Springboard Capstone 2 Project Report

By

Krishnan G. Raghavan

Course provided Springboard Data Science

2024

Table of Contents

Abbreviations, Notations and Variables

This section lists some of the commonly used abbreviations, notations and variables in this report.

Abbreviations:

EDA. – Exploratory

Data Analysis

df. - Dataframe

Abstract

Badly timed traffic lights can cause several problems like traffic congestion, dissatisfaction for drivers, dissatisfaction for drivers, increased risk of accidents and higher pollution. Many traffic controllers use real-time traffic data to time their signals. However, these are often expensive to set up and operate. One solution to this is implementing predictive models to time the signal, which will decrease dependence on real time sensor information, possibly eliminating the need for several sensors or allow cheaper sensors and reduce costs of installation and operation.

This project attempts to create such a model, using data from the Mill Road Project.

This is hourly data for the flow of vehicles and pedestrians at Mill Road and surround streets based on 16 sensors.

This data will be analyzed to develop a model that will predict the traffic for the next hour based on a combination of expected traffic from several days of data at the time, and recent traffic hourly data. This output can then be used to time the traffic signal appropriately.

Chapter 1. Introduction

With increasing population, technology and industrialization, travel traffic has increased substantially and is only expected to increase. According to a report by the International Energy Agency (IEA), global car sales are projected to increase from around 80 million units in 2020 to over 100 million by 2030, which is a 25% increase in a decade.

This increase will contribute to significant pollution and increased time wasted due to stop and go traffic.

Well timed traffic signals could be one solution to mitigate this problem, but the timing of the signals depends on traffic. Current signals use an array of sensors with mostly real time monitoring to time them. While this could be more accurate, it comes at high installation, maintenance and processing costs.

One way to reduce this cost would be to use predictive algorithms to estimate traffic conditions and use that information in conjunction with real time sensor data to reduce the number of sensors. This project aims to explore the development and efficacy of such an algorithm.

1.1 Business Case

Problem: How can temporal data from traffic sensors be used to reasonably predict hourly traffic flow by looking at near term and long-term patterns in Mill Road traffic data?

Customer's needs: Customer is a city official, or member of public concerned about worsening traffic conditions in the area leading to significant loss of time and reduced air quality.

1.2 Such a Project is useful for...

- The general public in the area, who are affected by the high traffic, such as increased commute time and health effects due to air and noise pollution.

1.3 Project Overview

In this project, we explore various modeling options to determine the feasibility of predicting traffic flow. We use traffic outflow as our metric. The salient features of this project are:

- Process for data extraction, exploration and processing with the goal of making predictive models for traffic outflow
- Training and performance of the models – ARIMA, XGBoost and Linear Regressor
- Comparison of the models – development time and performance on test data
- Correlation of traffic outflow captured at various locations by the respective sensors

1.4 Overview of this Report

The following is an overview of the contents of this report:

- Chapter 2: Data Extraction, Exploration and Pre-Processing
- Chapter 3: Modeling Data Preparation, Model Creation, Model validation
- Chapter 4: Conclusion
- Chapter 5: Discussion

Chapter 2. Data Extraction, Exploration and Pre-Processing

In this chapter, we talk about what kind of data was used in the project, where it was obtained from and an overview of how it was used in this project to solve the problem.

2.1 Data Acquisition

Data was acquired from the Mill Road Project Website [Mill Road Sensor Data], which contains hourly vehicle and pedestrian traffic counts. The following steps were involved in data extraction:

1. **Data Loading**: The dataset was loaded using Pandas, either directly from a URL or a local file.

```
#data_url='https://query.data.world/s/wj2wwszc2ldqikthjxld4zazaayexw?dws=00000'
# Import
#df=pd.read_csv(data_url)
df=pd.read_csv('mill_road_traffic_data.csv',index_col=0)
df.head()
```

✓ 0.6s Open 'df' in Data Wrangler

	Local Time (Sensor)	Date	Time	countlineName	direction	Car	Pedestrian	Cyclist	Motorbike	Bus	OGV1	OGV2	LGV
0	03/06/2019 01:00	03/06/2019	01:00:00	S10_EastRoad_CAM003	in	89	9	4	2	1	0	0	5
1	03/06/2019 01:00	03/06/2019	01:00:00	S10_EastRoad_CAM003	out	51	6	0	0	0	1	0	1
2	03/06/2019 01:00	03/06/2019	01:00:00	S12_DevonshireRoad_CyclePath_CAM003	in	0	0	0	0	0	0	0	0
3	03/06/2019 01:00	03/06/2019	01:00:00	S12_DevonshireRoad_CyclePath_CAM003	out	0	2	0	0	0	0	0	0
4	03/06/2019 01:00	03/06/2019	01:00:00	S13_MiltonRoad_CAM003	in	66	2	5	4	0	0	0	4

While it is possible to read html, it takes considerably longer than loading a locally saved file.

2. **Dataset Overview**: The dataset contains multiple features, including timestamps and traffic counts, categorized by sensor locations. Initial exploration involves inspecting the first few rows to understand the structure and content of the data.

Sensor location data was separately acquired from the website as well. The data was downloaded and then loaded into the notebook. The purpose of this data is to look at the location of sensors on a map

```
# Location information
df_loc=pd.read_csv('mill-road-trial-sensor-point-locations-2.csv')
df_loc
```

[6] ✓ 0.0s Open 'df_loc' in Data Wrangler

	Sensor Reference	Sensor Type	Street location	Column number	Easting	Northing
0	1	Vivacity sensor	362 Mill Rd	L6RAS	52.196510	0.153030
1	2	Vivacity sensor	Mill Rd (SO 1 Mortimer Rd)	L62RAS	52.201920	0.132450
2	3	Vivacity sensor	108 Coleridge Rd	L10RCI	52.190930	0.145920
3	4	Vivacity sensor	114 Vinery Rd	L11RJJ	52.199990	0.152570
4	5	Vivacity sensor	2 Tenison Rd	L1SAI (1)	52.199940	0.136790
5	6	Vivacity sensor	OP 6 Station Rd	L2SAO	52.194855	0.132876
6	7	Vivacity sensor	151/153 Coldhams Ln	L36RJD	52.203770	0.152010
7	8	Vivacity sensor	117 Cherry Hinton Rd	L13RCH	52.188320	0.142270
8	10	Vivacity sensor	142 Perne Road	L21RIW	52.192390	0.154670
9	11	Vivacity sensor	O/S ARU East Road	L29RBS	52.204070	0.132940
10	12	Vivacity sensor	41 Histon Road	L6WAS	52.215610	0.110890
11	13	Vivacity sensor	55 Devonshire Rd	L11RCD	52.196240	0.137130
12	14	Vivacity sensor	214 Milton Rd	L37RCG	52.220630	0.134160
13	15	Vivacity sensor	140 Hills Rd	L58SAN (39A)	52.190080	0.135340
14	16	Vivacity sensor	560 Newmarket Road	L104RCA	52.212880	0.156740

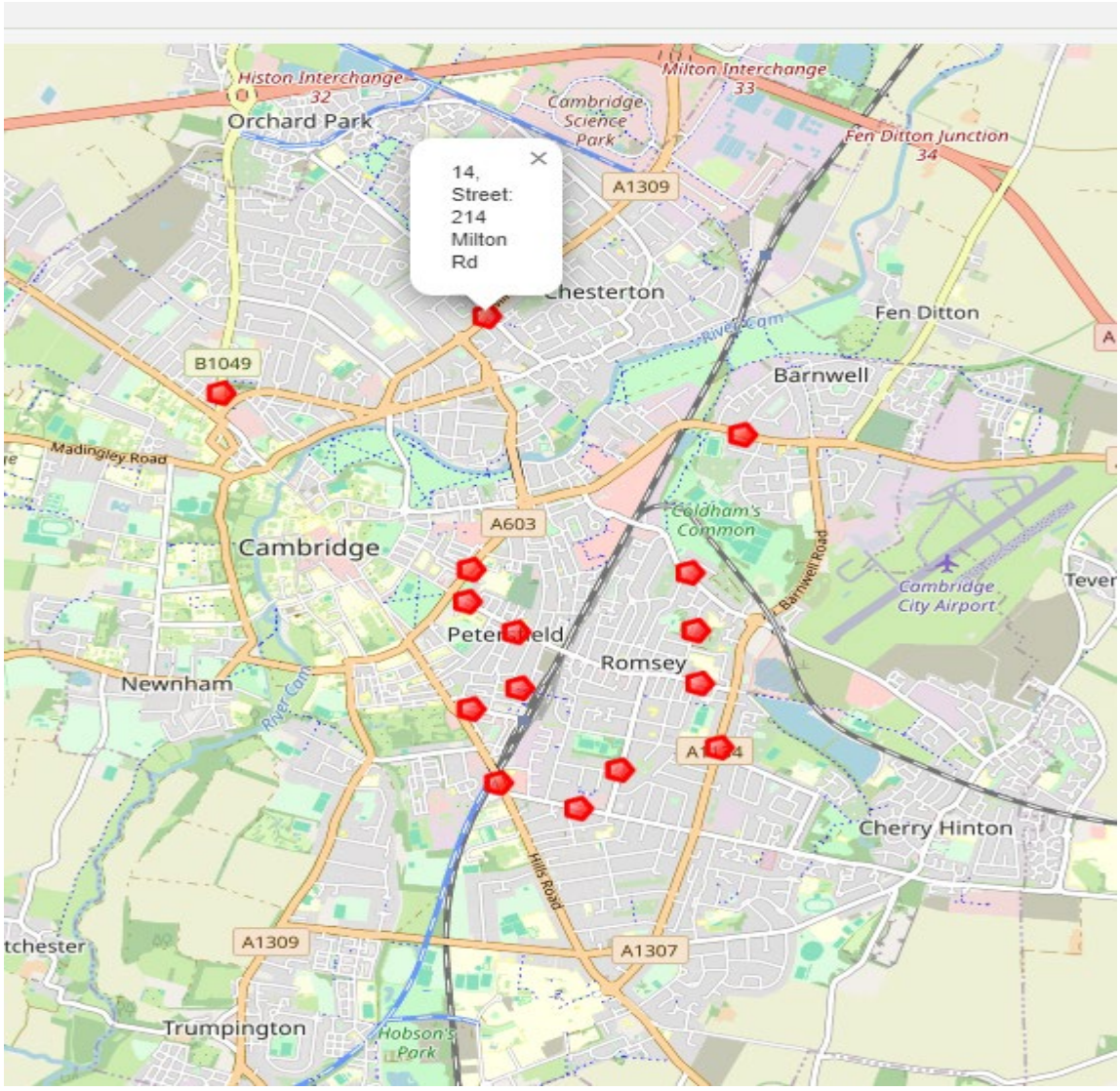
'Easting' and 'Northing' are just the Latitudes and Longitudes.

2.2 Exploration

1. First, we looked at the distribution of the sensors on an interactive map of the area, plotted using the map library 'Folium'.

Visualize sensors on map

```
# Sensor distribution on map
latitude=np.median(df_loc['Sensor Latitude'])
longitude=np.median(df_loc['Sensor Longitude'])

map_clusters = folium.Map(location=[latitude, longitude],zoom_start=12)

for index,row in df_loc.iterrows():

    num,street,lat,lon=row['Sensor Reference'],row['Street location'],row['Sensor Latitude'],row['Sensor Longitude']
    label = folium.Popup(f"{num}, Street: {street}", parse_html=True)

    folium.RegularPolygonMarker(
        [lat, lon],
        radius=7,
        number_of_sides=5,
        popup=label,
        color='red',
        fill=True,
        fill_color=None,
        fill_opacity=0.7,
                    ).add_to(map_clusters)

map_clusters
```



This shows how the sensors are located near Cambridge with the streets in the surrounding area. This gives us an idea of how the sensors are used to track traffic through a grid of streets in the area.

2. We looked at the time deltas by listing the unique timestamps using the pandas method .unique(). This showed us that the times are in intervals of hours as mentioned by the website

3. We created an ‘hour’ column incase we want to visualize the traffic flow by hour in combination with potentially other parameters.

4. We created a ‘Total’ column to obtain the sum of all traffic for that row (including pedestrians, cars, buses, motorcycles, cyclists, OGVs, LGVs).

5. The next important step is to merge the ‘in’ and ‘out’ data into a single entry. To do this, first the ‘in’ and ‘out’ direction data were separated, then again merged by timestamp and sensorname. Suffixes ‘_in’ and ‘_out’ were added to the corresponding traffic columns. This transformation makes it very easy to work with calculations for each sensor/timestamp combination by avoiding additional filtering based on direction.

Combine and in and out DFs into one dataframe showing all traffic for a given time

```
df_sensors_in_out=df_sensors_in.merge(df_sensors_out,on=['Local Time (Sensor)','countlineName'],suffixes=('_in', '_out'))
df_sensors_in_out.head()
```

✓ 0.2s 🈸 Open 'df_sensors_in_out' in Data Wrangler Python

	Local Time (Sensor)	Date	Time	countlineName	direction	Car_in	Pedestrian_in	Cyclist_in	Motorbike_in	Bus_in	...	Total_in	Car_out	Pedestrian_out	Cyclist_out	Motorb
0	03/06/2019 01:00	03/06/2019	01:00:00	S10_EastRoad_CAM003	in	89	9	4	2	1	...	110	51	6	0	
1	03/06/2019 01:00	03/06/2019	01:00:00	S12_DevonshireRoad_CyclePath_CAM003	in	0	0	0	0	0	...	0	0	2	0	

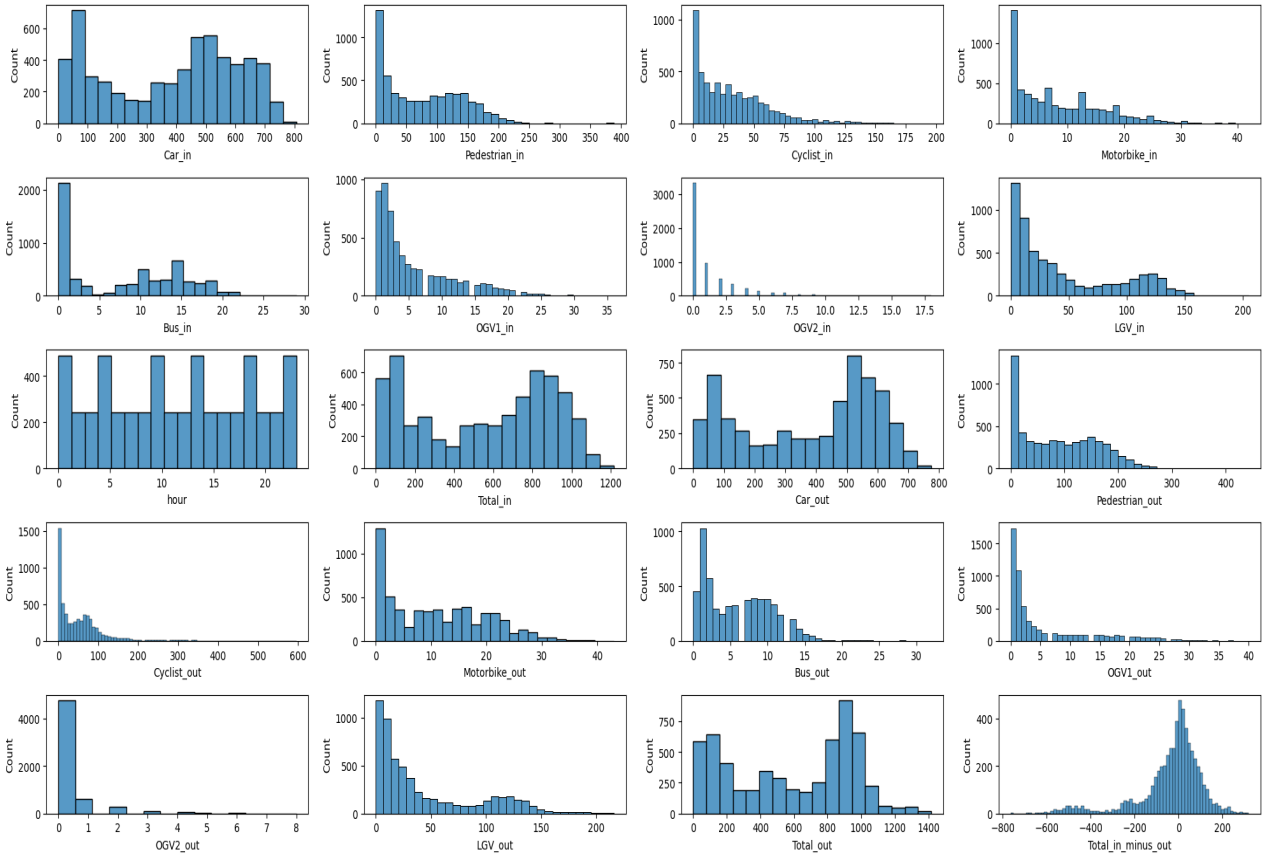
6. We calculate an ‘in-out’ column to show the net flow of traffic.

7. Do a quick check of the shape and features of the data using the pandas .info() method:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 262530 entries, 0 to 262529
Data columns (total 25 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   Local Time (Sensor)  262530 non-null  object
 1   Date                 262530 non-null  object
 2   Time                 262530 non-null  object
 3   countlineName        262530 non-null  object
 4   direction            262530 non-null  object
 5   Car_in               262530 non-null  int64
 6   Pedestrian_in        262530 non-null  int64
 7   Cyclist_in           262530 non-null  int64
 8   Motorbike_in         262530 non-null  int64
 9   Bus_in               262530 non-null  int64
 10  OGV1_in              262530 non-null  int64
 11  OGV2_in              262530 non-null  int64
 12  LGV_in               262530 non-null  int64
 13  hour                 262530 non-null  int64
 14  Total_in             262530 non-null  int64
 15  Car_out              262530 non-null  int64
 16  Pedestrian_out       262530 non-null  int64
 17  Cyclist_out          262530 non-null  int64
 18  Motorbike_out        262530 non-null  int64
 19  Bus_out              262530 non-null  int64
 20  OGV1_out             262530 non-null  int64
 21  OGV2_out             262530 non-null  int64
 22  LGV_out              262530 non-null  int64
 23  Total_out            262530 non-null  int64
 24  Total_in_minus_out   262530 non-null  int64
dtypes: int64(20), object(5)
```

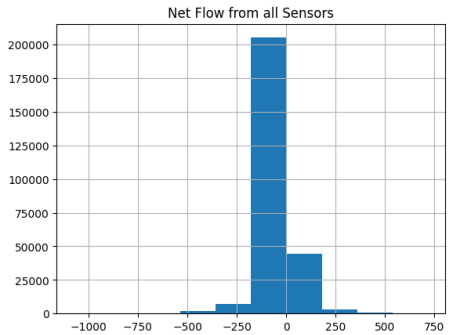
The columns have non-null values in the required format. If this weren’t the case, additional cleaning to remove / impute unavailable data would be needed.

8. Since the data is so large, we can try modeling by first isolating the data for one sensor. The below figure shows the distribution of parameters for sensor: ‘S10_EastRoad_CAM003’.



There is a problem, the total_in_minus_out, or net flow, is left-skewed. This means more traffic leaves than enters, which does not make practical sense.

9. Perhaps a more accurate picture could be obtained by considering data from all sensors instead of isolating by sensor?



Now the mean is closer to zero and the distribution is tighter around zero which makes more sense.

This tells us that we should include the data from all sensors in our analysis for an accurate representation of a real world intersection.

2.3 Variable Selection

What variable to focus on?

Real world traffic signals will probably time based on the traffic inflow.

High inflow may call for a higher green duty cycle to prevent stop-and-go behavior and congestion at the intersection.

Low inflow may require a more balanced duty cycle to prevent excessive wait times.

Therefore, we can choose **'Total_in'** as our dependent (target) variable.

Chapter 3. Modeling Data Preparation, Model Creation, Model validation

This chapter discusses the process utilized to go from 'Question to Answer'. It elaborates on how the data is transformed to analyze the target variable and the various analysis steps.

3.1 Data Preparation

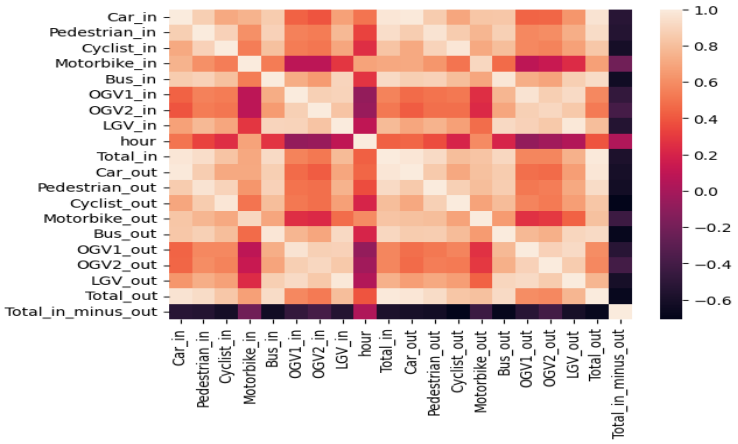
Since there are multiple sensor values at the same timestamp and we are interested in the total, we should sum the values at each timestamp. This has the added benefit of giving us a smaller array size that will be easier to work with. We do this by first grouping by Local time using the pandas 'groupby' method then obtaining the sum using the 'agg' method for the various traffic values, and mean for the 'hour'.

```
# Create aggregation dictionary (aggregate hour to mean and others to sum)
agg_dict = {col:'sum' if col != 'hour' else 'mean' for col in numeric_cols}
# Group by Local time using aggregation dictionary
df_sensors_in_out_timegrouped=df_sensors_in_out.groupby(by='Local Time (Sensor)').agg(agg_dict).sort_index()
df_sensors_in_out_timegrouped.head()
```

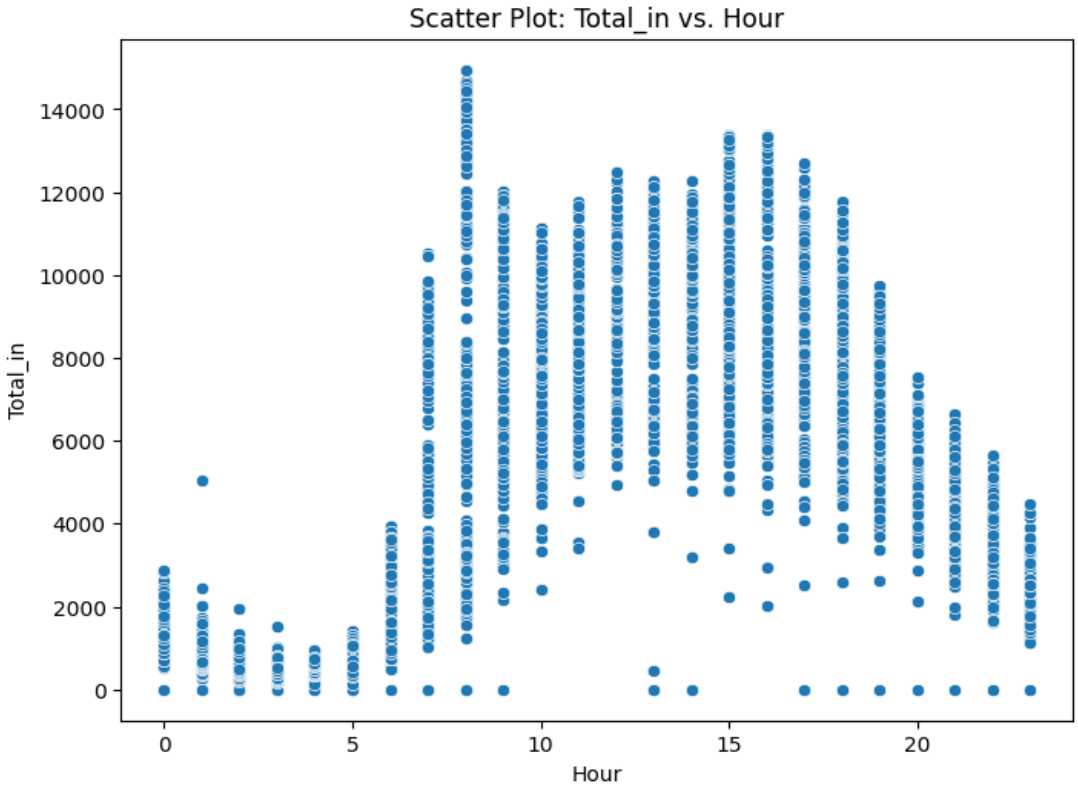
0.0s Open 'df_sensors_in_out_timegrouped' in Data Wrangler

Local Time (Sensor)	Car_in	Pedestrian_in	Cyclist_in	Motorbike_in	Bus_in	OGV1_in	OGV2_in	LGV_in	hour	Total_in	Car_out	Pedestrian_out	Cyclist_out	Motorbike_out	Bus_out	OC
2019-06-03 01:00:00	565	40	25	14	1	1	0	25	1.0	671	695	40	10	4	4	
2019-																

A quick look at the relationships between various kinds of traffic:

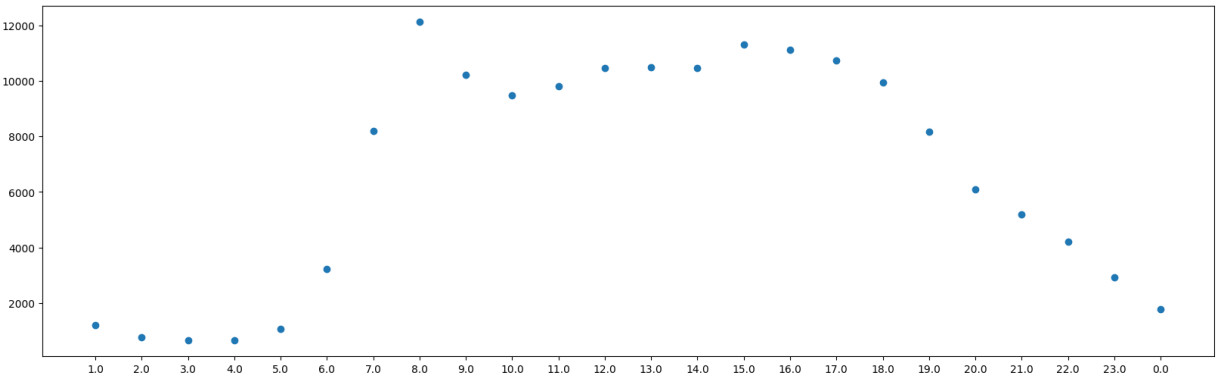


Most variables have significant correlation with 'Total_out'. While 'hour' has among the least correlation, we still want to see if there is any pattern between hour and Total_out.



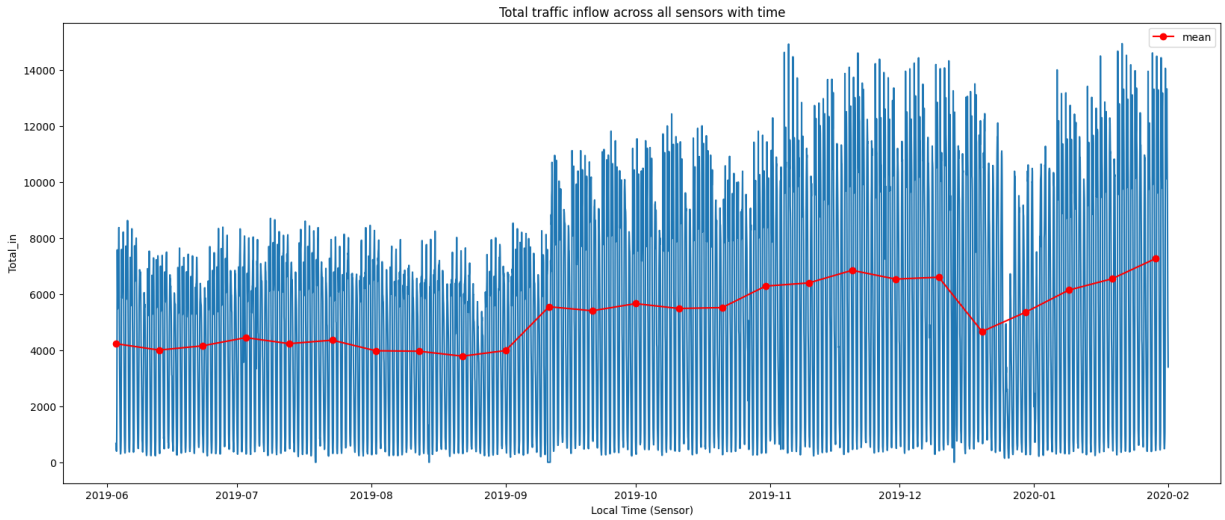
There indeed is a pattern, with the traffic being low till 5 am, increasing steadily till about 11 am, then dropping after around 4 pm.

To create a model of traffic by hour, lets take the following metric: median+1std (1std to be conservative in our traffic estimate). Doing that and plotting against hour gives us this plot:



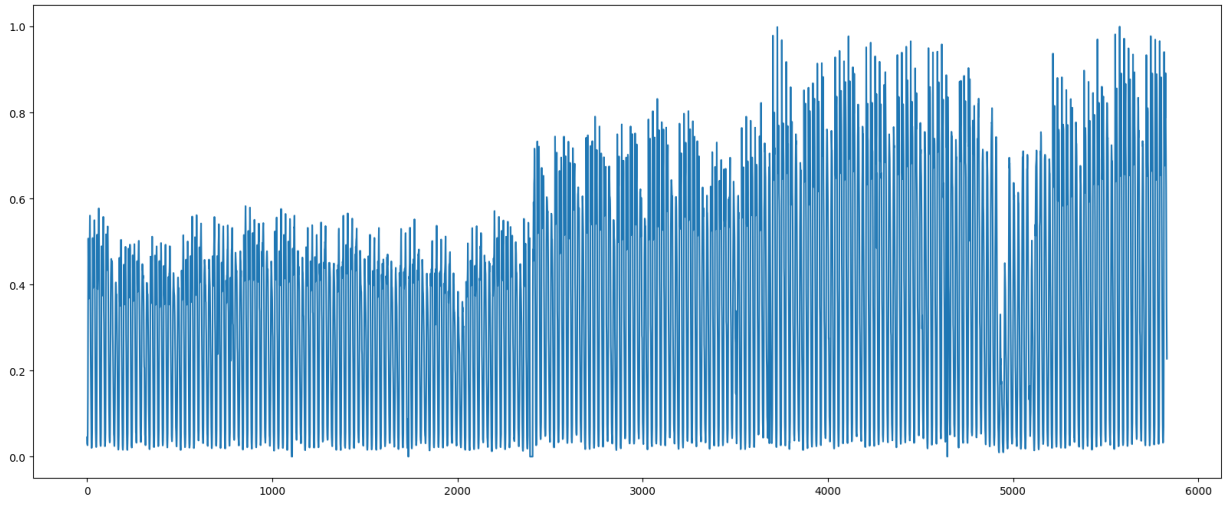
We can possibly use this metric later on in our analysis.

Let’s now look at the data vs. time.



There is non-stationarity, from both seasonality and trend. The drop in traffic during late December can be due to the holiday season.

Scaling: We explored Standard, Log and MinMax scaling options, and found MinMax Scaling to be the most appropriate since the data has a trend, making standard scaling unsuitable, and does not have a logarithmic distribution which makes the valleys stand out when using logarithmic scaling.



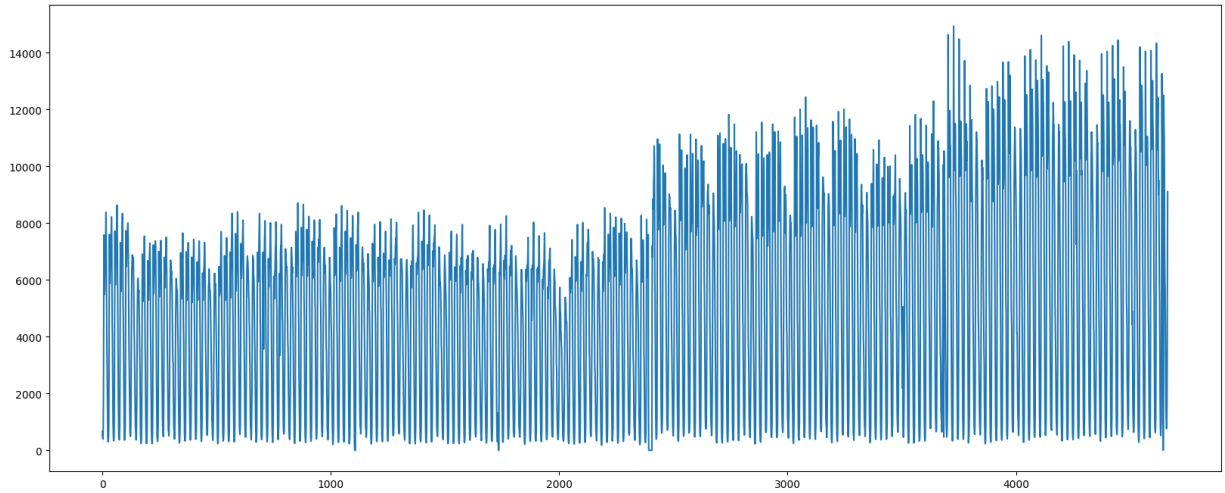
Train-Test Split: Since it is a time series data, instead of randomizing the dataset we split it into 80% training and 20% testing.

3.2 Model Creation and Performance

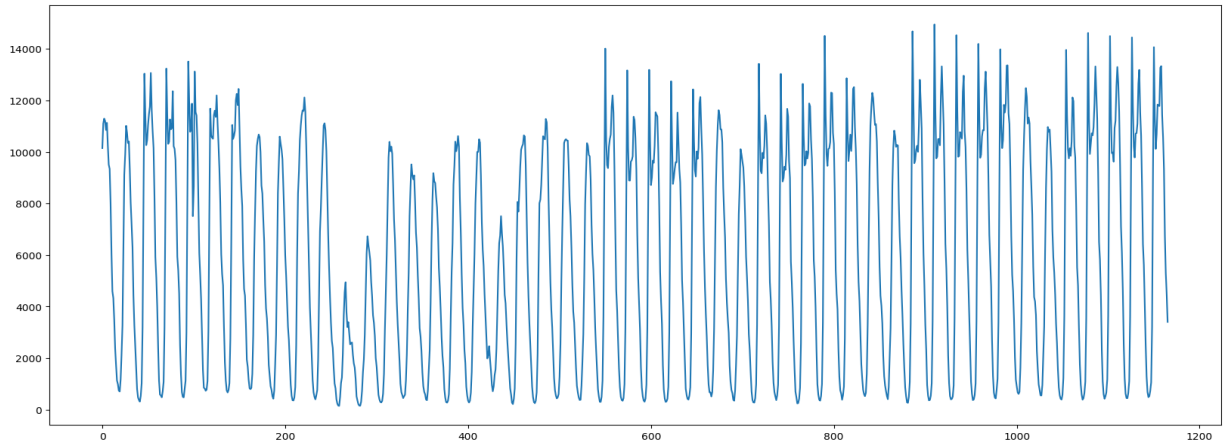
ARIMA Model:

An Autoregressive Moving Average model was developed using Scikit-Learn's ARIMA library to analyze the time series. ARMA (Autoregressive Moving Average) is a time series model used for forecasting and analyzing the behavior of time-series data. It is a combination of two components: the Autoregressive (AR) model and the Moving Average (MA) model.

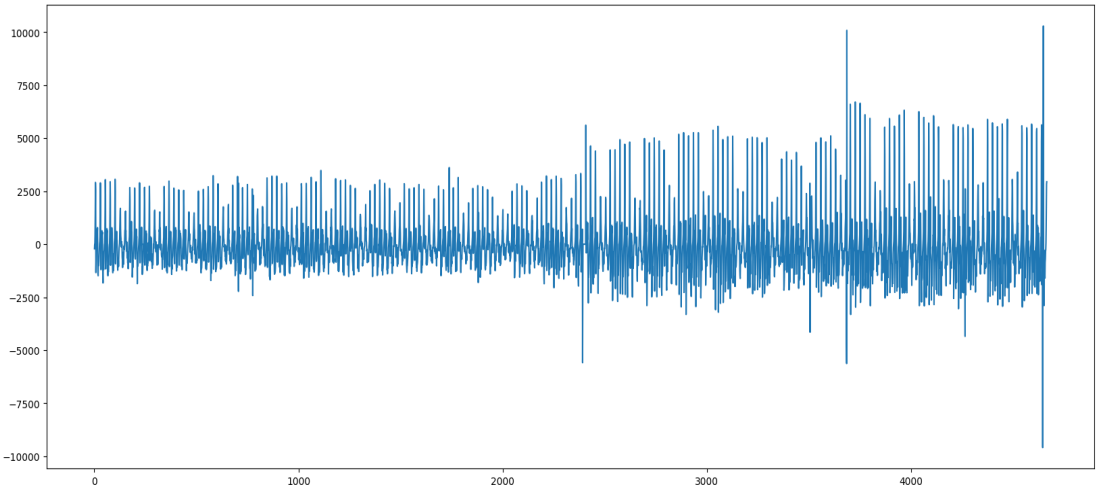
Below is a plot of the training data used for ARIMA:



and the test data:

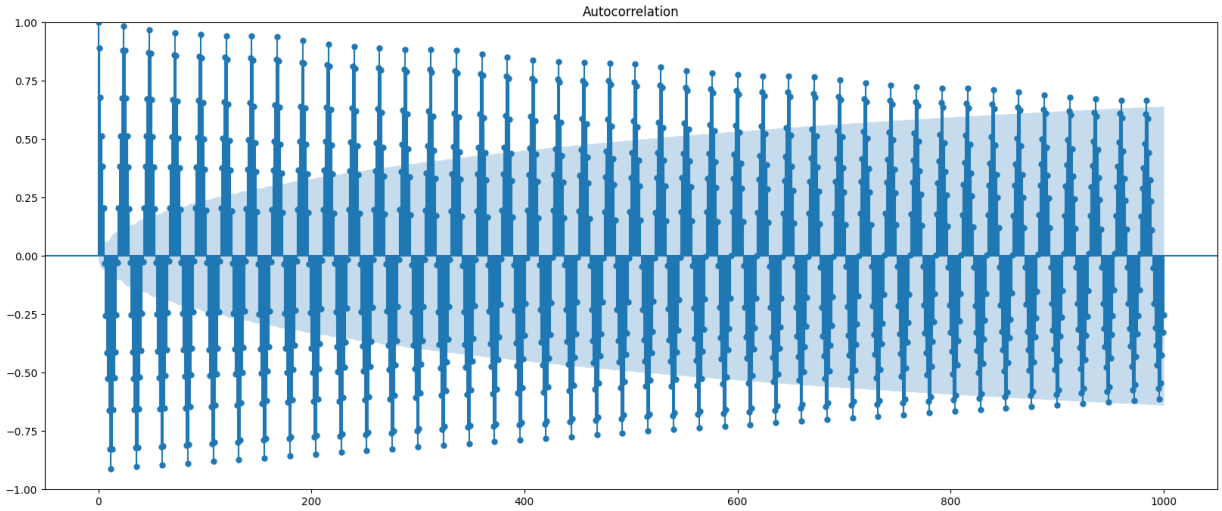


1. We first make the signal stationary (or close to) by taking the first difference. Stationary means the mean and other central tendencies tend to remain stable over time. A plot of the first difference:



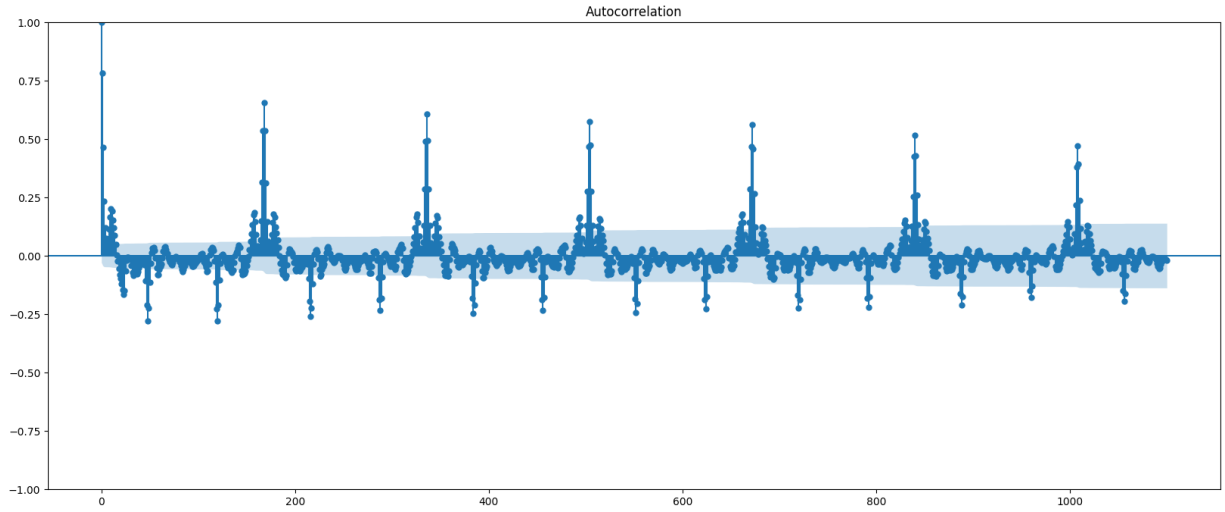
The plot indicates that the signal likely still has some seasonal effects. We can attempt to remove this by appropriate differencing, but how much? We can start by exploring the autocorrelation.

2. We plot the Autocorrelation using the statsmodel acf and plot_acf methods

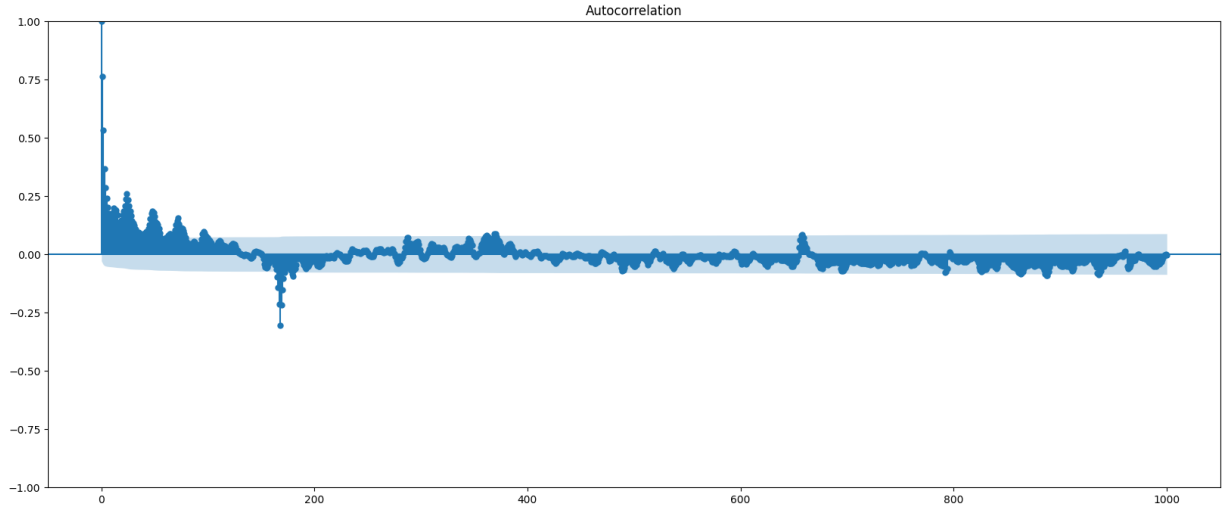


The shaded region represents the confidence interval. We see the confidence band get wider as the

number of lags gets higher. This suggests that the data is non-stationary. Further, we see a peaking trend even in the peaks of the acf plot. We can take the peaks using the scipy get_peaks function, and repeat it again.



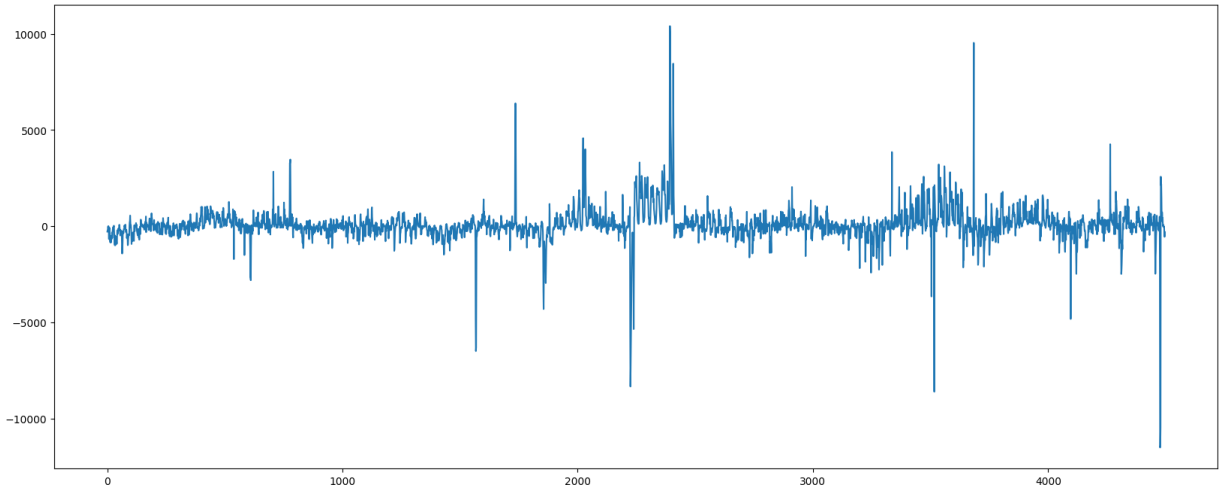
ACF plot after differencing using first peaks as interval



ACF plot after differencing using second peaks as interval

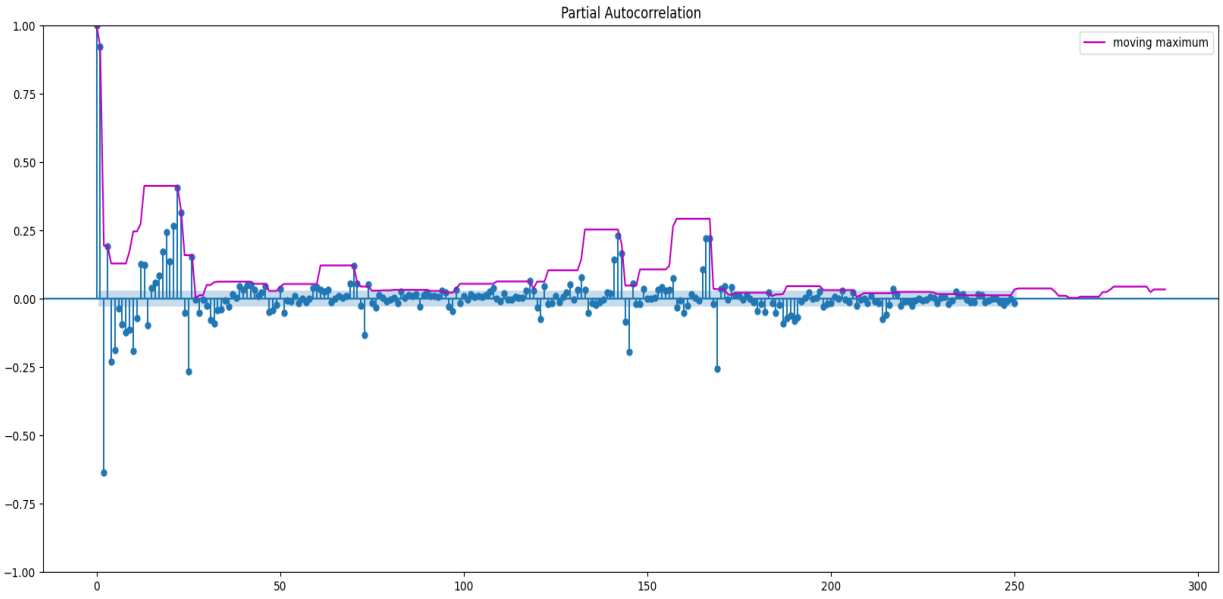
Now we have a more stationary ACF plot. But the differencing values is high (168)!

This is what the 168th difference plots looks like:



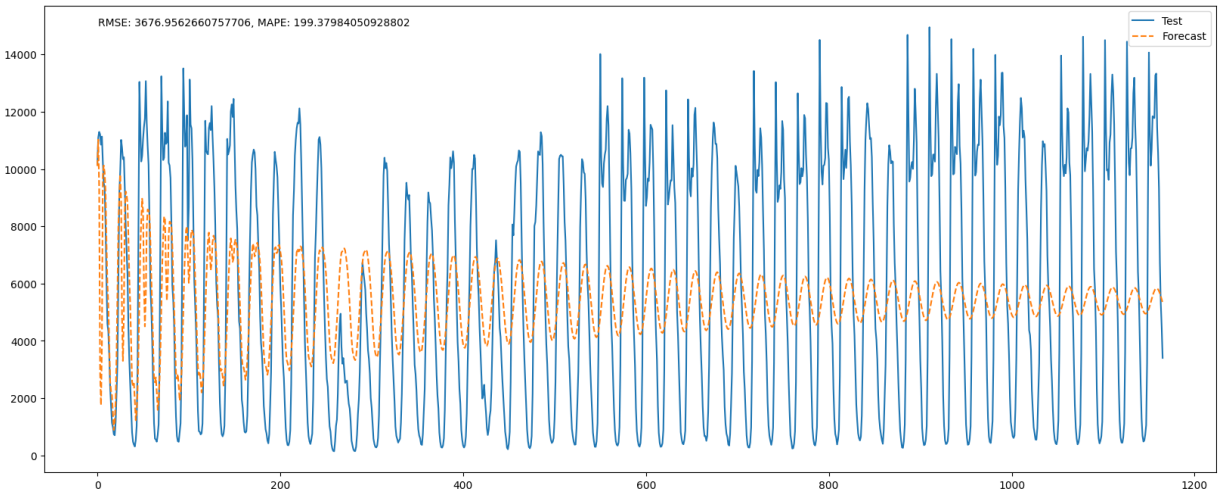
It's not perfect but better than difference 1. A lot of the intermediate perturbations have been removed. We can also consider the seasonality parameter of the ARIMA model to be 168 (a high number).

3. Next, we have to obtain the number of auto-regressors. For this, we look at the partial autocorrelation.



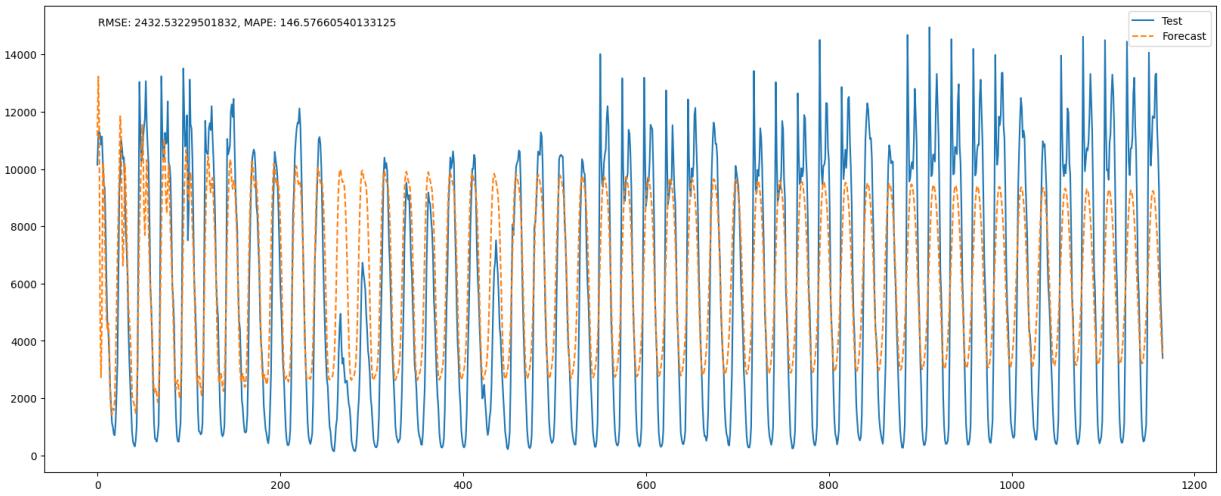
The maximum moving average line of window 10 goes within confidence bounds at 27. Hence we select this value as our autoregressive component number (p).

4. First, we try an ARIMA with few parameters: p=27, d=1, q=5:



Training time: 0.5 minutes

5. Repeat the model process with p=27, d=1, q=50:

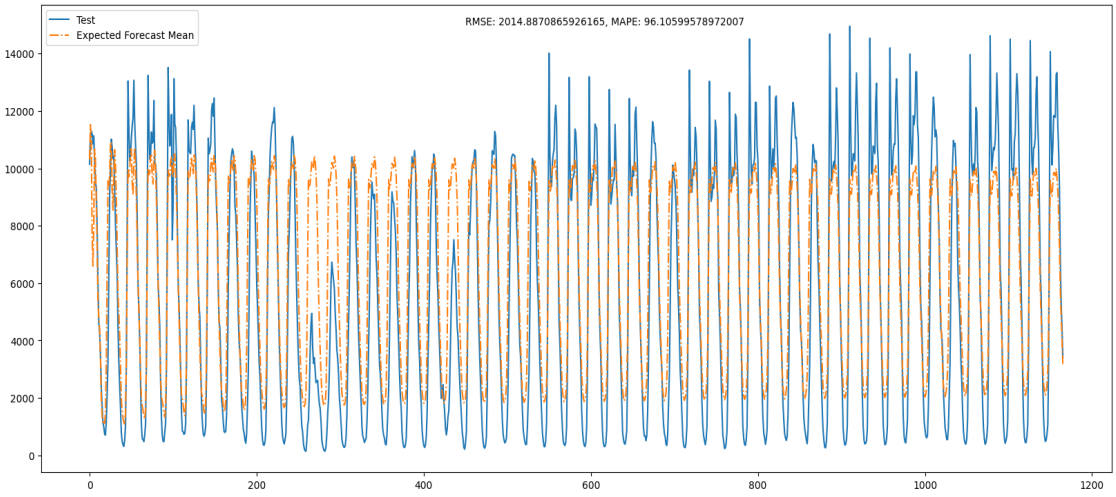


We see the RMSE go from 3677 to 2433, and the MAPE go from 199 to 147, suggesting a significant improvement.

Training time: 8.75 minutes

Ideally we should be able to go upto q=168, but the training time takes several hours, so it isn't feasible.

6. Instead, we can employ another strategy: Correct for the values using the expected hourly traffic flow we determined in the data processing phase (median + 1std). Taking the mean of the forecast output and the expected hourly:



This has an RMSE of 2014 that is better than the earlier 2433. The MAPE also went from 144 to 96. Notably it has the effect of reducing the noisy peaks in the earlier data. However, still performs poorly on future data (especially the holiday season).

XGB Regressor Model:

XGBoost (Extreme Gradient Boosting) is a highly efficient and scalable implementation of gradient boosting, a powerful machine learning technique that combines multiple weak models (such as decision trees) to create a strong predictive model. It has gained immense popularity in recent years for its exceptional performance on a wide range of machine learning tasks, including classification, regression, and ranking, and is widely used in industry and research due to its ability to handle large-scale data and complex problems effectively

Here we try out an XGBoost Regressor from the 'xgboost' library to try forecast the test data. To create a more robust model, we use grid search in combination with k-fold cross validation to iterate over transformer and hyperparameters to obtain the best performing model.

1. For an XGBoost regressor, the strategy is to predict the next point based on the previous few points. Hence, we have to first create a grid of X data containing 'n_back' consecutive points, and the y data of the next n_pred points for forecasting.

2. We then split the X and y into 80/20 as earlier. This gives us X_train, y_train,X_test and y_test.

3. Next we create a pipeline using the 'Pipeline' class of the scikit-learn preprocessing pipeline library. This allows us to create a pipeline with named steps. Our steps are a. minmax scaling, b. principal component analysis, c. modeling using XGBRegressor() . We perform PCA to ensure we do not have interdependencies in the independent variables.

4. We then create a parameter grid for each of the named steps as needed.

5. We create a pipeline using the 'Pipeline()' method by setting 'steps' as argument.

6. Finally we create the grid search using the GridSearchCV class of scikit-learn's model_selection library. The steps are shown below

```
# Create Pipeline and setup Grid Search
pca=PCA()
steps=[
    ('Transformer1',minmaxscaler),
    ('Transformer2',pca),
        ('Model',XGBRegressor())
        ]
# Set parameters
params = {
    'Transformer2__n_components':[1,10,30,70],
     'Model__max_depth':[1,5],
          'Model__learning_rate':[.1,.5],
            'Model__n_estimators':[2200,2500,2700]
}

# Create a TimeSeriesSplit object
tscv = TimeSeriesSplit(n_splits=3)

pipeline=Pipeline(steps)
cross_val=GridSearchCV(estimator=pipeline,param_grid=params,cv=tscv)
```

7. Fit the grid search component (cross_val) on X_train and y_train. The below snapshot shows the best values of hyperparameters for the best trained model:

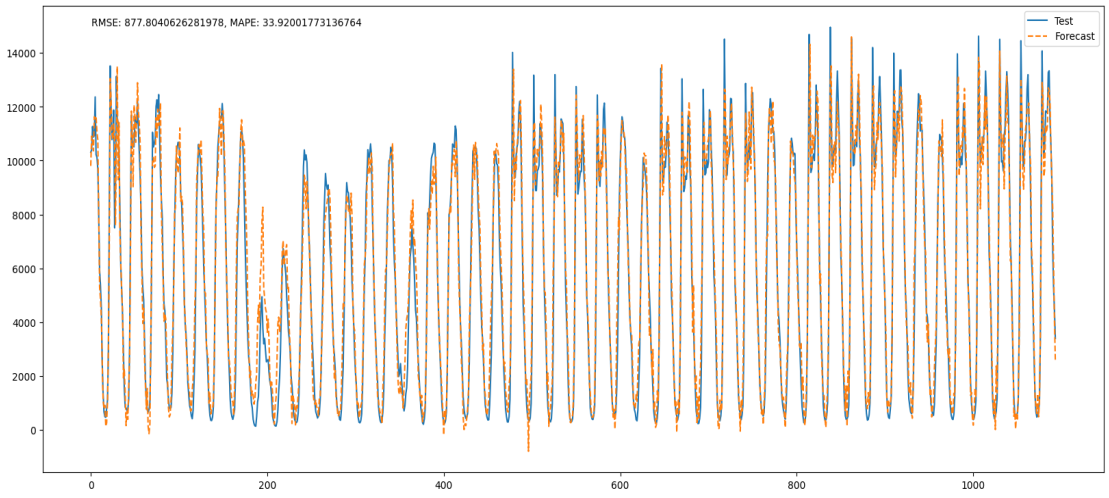
```
xgb_model=cross_val
xgb_model.fit(X_train,y_train)
(cross_val.best_score_,cross_val.best_params_)
```

✓ 4m 57.8s

```
(0.9050676015932981,
 {'Model__learning_rate': 0.5,
  'Model__max_depth': 1,
  'Model__n_estimators': 2700,
  'Transformer2__n_components': 70})
```

We also get a validation accuracy (R-squared) score of 0.905.

8. Plotting the forecast vs. test data:



In addition to the better RMSE of 878 compared to earlier 2014, and the MAPE is down to 34. We also see that the forecast can follow the peaks and valleys a lot better even for off-season traffic.

Linear Regressor Model:

A Linear Regressor is a type of supervised machine learning algorithm used for predicting a continuous

target variable based on one or more input features. It is a widely used technique for regression problems, where the goal is to estimate the relationship between the input variables and the output variable. Let's see how well it performs on forecasting the test data compared to the ARIMA and XGB earlier.

1. We use the same dataset as with XGB containing the history information.

2. The transformations include: a. minmax scaler, b. PCA, c. LinearRegressor(). LinearRegressor is the linear regression class from scikit-learn's 'linear_mode'l class.

3.

```
# Create Pipeline and setup Grid Search
pca=PCA()
steps=[
    ('Transformer1',minmaxscaler),
    ('Transformer2',pca),
        ('Model',LinearRegression())
        ]
# Set parameters
params = {
    'Transformer2__n_components':[1,10,30,70],
     #'Model__max_depth':[1,5],
         # 'Model__learning_rate':[.1,.5],
           #  'Model__n_estimators':[2200,2500,2700]
}
tscv = TimeSeriesSplit(n_splits=5)

pipeline=Pipeline(steps)

cross_val=GridSearchCV(estimator=pipeline,param_grid=params,cv=tscv)
```

✓ 0.0s

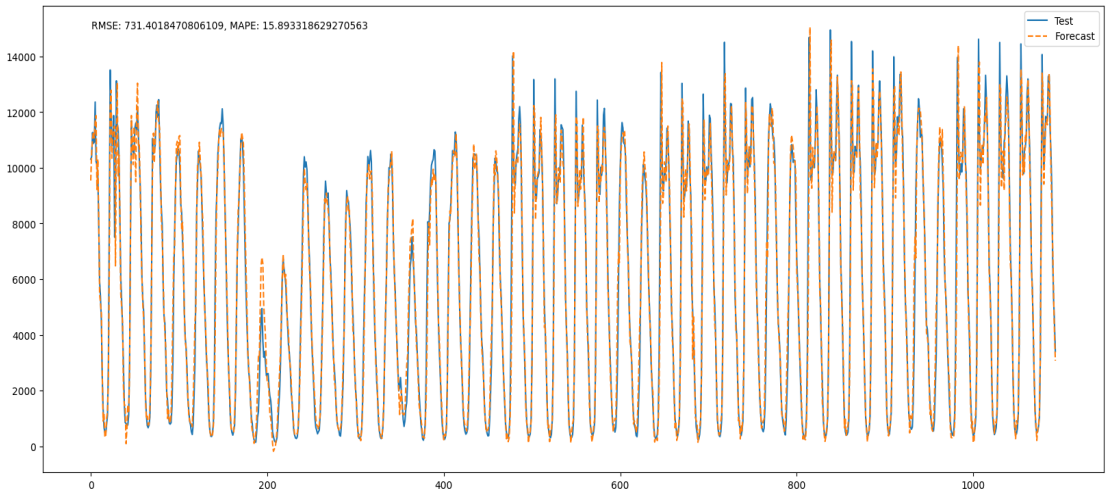
```
linear_model=cross_val
linear_model.fit(X_train,y_train)
(cross_val.best_score_,cross_val.best_params_)
```

✓ 0.4s

```
(0.9618473005720954, {'Transformer2__n_components': 70})
```

We get a higher training validation score than XGB.

4. Let's look at how well the model performs on test data



The model does atleast as well as XGB, and a lot better than the ARIMA model. Like XGB, it can forecast the peaks and valleys with reasonable accuracy. It also has a low MAPE of 16.

3.3 Model Comparison

The below table shows the model training time and RMSE:

Model	Training Time [minutes]	MAPE	RMSE
ARIMA (27,1,5)	0.5	199	3677
ARIMA (27,1,50)	8.75	147	2433
ARIMA (27,1,50) with expected sensor value	8.75	80	2205
XGBoost Regressor	5.0	33	878
Linear Regressor	0.4	16	731

Chapter 4. Conclusions

We've obtained and transformed the data, and trained an ARIMA, XGB Regressor and Linear Regressor model to forecast the traffic data.

Based on the model summary, we find the Linear Regressor to outperform XGB and ARIMA in both training time and forecasting accuracy. It also meets the benchmark of 80% accuracy we set in the proposal. This makes the Linear Regression Model a clear choice for the task.

Chapter 5. Discussion

While performing the data analysis and modeling, there were several challenges to overcome as well as identifying some areas of future work.

5.1 Challenges

1. The resolution of data was the same as what was trying to be predicted. Higher frequency data could give us more accurate predictions and better resolution.

2. There were some apparent discrepancies between the sensors data from the dataset and what was mentioned on the website – the website mentioned sixteen, but the data had labels for around 40 sensors

3. Determining the parameter to model traffic with was a major challenge, and involved some thought and research

4. The time complexity of the ARIMA model was a significant hinderance to effective ARIMA modeling, achieving models with higher 'q' values took unreasonable amounts of time

5. Finding the optimum set of parameters for the XGB pipeline took some trial and error

5.2 Suggestions for Improvement

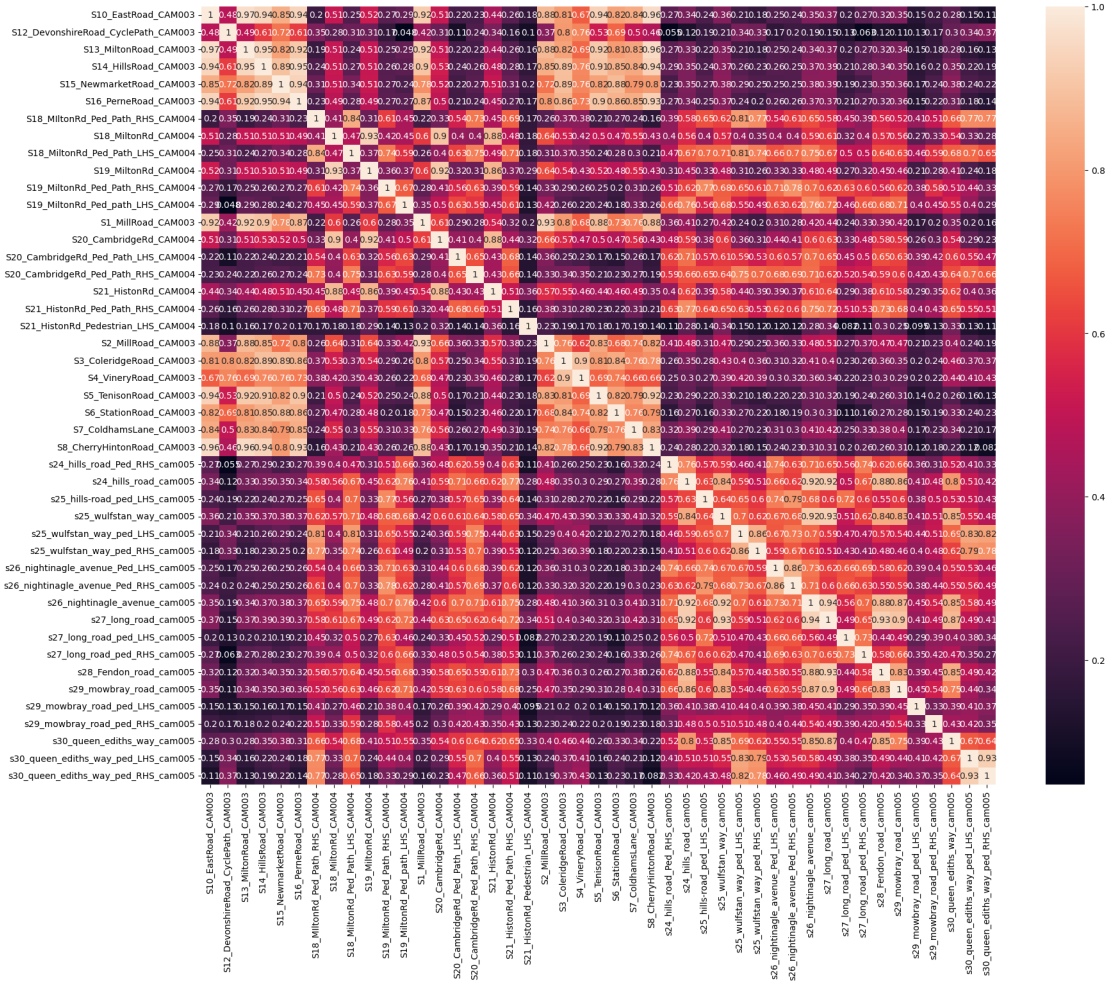
1. More complex ARIMA parameters could be investigated, and better strategies for data reduction and augmentation might help with this. For example, implementing a strategy that involves modeling for the specific hour of the day over the period of observation, to augment the overall ARIMA model.

2. Exploring strategies to reduce the training time of ARIMA and XGB

3. The model performance would need to be continuously monitored over time with actual

data to see if re-tuning is needed

4. The reliance on several sensors can also be reduced by considering the effect of correlated sensors. This can be a strategy to monitor the intersection / streets with less cost.



The strong correlation of the sensor data over such a long period suggests a reliable trend, even if it does not directly imply a causal relation. Measurements at regular intervals can help determine if the trend is still holding over time. Another thing to consider is short-term deviations.