_xgb_tuned = xgb_tuned_df_tr[['sby_need', 'train_pred']]
mape_df_tr_xgb_tuned = mape_df_fet_tr_xgb_tuned[mape_df_fet_tr_xgb_tuned['sby_need'] != 0]
```

```
train_mape_xgb_tuned= (1/mape_df_tr_xgb_tuned.shape[0]) * abs((mape_df_tr_xgb_tuned['sby_need'] -
mape_df_tr_xgb_tuned['train_pred'].dropna())/mape_df_tr_xgb_tuned['sby_need']).sum() * 100

# MSE Test
test_mse_xgb_tuned = mean_squared_error(y_test, test_pred_tuned_xgb)
# MAE Test
test_mae_xgb_tuned = mean_absolute_error(y_test, test_pred_tuned_xgb)
# MAPE Test
mape_df_fet_tst_xgb_tuned = xgb_tuned_df_tst[['sby_need', 'test_pred']]
mape_df_tst_xgb_tuned = mape_df_fet_tst_xgb_tuned[mape_df_fet_tst_xgb_tuned['sby_need'] != 0]
test_mape_xgb_tuned= (1/mape_df_tst_xgb_tuned.shape[0]) * abs((mape_df_tst_xgb_tuned['sby_need'] -
mape_df_tst_xgb_tuned['test_pred'].dropna())/mape_df_tst_xgb_tuned['sby_need']).sum() * 100

print(f'Training MSE: {train_mse_xgb_tuned}')
print(f'Training MAE: {train_mae_xgb_tuned}')
print(f'Training MAPE: {train_mape_xgb_tuned}')
print(f'Test MSE: {test_mse_xgb_tuned}')
print(f'Test MAE: {test_mae_xgb_tuned}')
print(f'Test MAPE: {test_mape_xgb_tuned}')

# XGBoost Model with Hyperparameter Tuning 80-20 Split

# In[52]:

# 80/20 Train-test split
x_2 = final_data.drop(columns=['sby_need', 'date']).copy()
y_2 = final_data['sby_need'].copy()

x_train_2, x_test_2, y_train_2, y_test_2 = train_test_split(x_2, y_2, test_size=0.2,shuffle=False, random_state=42)

# In[53]:

# XG Boost Hyperparameter Tuned Model

xgb_tuned_df_all_2= final_data[['sby_need']]
xgb_tuned_df_tr_2 = xgb_tuned_df_all_2[:math.floor(len(xgb_tuned_df_all_2) * 80/100)]
xgb_tuned_df_tst_2= xgb_tuned_df_all_2[math.floor(len(xgb_tuned_df_all_2)* 80/100):]

# Initialize XGBoost regressor
xgb_model_tuned_2 = xgb.XGBRegressor(objective='reg:squarederror', random_state=42)

# Define the parameter grid for hyperparameter tuning
param_grid_2 = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7]
}

# GridSearchCV to find the best hyperparameters
grid_search_xgb_2 = GridSearchCV(estimator=xgb_model_tuned_2, param_grid=param_grid_2, cv=3,
scoring='neg_mean_squared_error', n_jobs=1)
grid_result_xgb_2 = grid_search_xgb_2.fit(x_train_2, y_train_2)

# Print best parameters and best score
print("Best Parameters: ", grid_result_xgb_2.best_params_)
print("Best Score: ", np.sqrt(-grid_result_xgb_2.best_score_))

# Get the best model
best_xgb_2 = grid_result_xgb_2.best_estimator_

# Make predictions using the best model
train_pred_tuned_xgb_2= best_xgb_2.predict(x_train_2)
test_pred_tuned_xgb_2= best_xgb_2.predict(x_test_2)

# Storing the results
xgb_tuned_df_tr_2['train_pred']= train_pred_tuned_xgb_2
xgb_tuned_df_tst_2['test_pred']= test_pred_tuned_xgb_2

xgb_tuned_result_df_2 = xgb_tuned_df_all_2[['sby_need']]
xgb_tuned_result_df_2['train_pred']= xgb_tuned_df_tr_2['train_pred']
xgb_tuned_result_df_2['test_pred']= xgb_tuned_df_tst_2['test_pred']

# XGBoost Tuned Results Plot with actual and predicted values
sns.lineplot(data= xgb_tuned_result_df_2, x= xgb_tuned_result_df_2.index, y='sby_need')
sns.lineplot(data= xgb_tuned_result_df_2, x= xgb_tuned_result_df_2.index, y='train_pred')
sns.lineplot(data= xgb_tuned_result_df_2, x= xgb_tuned_result_df_2.index, y='test_pred')
plt.title('XGBoost Tuned #2')
plt.show()

# Evaluate XGBoost Tuned model 80-20 Split
# MSE Train
train_mse_xgb_tuned_2 = mean_squared_error(y_train_2, train_pred_tuned_xgb_2)
```

```
# MAE Train
train_mae_xgb_tuned_2 = mean_absolute_error(y_train_2, train_pred_tuned_xgb_2)
# MAPE Train
mape_df_fet_tr_xgb_tuned_2 = xgb_tuned_df_tr_2[['sby_need', 'train_pred']]
mape_df_tr_xgb_tuned_2 = mape_df_fet_tr_xgb_tuned_2[mape_df_fet_tr_xgb_tuned_2['sby_need'] != 0]
train_mape_xgb_tuned_2= (1/mape_df_tr_xgb_tuned_2.shape[0]) * abs((mape_df_tr_xgb_tuned_2['sby_need'] -
mape_df_tr_xgb_tuned_2['train_pred'].dropna())/mape_df_tr_xgb_tuned_2['sby_need']).sum() * 100

# MSE Test
test_mse_xgb_tuned_2 = mean_squared_error(y_test_2, test_pred_tuned_xgb_2)
# MAE Test
test_mae_xgb_tuned_2 = mean_absolute_error(y_test_2, test_pred_tuned_xgb_2)
# MAPE Test
mape_df_fet_tst_xgb_tuned_2 = xgb_tuned_df_tst_2[['sby_need', 'test_pred']]
mape_df_tst_xgb_tuned_2 = mape_df_fet_tst_xgb_tuned_2[mape_df_fet_tst_xgb_tuned_2['sby_need'] != 0]
test_mape_xgb_tuned_2= (1/mape_df_tst_xgb_tuned_2.shape[0]) * abs((mape_df_tst_xgb_tuned_2['sby_need'] -
mape_df_tst_xgb_tuned_2['test_pred'].dropna())/mape_df_tst_xgb_tuned_2['sby_need']).sum() * 100

print(f'Training MSE: {train_mse_xgb_tuned_2}')
print(f'Training MAE: {train_mae_xgb_tuned_2}')
print(f'Training MAPE: {train_mape_xgb_tuned_2}')
print(f'Test MSE: {test_mse_xgb_tuned_2}')
print(f'Test MAE: {test_mae_xgb_tuned_2}')
print(f'Test MAPE: {test_mape_xgb_tuned_2}')
```