# Machine learning for Intelligent Transportation Systems

## Traffic Volume Prediction Using Time Series Forecasting Machine Learning Models and Analysis

*Submitted By: Ahmed, Haseeb*
*Email: haseebravian756@gmail.com*

Budapest University of Technology and Economics

## TABLE OF CONTENTS

# INTRODUCTION

The concept of smart cities is more important than ever as cities grow and become denser due to urban ization and a growing world population. Smart cities can intelligently serve different kinds of needs, such as B. daily life, environmental protection, public security and city services, industrial and commercial activities. Among the highprofile goals of smart cities, building intelligent transportation systems coudhave a major impact on future urban dwellers. Advanced Traffic Management Systems (ATMSs) and Intelligent Transport Systems (ITSs) integrate technologies such as information and comm unication and apply them to build an integrated system of people, roads and vehicles

## Analysis of Machine Learning Models

A fundamental challenge in ATMSs and ITSs is to accurately predict the likely next traffic situation. This information helps prevent unlikely events such as traffic jams and other road anomalies. Therefore, this project focuses on researching prediction methods and techniques suitable for intelligent transportation systems using various sensor information. For this purpose, machine learning methods are analyzed and their performance on given data set will be compared as a conclusion.

*Source: www.analyticssteps.com*

# ML MODELS USED

- Multi-Layer Perceptron
  (Artificial Feedforward Neural Network)

- Support Vector Regression Model (SVR)

# DATA PREPROCESSING

The figure below shows the initial raw data that was received from the sensors.

| date | 6908 | 6909 | Total vol | 6908-mis% | 6909-mis% | Total-mis% |
|---|---|---|---|---|---|---|
| 5/25/2022 | 17815 | 22927 | 40742 | 0.1 | 0.1 | 0.1 |
| 5/26/2022 | 18711 | 24326 | 43037 | 0.3 | 0.3 | 0.3 |
| 5/27/2022 | 18636 | 24927 | 43563 | 0.1 | 0.1 | 0.1 |
| 5/28/2022 | 14784 | 19127 | 33911 | 0.1 | 0 | 0.1 |
| 5/29/2022 | 13187 | 17244 | 30431 | 0 | 0.1 | 0.1 |
| 5/30/2022 | 12265 | 15823 | 28088 | 0.1 | 0.1 | 0.1 |
| 5/31/2022 | 18782 | 24350 | 43132 | 0.1 | 0.1 | 0.1 |
| 6/1/2022 | 18883 | 24823 | 43706 | 0.2 | 0.2 | 0.2 |
| 6/2/2022 | 18876 | 24825 | 43701 | 0.1 | 0.1 | 0.1 |
| 6/3/2022 | 20268 | 26422 | 46690 | 0.2 | 0.2 | 0.2 |

.

Libraries used:

```
import numpy as np
import pandas as pd

from pylab import rcParams
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import csv
from pickletools import optimize
from turtle import color
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing

import seaborn as sns
from pylab import rcParams

from matplotlib import rc
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam


%matplotlib inline
%config InlineBackend.figure_format='retina'
sns.set(style='whitegrid', palette='muted', font_scale=1.5)
rcParams[ "figure.figsize" ] = 12, 8
```

Data pre-processing helps to prepare the raw data for machine learning algorithms and analysis and ultimately increases accuracy. It involves removal of noise, false positive, missing values etc. from the data.

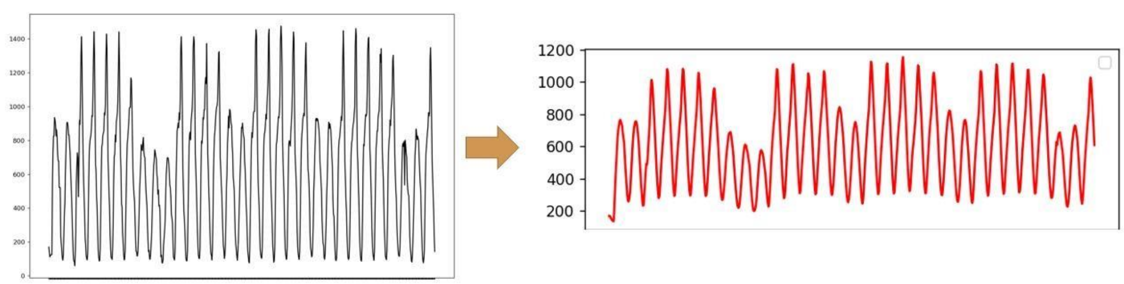
I used backward and forward fill approach to remove missing values (NaN) from data, as well as date and time columns were merged into one.

| 6908 | DateTime | Noise_Removed_Volume |
|---|---|---|
| 170 | 6/25/2022 1:00 | 170 |
| 141 | 6/25/2022 2:00 | 170 |
| 111 | 6/25/2022 3:00 | 164.2 |
| 118 | 6/25/2022 4:00 | 153.56 |
| 125 | 6/25/2022 5:00 | 146.448 |
| 125 | 6/25/2022 6:00 | 142.1584 |
| 125 | 6/25/2022 7:00 | 138.72672 |
| 550 | 6/25/2022 8:00 | 135.981376 |
| 717 | 6/25/2022 9:00 | 218.7851008 |
| 825 | 6/25/2022 10:00 | 318.4280806 |
| 825 | 6/25/2022 11:00 | 419.7424645 |

Further Exponential smoothing was applied to remove noise from the data.

Exponential smoothing is a time series forecasting method for univariate data. I used Simple Exponential smoothing function from “statsmodels.tsa.api” library , where smoothing factor or alpha was set to 0.2

The figure shows preprocessed data after noise removal.




```
# ---------------------  exponential smoothing (Removes Noise) -----------------
# -----MOving average ------

# ---- rolling average calculation
rollingseries = final_Data['6908'].rolling(window=10)

# THE GREATRE WINDOW SIZE< THE MORE THE SMOOTHNESS

rollingmean = rollingseries.mean()
print(rollingmean)
# ----------------------------------------------------------------------------
# ---- Exponential Smoothing Filter
data = final_Data['6908']
fit1 = SimpleExpSmoothing(data).fit(smoothing_level=0.2, optimized=False)
# -- Saving final smoothed data in a new data frame and then to new .csv file
final_Data['Noise_Removed_Volume'] = fit1.fittedvalues
final_Data.to_csv('./NoiseRemovedData.csv')
plt.plot(final_Data['DateTime'],
         final_Data['6908'], color='black')
plt.show()
plt.subplot(2, 1, 1)
plt.plot(fit1.fittedvalues, color='red')
```

# MULTI-LAYER PERCEPTRON

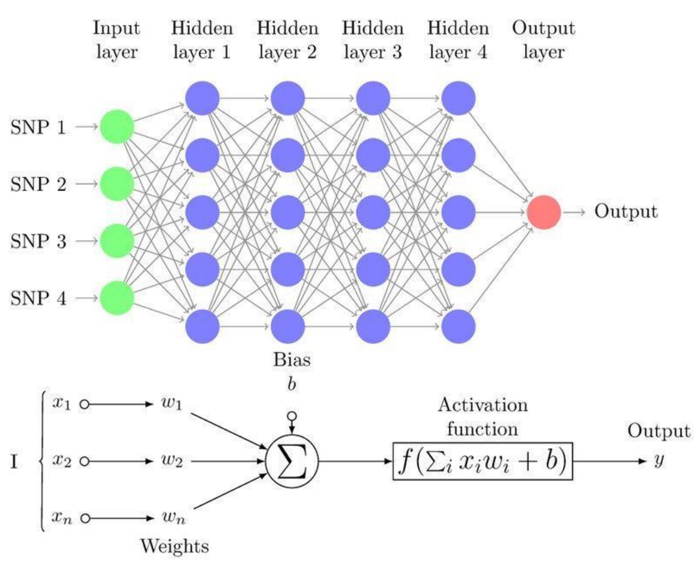
Multilayer Perceptron is a *feedforward Artificial Neural Network (ANN)* that learns the relationship between linear and non-linear data.

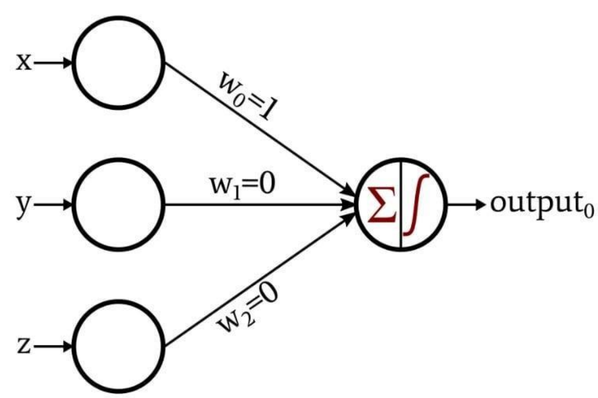
An MLP consists of at least three layers of nodes:

- input layer
- Hidden layer
- Output layer

Except for input nodes, each node is a neuron with a nonlinear activation function. MLP uses a supervised learning technique called backpropagation for training. Its complication and nonlinear activation distinguish MLP from linear perceptrons. You can distinguish between data that are not linearly separable.

After Loading the data further actions are taken to prepare the data for a specific model or algorithm.




*Figures sources: www.researchgate.net , [www.allaboutcircuits.com](http://www.allaboutcircuits.com) respectively* In the code block below, the same value of 'count' attribute from data.describe() function and and number of rows from data.shape confirms that there are no missing values.

```
data.shape
```

```
(744, 1)
```

```
data.describe(include='all')
```

| | Noise_Removed_Volume |
|---|---|
| count | 744.000000 |
| mean | 609.141547 |

The data is divided into 70% training set and 30& test set as shown below.

```python
from sklearn.model_selection import train_test_split

train,test = train_test_split(data_Series, test_size = 0.3) #test size is 30% and training is
```

Moving on, both training and test sets are changed into input-output samples where in this case, for every there 12 inputs there is 1 output. The model will use these sample to train itself.

```
[170.         170.         164.2        153.56       146.448
 142.1584     138.72672    135.981376   218.7851008  318.42808064
 419.74246451 500.79397161] 587.6351772876801
[170.         164.2        153.56       146.448      142.1584
 138.72672    135.981376   218.7851008  318.42808064 419.74246451
 500.79397161 587.63517729] 654.1081418301442
[164.2        153.56       146.448      142.1584     138.72672
 135.981376   218.7851008  318.42808064 419.74246451 500.79397161
 587.63517729 654.10814183] 702.0865134641153
```

Using tensorflow.keras, the multilayer-perceptron model is defined , and its hyperparameters(optimiser, loss, metrics etc.) are tuned according to the data. model.add() adds a layer in neural network while Sequential() groups linear stack of layers into a tendorflow.keras.model

```python
model = Sequential()
model.add(Dense(500, activation='relu', input_dim=n_steps))
# model.add(Dense(400, activation='relu'))
model.add(Dense(1))

model.compile(optimizer ='adam', loss='mse',metrics=['categorical_accuracy'])
```

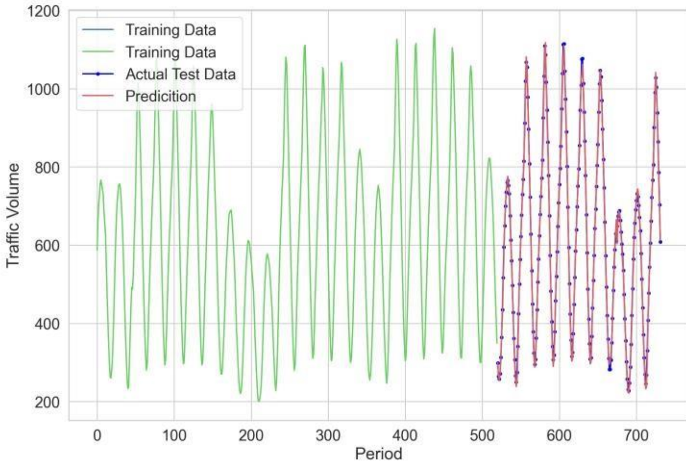
```python
# history = model.fit(X_train, y_train,batch_size=32, epochs=800,validation_split=0.33,verbose=2, callbacks=[es])
history = model.fit(X_train, y_train,batch_size=32, epochs=800,validation_split=0.33,verbose=2)
```

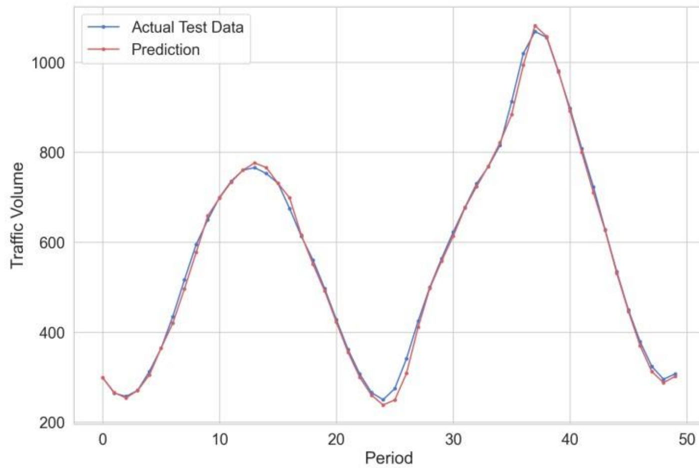
The model will iterate through the data 800 times i.e. 800 epochs.

```python
y_pred= model.predict(X_test, verbose=2)
```

```
11/11 - 0s - 63ms/epoch - 6ms/step
```

After the model is trained and predictions are checked on test set, the results are below, where the right figure is a zoomed version of the prediction on the left



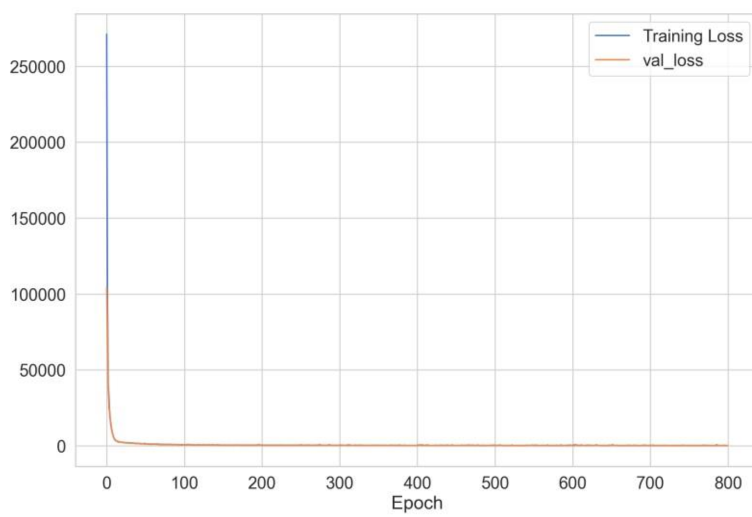




Errors and overfitting or underfitting checks tell us about the performance of our model.

Mean Squared Error: 178.337

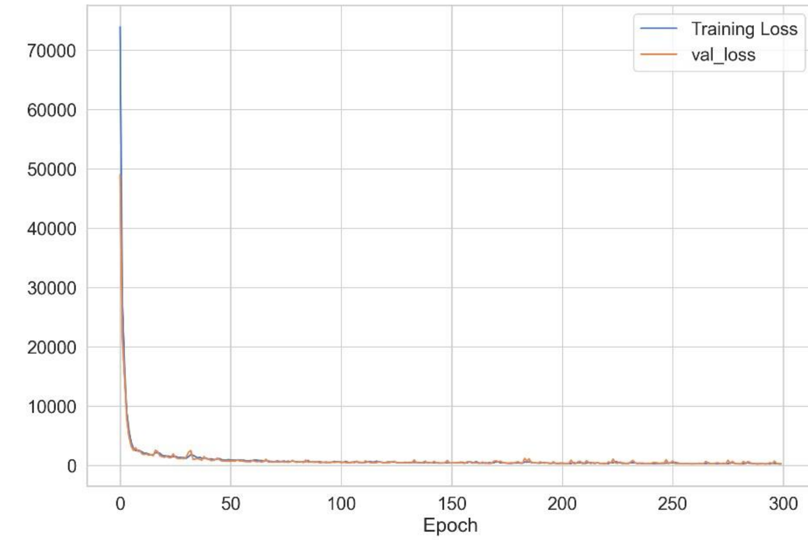
Mean Absolute Percentage Error: 1.870 %

Training data loss plotted against Test data loss, shows that model is not really over-fitting but the accuracy is not so good because model takes around 800 epochs and is not very fast.




We can use early stopping before training the model. It will stop the model before it starts over fitting and consequently reduce the number of epochs as well.

```
# simple early stopping
from tensorflow import keras
from keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
```



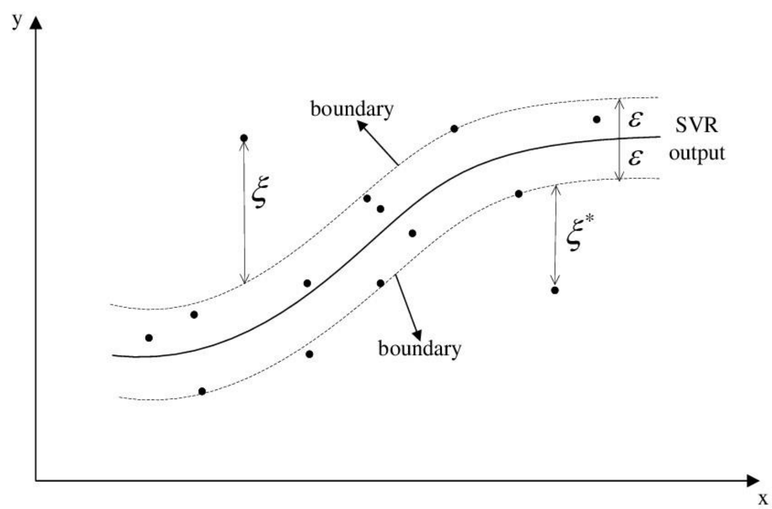

The number of epochs reduced from 800 to 300, on the cost of 1% increase in mean absolute percentage error.

# SUPPORT VECTOR REGRESSION (SVR)

Regression analysis helps us to understand how the value of the dependent variable is changing corresponding to an independent variable when other independent variables are held fixed.

Support vector regression is a supervised learning algorithm used to predict discrete values. Supports vector regression uses the same principle as SVM. The basic idea behind SVR is to find the most suitable line.

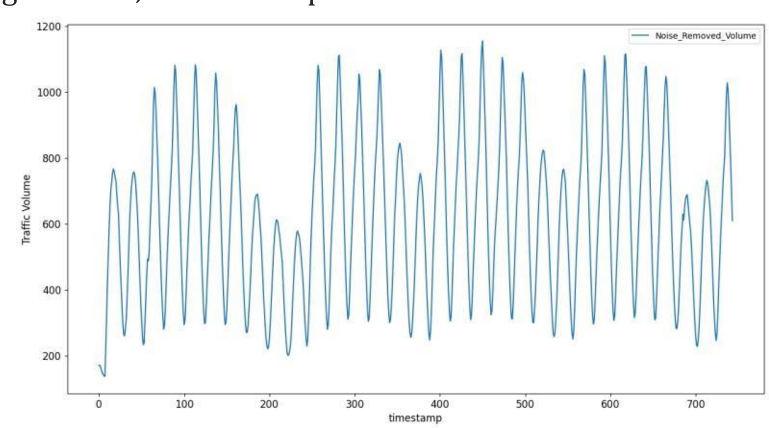
In SVR, the best fit line is the hyperplane with the maximum number of points.




Source:

- Allows to choose error tolerance (**ϵ**)
- Tolerance for values falling outside acceptable error frame (C) or $\xi^*$

After successfully loading the data, the next step is to visualize this data.




The libraries used and defined are:

```python
import os
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import datetime as dt
import math

from sklearn.svm import SVR
from sklearn.preprocessing import MinMaxScaler
# from common.utils import  mape
from sklearn.metrics import mean_absolute_percentage_error
```

The data is divided into 70% training set and 30% test set as shown below.

```python
train_start_dt = 0
test_start_dt = 521
```

```python
train = data.copy()[(data.index >= train_start_dt) & (data.index < test_start_dt)][['Noise_Removed_Volume']]
test = data.copy()[data.index >= test_start_dt][['Noise_Removed_Volume']]

print('Training data shape: ', train.shape)
print('Test data shape: ', test.shape)
```

```
Training data shape:  (521, 1)
Test data shape:  (223, 1)
```

Following, the train-test sets is divided into 2D tensor where for every 4 inputs, there is 1 output.

```python
# Converting data to 2D tensor

train_data_timesteps=np.array([[j for j in train_data[i:i+timesteps]] for i in range(0,len(train_data)-timesteps+1)])[:,:,0]
train_data_timesteps.shape
```

```
(517, 5)
```

```python
print(train_data_timesteps[0:5])
```

```
[[0.03340174 0.03340174 0.02770692 0.01725986 0.01027683]
 [0.03340174 0.02770692 0.01725986 0.01027683 0.00606501]
 [0.02770692 0.01725986 0.01027683 0.00606501 0.00269556]
 [0.01725986 0.01027683 0.00606501 0.00269556 0.        ]
 [0.01027683 0.00606501 0.00269556 0.         0.08130219]]
```

SVR() function is used from “sklearn.svm” pckage to define the regression based model. In hyperparameters

- Kernel defines the transformation function (non-linear in this case)

- Epsilon defines error tolerance (ϵ)
- 'C' defines Tolerance for values falling outside acceptable error frame • Gamma is Kernel coefficient and defines the reach of kernel

```
# Create model using RBF kernel

model = SVR(kernel='rbf',gamma=0.5, C=10, epsilon = 0.05)
```

```
# Fit model on training data

model.fit(x_train, y_train[:,0])
```

```
SVR
SVR(C=10, epsilon=0.05, gamma=0.5)
```

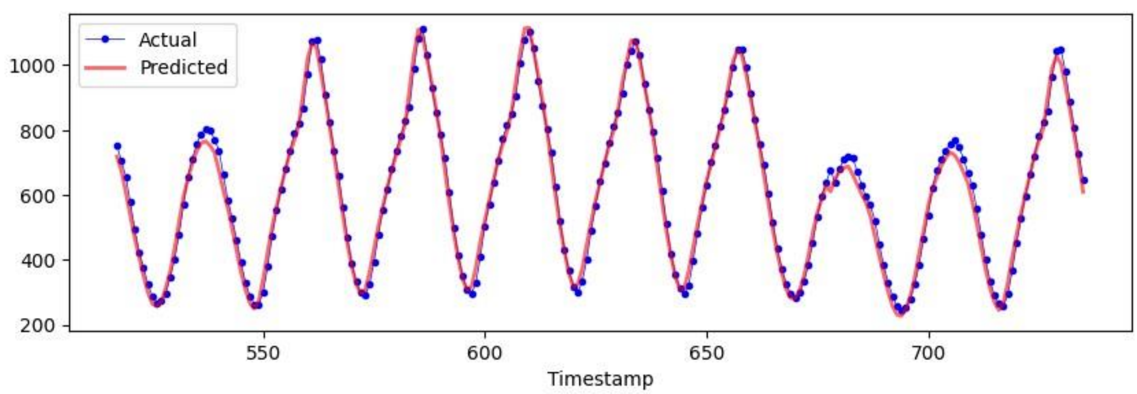
```
# Making predictions

# y_train_pred = model.predict(x_train).reshape(-1,1)
y_test_pred = model.predict(x_test).reshape(-1,1)

print( y_test_pred.shape)
```

```
(219, 1)
```

After tuning the model on the above defined hyperparameters, with the minimum error, we get the following prediction:




Mean Squared Error: 694.67
Mean Absolute Percentage Error: 4.19373217 %

# RESULTS

The training speed of MLP is less than SVR. But mean absolute percentage error of MLP (1.870 %) is approximately 3 times less than SVR (4.19373217 %) for our data. This shows that MLP model or neural networks is a better and optimal solution for traffic volume prediction for our defined data.

In general, I would conclude:

- SVM/SVR are good for data with less dimensions (features) while become overwhelmed with big sized data
- Neural networks work well with big data especially with modern hardware computing resources

| | Mean Squared Error | Mean Absolute % Error | Training Speed |
|---|---|---|---|
| MLP | 178.337 | 1.870 % | 85 |
| SVR | 694.67 | 4.19373217 % | 85 |