Model Engineering - DLMDSME01

IU International University of Applied Sciences

M.Sc. Data Science 120 ECTS

Forecasting Model of Rescue Drivers Using MS-TDSP

Aaron K. Althauser

Enrolment number: 92014910

May 31st, 2023

Table of Contents

List of Figures

List of Abbreviations

BNB	Bernoulli Naïve Bayes
DWH	Data Warehouse
GUI	Graphical User Interface
MAE	Mean Absolute Error
ML	Machine Learning
MRE	Max Residual Error
MS-TDSP	Microsoft Team Data Science Process
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
SVR	Support Vector Regression

1 Executive Summary

The Berlin Red Cross rescue service (BRC) answers to emergency calls when people are in need. On any given day, there are a set amount of rescue drivers residing on standby to answer these calls. When rescue drivers are not able to work due to temporary illnesses, the estimated number of drivers needed in a day can be interchangeable. The BRC allots a flat total of 90 standby-drivers every day; however, the HR planning department struggles with this approach since seasonal weather patterns affect employee health and allocating a flat number of standby-drivers often leads to having not enough or too many drivers standing by.

Our firm took up the task of fixing this organizational puzzle. We used our expertise in predictive modeling to develop a solution for the BRC that leverages machine learning (ML) techniques and data science. The solution developed can hereby be utilized by the organization to monitor, refine, and predict their approach to help redistribute budget towards cost efficient business goals.

1.1 Challenge

The task is to develop a solution that allows the planning department to assign standby drivers more accurately. Accuracy, in this case, means that there is minimal amount of extra standby drivers assigned that do not get used, while there is a maximized number of days where additional standby drivers do not need to be assigned.

In developing this solution, there are many variables that need to be considered. The challenge herein lies in the business' ability to understand which features of the dataset will help to predict an optimal number of standby drivers in need, which features can properly account for an accurate prediction, and the total volume, value, and variety of the organization's data.

1.2 Solution

Our solution takes the necessary steps to clean and preprocess the accumulation of data in BRC's data warehouses (DWH) before using it to train ML models that make predictions. As a result of the developments, the company was able to create a solution that not only predicts the optimal quantity of standby drivers needed on a given day, but also improved the coefficient of determination (R^2) score from the initial baseline models' 7.4% to over 99%, while lowering the root mean squared error (RMSE) rates from 33.73 to under 0.01.

2 Forecasting Model of Rescue Drivers Using MS-TDSP

The Microsoft Team Data Science Process (MS-TDSP) methodology is used to provide a team-oriented and methodical approach to this task. Using this industry standard method allowed the company to produce a solution that is in line with current data science methodologies for successful businesses. In the following section, the MS-TDSP approach will be briefly detailed by outlining the results of the company's research, followed by step-by-step breakdowns of the task results.

2.1 Business Understanding

In this phase, the necessary steps need to be taken to ensure that business goals are aligned with the end users of the business' efforts. Here, action taken by the business directly affects the rescue drivers employed by BRC. One step taken to ensure proper variables are being used in predictive models was to communicate with these drivers directly, as their input and interpretation is highly valuable. We also approached the IT department of BRC to understand what kind of data is being collected and how we could most efficiently extract, transfer, and load the data using our developed systems.

2.2 Data Acquisition & Understanding

In this phase, the company made the first statistical tests of the acquired data to understand how to approach the solution. This began with an exploratory data analysis and preprocessing, which lead to creating visualizations of the data. The complete code can be referenced in the Source Code annex. The main steps taken to achieve this step were to:

1. *Visualize*. **Error! Reference source not found.** shows an initial subsection of the data printed to gain an initial understanding the data types, columns, and rows. The first observation is that the type for the date column needs to be changed from 'object' to 'datetime' for proper time series indexing.
2. *Clean*. The data can be cleaned by deleting unnecessary columns, adjusting data types, and removing any duplicate rows as shown in Figure 2.
3. *Optimize*. Re-indexing data for time series querying by ensuring every row accounts for one day, and the interval between each row is standardized (Figure 3). This also includes a step to add columns to the data to separate individual aspects of dates (i.e., day, month, year) into features and join them with the data.

```
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1152 entries, 0 to 1151
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Unnamed: 0   1152 non-null   int64
 1   date         1152 non-null   object
 2   n_sick       1152 non-null   int64
 3   calls        1152 non-null   float64
 4   n_duty       1152 non-null   int64
 5   n_sby        1152 non-null   int64
 6   sby_need     1152 non-null   float64
 7   dafted       1152 non-null   float64
dtypes: float64(3), int64(4), object(1)
memory usage: 72.1+ KB
```

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

Figure 1. Initial view of dataset columns

4. Compute.
    a. Quick statistics of the dataset (mean, standard deviation, percentiles, and counts) to identify any outliers or significant trends, shown in Figure 4. There are clear outliers in the number of standby drivers needed (*sby_need*) and additional drivers (*dafted*). These are identified and removed in a later stage and can be seen in the full Source Code.
    b. The correlation matrix to visualize relationships between the features, shown in Figure 6. The matrix shows a few relationships that are noteworthy of exploring further: the number of sick drivers versus time of year (month, week, day, and season; Figure 5), the number of emergency calls versus the number of standby drivers needed (Figure 10), and the number of standby drivers needed on a given day versus the number of additional drivers needed (Figure 11).

```python
# drop columns
df = df.drop('Unnamed: 0', axis=1)
# fix data types
df['date'] = pd.to_datetime(df['date'])
# check for duplicates
df.drop_duplicates()
df.info()
df.head()
```

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

Figure 2. Data cleaning

```python
# Prepare for time-series
df = df.sort_values(by='date')
# Check time intervals
df['Time_Interval'] = df.date - df.date.shift(1)
df[['date', 'Time_Interval']].head()
```

	date	Time_Interval
0	2016-04-01	NaT
1	2016-04-02	1 days
2	2016-04-03	1 days
3	2016-04-04	1 days
4	2016-04-05	1 days

```python
print(f"{df['Time_Interval'].value_counts()}")
df = df.drop('Time_Interval', axis=1)
```

```
1 days    1151
Name: Time_Interval, dtype: int64
```

Figure 3. Preparation for time series indexing

```
df.describe()
```

	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

Figure 4. Quick statistics computation

5. *Plot*. Visualize the data using both time series graphs that show any trends or seasonality in the data, and individual graphs that show the relationships between selected variables.
   a. In the time series plots (Figure 7), there are a few observations made:
      i. Potential outliers can be seen in the graphs of ‘drivers called sick on duty’ and ‘standbys-activated on a given day’ (*sby_need*); the former is further explored in Figure 9 but found to be within three standard deviations of the mean and left in the dataset, and the latter is visualized and removed (see: ModelEng_ForecastRescueDrivers_EDA.py)
      ii. Trends can be observed in ‘drivers called sick on duty’ and ‘emergency calls’ that may be understood better using regression techniques and will be further tested in the benchmark model creation phase.
      iii. Seasonality patterns are observed in ‘standbys…’ and ‘add. drivers needed’. These are further explored in Figure 8, which shows the high shelf of need for standby drivers are between the months of May and August, within the first week of the month and the beginning of the week, and interestingly the number of drivers needed has been steadily increasing annually since 2018.
   b. Further relationships between variables were graphed in the following:
      i. Figure 5, which shows the seasonality patterns in the number of sick drivers. Here there is an expected seasonal correlation between variables which overall holds true: most sick drivers call in between the months of August and November which is known to be flu and cold season in Berlin. Another observation is that the number of sick drivers increases steadily

by day as the month progresses, and this could be explored in another study.

ii. Figure 10, which shows the relationship of emergency calls versus the number of standby drivers needed, explains an expected linear relationship. One interesting note about this relationship is the three tiers of scattered data points; these are separated by year, as with each year the number of both calls and drivers increases.

iii. Figure 11, which shows the number of standby drivers needed related to the number of additional drivers needed. Because BRC assigns 90 drivers to standby, there is nothing further to investigate as it is an obvious linear relationship as the number of drivers needed extends beyond 90.

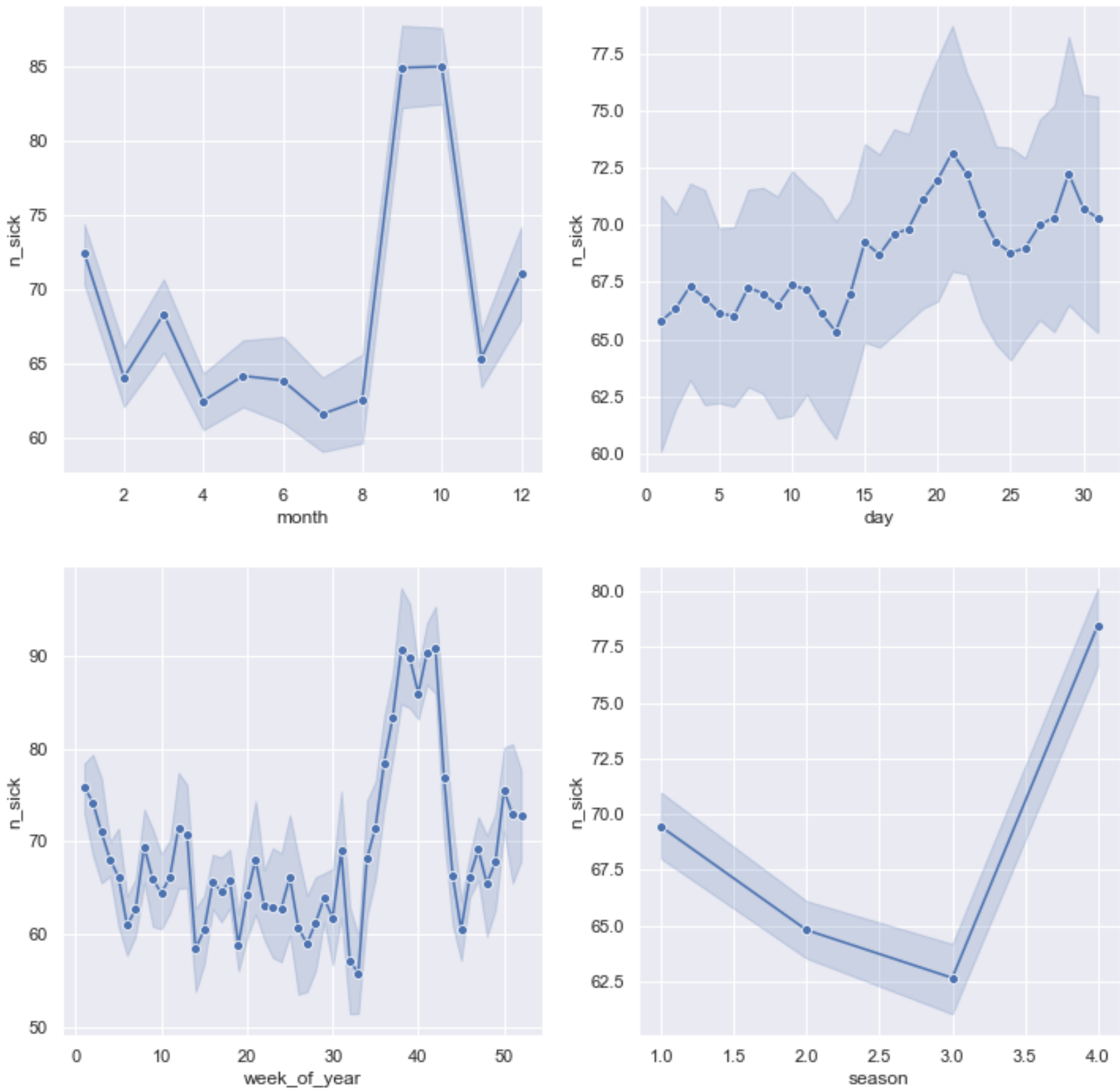


Figure 5. Number of drivers called in sick vs. time of year

```
df.corr()
```

	n_sick	calls	n_duty	n_sby	sby_need	dafted	year	month	day	day_of_week	day_of_year	week_of_year	quarter	season
n_sick	1.000000	0.162878	0.450137	NaN	0.014320	0.000239	0.399826	0.184001	0.116860	-0.060520	0.192167	0.195761	0.177704	0.194881
calls	0.162878	1.000000	0.374537	NaN	0.600676	0.427227	0.383679	-0.076633	-0.202063	-0.187590	-0.095613	-0.087755	-0.081681	0.160312
n_duty	0.450137	0.374537	1.000000	NaN	0.070021	0.072406	0.951256	-0.283837	-0.008700	0.002536	-0.285770	-0.278929	-0.287178	-0.225896
n_sby	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sby_need	0.014320	0.600676	0.070021	NaN	1.000000	0.861084	0.093016	-0.014943	-0.132877	-0.081426	-0.027149	-0.024666	-0.017073	0.098728
dafted	0.000239	0.427227	0.072406	NaN	0.861084	1.000000	0.092059	-0.017532	-0.100366	-0.037074	-0.026820	-0.025139	-0.015040	0.055320
year	0.399826	0.383679	0.951256	NaN	0.093016	0.092059	1.000000	-0.363946	-0.006020	0.007567	-0.364865	-0.357655	-0.365633	-0.287912
month	0.184001	-0.076633	-0.283837	NaN	-0.014943	-0.017532	-0.363946	1.000000	0.016645	0.004510	0.996687	0.986422	0.971747	0.571709
day	0.116860	-0.202063	-0.008700	NaN	-0.132877	-0.100366	-0.006020	0.016645	1.000000	-0.040221	0.097584	0.087544	0.014772	0.028956
day_of_week	-0.060520	-0.187590	0.002536	NaN	-0.081426	-0.037074	0.007567	0.004510	-0.040221	1.000000	0.001065	-0.008392	0.000450	0.003801
day_of_year	0.192167	-0.095613	-0.285770	NaN	-0.027149	-0.026820	-0.364865	0.996687	0.097584	0.001065	1.000000	0.988968	0.968539	0.570733
week_of_year	0.195761	-0.087755	-0.278929	NaN	-0.024666	-0.025139	-0.357655	0.986422	0.087544	-0.008392	0.988968	1.000000	0.959741	0.569694
quarter	0.177704	-0.081681	-0.287178	NaN	-0.017073	-0.015040	-0.365633	0.971747	0.014772	0.000450	0.968539	0.959741	1.000000	0.593431
season	0.194881	0.160312	-0.225896	NaN	0.098728	0.055320	-0.287912	0.571709	0.028956	0.003801	0.570733	0.569694	0.593431	1.000000

Figure 6. Correlation matrix

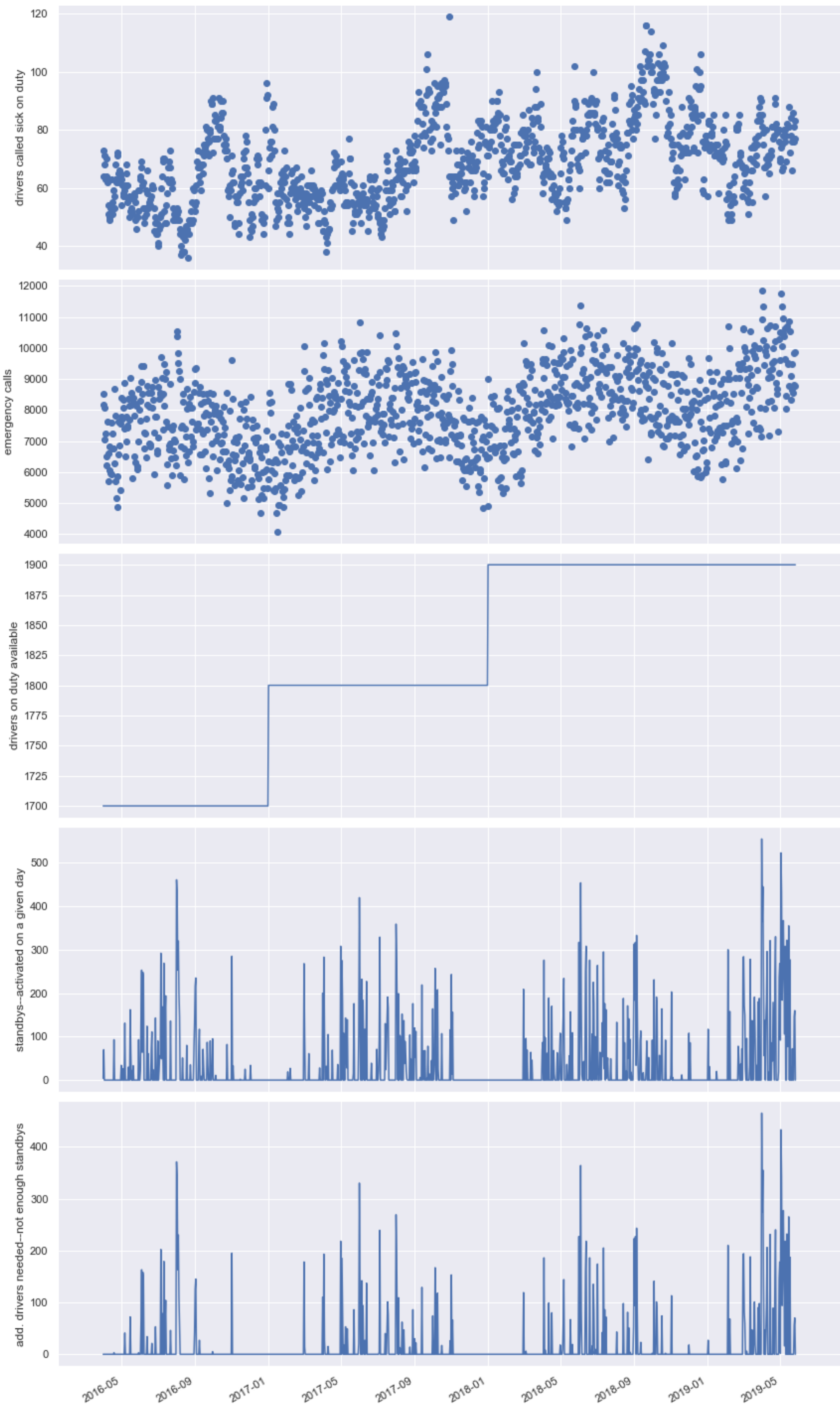


Figure 7. Time series visualization of the data

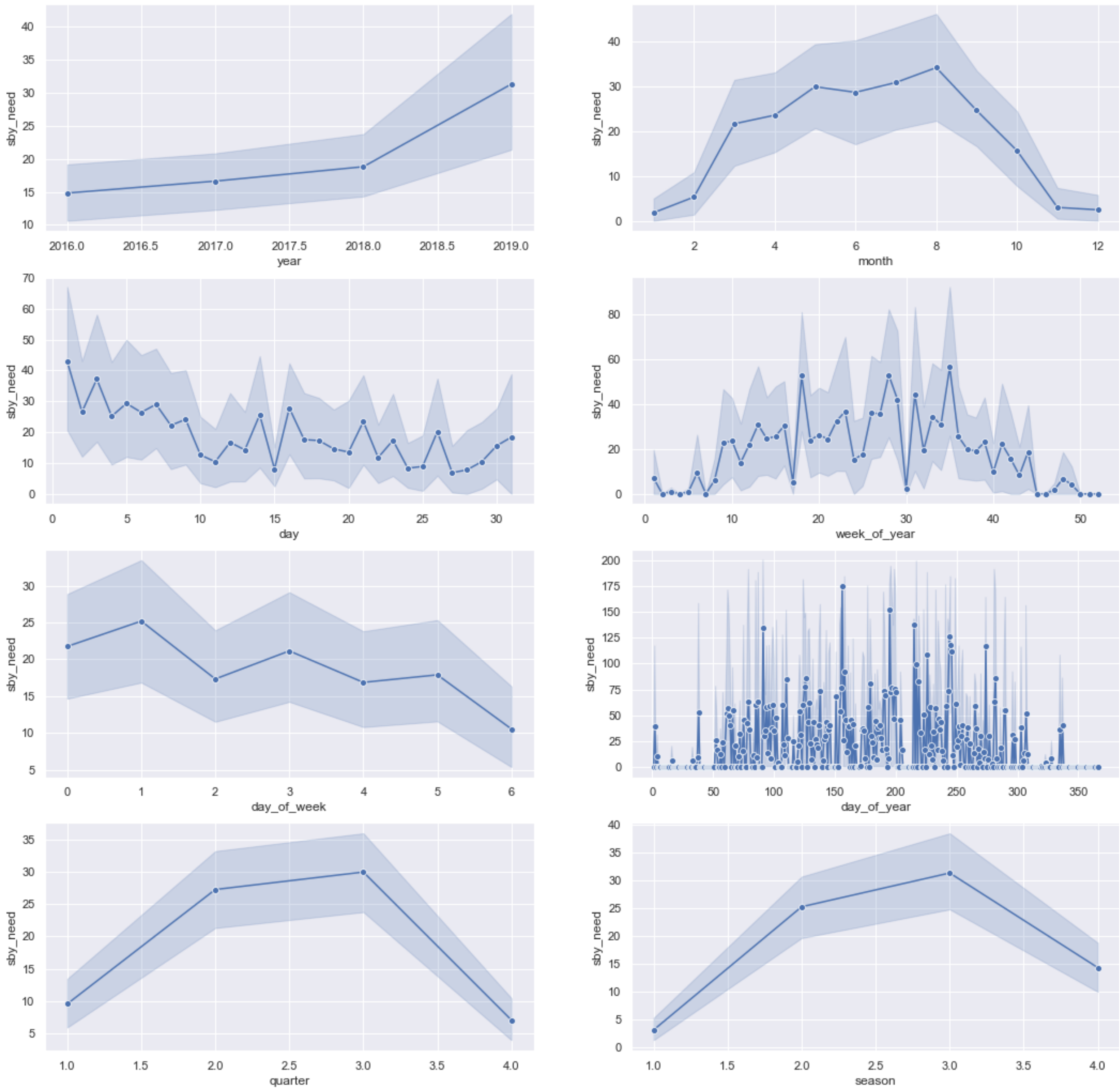


Figure 8. Seasonality trends for standby drivers needed

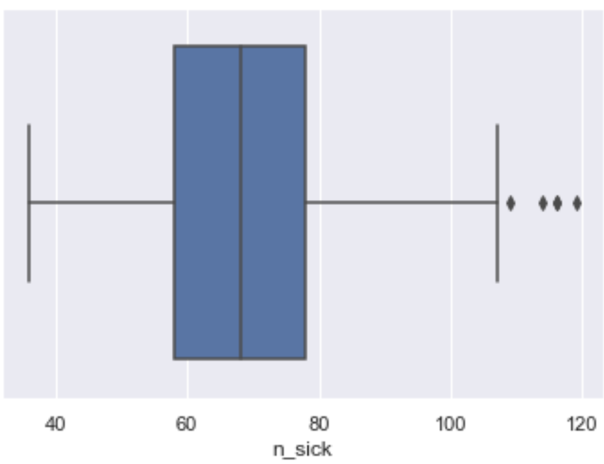


Figure 9. Outliers in number of sick drivers

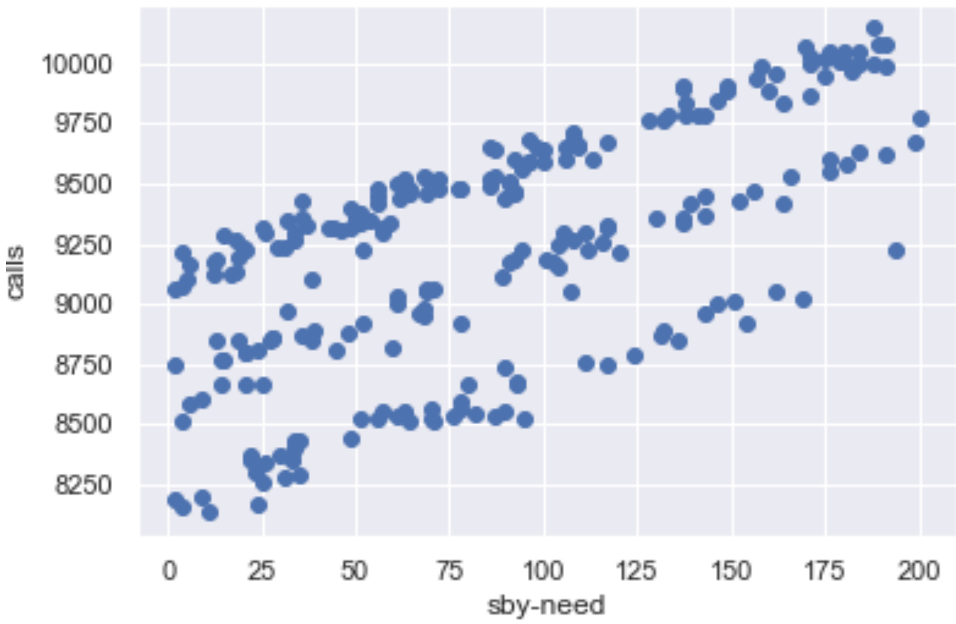


Figure 10. Emergency calls vs. standby drivers needed

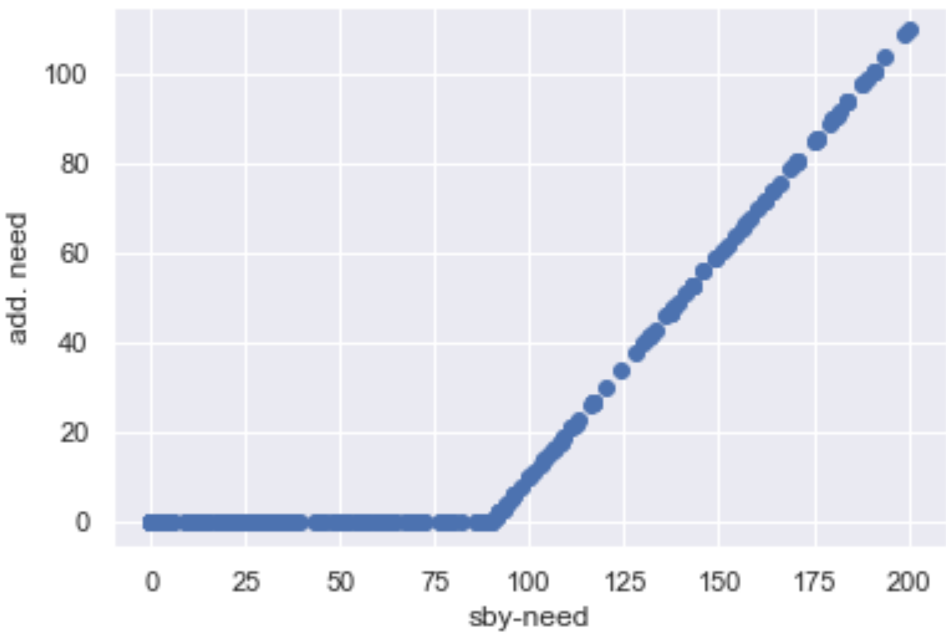


Figure 11. Standby drivers needed vs. additional drivers needed

2.3 Modeling

The first step to modeling is to understand the problem in which we are trying to solve with ML. In this solution, we were tasked with creating a model that optimally predicts how many standby drivers will be needed on any given day. For this problem, a regression model will be the most appropriate as the target variable for the dataset is numerical and there seem to be trends that can be explained with linear models. The next step would be to decide on the metrics that will be used to explain success of the model. For this problem, we are looking to minimize the number of days where not enough drivers are waiting on standby, while minimizing the days where too many drivers are assigned to standby.

2.3.1 Benchmark Models

We created baseline models that establish a benchmark for the final predictive models to be measured against. The baseline models were the calculated mean of standby drivers needed by different periods (day, day of week, season, e.g.), and a simple linear regression model. The steps in completing this phase are detailed as follows:

1. *Mean models*
    a. Calculate baseline mean of standby drivers needed (Figure 13); by day (month), day (week), and day (year) (Figure 14).
    b. Calculate the error metrics of baseline mean models. The output of these computations is shown in Figure 12.
2. *Linear regression model*. Split data in test and train datasets, fit data to the model and calculate model error (Figure 15). R^2 was an important metric to score in this model as it will be used in further regression analysis in more advanced models, so establishing a benchmark score was vital to the model building process.
3. *Compare benchmark models*. Seen in Figure 16, the models are compared using the same metrics (RMSE, MSE, MAE, and MRE).

From our creation and evaluation of benchmark models, it was clear that there is much room for improvement. The model with the best performance overall, meaning the lowest RMSE and MRE scores was the baseline mean model by year (Figure 16). The next step was to refine the results beyond these benchmark models.

```
=======
Baseline Mean (n=1085)
-------
| RMSE | 43.5718
| MSE  | 1898.50176
| MAE  | 29.45592
| Max  | 181.43318


=======
Baseline Mean - day/year (n=1085)
-------
| RMSE | 33.73285
| MSE  | 1137.90507
| MAE  | 18.66636
| Max  | 141.0


=======
Baseline Mean - day/month (n=1085)
-------
| RMSE | 42.74962
| MSE  | 1827.52995
| MAE  | 28.7447
| Max  | 180.0


=======
Baseline Mean - day/week (n=1085)
-------
| RMSE | 43.35615
| MSE  | 1879.75576
| MAE  | 29.40369
| Max  | 182.0
```

Figure 12. Model evaluation of baseline means

```python
# Calculate mean
pd.set_option('mode.chained_assignment',None)
mean = np.round(df['sby_need'].mean(), 5)
df['bl_mean'] = mean
df.head()
```

	date	n_sick	calls	n_duty	n_sby	sby_need	dafted	year	month	day	day_of_week	day_of_year	week_of_year	quarter	season	bl_mean
0	2016-04-01	73	8154.0	1700	90	4.0	0.0	2016	4	1	4	92	13	2	2	18.56682
1	2016-04-02	64	8526.0	1700	90	70.0	0.0	2016	4	2	5	93	13	2	2	18.56682
2	2016-04-03	68	8088.0	1700	90	0.0	0.0	2016	4	3	6	94	13	2	2	18.56682
3	2016-04-04	71	7044.0	1700	90	0.0	0.0	2016	4	4	0	95	14	2	2	18.56682
4	2016-04-05	63	7236.0	1700	90	0.0	0.0	2016	4	5	1	96	14	2	2	18.56682

Figure 13. Baseline mean

```python
# Calculate mean by day (month)
x = np.ceil(df.groupby('day')['sby_need'].mean()).to_frame('bl_mean_day_of_month').reset_index()
df = pd.merge(x, df, on='day')

# Calculate mean by day (week)
x = np.ceil(df.groupby('day_of_week')['sby_need'].mean()).to_frame('bl_mean_day_of_week').reset_index()
df = pd.merge(x, df, on='day_of_week')

# Calculate mean by day (year)
x = np.ceil(df.groupby('day_of_year')['sby_need'].mean()).to_frame('bl_mean_day_of_year').reset_index()
df = pd.merge(x, df, on='day_of_year')

df.head()
```

	day_of_year	bl_mean_day_of_year	day_of_week	bl_mean_day_of_week	day	bl_mean_day_of_month	date	n_sick	calls	n_duty	n_sby	sby_need	dafted	year	month	week_of_year	quarter	season	bl_mean
0	1	0.0	0	22.0	1	43.0	2018-01-01	64	9006.0	1900	90	0.0	0.0	2018	1	1	1	1	18.56682
1	1	0.0	1	26.0	1	43.0	2019-01-01	57	8382.0	1900	90	0.0	0.0	2019	1	1	1	1	18.56682
2	1	0.0	6	11.0	1	43.0	2017-01-01	60	6336.0	1800	90	0.0	0.0	2017	1	52	1	1	18.56682
3	2	39.0	0	22.0	2	27.0	2017-01-02	70	8550.0	1800	90	0.0	0.0	2017	1	1	1	1	18.56682
4	2	39.0	1	26.0	2	27.0	2018-01-02	75	7224.0	1900	90	0.0	0.0	2018	1	1	1	1	18.56682

Figure 14. Baseline mean of day/week/year

```python
# Split data into test/train sets
x1 = pd.get_dummies(df[['day_of_year', 'day', 'day_of_week', 'year', 'month', 'season']].astype(str))
X1, X2, y1, y2 = train_test_split(x1, df['sby_need'], random_state=5, train_size=0.7)

# Fit data to model
from sklearn.linear_model import Ridge
ridge_model = Ridge(alpha=20)
ridge_model.fit(X1, y1)
y_hat = ridge_model.predict(X2)
print(ridge_model.score(X2, y2))

# Evaluate model performance
lin_reg_bl_rmse, lin_reg_bl_max = metrics(y2, y_hat, 'Linear Reg.', 'just_print')
```

```
0.07440888623263897
=======
Linear Reg. (n=326)
-------
| RMSE | 44.31846
| MSE  | 1964.1263
| MAE  | 28.22533
| Max  | 167.70059
```

Figure 15. Linear regression model creation and evaluation

```python
results_rmse = [lin_reg_bl_rmse, bl_mean_week_rmse, bl_mean_month_rmse, bl_mean_year_rmse, bl_mean_rmse]
results_max = [lin_reg_bl_max, bl_mean_week_max, \
               bl_mean_month_max, bl_mean_year_max, bl_mean_max]
print(f'Benchmark Results')
print('========')
print(f'LinReg: RMSE = {lin_reg_bl_rmse}, Max = {lin_reg_bl_max}')
print(f'BL-mean: RMSE {bl_mean_rmse}, Max = {bl_mean_max}')
print(f'BL-mean-week: RMSE = {bl_mean_week_rmse}, Max = {bl_mean_week_max}')
print(f'BL-mean-month: RMSE = {bl_mean_month_rmse}, Max = {bl_mean_month_max}')
print(f'BL-mean-year: RMSE = {bl_mean_year_rmse}, Max = {bl_mean_year_max}\n')

print(f'*Best RMSE score* : {min(results_rmse)}')
print(f'*Best Max Error score* : {min(results_max)}')
```

```
Benchmark Results
========
LinReg: RMSE = 44.31846, Max = 167.70059
BL-mean: RMSE 43.5718, Max = 181.43318
BL-mean-week: RMSE = 43.35615, Max = 182.0
BL-mean-month: RMSE = 42.74962, Max = 180.0
BL-mean-year: RMSE = 33.73285, Max = 141.0

*Best RMSE score* : 33.73285
*Best Max Error score* : 141.0
```

Figure 16. Evaluation between baseline means and linear regression models

2.3.2 Final Prediction Model

To train and select a final prediction model for the task at hand, it was necessary to first split the dataset into training and testing sets. Then, the potential models were fit using the separated training data and evaluated on the testing data. Next, steps were taken to fine tune the selected models' hyperparameters to improve the models' accuracy. The models were evaluated again, and a final model was selected based on its generalizability and minimized error.

Validation. After an initial scan of ML technique options to solve this problem, two different regression techniques were chosen: Bernoulli Naïve Bayes[1] (BNB) and Support Vector Regression[2] (SVR). These models were chosen from their initial performance on the data, as they were the only models to provide error scores that showed a relatively high performance with room to fine tune their parameters. As a first step in refining the models, a function was written that iterates over a list of training set sizes, splits the dataset into test/train sets using each size, fit the model to the training data, and returns the best average model score after a simple two-fold cross-evaluation (VanderPlas, 2016) (Figure 17 & Figure 18).

```
Best R^2             : 0.7837477537817732
Best Train-set pct. : 0.66
```

Figure 17. Initial r-score and best training set size, SVR

```
Best R^2             : 0.7850634467396729
Best Train-set pct. : 0.72
```

Figure 18. Initial r-score and best training set size, BNB

Tune Hyperparameters. To improve the prediction ability of the models it is necessary to manipulate their hyperparameters. This was completed using what is known as a brute force cross-validation technique (VanderPlas, 2016). In this technique and for each model, a list of parameters with multiple variations is given. The algorithm will build and test a model for each combination of parameters in the given list. After each validation cycle, the best RMSE, R^2, and model parameters are recorded for further analysis, where the model with the lowest RMSE and highest R^2 is marked as the most accurate model.

Evaluate and select. After model creation and fine tuning of hyperparameters, each model was analyzed on its performance and a final prediction model was selected. After an initial fitting of

[1] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

the data to the BNB model, an r-score of 0.784 was recorded with a training set size of 66%; however, after fine tuning hyperparameters using *sklearn.model_selection.gridsearchCV*[3], the model saw a rapid decline in performance (Figure 19). This could be attributed to a few reasons:

1. The default parameters of sci-kit learn are optimized for many common cases of using the specific model and are more finely tuned than what even an expert data scientist could improve upon.
2. Based on the results, the model is not experiencing overfitting (the event in which models get hyper-tuned to their training data to a level where they're not able to generalize to new data). Evidently, the initial model was not overfitting the data originally due to the performance not increasing dramatically.
3. The decisions for how training data was randomized could be causing fluctuations in model performance, as the initial splitting of data into test/train sets was performed against a randomized seed of the data.

```
Best R^2 score: 0.035115492337631726
Best MSE: -2137.3649167733674
Best model: {'fit_prior': True, 'binarize': 0.5, 'alpha': 0.1}
```

Figure 19. BNB model scores after fine tuning

Finally, the SVR model was cycled after fine tuning/evaluation and selected as the best performing model due to its increased R^2 and minimized RMSE after cross-validation. The final error metrics and optimal model parameters are shown in Figure 20, which are stored for the company for further use and refinement of the model.

```
Best R^2 score: 0.9999999535634789
Best MSE: 7.872045324837176e-05
Best parameters: {'C': 0.1, 'epsilon': 0.01, 'gamma': 'scale', 'kernel': 'linear', 'shrinking': True}
```

Figure 20. SVR model scores and parameters

2.4 Deployment

Now that a prediction model has been created, the next steps in the MS-TDSP are to validate the model's performance with BRC to verify that the model provides a sufficient solution to the business' problem and that satisfaction is at its highest, and to deploy the model for use across all business departments and teams, so that engineering teams can develop the model further and BI teams can make ad hoc queries using insights gained from the model. The latter two points will be addressed in the following sections.

[3] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

2.4.1 Git Structure

The source code of the prediction model should be accessible to all teams within the organization. It is highly advised by the company that all code is placed in a private repository such as GitHub[4] and organized for separate business purposes. A recommended structure of organizing a repository can be found at: https://github.com/Azure/Azure-TDSP-ProjectTemplate. Here, the company will find a template for MS-TDSP projects that can be cloned and used in their private repository.

2.4.2 Dashboard Development

Like its source code, it is important in the MS-TDSP methodology for the model to be accessible via a graphical user interface (GUI). A conceptual GUI in the form of a dashboard is recommended by the company in Figure 21.

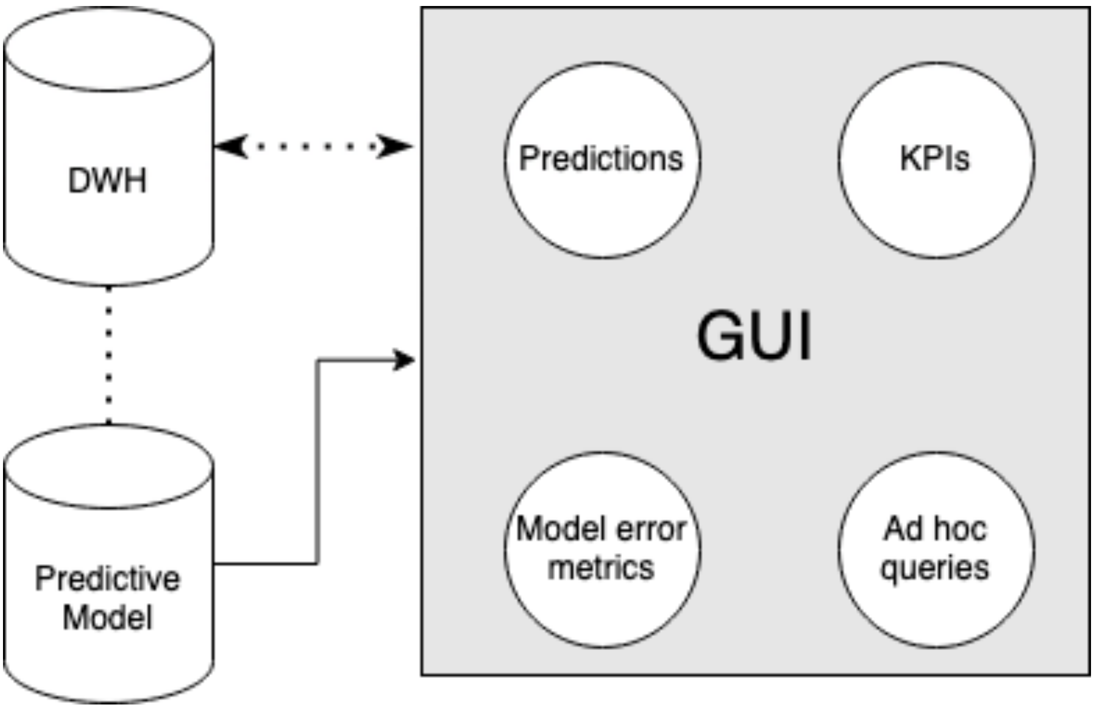


Figure 21. Conceptual GUI model

The proposed dashboard should be populated by extracted data from the DWH. Data is transferred back and forth between the model and the DWH. On the GUI, predictions, KPIs, and ad hoc queries that utilize the model are visualized with user inputs affecting the visualizations. Model error metrics and performance is visualized for users to understand the accuracy of its predictions.

[4] https://github.com/

3 Conclusion

The overall task to build a model that minimizes the number of occurrences where there are not enough standby drivers, and simultaneously predict with minimal error how many drivers will be needed on any given day, was solved by this solution. The proposed approach of using MS-TDSP as a methodology to structure and facilitate cooperation between teams within the organization is a proven structure for data science projects, and if the business approaches the task using this methodology, there will be guaranteed success.

A predictive model was presented to BRC that will help to optimize business operations. This model, while already finely tuned to solve the specific task, has the opportunity to be improved as more data becomes available to it. A GUI should be developed that allows all business units to interact and generate reports based on findings from the model. A conceptual model of a GUI was presented that the company can use as a starting point in its development.

References

Microsoft. (2022, March 1). *What is the Team Data Science Process?* Retrieved from learn.microsoft.com: https://learn.microsoft.com/en-us/azure/architecture/data-science-process/overview

VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data.* O'Reilly Media, Inc.

Appendix A. Source Code

A-1 ModelEng_ForecastRescueDrivers_EDA.py

```
#!/usr/bin/env python
# coding: utf-8

# # Exploratory Data Analysis
#
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
df = pd.read_csv('./use_case_2/sickness_table.csv', low_memory=False)

# drop columns
df = df.drop('Unnamed: 0', axis=1)
# fix data types
df['date'] = pd.to_datetime(df['date'])
# check for duplicates
df.drop_duplicates()
df.info()
df.head()

# Prepare for time-series
df = df.sort_values(by='date')
# Check time intervals
df['Time_Interval'] = df.date - df.date.shift(1)
df[['date', 'Time_Interval']].head()
print(f"{df['Time_Interval'].value_counts()}")
df = df.drop('Time_Interval', axis=1)
# check if no date appears more than once
df.date.duplicated().sum()

# check missing values
print(df.isnull().sum().sum())

df.describe()

# Visualize data
fig, ax = plt.subplots(5, sharex=True, figsize=(12, 24))
```

```
ax[0].scatter(df['date'], df['n_sick'])
ax[1].scatter(df['date'], df['calls'])
ax[2].plot(df['date'], df['n_duty'])
ax[3].plot(df['date'], df['sby_need'])
ax[4].plot(df['date'], df['dafted'])
ax[0].set_ylabel('drivers called sick on duty')
ax[1].set_ylabel('emergency calls')
ax[2].set_ylabel('drivers on duty available')
ax[3].set_ylabel('standbys--activated on a given day')
ax[4].set_ylabel('add. drivers needed--not enough standbys')
fig.suptitle('Seasonal Features Data', fontsize=16, y=1.005, x=0.50)
plt.tight_layout()
fig.autofmt_xdate()
plt.savefig('seasonal_features.png')
plt.show()


# ### What we can observe from visualization:
#
# - Potential outlier in *n_sick* around 2018, in *sby_need* and *dafted*
# - Possible regression trends in *n_sick* and *calls* (these will be tested during creation of benchmark models)
# - Time-series: seasonality patterns monthly (*sby_need* and *dafted*)


# ---
# Let's check for outliers in *n_sick* column:
sns.boxplot(df['n_sick'])
plt.savefig('n_sick_outliers.png')
df[df['n_sick'] > 105]


# From the plot it seemed that there were too many outliers above n=105, however, from the table we can see that this is just an upper limit of the data.
#
# Now let's check the outliers in sby_need:
df[df['sby_need'] > 200]


# It seems there are quite a bit of instances greater than two standard deviations above the mean, which could affect the data negatively. Let's remove those.
df = df.loc[((df['sby_need'] >= 0) & (df['sby_need'] <= 200))]
df.info()
```

```
# -----
# Now let's check the seasonality of the data on when standy drivers are needed
df['year'] = pd.DatetimeIndex(df['date']).year
df['month'] = pd.DatetimeIndex(df['date']).month
df['day'] = pd.DatetimeIndex(df['date']).day
df['day_of_week'] = pd.DatetimeIndex(df['date']).dayofweek
df['day_of_year'] = pd.DatetimeIndex(df['date']).dayofyear
df['week_of_year'] = pd.DatetimeIndex(df['date']).weekofyear
df['quarter'] = pd.DatetimeIndex(df['date']).quarter
df['season'] = df.month%12 // 3 + 1

df.to_csv('cleaned_data.csv')

plt.figure(figsize=(18, 18))

i = 0
cols = ['year', 'month', 'day', 'week_of_year', 'day_of_week', 'day_of_year',
'quarter', 'season']
for col in cols:
    i+=1
    plt.subplot(4, 2, i)
    ax = sns.lineplot(x=col, y='sby_need', marker='o', data=df)
plt.savefig('time_series_plot.png')
plt.show()



# -----
# Now let's explore the relationships between some variables by looking at the
correlation matrix

df.corr()

# This shows a few positive correlations worth visualizing:
#
# - Number of sick drivers vs. month/day/week/season
# - Number of emergency calls vs. number of standby drivers needed
# - Number of standby drivers needed vs. number of additional drivers needed

# Number of sick drivers vs. month/day/week/season
plt.figure(figsize=(12, 12))
```

```
i = 0
cols = ['month', 'day', 'week_of_year', 'season']
for col in cols:
    i+=1
    plt.subplot(2, 2, i)
    ax = sns.lineplot(x=col, y='n_sick', marker='o', data=df)
plt.savefig('sick_vs_season.png')
plt.show()

# Number of emergency calls vs. number of standby drivers needed
plt.figure(figsize=(10,10))

# Remove instances where 0 drivers are needed
only_need = df.where(df['sby_need'] > 0)

fig, ax = plt.subplots()
ax.scatter(only_need['sby_need'], only_need['calls'])
ax.set_xlabel('sby-need')
ax.set_ylabel('calls')
plt.savefig('standby_need_vs_calls.png')
plt.show()

# Number of standby drivers needed vs. number of additional drivers needed
plt.figure(figsize=(10,10))

fig, ax = plt.subplots()
ax.scatter(df['sby_need'], df['dafted'])
ax.set_xlabel('sby-need')
ax.set_ylabel('add. need')
plt.savefig('sby_need_vs_add.png')
plt.show()
```

A-2 ModelEng_ForecastRescueDrivers_Baseline-Model.py

```
#!/usr/bin/env python
# coding: utf-8

# # Baseline Model
# ------
```

```
#
# - Calculate driver need mean by day, day of week, season
# - Establish a simple linear regression model
# - Validate performance of the models to establish a benchmark
import pandas as pd
import numpy as np
import seaborn as sn
import datetime
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import max_error

import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import seaborn; seaborn.set()
df = pd.read_csv('./cleaned_data.csv', low_memory=False)
df = df.drop(['Unnamed: 0'], axis=1)

# Calculate mean
pd.set_option('mode.chained_assignment',None)
mean = np.round(df['sby_need'].mean(), 5)
df['bl_mean'] = mean
df.head()

# Calculate mean by day (month)
x                                                                              =
np.ceil(df.groupby('day')['sby_need'].mean()).to_frame('bl_mean_day_of_month'
).reset_index()
df = pd.merge(x, df, on='day')

# Calculate mean by day (week)
x                                                                              =
np.ceil(df.groupby('day_of_week')['sby_need'].mean()).to_frame('bl_mean_day_o
f_week').reset_index()
df = pd.merge(x, df, on='day_of_week')

# Calculate mean by day (year)
```

```
x                                                                            =
np.ceil(df.groupby('day_of_year')['sby_need'].mean()).to_frame('bl_mean_day_o
f_year').reset_index()
df = pd.merge(x, df, on='day_of_year')


df.head()


# Define a function to calculate error metrics for baseline means
def metrics(y, y_hat, title='baseline mean', save_or_print='just_print', tar-
get_var='sby_need'):
    mse = np.round(mean_squared_error(y, y_hat), 5)
    rmse = np.round(np.sqrt(mean_squared_error(y, y_hat)), 5)
    mae = np.round(mean_absolute_error(y, y_hat), 5)
    max_r = np.round(max_error(y, y_hat), 5)

    print('=======')
    print(f'{title} (n={len(y)})')
    print('-------')
    print(f'| RMSE | {rmse} ')
    print(f'| MSE  | {mse}')
    print(f'| MAE  | {mae}')
    print(f'| Max  | {max_r}')
    print('\n')

    if save_or_print is not 'just_print':
        with open('baseline_model_error_metrics.csv', 'a+') as file:
            date = datetime.datetime.now()
            row = f'\n{title}, {rmse}, {mse}, {mae}, {max_r}, {len(y)}, {tar-
get_var}, {date}'
            file.write(row)

    return rmse, max_r



bl_mean_rmse, bl_mean_max = metrics(df['bl_mean'], df['sby_need'], 'Baseline
Mean', 'save')
bl_mean_year_rmse,  bl_mean_year_max  =  metrics(df['bl_mean_day_of_year'],
df['sby_need'], 'Baseline Mean - day/year', 'save')
bl_mean_month_rmse, bl_mean_month_max = metrics(df['bl_mean_day_of_month'],
df['sby_need'], 'Baseline Mean - day/month', 'save')
bl_mean_week_rmse,  bl_mean_week_max  =  metrics(df['bl_mean_day_of_week'],
df['sby_need'], 'Baseline Mean - day/week', 'save')
```

```
# # Linear Regression Baseline Model
# ----
#
# - Split data into test-train sets
# - Fit data to model
# - Evaluate model performance

# Split data into test/train sets
x1 = pd.get_dummies(df[['day_of_year', 'day', 'day_of_week', 'year', 'month', 'season']].astype(str))
X1, X2, y1, y2 = train_test_split(x1, df['sby_need'], random_state=5, train_size=0.7)

# Fit data to model
from sklearn.linear_model import Ridge
ridge_model = Ridge(alpha=20)
ridge_model.fit(X1, y1)
y_hat = ridge_model.predict(X2)
print(ridge_model.score(X2, y2))

# Evaluate model performance
lin_reg_bl_rmse, lin_reg_bl_max = metrics(y2, y_hat, 'Linear Reg.', 'just_print')



# ## Compare Benchmark Models


results_rmse = [lin_reg_bl_rmse, bl_mean_week_rmse, bl_mean_month_rmse, bl_mean_year_rmse, bl_mean_rmse]
results_max = [lin_reg_bl_max, bl_mean_week_max, \
               bl_mean_month_max, bl_mean_year_max, bl_mean_max]
print(f'Benchmark Results')
print('========')
print(f'LinReg: RMSE = {lin_reg_bl_rmse}, Max = {lin_reg_bl_max}')
print(f'BL-mean: RMSE {bl_mean_rmse}, Max = {bl_mean_max}')
print(f'BL-mean-week: RMSE = {bl_mean_week_rmse}, Max = {bl_mean_week_max}')
print(f'BL-mean-month: RMSE = {bl_mean_month_rmse}, Max = {bl_mean_month_max}')
print(f'BL-mean-year: RMSE = {bl_mean_year_rmse}, Max = {bl_mean_year_max}\n')


print(f'*Best RMSE score* : {min(results_rmse)}')
```

```
print(f'*Best Max Error score* : {min(results_max)}')


# According to the initial results, the Baseline Mean - Days/Year model out performs all benchmarks.
```

A-3 ModelEng_ForecastRescueDrivers_Prediction-Model.py

```
#!/usr/bin/env python
# coding: utf-8

# # Prediction Model
# ----
#
# - Split data into test/train sets
# - Fit data to potential models: Support Vector Regression & Bernoulli Naive Bayes
# - Evaluate performance of models using cross-validation
# - Tune hyperparameters
# - Evaluate model performance again
# - Select best model for deployment

import pandas as pd
import numpy as np
import seaborn as sn
import datetime
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import max_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import seaborn; seaborn.set()
df = pd.read_csv('./cleaned_data.csv', low_memory=False)
```

```
df = df.drop(['Unnamed: 0'], axis=1)


# Split data into train/test sets
#
# Make feature matrix
x1 = pd.get_dummies(df[['day_of_year', 'day', 'day_of_week', 'year', 'month',
'season']].astype(str))
# Merge sby_need with matrix
x1['sby_need'] = df['sby_need']
# Split into train/test
X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=0.8)


# Make validation sets
Xval, Xval2, Yval, Yval2 = train_test_split(x1, x1['sby_need'],
train_size=0.5)



# ### Model validation
# ---
#
# - In order to verify optimal size of training set data, a function is used
to iterate through a list of sizes, split the data using each size, and return
the best average model score after two-fold cross-validation.


# Support vector regression
from sklearn import svm


r2_best = 0
trainset_size = 0


for i in np.arange(0.5, 0.98, 0.02):


    # Split into train/test
    X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=i, ran-
dom_state=42)


    svr_model = svm.SVR(gamma='auto').fit(X1, y1)
    y_hat_svr = svr_model.predict(X2)


    svr_model_val1 = svm.SVR(gamma='auto').fit(Xval, Yval)
    svr_model_val2 = svm.SVR(gamma='auto').fit(Xval2, Yval2)
    yhat_svrm1 = svr_model_val1.predict(Xval2)
```

```
    yhat_svrm2 = svr_model_val2.predict(Xval)

    r2 = svr_model.score(X2, y2)
    r2_test = svr_model.score(X1, y1)
    r2_val1 = svr_model_val1.score(Xval2, Yval2)
    r2_val2 = svr_model_val2.score(Xval, Yval)
    r2_mean = np.mean([r2, r2_test, r2_val1, r2_val2])

    if r2_mean > r2_best:
        r2_best = r2
        trainset_size = i

print(f'Best R^2            : {r2_best}')
print(f'Best Train-set pct. : {np.round(trainset_size, 2)}')

svr = {'best_r2': r2_best, 'best_train_size': np.round(trainset_size, 2)}

# Fit data to model - bernoulli naive bayes
from sklearn.naive_bayes import BernoulliNB

r2_best = 0
trainset_size = 0

for i in np.arange(0.5, 0.98, 0.02):

    # Split into train/test
    X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=i, ran-
dom_state=42)

    bnb_model = BernoulliNB(class_prior=None).fit(X1, y1)
    y_hat_bnb = bnb_model.predict(X2)

    bnb_val1 = BernoulliNB().fit(Xval, Yval)
    bnb_val2 = BernoulliNB().fit(Xval2, Yval2)
    y_hat_bnb_val1 = bnb_val1.predict(Xval2)
    y_hat_bnb_val2 = bnb_val2.predict(Xval)

    r2 = bnb_model.score(X2, y2)
    r2_test = bnb_model.score(X1, y1)
    r2_val1 = bnb_val1.score(Xval2, Yval2)
```

```
    r2_val2 = bnb_val2.score(Xval, Yval)
    r2_mean = np.mean([r2, r2_test, r2_val1, r2_val2])

    if r2_mean > r2_best:
        r2_best = r2_mean
        trainset_size = i

print(f'Best R^2            : {r2_best}')
print(f'Best Train-set pct. : {np.round(trainset_size, 2)}')

bnb = {'best_r2': r2_best, 'best_train_size': np.round(trainset_size, 2)}



# ## Tune Hyperparameters
# ----
#
# - Using brute force cross-validation to evaluate parameter performance, each model will have a list of the parameters used for each validation cycle along with the best score results.
# - The best scores from each model tuning will be printed and automatically selected.


# #### Bernoulli Naive Bayes model


from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import r2_score, mean_squared_error


# Split into train/test
X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=bnb['best_train_size'], random_state=42)


param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5],
    'fit_prior': [True, False],
    'binarize': [0.0, 0.5, 1.0]
}


scoring = {'R^2': 'r2', 'MSE': 'neg_mean_squared_error'}
search = RandomizedSearchCV(bnb_model, param_grid, cv=5, n_iter=10, scoring=scoring, refit='R^2').fit(X1, y1)
```

```
print("Best R^2 score:", search.best_score_)

best_r2_score = -float('inf')
best_mse_score = float('inf')
best_model = None

for mean_r2, mean_mse, params in zip(search.cv_results_['mean_test_R^2'], search.cv_results_['mean_test_MSE'], search.cv_results_['params']):
    if mean_r2 > best_r2_score:
        best_r2_score = mean_r2
        best_mse_score = mean_mse
        best_model = params
    elif mean_r2 == best_r2_score and mean_mse < best_mse_score:
        best_mse_score = mean_mse
        best_model = params

print("Best R^2 score:", best_r2_score)
print("Best MSE:", best_mse_score)
print("Best model:", best_model)


# The models performance is lower after tuning hyper-parameters. This suggests that:
#
# 1. The default parameters of Scikit-learn are more finely-tuned than what a novice data scientist could initially create.
# 2. The model is not experiencing overfitting, as the performance on initial model is not overly higher than the fine-tuned version.
# 3. The randomization of training data could be what is causing model perfro-mance to fluctuate


# #### Support Vector Regression model


# Split into train/test
X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=svr['best_train_size'], random_state=42)


param_grid = {
    'C': [0.1, 1, 10],
    'epsilon': [0.01, 0.1, 1],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto'],
```

```
    'shrinking': [True],
}
scoring = {'R^2': 'r2', 'MSE': 'neg_mean_squared_error'}
search = GridSearchCV(svr_model, param_grid, scoring=scoring, refit='R^2', cv=5).fit(X1, y1)

best_svr = search.best_estimator_

y_pred = best_svr.predict(X2)
r2 = r2_score(y2, y_pred)
mse = mean_squared_error(y2, y_pred)

print("Best R^2 score:", r2)
print("Best MSE:", mse)
print("Best parameters:", search.best_params_)

def select_best_model(models_dict):
    best_r2_score = -float('inf')
    best_mse_score = float('inf')
    best_model_params = None

    for (r2_score, mse_score), model_params in models_dict.items():
        if mse_score < best_mse_score or (mse_score == best_mse_score and r2_score > best_r2_score):
            best_r2_score = r2_score
            best_mse_score = mse_score
            best_model_params = model_params

    return best_model_params, best_r2_score, best_mse_score



score_models_dict = {}

# Iterate over all models in grid search results
for mean_r2, mean_mse, params in zip(search.cv_results_['mean_test_R^2'], search.cv_results_['mean_test_MSE'], search.cv_results_['params']):
    score_models_dict[(mean_r2, -mean_mse)] = params

print("\nDictionary of R^2 scores, MSE scores, and corresponding models:")
for (r2_score, mse_score), model_params in score_models_dict.items():
    print("R^2 score:", r2_score)
```

```
    print("MSE score:", mse_score)
    print("Model parameters:", model_params)
    print("--------------------")

best_params, best_r2, best_mse = select_best_model(score_models_dict)

print("\nBest model parameters:", best_params)
print("Best R^2 score:", best_r2)
print("Best MSE score:", best_mse)
```

INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

Appendix B. Jupyter Notebooks

B-1 Exploratory Data Analysis

Exploratory Data Analysis

In [115…

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
df = pd.read_csv('./use_case_2/sickness_table.csv', low_memory=False)
```

In [116…

```
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1152 entries, 0 to 1151
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Unnamed: 0   1152 non-null   int64
 1   date         1152 non-null   object
 2   n_sick       1152 non-null   int64
 3   calls        1152 non-null   float64
 4   n_duty       1152 non-null   int64
 5   n_sby        1152 non-null   int64
 6   sby_need     1152 non-null   float64
 7   dafted       1152 non-null   float64
dtypes: float64(3), int64(4), object(1)
memory usage: 72.1+ KB
```

Out[116]:

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

In [117…

```
# drop columns
df = df.drop('Unnamed: 0', axis=1)
# fix data types
df['date'] = pd.to_datetime(df['date'])
# check for duplicates
df.drop_duplicates()
df.info()
df.head()
```

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

```
dtypes: datetime64[ns](1), float64(3), int64(3)
memory usage: 63.1 KB
```

Out[117]:

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

In [118…

```
# Prepare for time-series
df = df.sort_values(by='date')
# Check time intervals
df['Time_Interval'] = df.date - df.date.shift(1)
df[['date', 'Time_Interval']].head()
```

Out[118]:

	date	Time_Interval
0	2016-04-01	NaT
1	2016-04-02	1 days
2	2016-04-03	1 days
3	2016-04-04	1 days
4	2016-04-05	1 days

In [119…

```
print(f"{df['Time_Interval'].value_counts()}")
df = df.drop('Time_Interval', axis=1)
```

```
1 days    1151
Name: Time_Interval, dtype: int64
```

In [120…

```
# check if no date appears more than once
df.date.duplicated().sum()
```

Out[120]: 0

In [121…

```
# check missing values
print(df.isnull().sum().sum())
```

```
0
```

In [122…

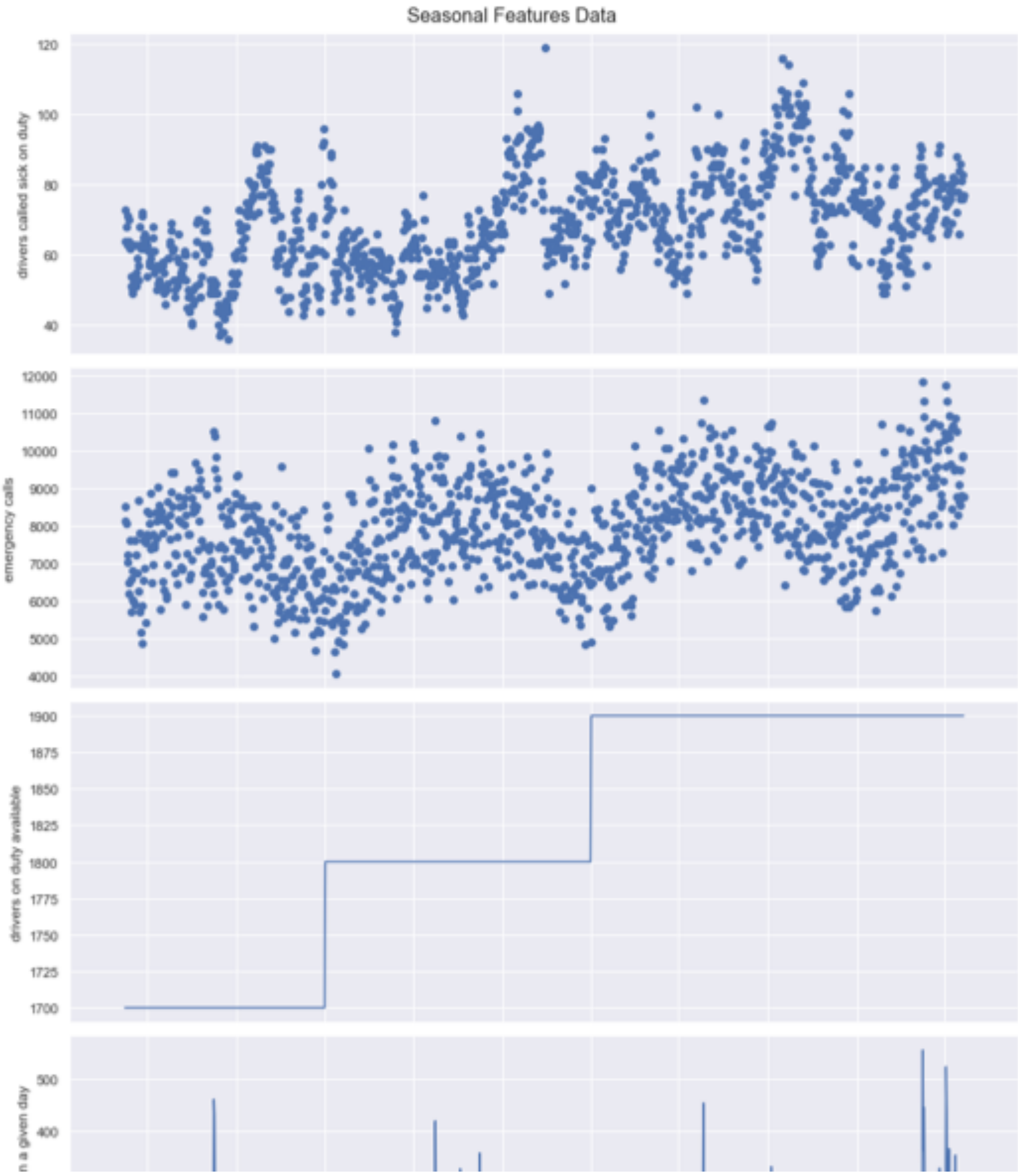
```
df.describe()
```

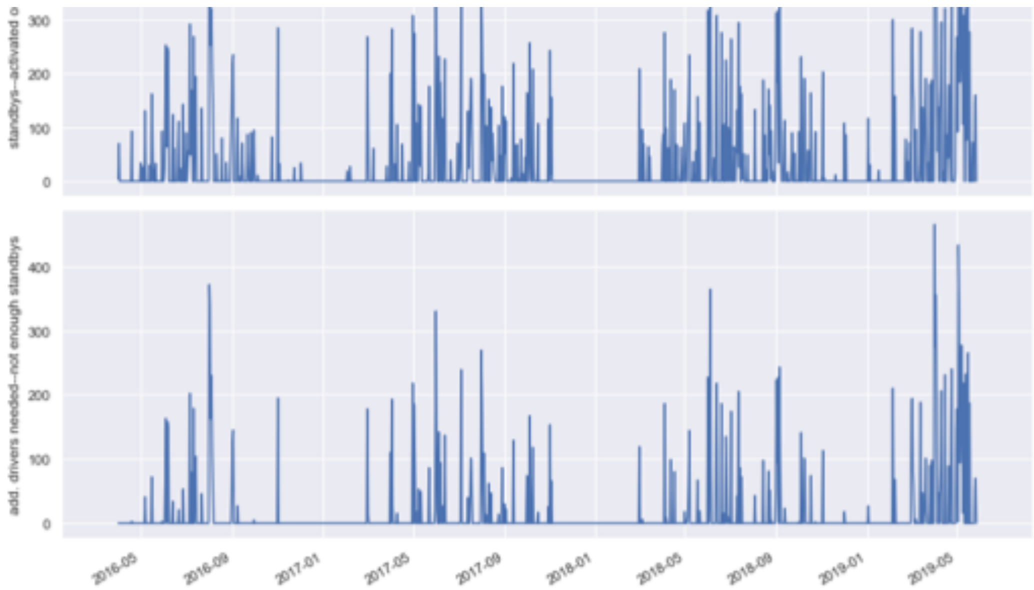
Out[122]:

	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

In [123…

```python
# Visualize data
fig, ax = plt.subplots(5, sharex=True, figsize=(12, 24))
ax[0].scatter(df['date'], df['n_sick'])
ax[1].scatter(df['date'], df['calls'])
ax[2].plot(df['date'], df['n_duty'])
ax[3].plot(df['date'], df['sby_need'])
ax[4].plot(df['date'], df['dafted'])
ax[0].set_ylabel('drivers called sick on duty')
ax[1].set_ylabel('emergency calls')
ax[2].set_ylabel('drivers on duty available')
ax[3].set_ylabel('standbys--activated on a given day')
ax[4].set_ylabel('add. drivers needed--not enough standbys')
fig.suptitle('Seasonal Features Data', fontsize=16, y=1.005, x=0.50)
plt.tight_layout()
fig.autofmt_xdate()
plt.show()
```





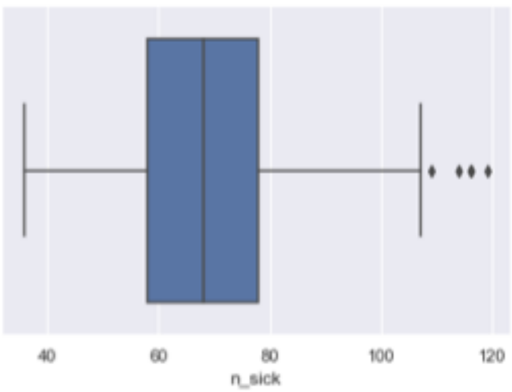
What we can observe from visualization:

- Potential outlier in *n_sick* around 2018, in *sby_need* and *dafted*
- Possible regression trends in *n_sick* and *calls* (these will be tested during creation of benchmark models)
- Time-series: seasonality patterns monthly (*sby_need* and *dafted*)

Let's check for outliers in *n_sick* column:

In [124…
```
sns.boxplot(df['n_sick'])
```

Out[124]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x132cdb6d0>
```



In [125…
```
df[df['n_sick'] > 105]
```

Out[125]:

	date	n_sick	calls	n_duty	n_sby	sby_need	dafted
539	2017-09-22	106	7794.0	1800	90	0.0	0.0

576	2017-10-29	119	7764.0	1800	90	0.0	0.0
901	2018-09-19	107	8040.0	1900	90	0.0	0.0
902	2018-09-20	116	8784.0	1900	90	0.0	0.0
903	2018-09-21	116	9108.0	1900	90	38.0	0.0
907	2018-09-25	106	9228.0	1900	90	52.0	0.0
909	2018-09-27	106	7800.0	1900	90	0.0	0.0
910	2018-09-28	114	7278.0	1900	90	0.0	0.0
924	2018-10-12	106	7506.0	1900	90	0.0	0.0
931	2018-10-19	109	6852.0	1900	90	0.0	0.0
994	2018-12-21	106	5844.0	1900	90	0.0	0.0

From the plot it seemed that there were too many outliers above n=105, however, from the table we can see that this is just an upper limit of the data.

Now let's check the outliers in sby_need:

In [126…

```
df[df['sby_need'] > 200]
```

Out[126]:

	date	n_sick	calls	n_duty	n_sby	sby_need	dafted
64	2016-06-04	67	9426.0	1700	90	253.0	163.0
66	2016-06-06	62	9426.0	1700	90	248.0	158.0
67	2016-06-07	53	9414.0	1700	90	236.0	146.0
96	2016-07-06	51	9702.0	1700	90	292.0	202.0
101	2016-07-11	70	9492.0	1700	90	269.0	179.0
...	...	...	...	...	...	...	...
1132	2019-05-08	80	10368.0	1900	90	254.0	164.0
1134	2019-05-10	80	10638.0	1900	90	308.0	218.0
1137	2019-05-13	82	10698.0	1900	90	322.0	232.0
1140	2019-05-16	81	10866.0	1900	90	355.0	265.0
1142	2019-05-18	72	10524.0	1900	90	277.0	187.0

67 rows × 7 columns

It seems there are quite a bit of instances greater than two standard deviations above the mean, which could affect the data negatively. Let's remove those.

In [127…

```
df = df.loc[((df['sby_need'] >= 0) & (df['sby_need'] <= 200))]
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1085 entries, 0 to 1151
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   date      1085 non-null   datetime64[ns]
 1   n_sick    1085 non-null   int64
 2   calls     1085 non-null   float64
```

```
 3   n_duty     1085 non-null   int64
 4   n_sby      1085 non-null   int64
 5   sby_need   1085 non-null   float64
 6   dafted     1085 non-null   float64
dtypes: datetime64[ns](1), float64(3), int64(3)
memory usage: 67.8 KB
```

Now let's check the seasonality of the data on when standy drivers are needed

In [128…

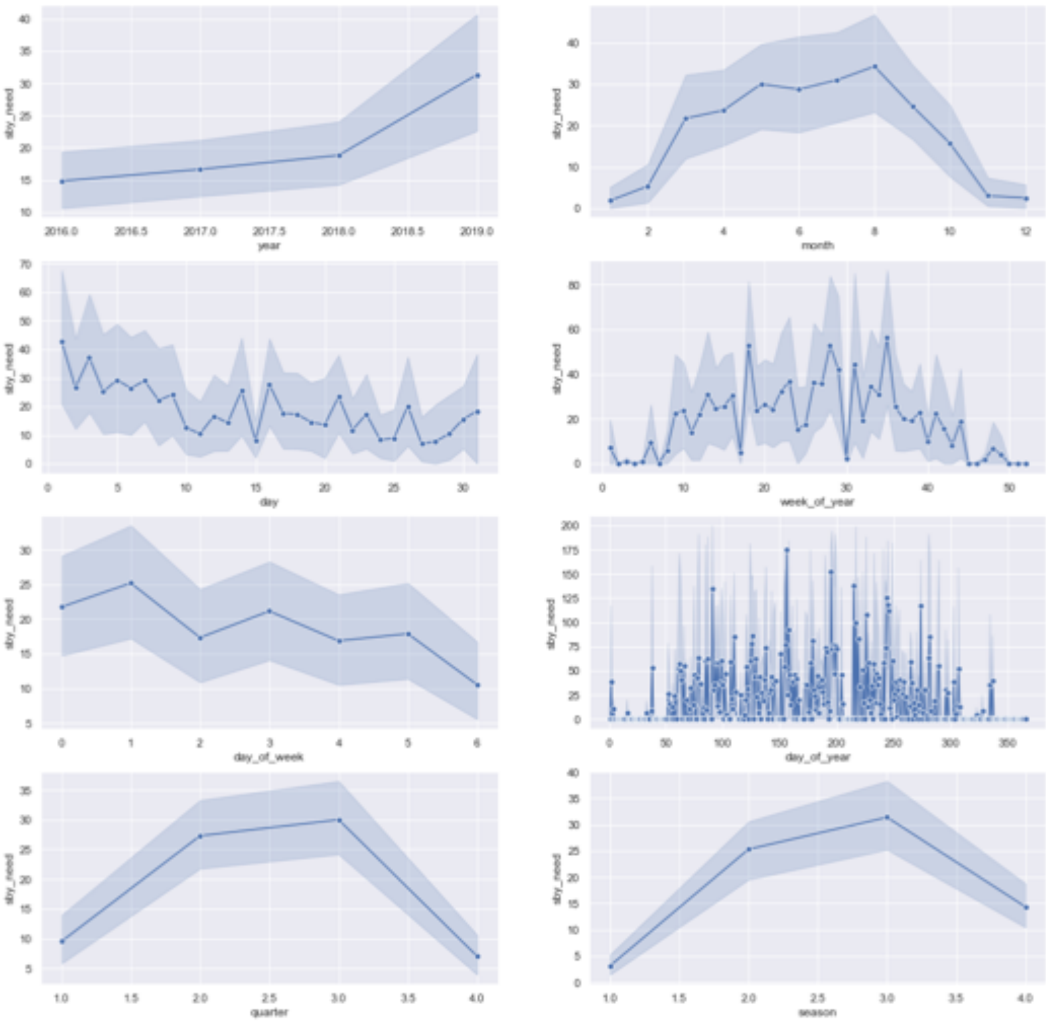
```
df['year'] = pd.DatetimeIndex(df['date']).year
df['month'] = pd.DatetimeIndex(df['date']).month
df['day'] = pd.DatetimeIndex(df['date']).day
df['day_of_week'] = pd.DatetimeIndex(df['date']).dayofweek
df['day_of_year'] = pd.DatetimeIndex(df['date']).dayofyear
df['week_of_year'] = pd.DatetimeIndex(df['date']).weekofyear
df['quarter'] = pd.DatetimeIndex(df['date']).quarter
df['season'] = df.month%12 // 3 + 1

df.to_csv('cleaned_data.csv')
```

In [129…

```
plt.figure(figsize=(18, 18))

i = 0
cols = ['year', 'month', 'day', 'week_of_year', 'day_of_week', 'day_of_year', 'quarter',
for col in cols:
    i+=1
    plt.subplot(4, 2, i)
    ax = sns.lineplot(x=col, y='sby_need', marker='o', data=df)
plt.show()
```



Now let's explore the relationships between some variables by looking at the correlation matrix

In [130…
```
df.corr()
```

Out[130]:

	n_sick	calls	n_duty	n_sby	sby_need	dafted	year	month	
n_sick	1.000000	0.162878	0.450137	NaN	0.014320	0.000239	0.399826	0.184001	0.11
calls	0.162878	1.000000	0.374537	NaN	0.600676	0.427227	0.383679	-0.076633	-0.20
n_duty	0.450137	0.374537	1.000000	NaN	0.070021	0.072406	0.951256	-0.283837	-0.00
n_sby	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
sby_need	0.014320	0.600676	0.070021	NaN	1.000000	0.861084	0.093016	-0.014943	-0.1
dafted	0.000239	0.427227	0.072406	NaN	0.861084	1.000000	0.092059	-0.017532	-0.10
year	0.399826	0.383679	0.951256	NaN	0.093016	0.092059	1.000000	-0.363946	-0.00
month	0.184001	-0.076633	-0.283837	NaN	-0.014943	-0.017532	-0.363946	1.000000	0.01

day	0.116860	-0.202063	-0.008700	NaN	-0.132877	-0.100366	-0.006020	0.016645	1.00
day_of_week	-0.060520	-0.187590	0.002536	NaN	-0.081426	-0.037074	0.007567	0.004510	-0.0
day_of_year	0.192167	-0.095613	-0.285770	NaN	-0.027149	-0.026820	-0.364865	0.996687	0.0
week_of_year	0.195761	-0.087755	-0.278929	NaN	-0.024666	-0.025139	-0.357655	0.986422	0.0
quarter	0.177704	-0.081681	-0.287178	NaN	-0.017073	-0.015040	-0.365633	0.971747	0.0
season	0.194881	0.160312	-0.225896	NaN	0.098728	0.055320	-0.287912	0.571709	0.0

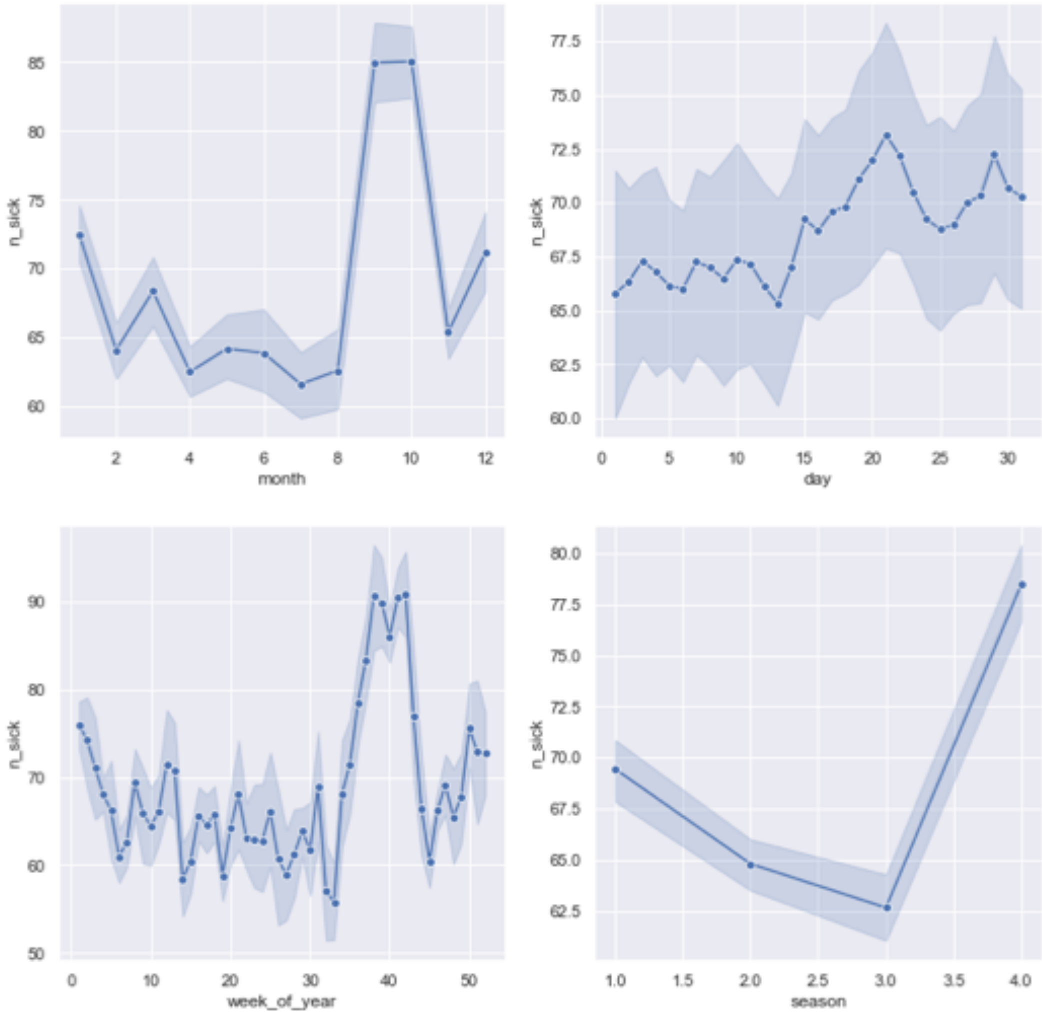
This shows a few positive correlations worth visualizing:

- Number of sick drivers vs. month/day/week/season
- Number of emergency calls vs. number of standby drivers needed
- Number of standby drivers needed vs. number of additional drivers needed

In [131…

```
# Number of sick drivers vs. month/day/week/season
plt.figure(figsize=(12, 12))

i = 0
cols = ['month', 'day', 'week_of_year', 'season']
for col in cols:
    i+=1
    plt.subplot(2, 2, i)
    ax = sns.lineplot(x=col, y='n_sick', marker='o', data=df)
plt.show()
```



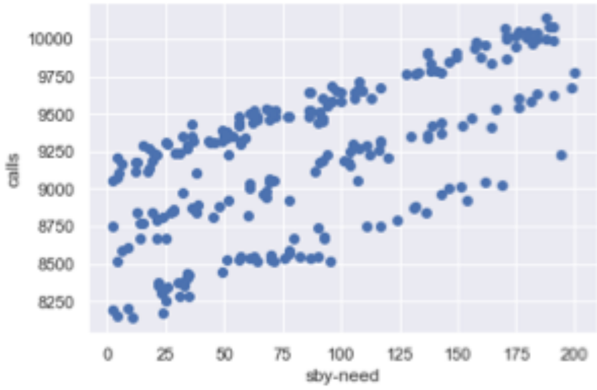
In [132…

```
# Number of emergency calls vs. number of standby drivers needed
plt.figure(figsize=(10,10))

# Remove instances where 0 drivers are needed
only_need = df.where(df['sby_need'] > 0)

fig, ax = plt.subplots()
ax.scatter(only_need['sby_need'], only_need['calls'])
ax.set_xlabel('sby-need')
ax.set_ylabel('calls')
plt.show()
```

```
<Figure size 720x720 with 0 Axes>
```

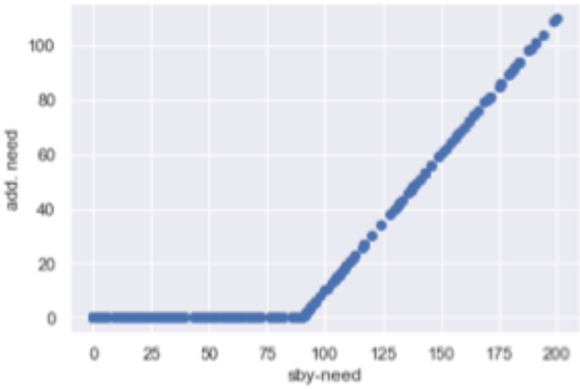


In [133…

```
# Number of standby drivers needed vs. number of additional drivers needed
plt.figure(figsize=(10,10))

fig, ax = plt.subplots()
ax.scatter(df['sby_need'], df['dafted'])
ax.set_xlabel('sby-need')
ax.set_ylabel('add. need')
plt.show()
```

```
<Figure size 720x720 with 0 Axes>
```



INTERNATIONAL UNIVERSITY OF APPLIED SCIENCES

B-2 Baseline Model

Baseline Model

- Calculate driver need mean by day, day of week, season
- Establish a simple linear regression model
- Validate performance of the models to establish a benchmark

In [47]:
```
import pandas as pd
import numpy as np
import seaborn as sn
import datetime
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import max_error

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn; seaborn.set()
df = pd.read_csv('./cleaned_data.csv', low_memory=False)
df = df.drop(['Unnamed: 0'], axis=1)
```

In [48]:
```
# Calculate mean
pd.set_option('mode.chained_assignment',None)
mean = np.round(df['sby_need'].mean(), 5)
df['bl_mean'] = mean
df.head()
```

Out[48]:

	date	n_sick	calls	n_duty	n_sby	sby_need	dafted	year	month	day	day_of_week	day_of_year
0	2016-04-01	73	8154.0	1700	90	4.0	0.0	2016	4	1	4	92
1	2016-04-02	64	8526.0	1700	90	70.0	0.0	2016	4	2	5	93
2	2016-04-03	68	8088.0	1700	90	0.0	0.0	2016	4	3	6	94
3	2016-04-04	71	7044.0	1700	90	0.0	0.0	2016	4	4	0	95
4	2016-04-05	63	7236.0	1700	90	0.0	0.0	2016	4	5	1	96

In [49]:
```
# Calculate mean by day (month)
x = np.ceil(df.groupby('day')['sby_need'].mean()).to_frame('bl_mean_day_of_month').reset
df = pd.merge(x, df, on='day')

# Calculate mean by day (week)
x = np.ceil(df.groupby('day_of_week')['sby_need'].mean()).to_frame('bl_mean_day_of_week'
df = pd.merge(x, df, on='day_of_week')

# Calculate mean by day (year)
```

```
x = np.ceil(df.groupby('day_of_year')['sby_need'].mean()).to_frame('bl_mean_day_of_year'
df = pd.merge(x, df, on='day_of_year')

df.head()
```

Out[49]:

	day_of_year	bl_mean_day_of_year	day_of_week	bl_mean_day_of_week	day	bl_mean_day_of_month	date
0	1	0.0	0	22.0	1	43.0	2018-01-01
1	1	0.0	1	26.0	1	43.0	2019-01-01
2	1	0.0	6	11.0	1	43.0	2017-01-01
3	2	39.0	0	22.0	2	27.0	2017-01-02
4	2	39.0	1	26.0	2	27.0	2018-01-02

In [50]:

```
# Define a function to calculate error metrics for baseline means
def metrics(y, y_hat, title='baseline mean', save_or_print='just_print', target_var='sby
    mse = np.round(mean_squared_error(y, y_hat), 5)
    rmse = np.round(np.sqrt(mean_squared_error(y, y_hat)), 5)
    mae = np.round(mean_absolute_error(y, y_hat), 5)
    max_r = np.round(max_error(y, y_hat), 5)

    print('=======')
    print(f'{title} (n={len(y)})')
    print('-------')
    print(f'| RMSE | {rmse} ')
    print(f'| MSE  | {mse}')
    print(f'| MAE  | {mae}')
    print(f'| Max  | {max_r}')
    print('\n')

    if save_or_print is not 'just_print':
        with open('baseline_model_error_metrics.csv', 'a+') as file:
            date = datetime.datetime.now()
            row = f'\n{title}, {rmse}, {mse}, {mae}, {max_r}, {len(y)}, {target_var}, {d
            file.write(row)

    return rmse, max_r
```

In [51]:

```
bl_mean_rmse, bl_mean_max = metrics(df['bl_mean'], df['sby_need'], 'Baseline Mean', 'sav
bl_mean_year_rmse, bl_mean_year_max = metrics(df['bl_mean_day_of_year'], df['sby_need'],
bl_mean_month_rmse, bl_mean_month_max = metrics(df['bl_mean_day_of_month'], df['sby_need
bl_mean_week_rmse, bl_mean_week_max = metrics(df['bl_mean_day_of_week'], df['sby_need'],
```

```
=======
Baseline Mean (n=1085)
-------
| RMSE | 43.5718
| MSE  | 1898.50176
| MAE  | 29.45592
| Max  | 181.43318


=======
Baseline Mean - day/year (n=1085)
-------
| RMSE | 33.73285
| MSE  | 1137.90507
| MAE  | 18.66636
```

```
| Max   | 141.0


=======
Baseline Mean - day/month (n=1085)
-------
| RMSE | 42.74962
| MSE  | 1827.52995
| MAE  | 28.7447
| Max  | 180.0


=======
Baseline Mean - day/week (n=1085)
-------
| RMSE | 43.35615
| MSE  | 1879.75576
| MAE  | 29.40369
| Max  | 182.0
```

Linear Regression Baseline Model

- Split data into test-train sets
- Fit data to model
- Evaluate model performance

In [52]:

```
# Split data into test/train sets
x1 = pd.get_dummies(df[['day_of_year', 'day', 'day_of_week', 'year', 'month', 'season']]
X1, X2, y1, y2 = train_test_split(x1, df['sby_need'], random_state=5, train_size=0.7)

# Fit data to model
from sklearn.linear_model import Ridge
ridge_model = Ridge(alpha=20)
ridge_model.fit(X1, y1)
y_hat = ridge_model.predict(X2)
print(ridge_model.score(X2, y2))

# Evaluate model performance
lin_reg_bl_rmse, lin_reg_bl_max = metrics(y2, y_hat, 'Linear Reg.', 'just_print')
```

```
0.07440888623263897
=======
Linear Reg. (n=326)
-------
| RMSE | 44.31846
| MSE  | 1964.1263
| MAE  | 28.22533
| Max  | 167.70059
```

Compare Benchmark Models

In [53]:

```
results_rmse = [lin_reg_bl_rmse, bl_mean_week_rmse, bl_mean_month_rmse, bl_mean_year_rms
results_max = [lin_reg_bl_max, bl_mean_week_max, \
               bl_mean_month_max, bl_mean_year_max, bl_mean_max]
print(f'Benchmark Results')
print('========')
```

```
print(f'LinReg: RMSE = {lin_reg_bl_rmse}, Max = {lin_reg_bl_max}')
print(f'BL-mean: RMSE {bl_mean_rmse}, Max = {bl_mean_max}')
print(f'BL-mean-week: RMSE = {bl_mean_week_rmse}, Max = {bl_mean_week_max}')
print(f'BL-mean-month: RMSE = {bl_mean_month_rmse}, Max = {bl_mean_month_max}')
print(f'BL-mean-year: RMSE = {bl_mean_year_rmse}, Max = {bl_mean_year_max}\n')

print(f'*Best RMSE score* : {min(results_rmse)}')
print(f'*Best Max Error score* : {min(results_max)}')
```

```
Benchmark Results
========
LinReg: RMSE = 44.31846, Max = 167.70059
BL-mean: RMSE 43.5718, Max = 181.43318
BL-mean-week: RMSE = 43.35615, Max = 182.0
BL-mean-month: RMSE = 42.74962, Max = 180.0
BL-mean-year: RMSE = 33.73285, Max = 141.0

*Best RMSE score* : 33.73285
*Best Max Error score* : 141.0
```

According to the initial results, the Baseline Mean - Days/Year model out performs all benchmarks.

B-3 Prediction Model

Prediction Model

- Split data into test/train sets
- Fit data to potential models: Support Vector Regression & Bernoulli Naive Bayes
- Evaluate performance of models using cross-validation
- Tune hyperparameters
- Evaluate model performance again
- Select best model for deployment

In [342…

```
import pandas as pd
import numpy as np
import seaborn as sn
import datetime
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import max_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn; seaborn.set()
df = pd.read_csv('./cleaned_data.csv', low_memory=False)
df = df.drop(['Unnamed: 0'], axis=1)
```

In [ ]:

```
# Split data into train/test sets
#
# Make feature matrix
x1 = pd.get_dummies(df[['day_of_year', 'day', 'day_of_week', 'year', 'month', 'season']]
# Merge sby_need with matrix
x1['sby_need'] = df['sby_need']
# Split into train/test
X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=0.8)

# Make validation sets
Xval, Xval2, Yval, Yval2 = train_test_split(x1, x1['sby_need'], train_size=0.5)
```

Model validation

- In order to verify optimal size of training set data, a function is used to iterate through a list of sizes, split the data using each size, and return the best average model score after two-fold cross-validation.

In [318…

```
# Support vector regression
from sklearn import svm

r2_best = 0
trainset_size = 0
```

```
for i in np.arange(0.5, 0.98, 0.02):

    # Split into train/test
    X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=i, random_state=42)

    svr_model = svm.SVR(gamma='auto').fit(X1, y1)
    y_hat_svr = svr_model.predict(X2)

    svr_model_val1 = svm.SVR(gamma='auto').fit(Xval, Yval)
    svr_model_val2 = svm.SVR(gamma='auto').fit(Xval2, Yval2)
    yhat_svrm1 = svr_model_val1.predict(Xval2)
    yhat_svrm2 = svr_model_val2.predict(Xval)

    r2 = svr_model.score(X2, y2)
    r2_test = svr_model.score(X1, y1)
    r2_val1 = svr_model_val1.score(Xval2, Yval2)
    r2_val2 = svr_model_val2.score(Xval, Yval)
    r2_mean = np.mean([r2, r2_test, r2_val1, r2_val2])

    if r2_mean > r2_best:
        r2_best = r2
        trainset_size = i

print(f'Best R^2              : {r2_best}')
print(f'Best Train-set pct. : {np.round(trainset_size, 2)}')

svr = {'best_r2': r2_best, 'best_train_size': np.round(trainset_size, 2)}
```

```
Best R^2            : 0.7837477537817732
Best Train-set pct. : 0.66
```

In [349…

```
# Fit data to model - bernoulli naive bayes
from sklearn.naive_bayes import BernoulliNB

r2_best = 0
trainset_size = 0

for i in np.arange(0.5, 0.98, 0.02):

    # Split into train/test
    X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=i, random_state=42)

    bnb_model = BernoulliNB(class_prior=None).fit(X1, y1)
    y_hat_bnb = bnb_model.predict(X2)

    bnb_val1 = BernoulliNB().fit(Xval, Yval)
    bnb_val2 = BernoulliNB().fit(Xval2, Yval2)
    y_hat_bnb_val1 = bnb_val1.predict(Xval2)
    y_hat_bnb_val2 = bnb_val2.predict(Xval)

    r2 = bnb_model.score(X2, y2)
    r2_test = bnb_model.score(X1, y1)
    r2_val1 = bnb_val1.score(Xval2, Yval2)
    r2_val2 = bnb_val2.score(Xval, Yval)
    r2_mean = np.mean([r2, r2_test, r2_val1, r2_val2])

    if r2_mean > r2_best:
        r2_best = r2_mean
        trainset_size = i

print(f'Best R^2              : {r2_best}')
print(f'Best Train-set pct. : {np.round(trainset_size, 2)}')

bnb = {'best_r2': r2_best, 'best_train_size': np.round(trainset_size, 2)}
```

```
Best R^2            : 0.7850634467396729
```

```
Best Train-set pct. : 0.72
```

Tune Hyperparameters

- Using brute force cross-validation to evaluate parameter performance, each model will have a list of the parameters used for each validation cycle along with the best score results.
- The best scores from each model tuning will be printed and automatically selected.

Bernoulli Naive Bayes model

In [368…

```
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import r2_score, mean_squared_error

# Split into train/test
X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=bnb['best_train_size'],

param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5],
    'fit_prior': [True, False],
    'binarize': [0.0, 0.5, 1.0]
}

scoring = {'R^2': 'r2', 'MSE': 'neg_mean_squared_error'}
search = RandomizedSearchCV(bnb_model, param_grid, cv=5, n_iter=10, scoring=scoring, ref

print("Best R^2 score:", search.best_score_)

best_r2_score = -float('inf')
best_mse_score = float('inf')
best_model = None

for mean_r2, mean_mse, params in zip(search.cv_results_['mean_test_R^2'], search.cv_resu
    if mean_r2 > best_r2_score:
        best_r2_score = mean_r2
        best_mse_score = mean_mse
        best_model = params
    elif mean_r2 == best_r2_score and mean_mse < best_mse_score:
        best_mse_score = mean_mse
        best_model = params

print("Best R^2 score:", best_r2_score)
print("Best MSE:", best_mse_score)
print("Best model:", best_model)
```

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/sklearn/mo
del_selection/_split.py:657: Warning: The least populated class in y has only 1 members,
which is too few. The minimum number of members in any class cannot be less than n_split
s=5.
  % (min_groups, self.n_splits)), Warning)
Best R^2 score: 0.035115492337631726
Best R^2 score: 0.035115492337631726
Best MSE: -2137.3649167733674
Best model: {'fit_prior': True, 'binarize': 0.5, 'alpha': 0.1}
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/sklearn/mo
del_selection/_search.py:813: DeprecationWarning: The default of the `iid` parameter wil
l change from True to False in version 0.22 and will be removed in 0.24. This will chang
e numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

The models performance is lower after tuning hyper-parameters. This suggests that:

1. The default parameters of Scikit-learn are more finely-tuned than what a novice data scientist could initially create.
2. The model is not experiencing overfitting, as the performance on initial model is not overly higher than the fine-tuned version.
3. The randomization of training data could be what is causing model perfromance to fluctuate

Support Vector Regression model

In [341…

```
# Split into train/test
X1, X2, y1, y2 = train_test_split(x1, x1['sby_need'], train_size=svr['best_train_size'],

param_grid = {
    'C': [0.1, 1, 10],
    'epsilon': [0.01, 0.1, 1],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto'],
    'shrinking': [True],
}
scoring = {'R^2': 'r2', 'MSE': 'neg_mean_squared_error'}
search = GridSearchCV(svr_model, param_grid, scoring=scoring, refit='R^2', cv=5).fit(X1,

best_svr = search.best_estimator_

y_pred = best_svr.predict(X2)
r2 = r2_score(y2, y_pred)
mse = mean_squared_error(y2, y_pred)

print("Best R^2 score:", r2)
print("Best MSE:", mse)
print("Best parameters:", search.best_params_)

def select_best_model(models_dict):
    best_r2_score = -float('inf')
    best_mse_score = float('inf')
    best_model_params = None

    for (r2_score, mse_score), model_params in models_dict.items():
        if mse_score < best_mse_score or (mse_score == best_mse_score and r2_score > bes
            best_r2_score = r2_score
            best_mse_score = mse_score
            best_model_params = model_params

    return best_model_params, best_r2_score, best_mse_score


score_models_dict = {}

# Iterate over all models in grid search results
for mean_r2, mean_mse, params in zip(search.cv_results_['mean_test_R^2'], search.cv_resu
    score_models_dict[(mean_r2, -mean_mse)] = params

print("\nDictionary of R^2 scores, MSE scores, and corresponding models:")
for (r2_score, mse_score), model_params in score_models_dict.items():
    print("R^2 score:", r2_score)
    print("MSE score:", mse_score)
    print("Model parameters:", model_params)
    print("--------------------")

best_params, best_r2, best_mse = select_best_model(score_models_dict)

print("\nBest model parameters:", best_params)
print("Best R^2 score:", best_r2)
print("Best MSE score:", best_mse)
```

```
Best R^2 score: 0.9999999535634789
Best MSE: 7.872045324837176e-05
Best parameters: {'C': 0.1, 'epsilon': 0.01, 'gamma': 'scale', 'kernel': 'linear', 'shri
nking': True}
Dictionary of R^2 scores, MSE scores, and corresponding models:
R^2 score: 0.9999999598546738
MSE score: 7.743527524104955e-05
Model parameters: {'C': 10, 'epsilon': 0.01, 'gamma': 'auto', 'kernel': 'linear', 'shrin
king': True}
--------------------
R^2 score: 0.04571005143855864
MSE score: 1911.7009623166643
Model parameters: {'C': 0.1, 'epsilon': 0.01, 'gamma': 'scale', 'kernel': 'rbf', 'shrink
ing': True}
--------------------
R^2 score: 0.04800308663814628
MSE score: 1907.6581351291327
Model parameters: {'C': 0.1, 'epsilon': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'shrinki
ng': True}
--------------------
R^2 score: 0.9999956869757741
MSE score: 0.008303550524077632
Model parameters: {'C': 10, 'epsilon': 0.1, 'gamma': 'auto', 'kernel': 'linear', 'shrink
ing': True}
--------------------
R^2 score: 0.047258474712806586
MSE score: 1908.6311988305708
Model parameters: {'C': 0.1, 'epsilon': 0.1, 'gamma': 'scale', 'kernel': 'rbf', 'shrinki
ng': True}
--------------------
R^2 score: 0.04952557765021202
MSE score: 1904.6362925620194
Model parameters: {'C': 0.1, 'epsilon': 0.1, 'gamma': 'auto', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.9995617151084139
MSE score: 0.8441344048017452
Model parameters: {'C': 10, 'epsilon': 1, 'gamma': 'auto', 'kernel': 'linear', 'shrinkin
g': True}
--------------------
R^2 score: 0.062269658421828265
MSE score: 1878.8455090174475
Model parameters: {'C': 0.1, 'epsilon': 1, 'gamma': 'scale', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.06427348825334182
MSE score: 1875.3385613974108
Model parameters: {'C': 0.1, 'epsilon': 1, 'gamma': 'auto', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.8372435978142292
MSE score: 333.686435817931
Model parameters: {'C': 1, 'epsilon': 0.01, 'gamma': 'scale', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.718968038316113
MSE score: 571.2616824252777
Model parameters: {'C': 1, 'epsilon': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.8374365928527591
MSE score: 333.3063220493743
Model parameters: {'C': 1, 'epsilon': 0.1, 'gamma': 'scale', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
```

```
R^2 score: 0.7188120239564485
MSE score: 571.6731313808535
Model parameters: {'C': 1, 'epsilon': 0.1, 'gamma': 'auto', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.8391010621291095
MSE score: 329.84998274868286
Model parameters: {'C': 1, 'epsilon': 1, 'gamma': 'scale', 'kernel': 'rbf', 'shrinking':
True}
--------------------
R^2 score: 0.7199407258196721
MSE score: 569.2082298309839
Model parameters: {'C': 1, 'epsilon': 1, 'gamma': 'auto', 'kernel': 'rbf', 'shrinking':
True}
--------------------
R^2 score: 0.9988863324366494
MSE score: 2.3761025137285294
Model parameters: {'C': 10, 'epsilon': 0.01, 'gamma': 'scale', 'kernel': 'rbf', 'shrinki
ng': True}
--------------------
R^2 score: 0.9975393902709314
MSE score: 5.284376039466548
Model parameters: {'C': 10, 'epsilon': 0.01, 'gamma': 'auto', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.9988792045127851
MSE score: 2.3874817570292985
Model parameters: {'C': 10, 'epsilon': 0.1, 'gamma': 'scale', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.9975091672560007
MSE score: 5.3462195250408735
Model parameters: {'C': 10, 'epsilon': 0.1, 'gamma': 'auto', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.9984749449128648
MSE score: 3.1859373982539414
Model parameters: {'C': 10, 'epsilon': 1, 'gamma': 'scale', 'kernel': 'rbf', 'shrinkin
g': True}
--------------------
R^2 score: 0.996825652151134
MSE score: 6.699423817466172
Model parameters: {'C': 10, 'epsilon': 1, 'gamma': 'auto', 'kernel': 'rbf', 'shrinking':
True}
--------------------

Best model parameters: {'C': 10, 'epsilon': 0.01, 'gamma': 'auto', 'kernel': 'linear',
'shrinking': True}
Best R^2 score: 0.9999999598546738
Best MSE score: 7.743527524104955e-05
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/sklearn/mo
del_selection/_search.py:813: DeprecationWarning: The default of the `iid` parameter wil
l change from True to False in version 0.22 and will be removed in 0.24. This will chang
e numeric results when test-set sizes are unequal.
  DeprecationWarning)
```