Predicting Road Traffic in the UK

Handle time series data to predict a continuous outcome in Python

Kieu Van Dang Email: dang.v@northeastern.edu

LinkedIn: https://www.linkedin.com/in/kirudang/ Phone: +1 647 782 4558

Data science enthusiast with a great passion for data pre-processing and prediction with a strong background in business. Relevant skills include machine learning, statistics, problem-solving, programming (including SQL, Python, and R), and critical & creative thinking.

INDEX

I. INTRODUCTION

1. Time-Series Analysis

A very significant and effective research in data science, data analytics, and artificial intelligence is time series analysis and forecasting. Based on data that has been stored with consideration for changing time, it aids in analysis, forecasting, or computing the likelihood of an incidence in the future. ARIMA, smooth-based models, and moving averages are typical types of time series analysis. Time series are also frequently used in healthcare analytics, geographical analysis, weather forecasting, and to predict the future of continuously changing data.

2. Project Objectives

By forecasting traffic on numerous UK roads, this project aims to help users gain an intuitive understanding of various time series analysis techniques. This includes data cleaning to ensure the highest quality, the use of multiple time series models to forecast traffic for a given period of time, and the evaluation of each model's performance. The author will adjust some parameters or enhance each model using a seasonal ARIMA model to produce the best possible results.

3. Dataset Introduction

UK Road Traffic dataset is a simple time series dataset, which provides fundamental information about traffic for each road with according timestamps. Since variables are transparent, the author will delve into data preparation in the next step without listing all definitions here.

II. ANALYSIS AND RESULTS

1. Data Validation and Cleaning

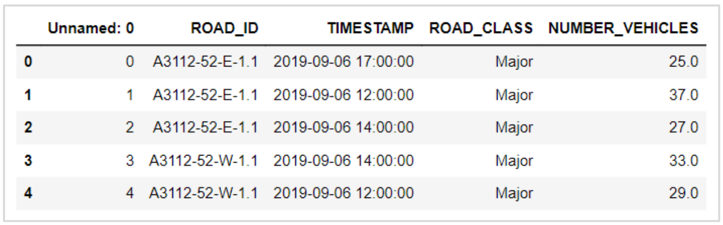
Data validation and preprocessing are crucial steps to validate the set to make sure it is sufficiently clean before conducting any analysis including checking on the dataset purpose, its structure, variables and fields within the set, data types, duplication, outliers, and missing values. Therefore, the author executes several fundamental steps of data validation in this phase before moving further.

a. Data Overview

Overview: First, the author loads the dataset in a data frame to check the overall view, as displayed in **Figure 1** below. Initially, the data look good with all major variables as described.

Figure 1

Overview of the data set

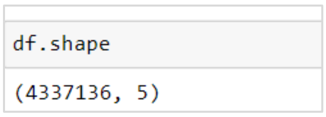


	Unnamed: 0	ROAD_ID	TIMESTAMP	ROAD_CLASS	NUMBER_VEHICLES
0	0	A3112-52-E-1.1	2019-09-06 17:00:00	Major	25.0
1	1	A3112-52-E-1.1	2019-09-06 12:00:00	Major	37.0
2	2	A3112-52-E-1.1	2019-09-06 14:00:00	Major	27.0
3	3	A3112-52-W-1.1	2019-09-06 14:00:00	Major	33.0
4	4	A3112-52-W-1.1	2019-09-06 12:00:00	Major	29.0

Dimension: The set also contains more than 4 million observations spanning 5 variables, as shown in **Figure 2** below. This data is considered relatively large and suitable for time series analytics.

Figure 2

Dataset dimension

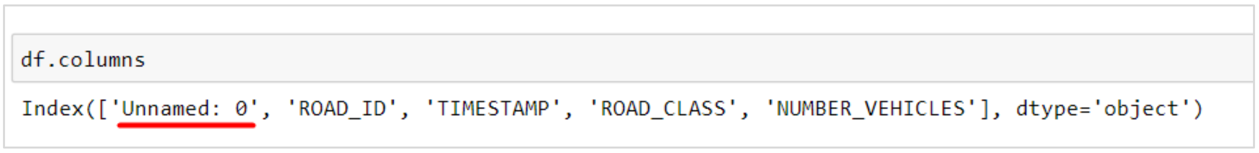


```
df.shape
(4337136, 5)
```

Features: There is one variable of **'Unnamed: 0'**, which shows the index of the set, shown in **Figure 3**. The author will remove this as it is redundant.

Figure 3

Listing of all variables



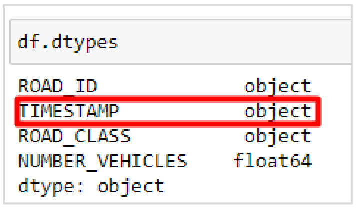
```
df.columns
Index(['Unnamed: 0', 'ROAD_ID', 'TIMESTAMP', 'ROAD_CLASS', 'NUMBER_VEHICLES'], dtype='object')
```

b. Data Type and Data Structure

Data type: As per **Figure 4**, we could see the details of the set, in which almost all data types are in the correct format. However, the **TIMESTAMP** should be in date-time format instead of categorical data, for that reason, the author converts it into time-date format.

Figure 4

Data type

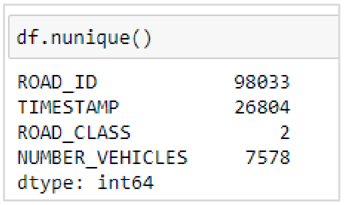


```
df.dtypes
ROAD_ID              object
TIMESTAMP            object
ROAD_CLASS           object
NUMBER_VEHICLES     float64
dtype: object
```

Count of unique values: Next, the author also checks the unique count of each variable to see if we should keep or remove them from the set. In **Figure 5**, everything looks fine, but the number of roads in the set appear a bit too many.

Figure 5

Unique value count

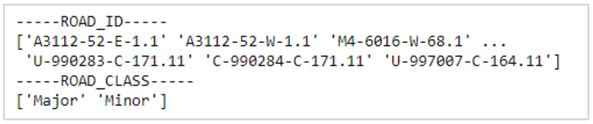


```
df.nunique()

ROAD_ID            98033
TIMESTAMP          26804
ROAD_CLASS             2
NUMBER_VEHICLES     7578
dtype: int64
```

Unique values: for categorical variables, it is a must to check on the number of classes within each feature. We have 2 categorical variables left in the set. However, there is nothing to modify in this case, except for the **ROAD_ID** with many classes inside.

Figure 6

Check on unique values of categorical variables

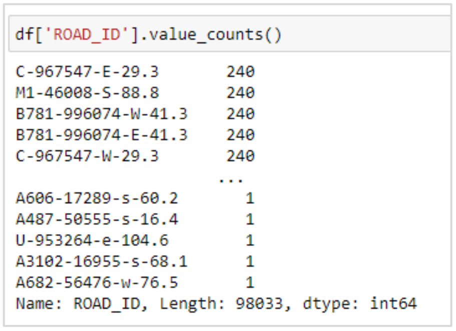


```
-----ROAD_ID-----
['A3112-52-E-1.1' 'A3112-52-W-1.1' 'M4-6016-W-68.1' ...
 'U-990283-C-171.11' 'C-990284-C-171.11' 'U-997007-C-164.11']
-----ROAD_CLASS-----
['Major' 'Minor']
```

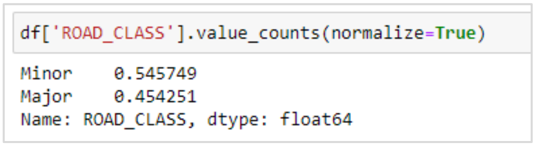
Detailed check: For further investigation, the author checks on the distribution of **ROAD_ID** and **ROAD_CLASS**. As we can observe from **Figure 7**, while ROAD_ID has many classes, and there are several of which have only 1 timestamp, ROAD_CLASS has only 2 elements major (54.6%) and minor (45.4%).

Figure 7

Check on the distribution of City ROAD_ID and ROAD_CLASS



```
df['ROAD_ID'].value_counts()

C-967547-E-29.3       240
M1-46008-S-88.8       240
B781-996074-W-41.3    240
B781-996074-E-41.3    240
C-967547-W-29.3       240
                     ...
A606-17289-s-60.2       1
A487-50555-s-16.4       1
U-953264-e-104.6        1
A3102-16955-s-68.1      1
A682-56476-w-76.5       1
Name: ROAD_ID, Length: 98033, dtype: int64
```



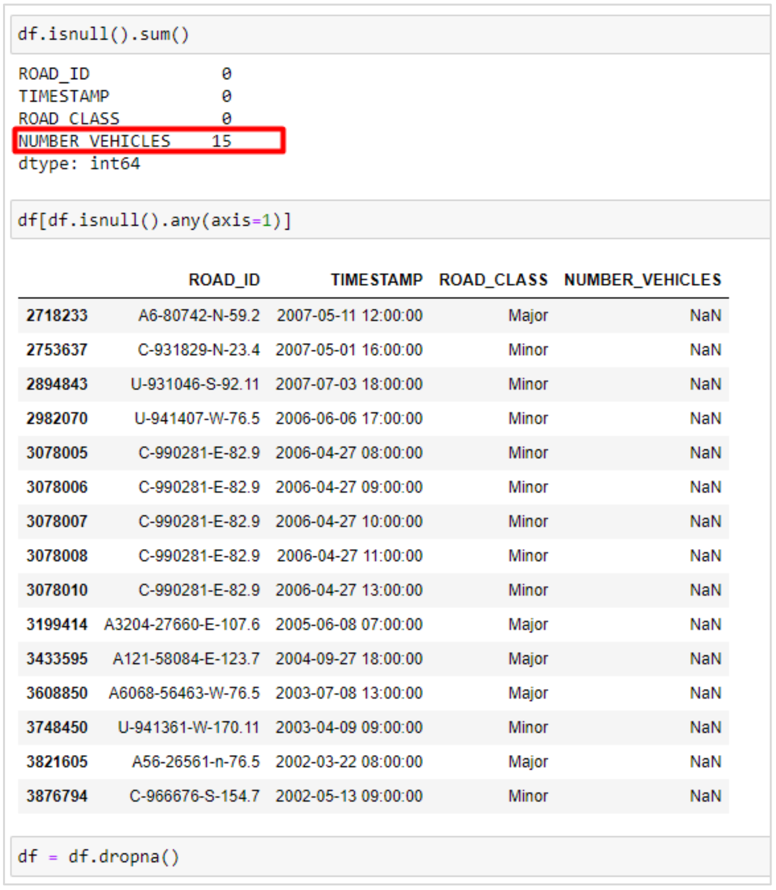
```
df['ROAD_CLASS'].value_counts(normalize=True)

Minor    0.545749
Major    0.454251
Name: ROAD_CLASS, dtype: float64
```

c. Missing Values

It is noticeable that there are 15 NA observations for **NUMBER_VEHICLES** as shown in **Figure 08** below. As the number of missing data is small, the author removes these observations from the set.

Figure 08

Missing value summary and drop



```
df.isnull().sum()
```

```
ROAD_ID             0
TIMESTAMP           0
ROAD_CLASS          0
NUMBER_VEHICLES    15
dtype: int64
```

```
df[df.isnull().any(axis=1)]
```

	ROAD_ID	TIMESTAMP	ROAD_CLASS	NUMBER_VEHICLES
2718233	A6-80742-N-59.2	2007-05-11 12:00:00	Major	NaN
2753637	C-931829-N-23.4	2007-05-01 16:00:00	Minor	NaN
2894843	U-931046-S-92.11	2007-07-03 18:00:00	Minor	NaN
2982070	U-941407-W-76.5	2006-06-06 17:00:00	Minor	NaN
3078005	C-990281-E-82.9	2006-04-27 08:00:00	Minor	NaN
3078006	C-990281-E-82.9	2006-04-27 09:00:00	Minor	NaN
3078007	C-990281-E-82.9	2006-04-27 10:00:00	Minor	NaN
3078008	C-990281-E-82.9	2006-04-27 11:00:00	Minor	NaN
3078010	C-990281-E-82.9	2006-04-27 13:00:00	Minor	NaN
3199414	A3204-27660-E-107.6	2005-06-08 07:00:00	Major	NaN
3433595	A121-58084-E-123.7	2004-09-27 18:00:00	Major	NaN
3608850	A6068-56463-W-76.5	2003-07-08 13:00:00	Major	NaN
3748450	U-941361-W-170.11	2003-04-09 09:00:00	Minor	NaN
3821605	A56-26561-n-76.5	2002-03-22 08:00:00	Major	NaN
3876794	C-966676-S-154.7	2002-05-13 09:00:00	Minor	NaN

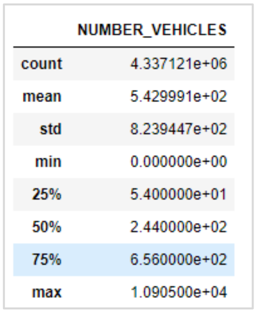
```
df = df.dropna()
```

d. Outliers / Anomalies

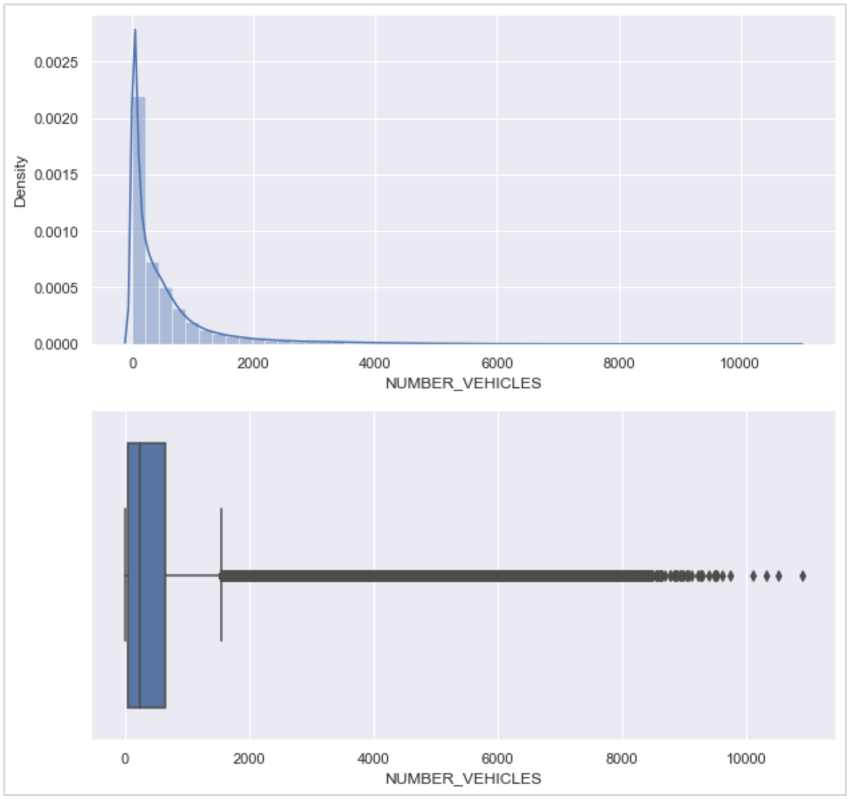
To check for outliers, the author draws a descriptive table for all numerical data to observe the range, and the minimum and maximum values for each variable. Referring to **Figure 09**, it seems that **NUMBER_VEHICLES** is highly skewed, but we should just keep all observations.

Figure 09

Descriptive summary and distribution of NUMBER_VEHICLES



	NUMBER_VEHICLES
count	4.337121e+06
mean	5.429991e+02
std	8.239447e+02
min	0.000000e+00
25%	5.400000e+01
50%	2.440000e+02
75%	6.560000e+02
max	1.090500e+04

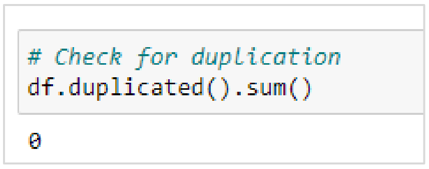


e. Duplications

Fortunately, there are no duplicated observations in our set; therefore, all data are kept the same.

Figure 10

Detect and remove duplications



```
# Check for duplication
df.duplicated().sum()
```

```
0
```

Note: Up to this step, we have clean data to work further.

2. Explanatory Descriptive Analysis

a. How to predict when there are too many roads?

When it comes to predicting traffic for UK roads, it's important to consider the scope and granularity of the prediction task. If you have too many roads to predict traffic for, it may not be practical to build a single model that can accurately predict traffic for all roads.

One approach is to build separate models for different groups of roads based on certain characteristics. For example, we could group roads by their location, traffic volume, or type of road (e.g. major or minor in this case). This can help you us more targeted and accurate models for specific subsets of roads, rather than trying to create a one-size-fits-all model.

Another approach is to use a hierarchical model that can capture the relationships between different levels of geographic regions, such as country, region, city, and street. This can allow us to make predictions at different levels of granularity, depending on your specific needs.

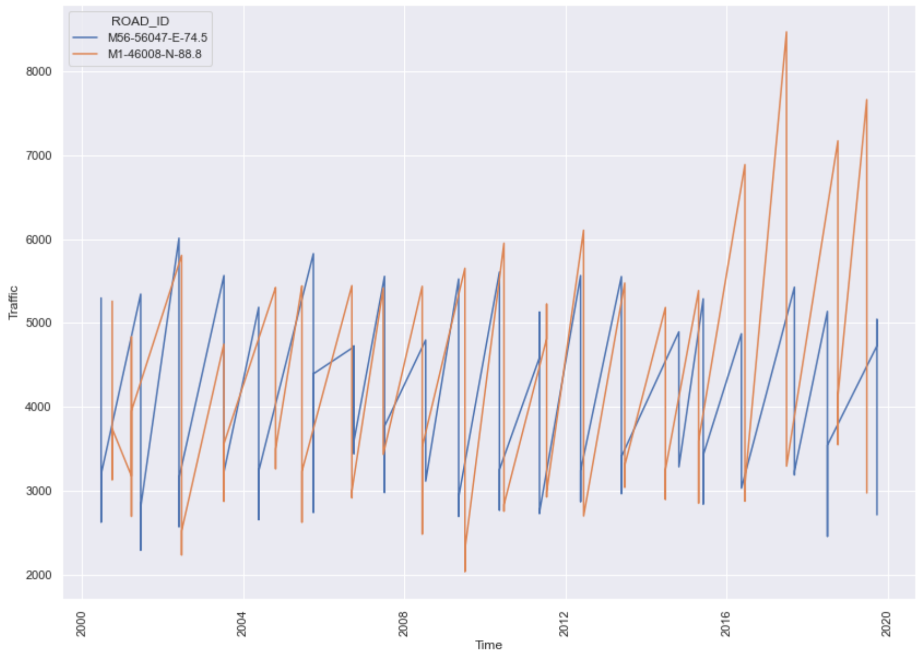
b. Prediction for Specific Road

As analyzed earlier, it would be the most meaningful to predict traffic for a specific road, especially for roads with more observations through time data.

Figure 11 below illustrates traffic for 2 specific roads over time.

Figure 11

The traffic of 2 roads



However, there are several roads where building a time series model will not work. They are:

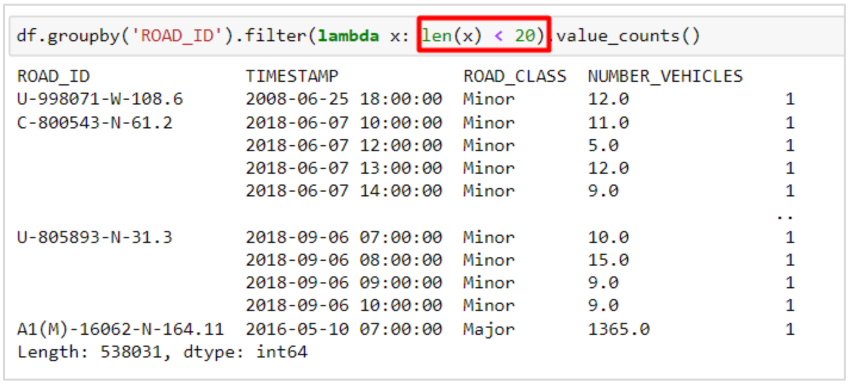
- Road with too few observations (let's say less than 20).
- Road with unevenly spaced time intervals.

To display this, please consult the following visualization.

Road with too few observations: We cannot run models to predict future events when there are insufficient data to train models. In **Figures 12** and **13**, there are many roads like this.

Figure 12

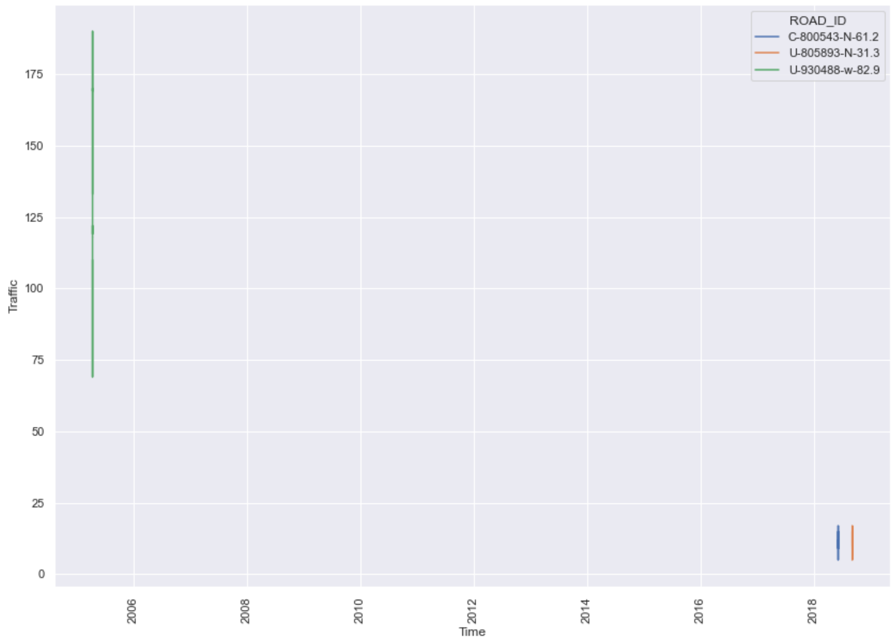
Insufficient data road



```
df.groupby('ROAD_ID').filter(lambda x: len(x) < 20).value_counts()

ROAD_ID              TIMESTAMP            ROAD_CLASS  NUMBER_VEHICLES
U-998071-W-108.6     2008-06-25 18:00:00  Minor       12.0              1
C-800543-N-61.2      2018-06-07 10:00:00  Minor       11.0              1
                     2018-06-07 12:00:00  Minor       5.0               1
                     2018-06-07 13:00:00  Minor       12.0              1
                     2018-06-07 14:00:00  Minor       9.0               1
                                                                       ..
U-805893-N-31.3      2018-09-06 07:00:00  Minor       10.0              1
                     2018-09-06 08:00:00  Minor       15.0              1
                     2018-09-06 09:00:00  Minor       9.0               1
                     2018-09-06 10:00:00  Minor       9.0               1
A1(M)-16062-N-164.11 2016-05-10 07:00:00  Major       1365.0            1
Length: 538031, dtype: int64
```

Figure 13

Visualization for insufficient data road

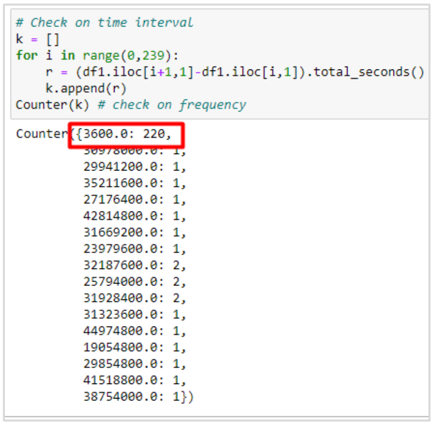


As we can see from the plot above, these 3 roads have very few data, which is not enough to train for a valid predictive model.

Road with unevenly spaced time intervals: One fundamental condition for time series analysis is that all time intervals should be equally spaced. In this case, the author conducts a test on a specific road id with a maximum number of observations; conversely, they are not spaced evenly. Referring to **Figure 14** below, we can see that only 220 out of 240 observations are 1 hour (3600 seconds) difference, while the rest of the 20 timestamps are gapped by a distinct distance.

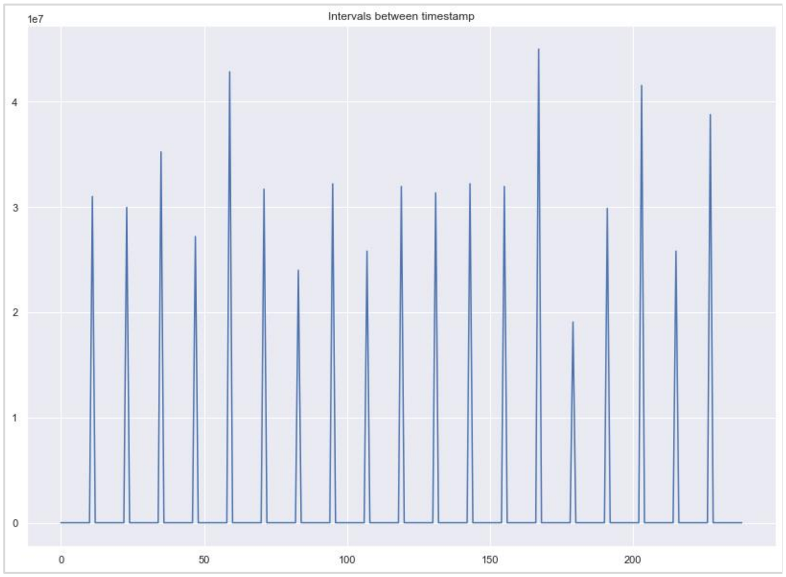
Figure 14

Unevenly spaced timestamps



```
# Check on time interval
k = []
for i in range(0,239):
    r = (df1.iloc[i+1,1]-df1.iloc[i,1]).total_seconds()
    k.append(r)
Counter(k) # check on frequency
```

```
Counter({3600.0: 220,
         30978000.0: 1,
         29941200.0: 1,
         35211600.0: 1,
         27176400.0: 1,
         42814800.0: 1,
         31669200.0: 1,
         23979600.0: 1,
         32187600.0: 2,
         25794000.0: 2,
         31928400.0: 2,
         31323600.0: 1,
         44974800.0: 1,
         19054800.0: 1,
         29854800.0: 1,
         41518800.0: 1,
         38754000.0: 1})
```



3. Data Transformation for Evenly Spaced Time-Series

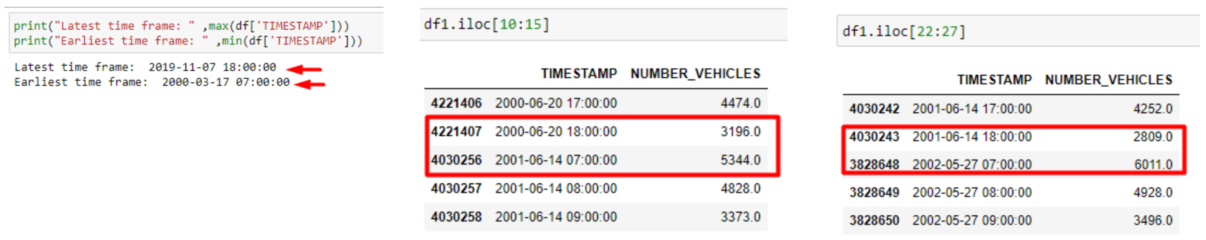
The author has checked all roads to see if any roads satisfy the condition of the equal interval, but unfortunately, there are none of them. Hence, the author must transform data to ensure the quality of modeling. From this step, the author will pick the 'ROAD_ID' of **M56-56047-E-74.5** and work with this specific road.

Our first timestamp in the set is 2000-03-17 07:00:00, and the last is 2019-11-07 18:00:00. From data, we also could see that for a specific day of record, data are observed from 07:00 to 18:00 with 1-hour distance. However, the days are not consecutive, but jump in an unorganized manner, as shown in **Figure 15**. In the second plot, time jumps from 2000-06-20 18:00:00 to 2001-

06-14 07:00:00, while in the third plot below, time jumps from 2001-06-14 18:00:00 to 2002-05-27 07:00:00.

Figure 15

Timestamps analysis



```
print("Latest time frame: " ,max(df['TIMESTAMP']))
print("Earliest time frame: " ,min(df['TIMESTAMP']))
```

```
Latest time frame:  2019-11-07 18:00:00
Earliest time frame:  2000-03-17 07:00:00
```

```
df1.iloc[10:15]
```

	TIMESTAMP	NUMBER_VEHICLES
4221406	2000-06-20 17:00:00	4474.0
4221407	2000-06-20 18:00:00	3196.0
4030256	2001-06-14 07:00:00	5344.0
4030257	2001-06-14 08:00:00	4828.0
4030258	2001-06-14 09:00:00	3373.0

```
df1.iloc[22:27]
```

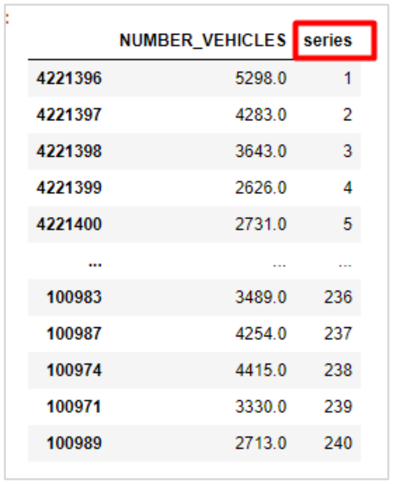
	TIMESTAMP	NUMBER_VEHICLES
4030242	2001-06-14 17:00:00	4252.0
4030243	2001-06-14 18:00:00	2809.0
3828648	2002-05-27 07:00:00	6011.0
3828649	2002-05-27 08:00:00	4928.0
3828650	2002-05-27 09:00:00	3496.0

How to deal with unevenly spaced time series: There are multiple ways to handle interrupted time series. One common method is to impute them. But since the interruptions between time data are too big, for example, 44974800-second jump, which is 12,943 times higher than a common gap of 1-hour jump. Therefore, creating a time series from early to the end with imputed data would destroy the whole model.

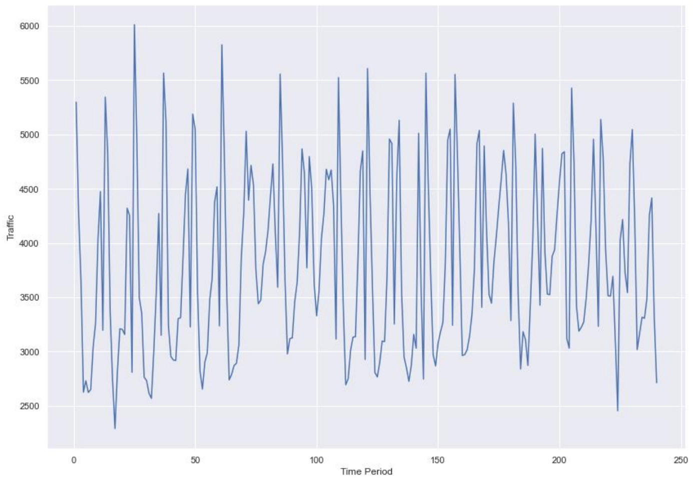
In this case, since our data do not have consecutive days but they have a cycle of 12 hours every day from 07:00 to 18:00, therefore, the author will create a series of numbers from 1 to 240 (a total of 20 cycles of observations) to replace for timestamps. **Figure 16** displays my code and time data in a series of numbers. Up to this step, we have our final data for time series prediction.

Figure 16

Unevenly spaced timestamps



	NUMBER_VEHICLES	series
4221396	5298.0	1
4221397	4283.0	2
4221398	3643.0	3
4221399	2626.0	4
4221400	2731.0	5
...	...	...
100983	3489.0	236
100987	4254.0	237
100974	4415.0	238
100971	3330.0	239
100989	2713.0	240



4. Simple Moving Average

a. Note on Test Data

Since the author already modifies the timestamps to a number of periods and since we need to evaluate model performance, the author will split data into two parts: training set on the first 18 days (216 observations) and test data on last 2 days (24 observations).

b. Model Performance

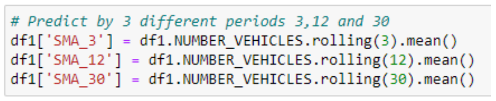
Rolling windows are provided over the data by the simple moving average method. We can run calculations using a statistical function on the generated windows (in this case the mean). The window option specifies the number of periods and *pandas.Series.rolling()* is one of the simplest techniques to calculate this method.

The author runs 3 simple moving averages (SMA) with rolling windows of 3, 12, and 30 to check for their performance.

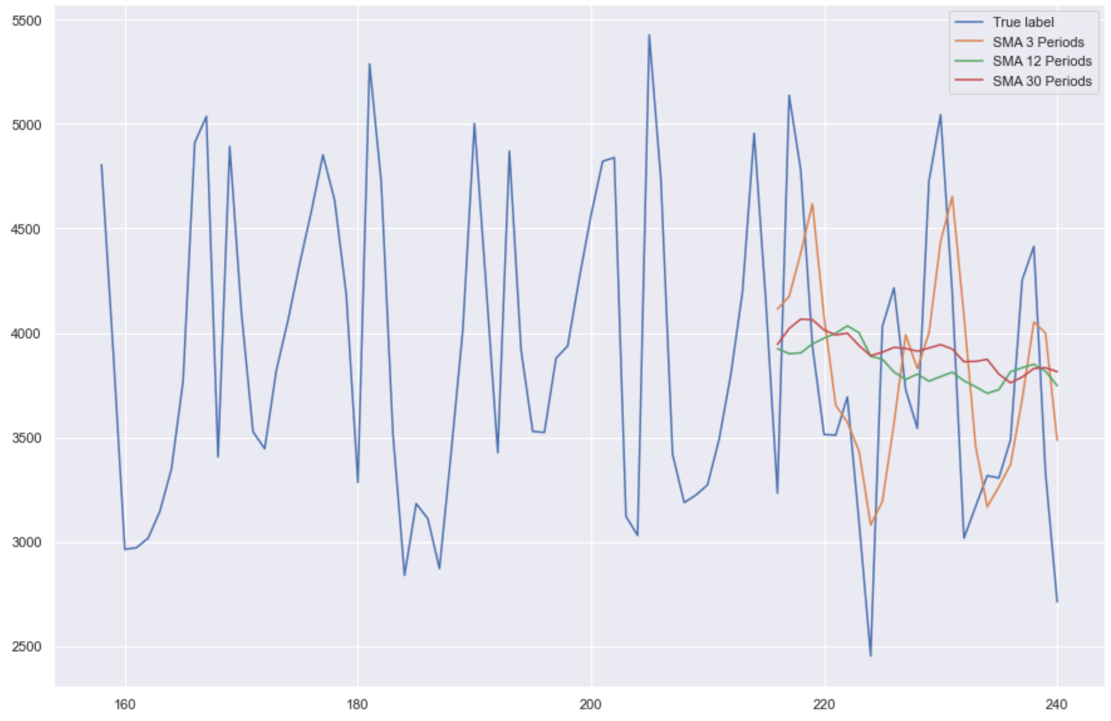
It can be seen from **Figure 17** that the SMA of 3 periods performs well, following the trend of actual data with a slight lag behind. While for SMA of 12 and 30 periods, the predictions seem flattened and smooth because they are taken by the average of many past data. In this case, the more rolling windows of the model, the smoother the line is.

Figure 17

SMA and visualization of model performance



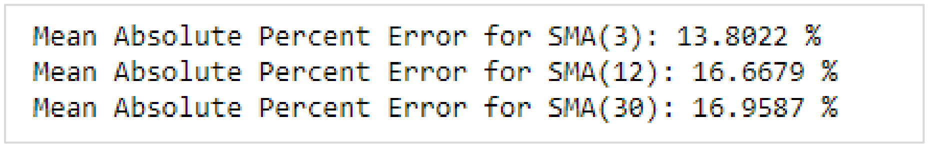
```
# Predict by 3 different periods 3,12 and 30
df1['SMA_3'] = df1.NUMBER_VEHICLES.rolling(3).mean()
df1['SMA_12'] = df1.NUMBER_VEHICLES.rolling(12).mean()
df1['SMA_30'] = df1.NUMBER_VEHICLES.rolling(30).mean()
```



To quantify the model performance, the author employs Mean Absolute Percent Error (MAPE) to confirm in **Figure 18**. The results enhance our initial evaluation when MAPE of 3-period SMA is the lowest with 13.8% only. The figures for 12-period SMA and 30-period SMA are 16.66% and 16.95% consecutively.

Figure 18

MAPE of 3 models



```
Mean Absolute Percent Error for SMA(3): 13.8022 %
Mean Absolute Percent Error for SMA(12): 16.6679 %
Mean Absolute Percent Error for SMA(30): 16.9587 %
```

Conclusion: These results are expected for SMA. For short-term SMA like 3 periods: The likelihood of seeing a change in trend at a very early stage increases with the shorter the SMA term, although it is vulnerable to false signals or an abrupt change of data. Whereas, medium or long-term SMAs (12 and 30 periods in this case) will smoothen the predictions, which results in missing out on the change in trend, but at the same time, they are more stable in an environment where significant changes are common.

5. Auto-Regressive Integrated Moving Average - ARIMA

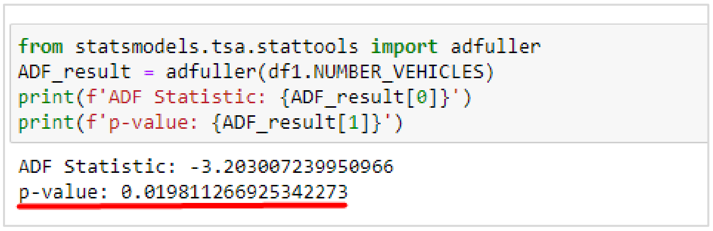
a. Check Stationarity

The first step is to test for the stationarity of data. In this work, the author employs both visualization and hypothesis testing to evaluate this.

In **Figure 17** above, we can see that our data are stationary as they have constant mean and variance over time. Furthermore, in the Dickey-Fuller Test – ADF (in **Figure 19**) with the null hypothesis that our time series data are not stationary, we have a p-value of 0.0198; thus, we can reject this null hypothesis and conclude that our data are already stationary.

Figure 19

ADF test for stationarity



```
from statsmodels.tsa.stattools import adfuller
ADF_result = adfuller(df1.NUMBER_VEHICLES)
print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

```
ADF Statistic: -3.203007239950966
p-value: 0.019811266925342273
```

Since we have this property for our data, we do not need to transform data, hence our differencing level is equal to 0 (**d = 0**) for all models later.

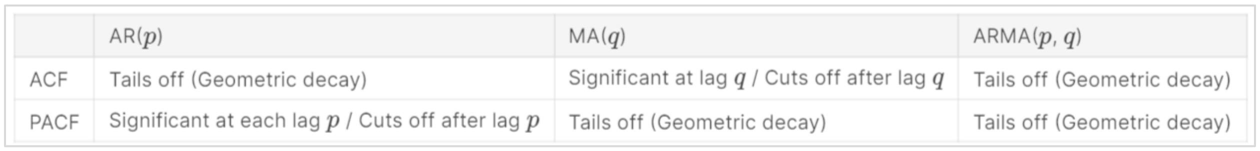
b. Order (p and q) Identifying

Let's now examine the PACF and ACF to determine whether we can infer the sequence of one of the processes at work.

The method to determine the order for ARMA is shown in the table below:

Table 1

Method to determine the order for time series

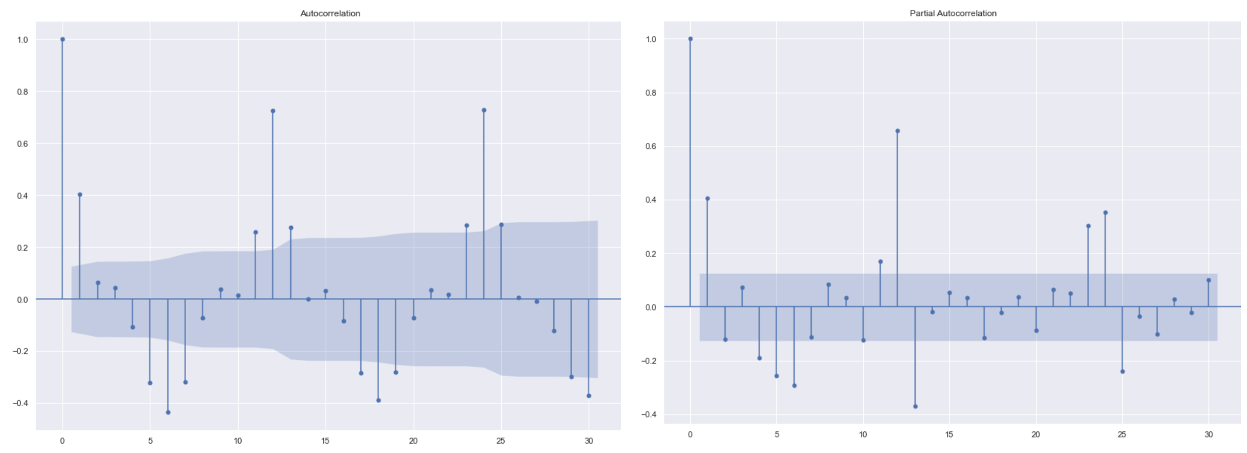


	AR(p)	MA(q)	ARMA(p, q)
ACF	Tails off (Geometric decay)	Significant at lag q / Cuts off after lag q	Tails off (Geometric decay)
PACF	Significant at each lag p / Cuts off after lag p	Tails off (Geometric decay)	Tails off (Geometric decay)

Now, let's take a look at our Autocorrelation (ACF) and Partial Autocorrelation (PACF) plots to pick the appropriate values for p and q.

Figure 20

ACF and PACF plot



Referring to **Figure 20**, both ACF and PACF are not informative as they are in sinusoidal and seasonal shape. Details are as follows:

AR:

- AFC has not tailed off (slowly decreased) since it peaks at 1 and again at 6 and 12.
- PAFC has no significant cut-off, it has several cut-off values: 1, 6, 12, and 13.

MA:

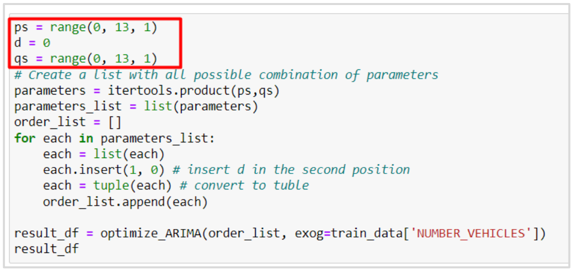
- AFC has no significant cut-off, it has several cut-off values: 1, 6, 12, and 13.
- PAFC has not tailed off (slowly decreased) since it peaks again at 6 and 12.

To solve this issue, we must test several order combinations, fit an ARIMA model using those orders, then choose orders using Akaike's Information Criterion, or AIC.

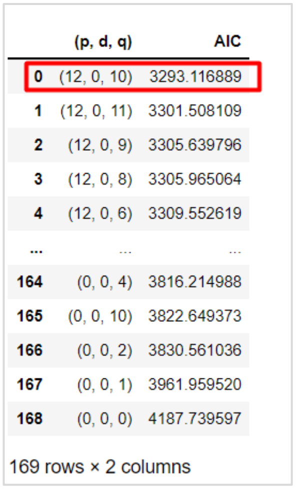
We already have d = 0, so in this case, the author creates 2 same ranges for p and q from 0 to 13 and fits the model by different combinations by a manual function. Codes and results are detailed in the markdown file. Please consult the result in **Figure 21**, in which in a total of 169 combinations of orders, the order (12,0,10) for p,d,q yields the lowest AIC. Thus, the author would pick this order to run on our model.

Figure 21

The best combination of order



```
ps = range(0, 13, 1)
d = 0
qs = range(0, 13, 1)
# Create a list with all possible combination of parameters
parameters = itertools.product(ps,qs)
parameters_list = list(parameters)
order_list = []
for each in parameters_list:
    each = list(each)
    each.insert(1, 0) # insert d in the second position
    each = tuple(each) # convert to tuble
    order_list.append(each)

result_df = optimize_ARIMA(order_list, exog=train_data['NUMBER_VEHICLES'])
result_df
```



	(p, d, q)	AIC
0	(12, 0, 10)	3293.116889
1	(12, 0, 11)	3301.508109
2	(12, 0, 9)	3305.639796
3	(12, 0, 8)	3305.965064
4	(12, 0, 6)	3309.552619
...	...	...
164	(0, 0, 4)	3816.214988
165	(0, 0, 10)	3822.649373
166	(0, 0, 2)	3830.561036
167	(0, 0, 1)	3961.959520
168	(0, 0, 0)	4187.739597

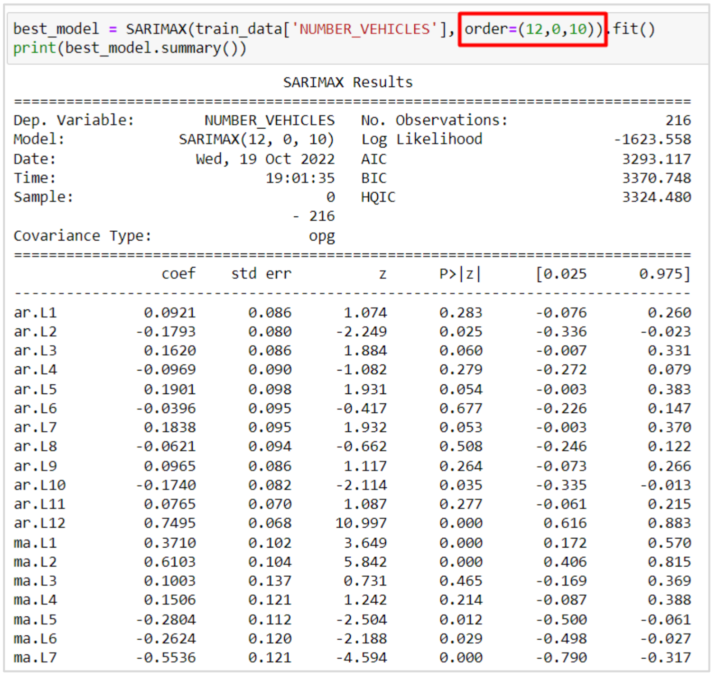
169 rows × 2 columns

c. *Model Fit and Performance Evaluation*

Fit the ARIMA model with the defined order, we have the result as below:

Figure 22

ARIMA summary



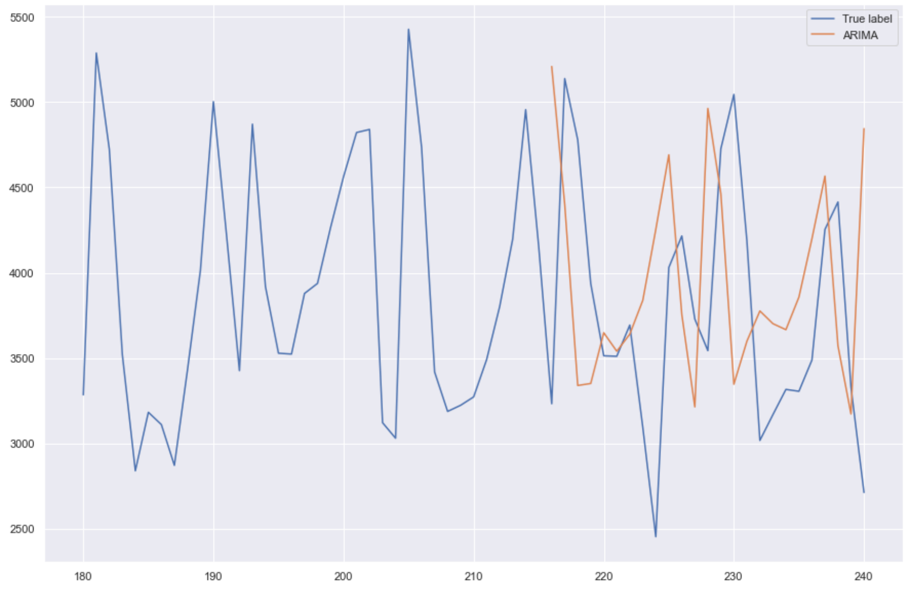
```
best_model = SARIMAX(train_data['NUMBER_VEHICLES'], order=(12,0,10)).fit()
print(best_model.summary())
```

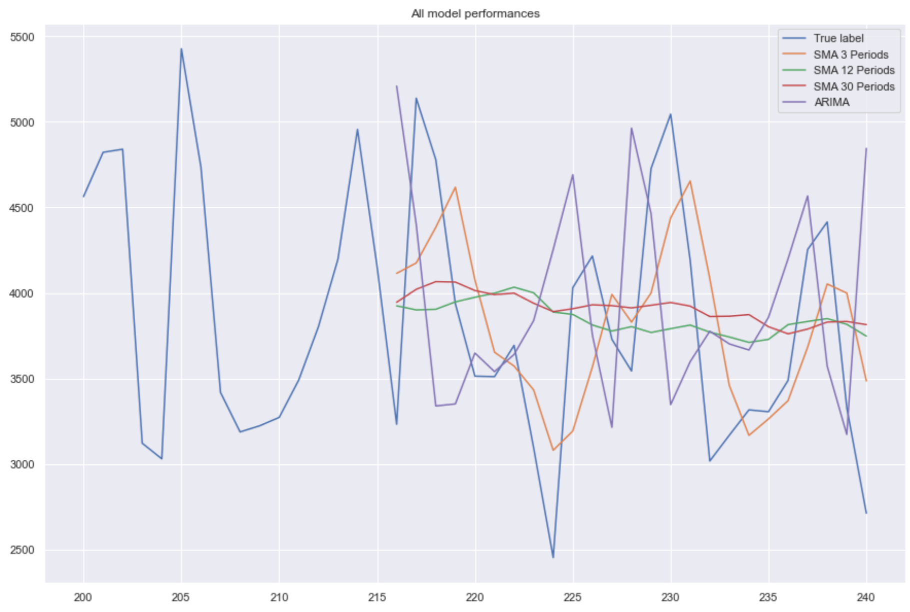
```
                               SARIMAX Results
==============================================================================
Dep. Variable:        NUMBER_VEHICLES   No. Observations:                  216
Model:               SARIMAX(12, 0, 10)   Log Likelihood               -1623.558
Date:                Wed, 19 Oct 2022   AIC                           3293.117
Time:                        19:01:35   BIC                           3370.748
Sample:                             0   HQIC                          3324.480
                                - 216
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.0921      0.086      1.074      0.283      -0.076       0.260
ar.L2         -0.1793      0.080     -2.249      0.025      -0.336      -0.023
ar.L3          0.1620      0.086      1.884      0.060      -0.007       0.331
ar.L4         -0.0969      0.090     -1.082      0.279      -0.272       0.079
ar.L5          0.1901      0.098      1.931      0.054      -0.003       0.383
ar.L6         -0.0396      0.095     -0.417      0.677      -0.226       0.147
ar.L7          0.1838      0.095      1.932      0.053      -0.003       0.370
ar.L8         -0.0621      0.094     -0.662      0.508      -0.246       0.122
ar.L9          0.0965      0.086      1.117      0.264      -0.073       0.266
ar.L10        -0.1740      0.082     -2.114      0.035      -0.335      -0.013
ar.L11         0.0765      0.070      1.087      0.277      -0.061       0.215
ar.L12         0.7495      0.068     10.997      0.000       0.616       0.883
ma.L1          0.3710      0.102      3.649      0.000       0.172       0.570
ma.L2          0.6103      0.104      5.842      0.000       0.406       0.815
ma.L3          0.1003      0.137      0.731      0.465      -0.169       0.369
ma.L4          0.1506      0.121      1.242      0.214      -0.087       0.388
ma.L5         -0.2804      0.112     -2.504      0.012      -0.500      -0.061
ma.L6         -0.2624      0.120     -2.188      0.029      -0.498      -0.027
ma.L7         -0.5536      0.121     -4.594      0.000      -0.790      -0.317
```

To evaluate its performance, the author predicts the test set and calculates MAPE this time. It’s not surprising that MAPE is up to 22.3092 %. To see the performance visually, please see two plots below:

Figure 23

ARIMA performance and comparison with other conducted models.





```
Mean Absolute Percent Error for ARIMA: 22.3092 %
```

Conclusion: Firstly, a MAPE of 22.3% is not a terrible prediction. However, ARIMA does not improve the accuracy of prediction from SMA, even though the author already picked the best order for it. This is because the patterns for ACF and PACF show seasonal trends instead of tail-offs or significant cut-offs as described for ARMA.

6. Seasonal Auto-Regressive Integrated Moving Average - SARIMA

In the previous works, the author already covered SMA and ARIMA; however, our data explicitly depict seasonality, which makes sense as we record traffic from 7 AM to 6 PM every day and repeat the process in the next cycle.

In this part, the author will demonstrate SMARIA to improve model performance.

A seasonal ARIMA model or SARIMA is written as follows:

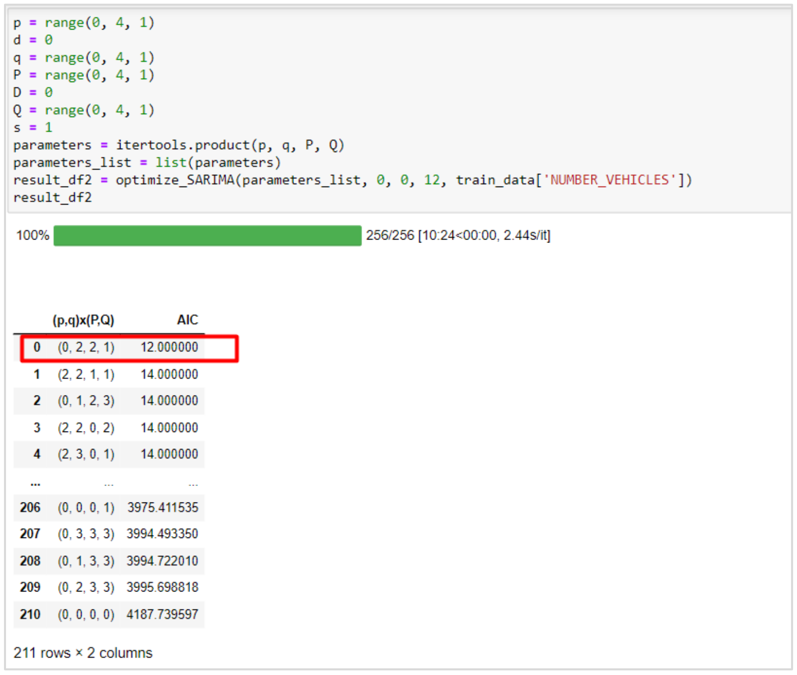
$$SARIMA \underbrace{(p, d, q)}_{non-seasonal} \underbrace{(P, D, Q)_m}_{seasonal}$$

In this formula, P, D, and Q are the same for non-seasonal time series, and m is the number of observations per 1 seasonal period.

Looking at ACF and PACF, we cannot tell which values for p, d, P and D for SARIMA as the same as when we conduct it for ARIMA, but we can induce the parameter m. In this case, **m = 12** as we can see 12 observations per 1 seasonal period.

The process to determine p, d, P and D is relatively equivalent to the one we conducted earlier by creating multiple combinations of orders and then picking up the most optimal one with the lowest AIC. *However, since it is too computationally expensive for the author's computer to conduct a range of every element from 0 to 13 (we have 28,561 combinations to fit the model), the author only runs a small range of each factor, which may not yield the best result for SARIMA. But again, this is for illustration only.*

The best order for our model is as below:



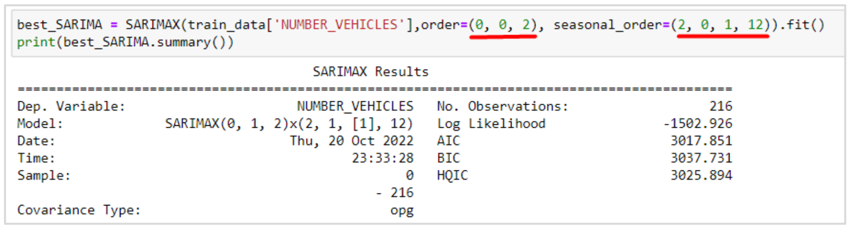
```
p = range(0, 4, 1)
d = 0
q = range(0, 4, 1)
P = range(0, 4, 1)
D = 0
Q = range(0, 4, 1)
s = 1
parameters = itertools.product(p, q, P, Q)
parameters_list = list(parameters)
result_df2 = optimize_SARIMA(parameters_list, 0, 0, 12, train_data['NUMBER_VEHICLES'])
result_df2
```

```
100% 256/256 [10:24<00:00, 2.44s/it]
```

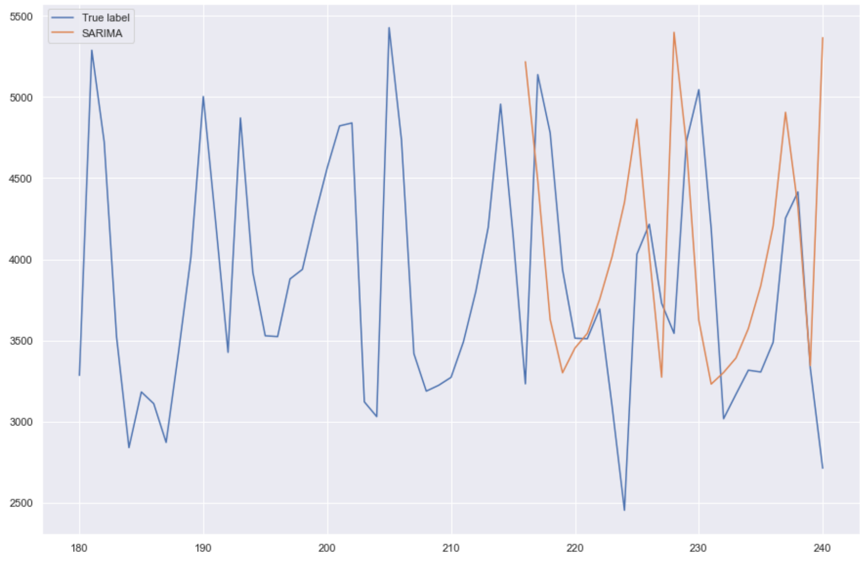
	(p,q)x(P,Q)	AIC
0	(0, 2, 2, 1)	12.000000
1	(2, 2, 1, 1)	14.000000
2	(0, 1, 2, 3)	14.000000
3	(2, 2, 0, 2)	14.000000
4	(2, 3, 0, 1)	14.000000
...	...	...
206	(0, 0, 0, 1)	3975.411535
207	(0, 3, 3, 3)	3994.493350
208	(0, 1, 3, 3)	3994.722010
209	(0, 2, 3, 3)	3995.698818
210	(0, 0, 0, 0)	4187.739597

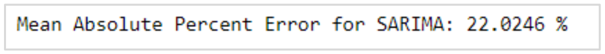
211 rows × 2 columns

From this result, the author runs an optimal SARIMA model and evaluates its performance:



```
best_SARIMA = SARIMAX(train_data['NUMBER_VEHICLES'],order=(0, 0, 2), seasonal_order=(2, 0, 1, 12)).fit()
print(best_SARIMA.summary())
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                    NUMBER_VEHICLES   No. Observations:                  216
Model:             SARIMAX(0, 1, 2)x(2, 1, [1], 12)   Log Likelihood               -1502.926
Date:                            Thu, 20 Oct 2022   AIC                           3017.851
Time:                                    23:33:28   BIC                           3037.731
Sample:                                         0   HQIC                          3025.894
                                            - 216
Covariance Type:                              opg
```





```
Mean Absolute Percent Error for SARIMA: 22.0246 %
```

Model performance is slightly improved compared to traditional ARIMA with MAPE now of 22.02%. But again, if we could conduct a wider range for each order, the author believes that model performance would be significantly uplifted.

III. CONCLUSION AND KEY TAKEAWAYS

The author has several highlights to list below:

Evenly spaced interval: One fundamental condition for time series is that its timestamps should be equally different. In case, its intervals are not equally spaced, we need to handle this issue by multiple approaches.

Important components of time series: There is a checklist for time series if we want to work in depth, which includes: stationarity, ACF, PACF, white noise, decomposing time series, unit roots, and seasonality model.

IV. REFERENCES

Kumar, S. (2022, Apr 28). *4 Techniques to Handle Missing values in Time Series Data*. https://towardsdatascience.com/4-techniques-to-handle-missing-values-in-time-series-data-c3568589b5a8

Monigatti, L. (2022, Aug 2). *Interpreting ACF and PACF Plots for Time Series Forecasting*. https://towardsdatascience.com/interpreting-acf-and-pacf-plots-for-time-series-forecasting-af0d6db4061c

Moreno, A. I. (2020, Jul 8). *Moving averages with Python*. https://towardsdatascience.com/moving-averages-in-python-16170e20f6c

Peixeiro, M. (2020, Jun 10). *Time Series Forecasting with Autoregressive Processes*. https://towardsdatascience.com/time-series-forecasting-with-autoregressive-processes-ba629717401

Peixeiro, M. (2020, Jun 24). *Advanced Time Series Analysis with ARMA and ARIMA*. https://towardsdatascience.com/advanced-time-series-analysis-with-arma-and-arima-a7d9b589ed6d

Peixeiro, M. (2020, Jun 29). *Time Series Forecasting with SARIMA in Python*. https://towardsdatascience.com/time-series-forecasting-with-sarima-in-python-cda5b793977b

Peixeiro, M. (2021, Dec 6). *Defining the Moving Average Model for Time Series Forecasting in Python*. https://towardsdatascience.com/defining-the-moving-average-model-for-time-series-forecasting-in-python-626781db2502 ritvikmath. (2020, Feb 18). *Why Are Time Series Special? : Time Series Talk*. https://www.youtube.com/watch?v=ZoJ2OctrFLA&list=PLvcbYUQ5t0UHOLnBzl46_Q6QKtFgfMGc3