

Identifying Machine Learning Algorithms to Predict Pollution, Weather, and Traffic Conditions for Smart City Applications

Final Report

Muhammad Hasif Bin Muhammad Uzir

Supervisor: Dr Miguel Rodrigues

Second Assessor: Dr Christos Masouros

March 2018

Abstract

A smart city is an area with an urban population that leverages technology to manage resources and assets using information obtained from various electronic sources to efficiently improve quality of life. An interesting development is the increased interest in utilising machine learning to process large amounts of data to identify trends and predict the future. However, there is the question of how to leverage multiple data sources to predict relevant parameters, what algorithms are suitable and which data sources can be useful. These are all emerging problems involved in the prediction various parameters such as pollution levels, traffic conditions and weather. The objective of this research project is to explore the relationship between data inputs, strength of various algorithms and develop a machine learning algorithm that can combine multiple data sources in order to predict the future. This can then be used to leverage real-time data and used in the real-world.

First, the performance of a system with a single input is compared against a system with multiple inputs to identify whether there exists a correlation between data. Then, the performance of four algorithms are compared to determine suitability for smart city applications. The better algorithm is then used in a system to leverage real-time data as an application to prove real-world suitability. It was found that a system with multiple inputs performed better than as system with only a single input, suggesting a correlation between inputs. Thus, in the future, it is better for smart cities to leverage various data types to better predict parameters.

The performance of four different methods in time-series predictions are compared, which are Moving Average, Autoregressive Integrated Moving Average, Recurrent Neural Network and Sequence-to-Sequence (Seq2seq). It was concluded that the Deep Learning methods provided far better performance especially the Seq2seq model, with less errors committed. However, leveraging machine learning required a significant amount of computing power and the system requires heavy training and as well as further development for better predictions.

In conclusion, a system utilising the Seq2seq algorithm with multiple inputs will provide the best performance in predicting pollution, weather and traffic patterns in a smart city. It is also possible to develop a system leveraging real-time data for applications in the real world, however further development is required.

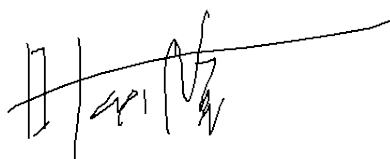
Declaration

I have read and understood the College and Department's statements and guidelines concerning plagiarism.

I declare that all material described in this report is all my own work except where explicitly individually indicated in the text. This includes ideas described in the text, figures and computer programs.

Name: Muhammad Hasif Bin Muhammad Uzir

Signature:

A handwritten signature in black ink, appearing to be 'Muhammad Hasif Bin Muhammad Uzir', written over a horizontal line.

Date: 6 March 2018

Table of Contents

	Page
Abstract	ii
Declaration	iii
Introduction	1
Literature Review	3
Theory	5
<i>Moving Average</i>	5
<i>Autoregressive Integrated Moving Average</i>	5
<i>Recurrent Neural Network</i>	6
<i>Sequence-to-Sequence</i>	7
Methodology	8
<i>Software</i>	8
<i>Datasets</i>	8
<i>Comparing Single Inputs Against Multiple Inputs</i>	12
<i>Machine Learning Algorithms Comparison</i>	14
<i>Real-world Applications</i>	16
Results and Analysis	17
<i>Comparing Single Inputs Against Multiple Inputs</i>	17
<i>Machine Learning Algorithms Comparison</i>	24
Real-world Applications	28
Conclusion	29
References	30
Appendix	31
A. <i>Moving Average Model Python Source Code</i>	31
B. <i>ARIMA Model Python Source Code</i>	32
C. <i>RNN Model Python Source Code</i>	34
D. <i>Seq2seq Model Python Source Code</i>	37

Introduction

Machine Learning allows computers to discover patterns and learn without being explicitly programmed to do so. Smart Cities, are urban areas utilising electronics and data to manage its resources efficiently. Today, especially with the advent of Big Data and Internet of Things (IoT), the role of Machine Learning becomes increasingly important to allow us to sift through vast amounts of data to better predict the future.

For efficient utilisation of Machine Learning in a Smart City, several emerging problems exist. First, are multiple data sources useful in predicting relevant smart city parameters such as pollution, traffic and weather? If so, how can they be leveraged? Is there a correlation between data to allow better predictions? There is also the question of what algorithms are suitable for a time-series prediction of future results. This is important because as Big Data becomes increasingly widespread, there exists potential to allow more efficient applications in a Smart City using Machine Learning, such as allocating resources or simply letting citizens know when a good time to avoid rainfall is.

There have been many studies suggesting the potential of Machine Learning for usage in smart cities and leveraging Big Data. The Literature Review explores a selection of studies done in the field of Smart Cities and Machine Learning.

In this report, machine algorithms for the purpose of smart cities, will be explored in terms of its inputs, the algorithms and real-world applications. First, a comparison is done between a single input system against a multiple input system is done to identify the differences in performance as well identify if there exists a correlation between data types typically obtained from a city. Then, the performances of four different algorithms are compared to identify which is best suited for smart city applications and their individual characteristics. Next, having identified the strengths of different algorithms, a machine learning algorithm best suited to predict parameters in a smart city is developed. Finally, the system will be modified to utilise real-time data. This serves as a proof-of-concept for a real-world application.

In the rest of the report, relevant literature and studies will be explored and compared. Then, a section on theory is presented to familiarise the reader with theory essential to the report. The Methodology section explains all tests done in detail which is then discussed in the Results and Analysis section. The Methodology also provides a detailed plan on completing the report's entire proposed studies. The Real-world Applications section will provide an insight into utilising machine learning for real-time predictions. Finally, the Conclusion will summarise all findings as well answer relevant questions regarding the studies done and provide insight into future applications.

Literature Review

Deryckere [1] explores the potential of machine learning in a smart city and discovered that it can be useful in almost every aspect in city infrastructure. Machine learning allows patterns in both structured and unstructured data to be recognised and create an optimised solution. However, potential may be limited due to the fact that a certain model only fits a predetermined problem. He posits that the issue can be overcome through the development of Cognitive Computing, however this is beyond the scope of this paper. However, he doesn't explore various algorithms that exist, which could provide sufficient accuracy and suitability for a smart city. An interesting point in his dissertation, he introduces consumer tools for implementing Machine Learning which can be easy to use. The ability for consumers to leverage the power of Machine Learning is a recent development due to the exponential increase in the computing power of consumer hardware. This paper in particular, uses TensorFlow, an open source library from Google to create Machine Learning systems.

It was found that there is a strong correlation between weather-based attributes and traffic data [2] in a paper by Chin, Callaghan and Lam. The degree of correlation was left unexplored. In the paper, the relationship between bicycle hires in London and rainfall was analysed to provide an idea on the role of Big Data and Machine Learning for smart cities. This strengthens the arguments for the usage of Machine Learning as it is able to determine the relationship between seemingly unrelated datasets. However, in the paper, classifier algorithms were used, which classified rainfall based on bicycle hires. For time-series predictions, regressions are usually used as it takes continuous variables instead of class labels. Thus, the performance comparison between algorithms provides little insight for the purpose of this paper.

A paper by Zickus, Greig and Niranjana [3], presents a comparison between four machine-learning methods; Logistic regression, decision tree, multivariate adaptive regression splines and neural network in an attempt to offer machine learning as a practical alternative to deterministic and statistical methods in predicting air pollution concentrations in Helsinki, Finland. It was discovered that with the exception of the decision tree, which was described as significantly inferior, the performance of the other three models were similar. It was noted that the Neural Network was prone to overfitting and sensitive to noise in data due to its flexibility. It was the only method to identify precipitation, which may significantly affect PM10 concentrations in the atmosphere as an important input feature. This implies the neural network was more adept in more complex input variable selection, an important behaviour when dealing with big data for smart cities. However, it was noted that that made it more sensitive to errors in the input data which meant data processing was equally important to training the model. Ultimately, the researchers concluded machine learning methods as a powerful analysis tool in

situations where the underlying equations are unknown or convoluted. Using machine learning, air quality forecasts were not only improved, factors affecting atmospheric concentrations were also better understood such as the case of precipitations. Other than several small differences, the performances of a logistic regression, multivariate adaptive regression and neural network was similar and up to whichever method is more easily interpreted. This paper provided valuable insight into neural networks as it is one of the methods explored in this project.

References

- [1] N. Deryckere, "What is the Potential of Machine Learning in a Smart City?", Undergraduate, Howest, University College West Flanders, 2016.
- [2] J. Chin, V. Callaghan and I. Lim, "Understanding and personalising smart city services using machine learning, The Internet-of-Things and Big Data", IEEE, Edinburgh, 2017.
- [3] M. Zickus, A. Greig and M. Niranjana, "Comparison of Four Machine Learning Methods for Predicting PM10 Concentrations in Helsinki, Finland", Water, Air and Soil Pollution: Focus, vol. 2, no. 56, pp. 717-729, 2002.

Theory

Moving Average

A Moving Average or rolling mean is a calculation model used to analyse several data points, usually those of in time series. A series of averages of different parts of the dataset is calculated and recalculated as more data points are added to the complete dataset. A fixed subset of a fixed size is used to obtain and calculate the average of a series, the subset is then shifted in time by excluding the first data and including the next data in the series. [1] The formula for a simple moving average is given below:

$$M.A = \frac{a_1 + a_2 + a_3 + a_4 + \dots + a_n}{n}$$

Equation 1 Simple Moving Average Formula

where a are values in a time series, n is size of subset

Thus, the variations in the mean are affected by variations in the data. It has an advantage of determining and highlighting long-term trends by smoothing out short-term fluctuations.

Autoregressive Integrated Moving Average

The Autoregressive Integrated Moving Average (ARIMA) model is a statistical model for forecasting time series data. By examining the differences between values in the series instead of the actual values, future trends can be predicted through calculations. The model may include autoregressive and moving average terms. In regards to the autoregressive terms, data from the time series is used to predict data in the same time series. The variable of interest is regressed on its prior values in the dataset. The moving average terms are used to predict future values within the same time series. Integrated means that actual data values are not used and instead differences between the actual values and previous values are referenced. Using the various features of the model, the model can fit data as well as possible. A typical non-seasonal ARIMA model is denoted below [2]:

$$\text{ARIMA}(p, d, q)$$

Where p is the order of the AR model, d is the degree of differencing, and q is the order or the MA model

Recurrent Neural Network

Recurrent Neural Networks (RNN) are a class of Artificial Neural Networks (ANN) able to make use of sequential information. Compared to traditional ANN, which uses scalar inputs for learning and assumes independency between all inputs and outputs, RNNs learn from a sequence of scalar inputs.

RNNs are recurrent because in every element of the sequence, the same task is performed and the output is dependent on previous calculations. This allows the network to exhibit memory-like properties, reusing information obtained from the sequence. [3] A RNN contains loops which allow information to be reused as shown below:

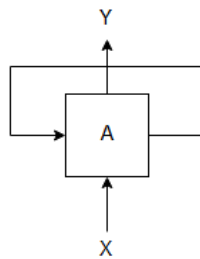


Figure 1 Rolled Recurrent Neural Network depicting the output being “looped” back into the network.

If the RNN is unrolled or unfolded into a full network, the full operation can be seen below. As new inputs from the sequence is obtained, each layer receives the output of the previous layer’s calculations based on previous inputs.

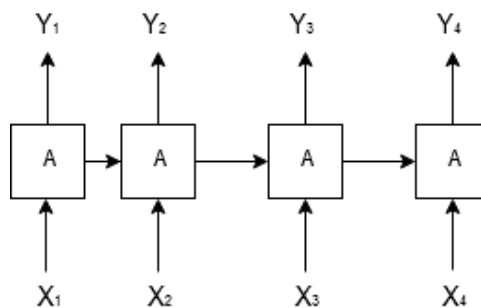


Figure 2 Unrolled Recurrent Neural Network, allowing the network to simultaneously obtain new inputs and utilise the computations based on previous inputs.

Outputs from each calculation is compared to test data (a subset of original data used in training) to obtain an error rate. After comparison, backpropagation is undergone to adjust the weight of the network and ensure optimal learning.

The most commonly used type of RNNs are Long Short-Term Memory (LSTM). Each LSTM is a building block for layers of a RNN. Internally, each LSTM cell decide what information is necessary to retain which is then combined with the previous state, current memory and input. LSTMs are suitable at modelling short-term memory which can last for a long period of time.

Sequence-to-Sequence

Sequence-to-Sequence (seq2seq) is a model utilising two connected Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNN). It is also known as an RNN Encoder-Decoder. Seq2seq performs well in a variety of tasks such translation and image captioning. The Seq2seq model maps arbitrary-length sequences to another arbitrary-length sequence. [4] This solves the spatial problem of other architectures requiring a fixed sized for the input and outputs.

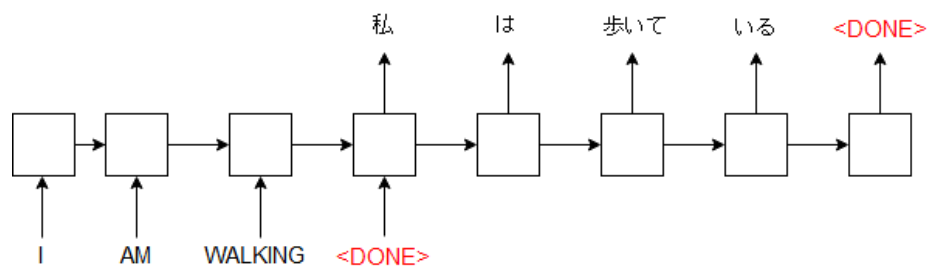


Figure 3 A basic diagram of the seq2seq model with Encoder RNNs on the left and Decoder RNNs on the right, translating an English sentence into Japanese

Based on the diagram above, two RNNs are connected together, with an Encoder RNN on the left, and the Decoder RNN on the right. The encoder RNN receives the entire input sequence, encoding them into a fixed-length vector until a final token is received, signifying the end of the sequence. The Encoder RNN's internal state will change as it receives each input until the final token is received. The vector is then passed to the Decoder RNN to map into an output sequence. The Decoder RNN's internal state also changes as it decodes the sequence. Finally, another final token is outputted, signifying the end of the output sequence.

This model is used especially for the *Comparing Single Inputs Against Multiple Inputs* part of the project. Seq2Seq is used to build a time series forecasting model to predict parameters.

References

- [1] I. Lurie, "How do I calculate a rolling average?", Portent, 2009. [Online]. Available: <https://www.portent.com/blog/analytics/rolling-averages-math-moron.htm>.
- [2] "Moving Average Models (MA models)", Applied Time Series Analysis, 2018. [Online]. Available: <https://onlinecourses.science.psu.edu/stat510/node/48>.
- [3] D. Britz, "Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs", WildML, 2018. [Online]. Available: <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>.
- [4] D. Nag, "seq2seq: the clown car of deep learning – Dev Nag – Medium", Medium, 2018. [Online]. Available: <https://medium.com/@devnag/seq2seq-the-clown-car-of-deep-learning-f88e1204dac3>.

Methodology

Software

For the purpose of this research project, the Python language and several libraries are used to develop, run and analyse machine learning algorithms. The most important library used is TensorFlow, an open source library used in various applications and especially in machine learning and neural networks. Computations are done using an architecture involving nodes, which represent operations and tensors representing arrays of values. A computational graph is then created consisting of nodes which pass tensors from one another. Other libraries used are pandas, Matplotlib, NumPy and Seaborn.

Datasets

Next, datasets suitable towards the application of smart cities are used. These will be used as the inputs to be used in the machine learning system to predict parameters. Three types of data categories have been identified for use; Pollution, Weather, Traffic and Time. All data are hourly recorded data between 31st May 2013 to 31st May 2017.

The location used in all datasets will be restricted to Marylebone Road, London. This location was used due to the availability of useful datasets compared to other locations in London. Marylebone Road is classified as a roadside site, meaning that there is a road within 5 meters of measurements recorded.

Pollution data was obtained from the London Air Quality Network (LAQN) [1][2] and the Automatic Urban and Rural Network (AURN) [3]. From the datasets, several species of pollutants are available; Carbon Monoxide, Nitric Oxide, Nitrogen Dioxide, Oxides of Nitrogen, Ozone, PM 10 Particulates (by Tapered Element Oscillating Microbalance (TEOM) and Filter Dynamic Measurement System (FDMS)), and PM 2.5 Particulates (by FDMS), and Sulphur Dioxide. Below are details on the pollutant species:

Name	Units	Source
Carbon Monoxide	mg/m ³	LAQN
Nitric Oxide	µg/m ³	LAQN
Nitrogen Dioxide	µg/m ³	LAQN
Oxides of Nitrogen	µg/m ³	LAQN
Ozone	µg/m ³	LAQN
PM10 Particulates (by TEOM)	µg/m ³	LAQN
PM10 Particulates (by FDMS)	µg/m ³	LAQN
PM2.5 Particulates (by FDMS)	µg/m ³	LAQN
Sulphur Dioxide	µg/m ³	AURN

Table 1 Pollutants with units and sources

Weather data was obtained from the AURN [3]. Weather data includes Temperature, Wind Direction, and Wind Speed. Below are details on the data:

Name	Units	Source
Temperature	°C	AURN
Wind Direction	° (from North)	AURN
Wind Speed	ms ⁻¹	AURN

Table 2 Weather data with units and source

LAQN also provides weather data as well, however, the data is incomplete, with recorded data available only about 70% of the time for the chosen period of this project.

Currently, Transport for London (TfL) does not make hourly traffic counts readily available online however, the hourly traffic counts for Marylebone Road was able to obtained from a previous Freedom for Information Request [4]. The data source is a Patched Automatic Traffic Counter Database. The traffic data are the hourly counts for Marylebone Road running towards the East and West. Below are details on the data:

Name	Units	Source
East	-	TfL
West	-	TfL

Table 3 Traffic data with units and source

Time was also used as an input. For this project, the hours from each day were used. Since all previous data were hourly counts, time can be validly sourced from the LAQN, AURN or TfL datasets.

Name	Units	Source
Hour	Hour	LAQN/ AURN/ TfL

Table 4 Time data with units and sources

All data was then compiled into a single Comma-Separated Values (csv) file with each data type populating its own column. Each data type will then be used as the features and labels desired.

One thing to take note of is that all data have at least a few empty cells, which suggest that data was not recorded for the specific hour. All empty cells were then populated with zeros. Below is a comparison on the completeness of each feature:

Name	Empty Cells	Validity
Carbon Monoxide	2160	93.85%
Nitric Oxide	474	98.65%
Nitrogen Dioxide	474	98.65%
Oxides of Nitrogen	474	98.65%
Ozone	897	97.44%
PM10 Particulates (by TEOM)	1481	95.78%
PM10 Particulates (by FDMS)	1517	95.68%
PM2.5 Particulates (by FDMS)	1084	96.91%
Sulphur Dioxide	2276	93.51%
Temperature	1386	96.05%
Wind Direction	1351	96.15%
Wind Speed	1350	96.16%
East	4	99.99%
West	4	99.99%
Hour	0	100%

Table 5 Comparison on data validity

Comparing Single Inputs Against Multiple Inputs

The first part of the project involves comparing the performance and accuracy of a system given a single type of input against multiple types of inputs to predict a single output. The main purpose is to discern whether there are correlations between different input types and whether having more features will lead to a better label prediction.

A python file is created to generate results for this study. The file will read the dataset csv file, extract relevant features, process them, build, train, run and test a TensorFlow model based on the Seq2seq architecture. This model will predict labels for the last 30 days of the time series and compare the results with the actual last 30 days.

First, all relevant modules are imported. Then, the csv file containing the data is read using pandas. Then, the last 30 days of data is split to be used for testing with the remaining data used for training.

Then, relevant columns are extracted to be used for the features and label. To test single inputs, only one column is used for both the feature and label. To test multiple inputs, multiple columns are chosen for the features and a single column is chosen for the label. The chosen data is then normalised via the z-score. This allows the comparison of two different features.

The training and test data is then transformed into 3D formats for time series use (size of batch, timestep, feature dimensions). Size of batch determines the size of each batch used in training and testing. Timestep determines the sequence length of inputs and outputs, and feature dimensions determine the columns used for both the features and label.

Two functions are created to facilitate the generation of training and testing samples. In the function to create training samples, random batches are sampled from the available data for training.

A Seq2seq model is then created which is able to take in any number of inputs and return a single output. A TensorFlow session is run for training. The model is then trained using guided training. During training the correct output is fed into the decoder at every time step. An example of this is presented below:

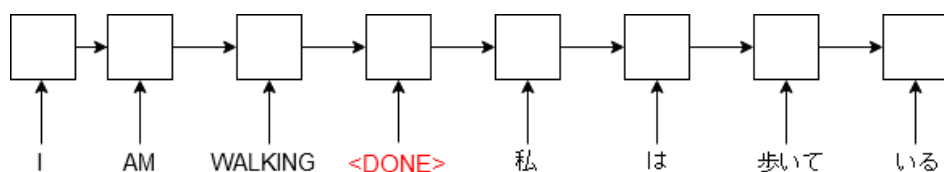


Figure 4 A Seq2seq model for training a model to translate an English sentence into Japanese.

The model is then tested using another TensorFlow session. During the testing, the true output is not fed into the decoder, however, the output of the decoder is fed back and becomes the decoder input at the next time step. An example of this is presented below:

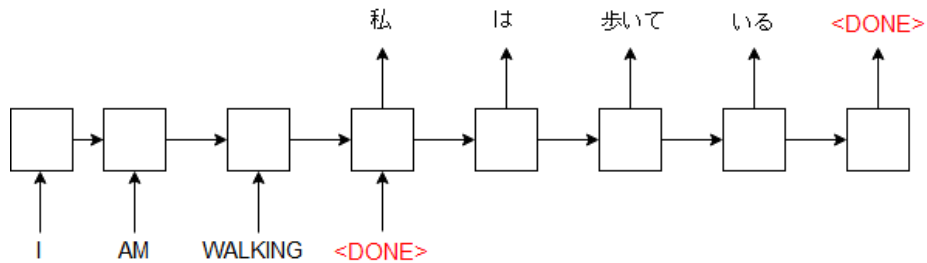


Figure 5 A Seq2seq model for testing, translating an English sentence into Japanese.

The Mean Squared Error (MSE) is then calculated from the results by comparing it to the true results. A new csv file is then created to save the results, containing the predicted and actual results to create graphs. A plot based on the results is also created using Matplotlib for quick comparison when running the file.

For the univariate case, the labels chosen are Nitric Oxide, East and Temperature. For the multivariate case, the labels chosen are same as above, however all data were used as features. Each instance is run five times, results recorded and the average MSE is obtained.

Machine Learning Algorithms Comparison

Next, the performance and accuracy of 4 different machine learning algorithms are compared to identify how each system differs from one another in smart city applications. The same datasets as well as hardware are used. The four algorithms used are Moving Average, Autoregressive Integrated Moving Average, Recurrent Neural Network and Sequence-to-Sequence.

Moving Average

A python file is created to generate results. The file will read the dataset csv file, process and calculate the moving average based on the data. The model will predict the hourly values for Nitrogen Dioxide (NO₂) pollution for the last 30 days of the time series and compare the results with the actual last 30 days.

First, the required libraries are imported which are pandas, matplotlib, and NumPy. Then, the csv containing the dataset is read using pandas and the relevant data is saved to a variable. A window of 24 hours is set which will be the number of values averaged. The dataset is split into two sets; Actual, which will be used in the prediction and Test, which will be used to compare the prediction's accuracy.

The last 30 days of the dataset is then predicted by running a rolling mean function. This is done by averaging the previous 24 values (signifying 24 hours) to predict the next hourly NO₂ level. This is repeated until 30 days' worth of predictions are generated. The mean squared error is then calculated.

The results are saved to a csv file for further analysis and the entire process is repeated 5 times to obtain the average MSE.

Autoregressive Integrated Moving Average

A python file is created to generate results. The file will read the dataset csv file, process the dataset and run an ARIMA model to predict the last 30 days of the hourly NO₂ levels. The results will be saved to a separate csv file and the mean squared error is calculated for comparison with other algorithms.

First the relevant libraries are imported the most important library being the ARIMA model library from statsmodels. The dataset is then read and saved to a new variable. The data is then split for running the ARIMA model and validation. The final 30 days are to be used as validation to compare with the predicted NO₂ levels.

A function is then created to remove any seasonal differences in the data. The function subtracts values from the same day a year ago to create a differenced series. The ARIMA model is then fit using the built-in functions from the imported libraries. A multi-step out-of-sample forecast is then run to obtain future NO₂ values.

The data is then resplit to only obtain the last 30 days of the series. The MSE is calculated and the results are then saved into a separate csv file for further analysis. The file is then run another 4 times and the average MSE is obtained.

Recurrent Neural Network

A python file is created to generate results. The file will read the dataset csv file, process the dataset and run an RNN to predict the last 30 days of the hourly NO₂ levels. The results will be saved to a separate csv file and the mean squared error is calculated for comparison with other algorithms.

First, the relevant libraries are imported as required. The dataset is then read using pandas and the date and NO₂ values columns are saved in different NumPy arrays. The data is then split into two for training and testing. The final 30 days will be used for testing and the previous days are used for training using TensorFlow.

The data is then normalised via a min-max normalisation. A graph is plot to ensure the data is properly normalised. The computational graph used for the neural network is then initialised using default parameters for a RNN.

A computational graph is then constructed using a Basic RNN Cell and basic functions native to TensorFlow. Batches of training inputs and outputs are then generated using a new function that is defined. 50 random values are taken as batches to be used in training. The RNN is the trained by running a new TensorFlow session.

Once the RNN is trained and loss is minimised, predictions for the last 30 days of readings are generated based on the previous session. The predictions are then denormalised and graphs are plotted to compare with the actual values of NO₂. The file is run for a total of 5 times to calculate and obtain the average MSE.

Sequence-to-Sequence

The python file from the previous section, Comparing Single Inputs against Multiple Inputs is modified to only predict NO2 levels. Based on the same dataset used by the previous 3 tests, the data is split, processed and used to train a Seq2Seq model. The final 30 days are then predicted based on the trained model and the MSE is calculated. The model is repeated another 4 times to calculate and record the average MSE.

Real-world Applications

Finally, an application for real-time prediction of parameters will be explored and demonstrated as a proof-of-concept. The system would leverage real-time data to predict future values. This would be useful for real-world applications. Further details are provided in the Real-world Applications section of the paper.

References

- [1] "London Air Quality Network: Marylebone Road", Londonair, 2017. [Online]. Available: <http://www.londonair.org.uk/london/asp/publicdetails.asp?site=MY1>.
- [2] "London Air Quality Network: Marylebone Road FDMS", Londonair, 2017. [Online]. Available: <http://www.londonair.org.uk/london/asp/publicdetails.asp?site=MY7>.
- [3] "London Marylebone Road - Air Quality England", Airqualityengland, 2017. [Online]. Available: http://www.airqualityengland.co.uk/site/exceedence?site_id=MY1.
- [4] "FOI request detail: Hourly Traffic Counts Data for London Marylebone Rd", Transport for London, 2017. [Online]. Available: <https://tfl.gov.uk/corporate/transparency/freedom-of-information/foi-request-detail?referenceId=FOI-0660-1718>.

Results and Analysis

Comparing Single Inputs Against Multiple Inputs

For each case, the average Mean Squared Error (MSE) of 5 runs will be used for analysis.

Nitric Oxide

The average MSE for the single input case is 0.467. The average MSE for the multiple input case is 0.276. This means that when multiple inputs are fed into the system, it predicted future Nitric Oxide values more accurately compared to when Nitric Oxide is the only feature. The multiple input system was 45.21% more accurate compared to the single input system.

Below are line graphs comparing the two cases. It can be observed from the first graph that the predicted values vary wildly for the first 190 hours. This can perhaps be attributed to abnormally low Nitric Oxide levels for that period, which the single input system was unable to predict. Compared to the second graph, the prediction was very close, with the only major mistake being the missing peaks around the 79-hour mark. For the rest of the month, the single input model tended to oscillate wildly while the multiple input model closely predicted the peaks and falls with less wild oscillations.

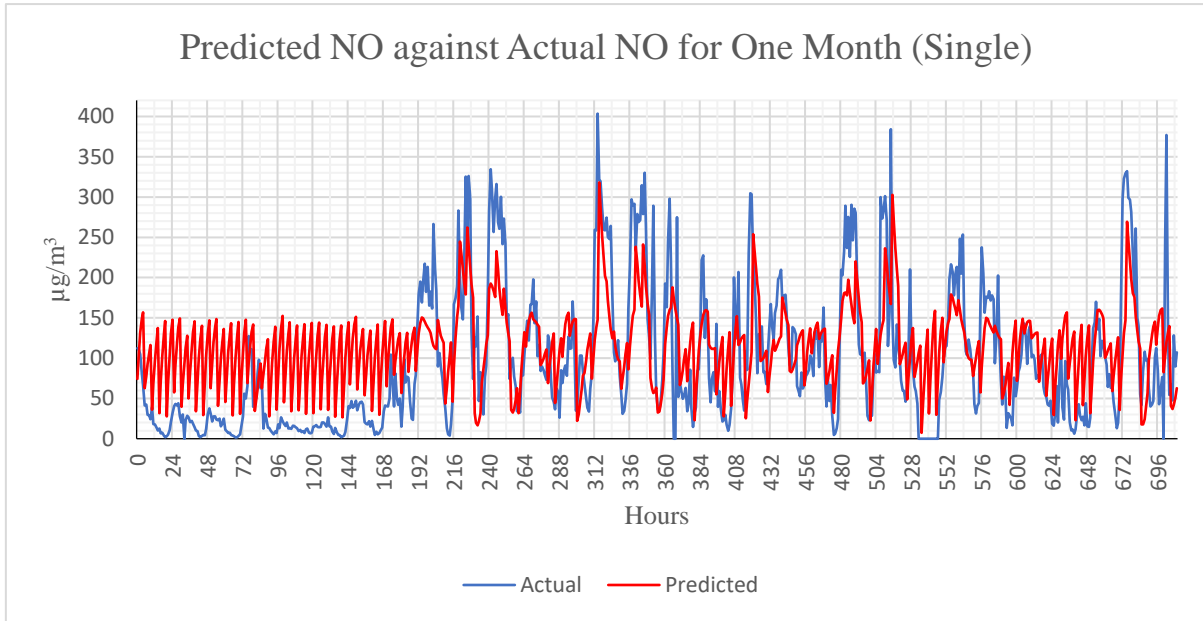


Figure 6 Predicted NO values against actual NO values for a period of one month using one input type

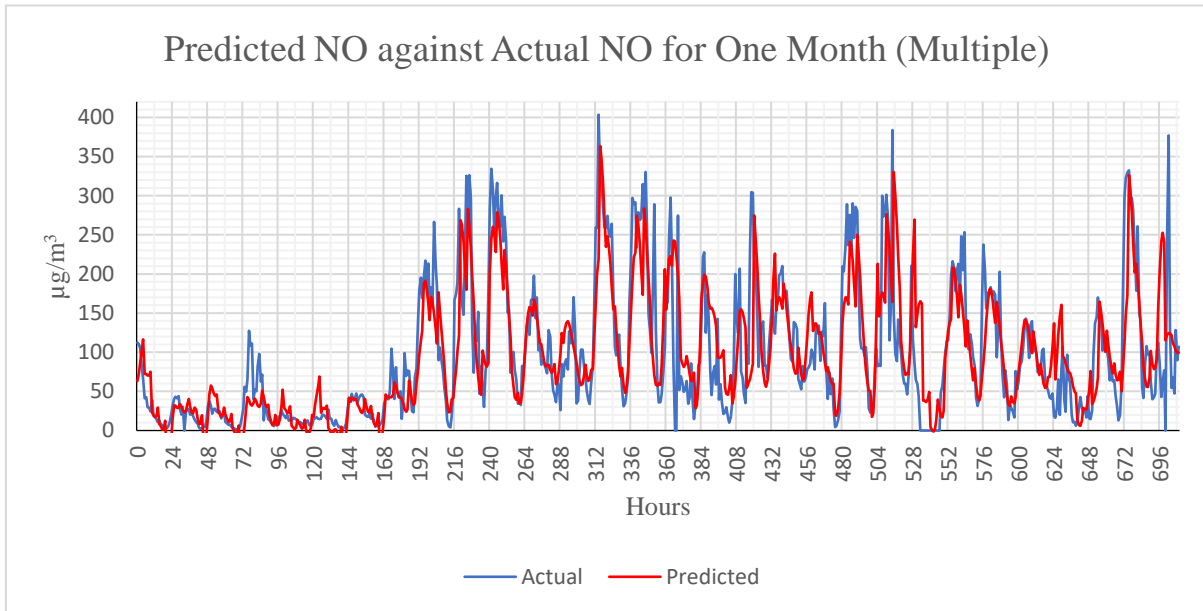


Figure 7 Predicted NO values against actual NO values for a period of one month using multiple input types

East

The average MSE for the single input case is 0.498. The average MSE for the multiple input case is 0.384. This suggests that the multiple input system performed better than the single input system, which, in percentage terms is 25.81%.

Below are line graphs comparing the two cases. Generally, both systems performed well to predict the general shape of the dips in the traffic. However, both cases incorrectly predicted the maximum dip and peak in traffic. For the single input case, it can be seen that it incorrectly predicted the dip in traffic at the 45th hour, predicting 300 when it actually was around 600. This point is perhaps an outlier in the prediction. Also, the single input model tended to predict peaks at incorrect times as well as having less steep peaks. In the actual data, traffic tended to peak and dip quickly. This can also be said of the multiple input system, the predictions tended to miss the wild oscillations during peak hours. However, it did not incorrectly predict peaks as often as the first case.

Predicted East Traffic against Actual East Traffic for One Month (Single)

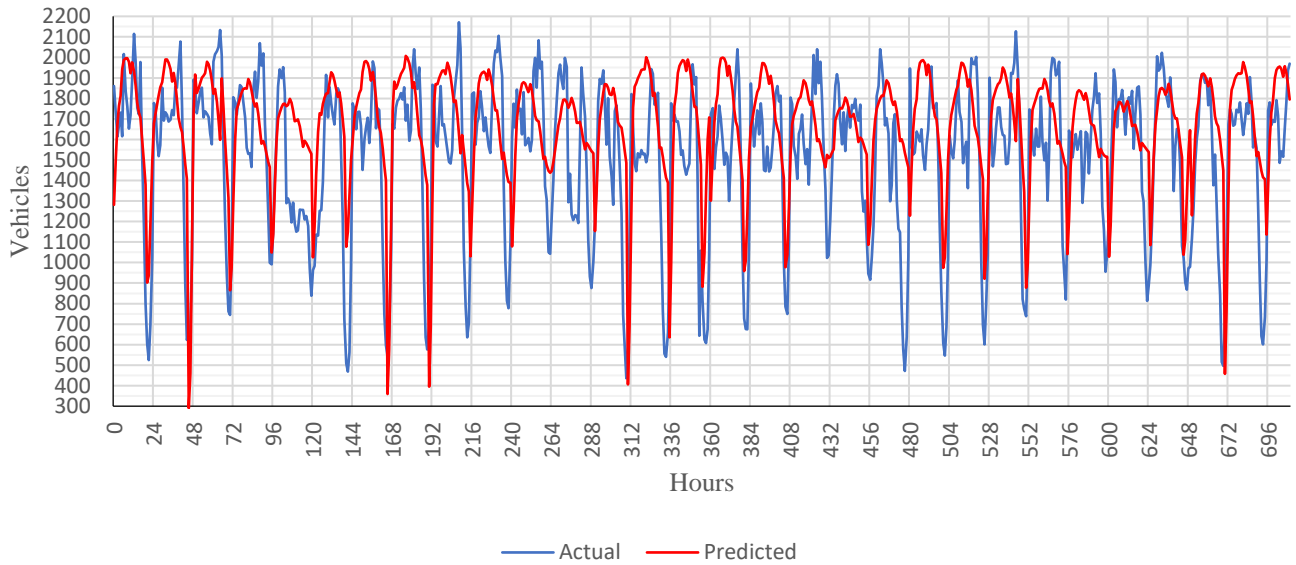


Figure 8 Predicted East Traffic values against actual East Traffic values for a period of one month using one input type

Predicted East Traffic against Actual East Traffic for One Month (Multiple)

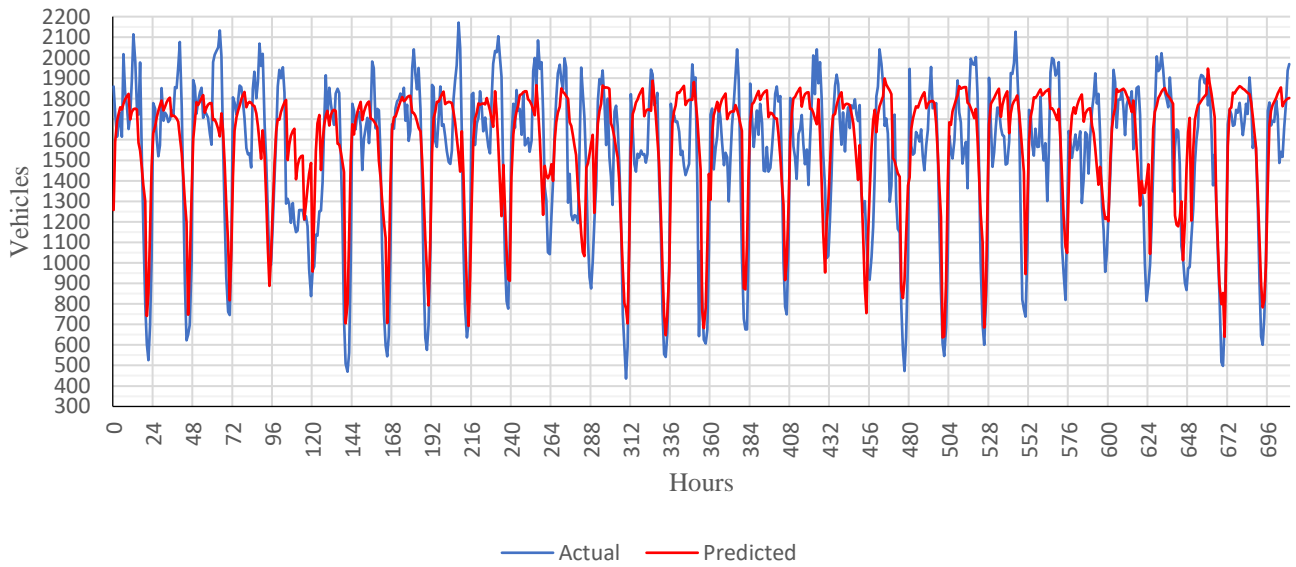


Figure 9 Predicted East Traffic values against actual East Traffic values for a period of one month using multiple input types

Temperature

The average MSE for the single input case is 0.165. The average MSE for the multiple input case is 0.150. This suggests that the multiple input system performed better than the single input system, with an improvement of 9.64%.

Below are line graphs comparing the two cases. It is observed that the single input system incorrectly predicted dips in the temperature and this can be seen at hours before 200. At hour 144, 168 and 192, the system predicted that temperatures would dip below 0 when in reality, temperature never fell below 0 for the entire month. Compared to the multiple input system, the predicted results modelled the actual results very closely, predicting the peaks and troughs in temperature with differences of less than 2°C. Minor mistakes can be seen in incorrectly predicting the minimum temperature such as in hour 384 or 431.

Generally, both systems performed well to predict the general shape of the dips in the traffic. However, both cases incorrectly predicted the maximum dip and peak in traffic. For the single input case, it can be seen that it incorrectly predicted the dip in traffic at the 45th hour, predicting 300 when it actually was around 600. This point is perhaps an outlier in the prediction. Also, the single input model tended to predict peaks at incorrect times as well as having less steep peaks. In the actual data, traffic tended to peak and dip quickly. This can also be said of the multiple input system, the predictions tended to miss the wild oscillations during peak hours. However, it did not incorrectly predict peaks as often as the first case. Overall, both systems still managed to predict temperatures quite well.

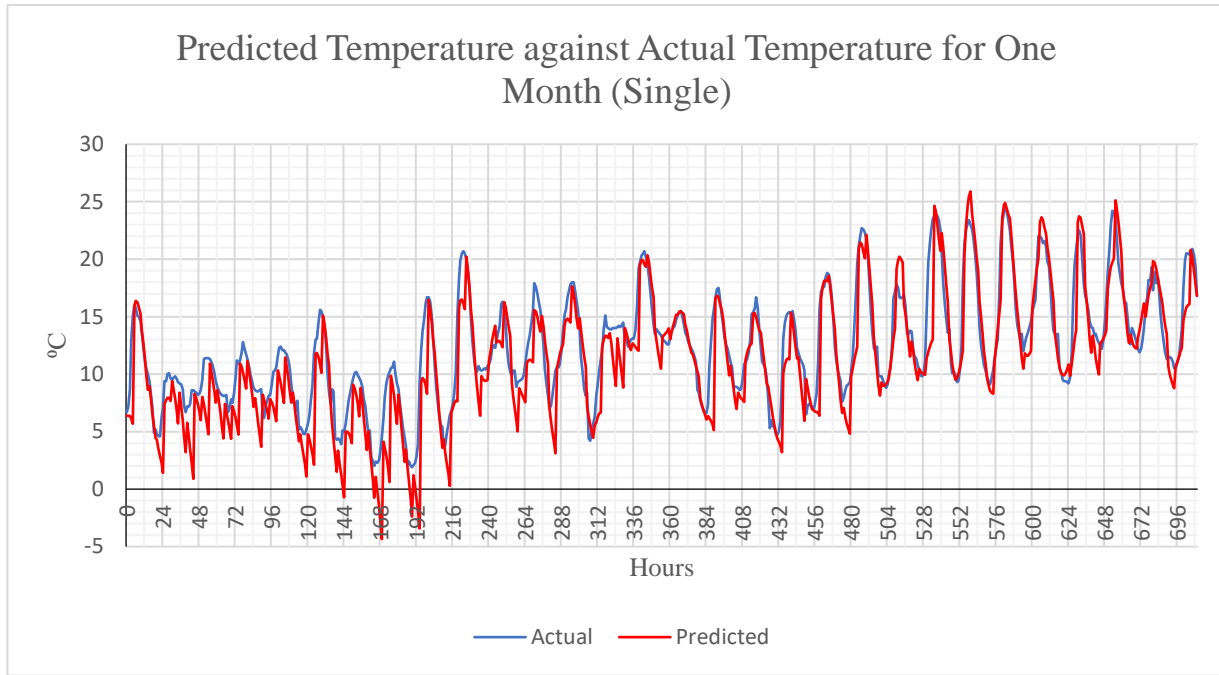


Figure 10 Predicted Temperature values against actual Temperature values for a period of one month using one input type

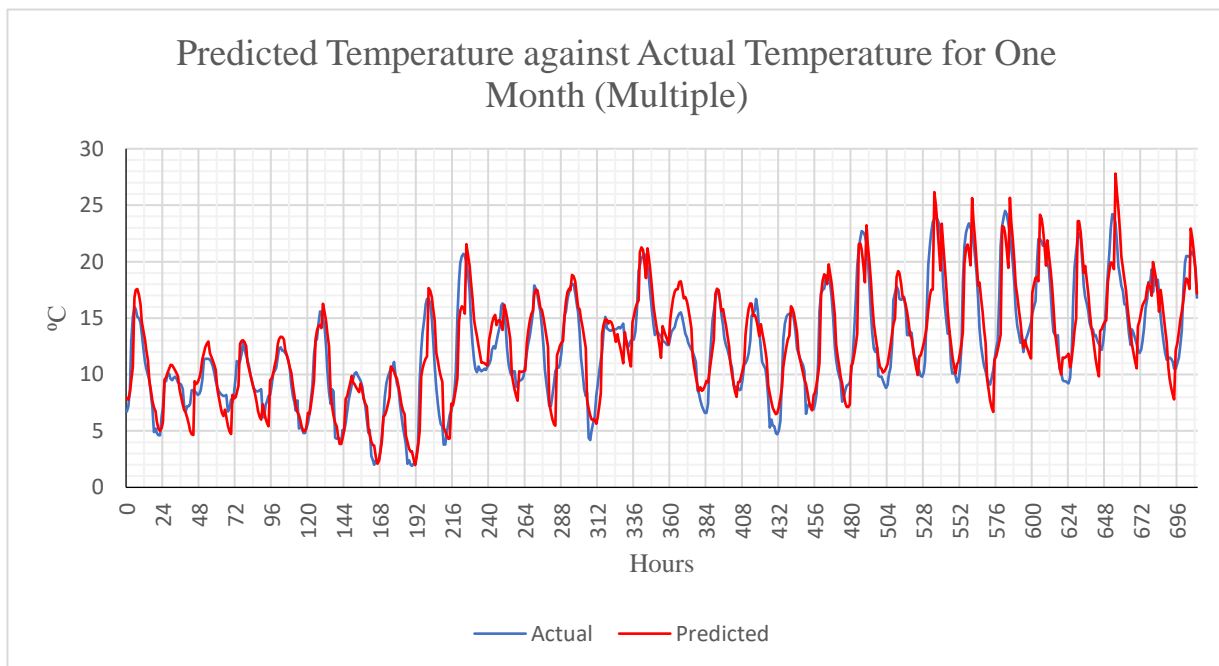


Figure 11 Predicted Temperature values against actual Temperature values for a period of one month using multiple input types

Based on the three cases above, it can be concluded that the Seq2seq model performed better when given multiple features compared to if given only one feature. In all cases, there were improvements in performance which suggests that all the data types are correlated. Although the system was only predicting one type of parameter, having other seemingly unrelated data improved performance.

However, it is not known how closely correlated each data type since there is no metric of measurement. Also, it also unknown whether there is a point of diminishing returns in having more features in the system. This is important to know as to prevent wastage of resources when predicting parameters in a smart city.

Another concern is that the improvements in the system can only be truly seen once the MSE is averaged over a few tests. In every case, the model sometimes tended to perform worse in the multivariate case compared to the univariate case. Below is a table depicting the MSE of each test:

Nitric Oxide (Single)	Nitric Oxide (Multiple)	East Traffic (Single)	East Traffic (Multiple)	Temperature (Single)	Temperature (Multiple)
0.415936	0.245533	0.596377	0.392438	0.135042	0.195992211
0.337178	0.227954	0.552437	0.310874	0.138709	0.177099777
0.388463	0.252087	0.394892	0.470995	0.16665	0.107781206
0.851243	0.26913	0.390394	0.355288	0.213263	0.116371464
0.342275	0.383372	0.556343	0.295365	0.171948	0.152445296

Table 6 MSE for each test case

For example, it can be seen that the best MSE from the single input Nitric Oxide case of 0.337 is better than the worst MSE from the multiple input case, 0.383. This puts the reliability of the model into question, however as explained above, once the values are averaged over a few tests, it is observed that the multiple input case model's performance was better.

Thus, for a smart city, it is better to utilise multiple inputs to predict any parameters as it can boost performance. The datasets used are all real-world data suitable for smart cities.

Machine Learning Algorithms Comparison

For each case, the average Mean Squared Error (MSE) of 5 runs will be used for analysis.

Moving Average

The average MSE for the Moving Average system is 1262.462.

Below is a graph comparing the actual Nitrogen Dioxide (NO₂) values against the values predicted by the Moving Average method. In general, the algorithm was able to model the general trend changes in the NO₂ levels in the atmosphere. However, the system failed to make any significant predictions at any point in the time series. Every single peak and trough was averaged to only map the overall trendline. As a predictor of actual NO₂ values, the model performed extremely poorly, and no useful information can be obtained. However, it is a useful tool in obtaining the general trend change in NO₂ levels as the trendline seemed to cross the centre points of the actual trendline, meaning its average.

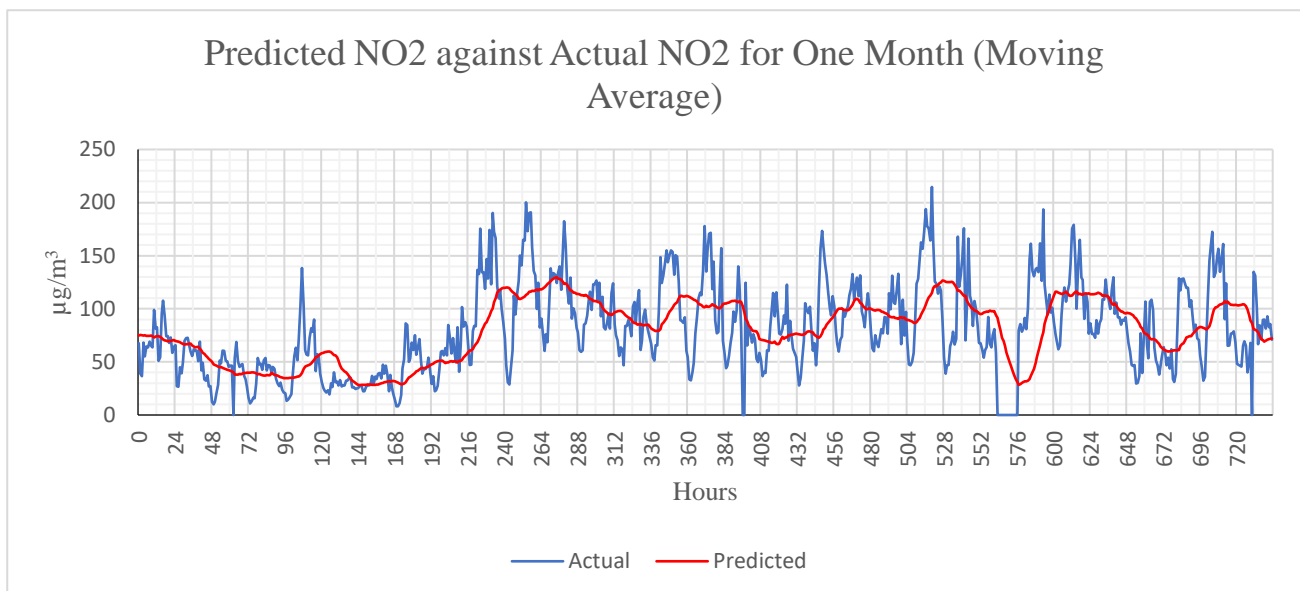


Figure 12 NO₂ values against actual NO₂ values for a period of one month using a moving average model

ARIMA

The average MSE for the ARIMA system is 3672.03. Compared to the Moving Average method, the MSE increased by 190.86%, indicating worse performance in predictions.

The line graph below depicts the comparison between predicted NO₂ levels by the ARIMA model compared to the actual values. Overall, the prediction line presents a very poor fit of the actual line. At many points in the time series, the prediction was either too high or too low by at most 110 $\mu\text{g}/\text{m}^3$. Between the 216 and 264th hour, the predicted NO₂ levels was considerably accurate, with the model predicting the 43 $\mu\text{g}/\text{m}^3$ low on the 244th hour. However, it did not model the peaks accurately being short by at least 40 $\mu\text{g}/\text{m}^3$. Other than that, on the 613rd hour, the system overpredicted the NO₂ level but placed the peak at the correct hour. This shows that the system shows promise in predicting smart city parameters, however it requires better modelling.

The ARIMA method was modelled using an internal function provided by the library used, which may be inaccurate given the dataset and requires manual modelling to determine the best ARIMA parameters. Basing performance purely on the MSE may also not be accurate as it considers the mean values of the errors. In the Moving Average method, it can be seen that the system either overpredicted or underpredicted actual NO₂ levels, since it was a rolling mean and that may prove to offset the MSE value. When compared to the ARIMA method, there are times when the model proved to accurately predict the pollution levels. Thus, the ARIMA model, if better modelled, may prove more suited for smart city applications because in predicting parameters, a user would want accurate value prediction rather than the general trend.

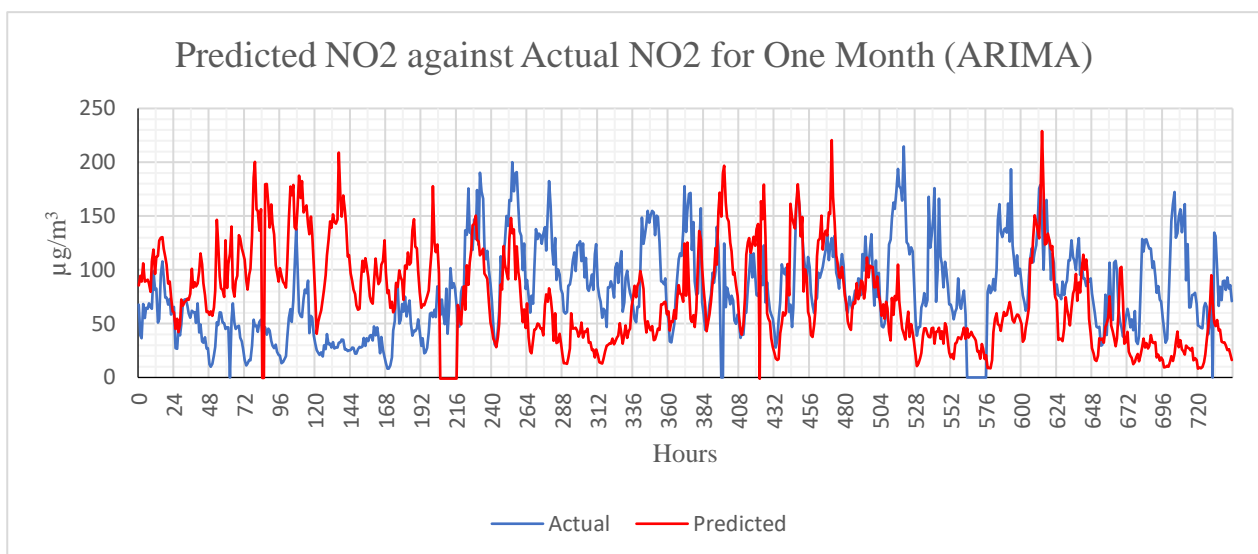


Figure 13 Predicted NO₂ values against actual NO₂ values for a period of one month using an ARIMA model.

Recurrent Neural Network

The average MSE for the Recurrent Neural Network (RNN) model is 0.524. Compared to the ARIMA model, this represents a 99.99% decrease while compared to the Moving Average model, a 99.96% decrease. This proves an immense improvement in performance when a neural network method is used compared to statistical methods.

Depicted below is a line graph of the RNN model predictions against actual hourly NO₂ levels for 30 days. In general, the RNN was able to accurately model the general trendlines of the pollution. At every peak and trough, the prediction was correct and only missed a peak around the 75th hour and 700th hour. The system was unable to predict the accurate highest NO₂ levels between hour 0 and 250. At hour 75, the unusually high NO₂ level peaking at 142, was unable to be modelled, which suggest that the system may require further training to predict anomalies in pollution patterns. The system also ignored the error in the actual data at hour 367, which also suggests that the system may be more resistant to errors in datasets. Overall, the system was able to accurately model the pollution levels and that further training and modifications to the algorithm may further improve the accuracy in predictions.

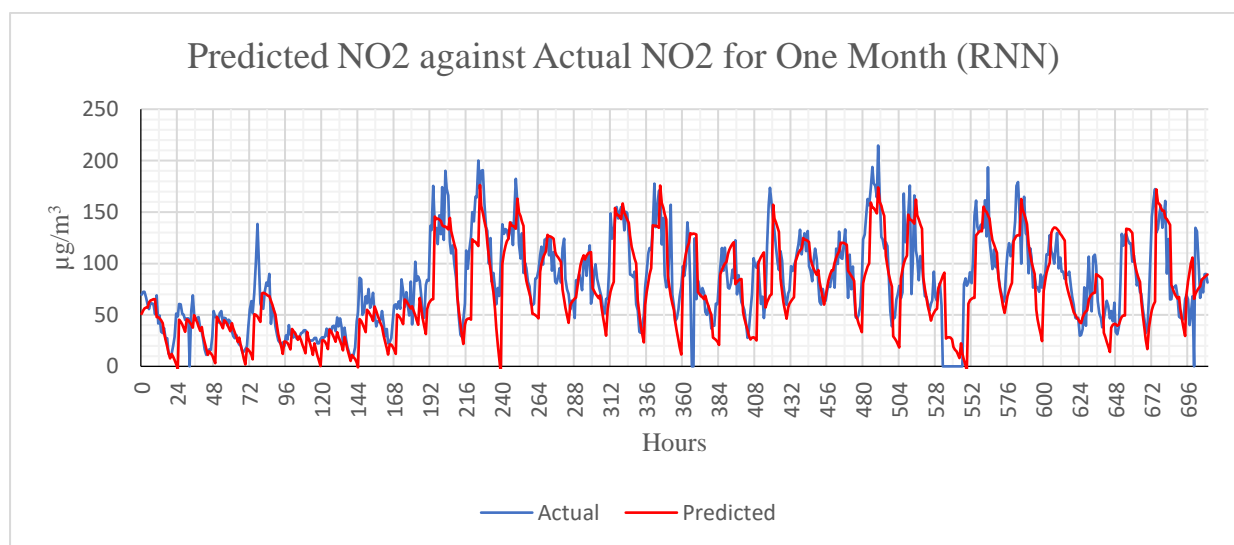


Figure 14 Predicted NO₂ values against actual NO₂ values for a period of one month using a RNN.

Sequence-to-Sequence

The average MSE for the Seq2seq system is 0.466. This represents an 11.07% improvement in performance compared to an RNN. Compared to the Moving Average and ARIMA methods, the system showed similar improvements to that of the RNN. Thus, between the two artificial neural network methods, the Seq2seq performed better than a standard RNN.

Based on the line graph below, the Seq2seq model accurately modelled the general trend of the hourly NO₂ levels. The method was able to predict every peak and trough with the exception of a misplaced peak at hours 75 and 133. When graphed, the predictions were smoother compared to actual NO₂ values, which erratically oscillated more often with many sudden dips and rises in NO₂ levels. As with the RNN, the Seq2seq model also was unable to predict the unusually high NO₂ level at the 75th hour. The model also ignored the error in the dataset at hour 367. Thus, it can be concluded from the previous observations that the method innately “smoothens” predictions, which may be somewhat inaccurate compared to real-world conditions which is far more random. The model was also unable to predict anomalies in real-world conditions, which may suggest that further training needed. It was also resistant to errors in datasets because it ignored an empty value in the dataset, however, this may also be detrimental if actual real-world conditions experienced a sudden change and the system did not model this behaviour as it expects it to be an error. Overall, the system was able to accurately model the pollution levels and that further training and modifications to the algorithm may further improve the accuracy in predictions.

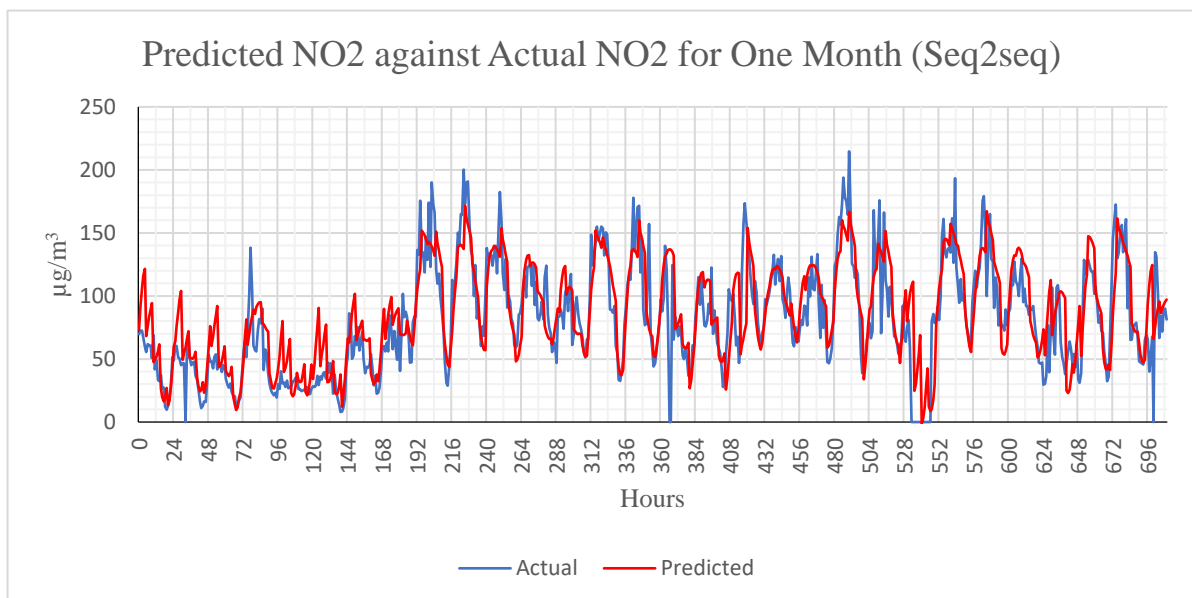


Figure 15 Predicted NO₂ values against actual NO₂ values for a period of one month using a Seq2Seq model.

Real-world Applications

To prove the potential of machine learning for real-world applications utilising consumer-grade hardware and software, a real-time program was developed to visualise predictions of pollution, traffic and weather values in London. To create the program, HTML, jQuery (a JavaScript library), and an external Python program was used. For the dataset, real-time data was only available for pollution and weather in selected parts in London. Traffic data was only available through TfL and can only be acquired through Freedom of Information requests, making acquiring real-time data impossible at this time.

To generate real-time predictions, the Seq2seq python file was modified to continuously run and predict readings every hour and update a csv file which was used as the input data for the real-time program. Then, a HTML file was created for the actual program. Using jQuery, a library to visualise data as heatmaps, heatmap.js was imported and used. The library will read data from the generated csv file to produce heatmaps across London, however, the location for each data point had to manually set. Next, using the Google Maps API, a client-side map of London was able to be imported. The API allows the program to leverage standard Google Maps functions without the need for further coding.

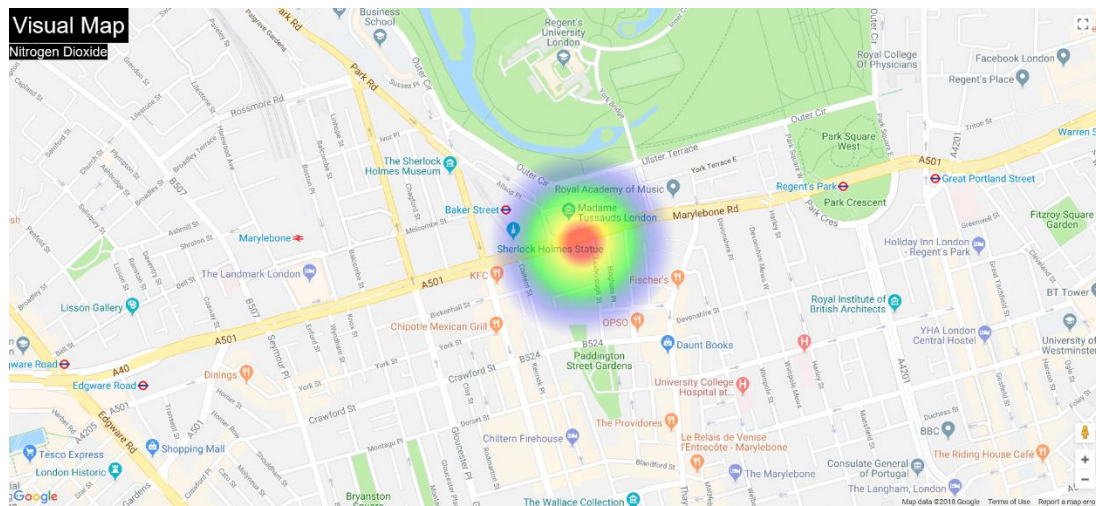


Figure 16 A screenshot of the Visual Map program created to visualise real-time predictions of smart city parameters. The program is visualising NO levels in Marylebone Road on the 1st March 2018.

Using the program, it was possible to visualise predictions on smart city parameters in real-time. However, the heatmaps plotted may not be accurate as actual modelling of pollution spread was not done at this time, though possible if given the data. The program also requires a constantly running external program, which requires a significant amount of computing power for machine learning, making it not lightweight. The program also was missing traffic predictions, as real-time data was unavailable.

Conclusions

With regards to inputs, it was shown that having more features on average will improve the accuracy of a machine learning system. There exists correlation between pollution, weather and traffic in a city and this relationship should be taken advantage of. In the case of smart cities, there will be no shortage of data with the advent of the Internet of Things (IoT). With more data types available, machine learning systems will be able to better predict future parameters, thus ensuring better quality of life for the general population. However, the degree of correlation between data is unknown and should be further explored.

Other than that, it was proven that the performance of Deep Learning or Neural Network methods were superior compared to that of statistical methods, as they were able to better predict parameters with less errors. The Seq2seq model produced slightly better performance compared to the RNN model however both methods exhibited similar drawbacks due to utilising the same Deep Learning architecture. Predictions tend to be “smoother”; The volatility of real-world values were not accurately predicted. However, they also tend to be more resistant to errors in the dataset. Utilising machine learning also requires more computing power to produce results in a short amount of time. The advancement of computing power will make machine learning an invaluable tool for every ordinary citizen of a smart city of the future. The Seq2seq may also be improved through better modifications in the main code as well as general improvements to the core TensorFlow library.

Machine learning also has real-world applications in providing real-time predictions in smart city parameters. Using only consumer-grade hardware and software, it is now possible for anyone to leverage machine learning in various applications. However, further development and better access to real-time data will improve the usability and reliability of the program. Other than that, other algorithms could also provide further insight to the potential of Machine Learning especially with the recent development of Continuous Learning or Incremental Learning.

In conclusion, Machine Learning, especially Deep Learning techniques provide an avenue for better predictions in traffic, weather and pollution. Further development will allow anyone to leverage cognitive computing through readily available tools such as TensorFlow for a variety of real-world applications. Machine Learning provides far better performance and accuracy and should be the focus of future development to overcome the system’s weaknesses.

References

Literature Review

- [1] N. Deryckere, "What is the Potential of Machine Learning in a Smart City?", Undergraduate, Howest, University College West Flanders, 2016.
- [2] J. Chin, V. Callaghan and I. Lim, "Understanding and personalising smart city services using machine learning, The Internet-of-Things and Big Data", IEEE, Edinburgh, 2017.
- [3] M. Zickus, A. Greig and M. Niranjana, "Comparison of Four Machine Learning Methods for Predicting PM10 Concentrations in Helsinki, Finland", Water, Air and Soil Pollution: Focus, vol. 2, no. 56, pp. 717-729, 2002.

Theory

- [1] I. Lurie, "How do I calculate a rolling average?", Portent, 2009. [Online]. Available: <https://www.portent.com/blog/analytics/rolling-averages-math-moron.htm>.
- [2] "Moving Average Models (MA models)", Applied Time Series Analysis, 2018. [Online]. Available: <https://onlinecourses.science.psu.edu/stat510/node/48>.
- [3] D. Britz, "Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs", WildML, 2018. [Online]. Available: <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>.
- [4] D. Nag, "seq2seq: the clown car of deep learning – Dev Nag – Medium", Medium, 2018. [Online]. Available: <https://medium.com/@devnag/seq2seq-the-clown-car-of-deep-learning-f88e1204dac3>.

Methodology

- [1] "London Air Quality Network: Marylebone Road", Londonair, 2017. [Online]. Available: <http://www.londonair.org.uk/london/asp/publicdetails.asp?site=MY1>.
- [2] "London Air Quality Network: Marylebone Road FDMS", Londonair, 2017. [Online]. Available: <http://www.londonair.org.uk/london/asp/publicdetails.asp?site=MY7>.
- [3] "London Marylebone Road - Air Quality England", Airqualityengland, 2017. [Online]. Available: http://www.airqualityengland.co.uk/site/exceedence?site_id=MY1.
- [4] "FOI request detail: Hourly Traffic Counts Data for London Marylebone Rd", Transport for London, 2017. [Online]. Available: <https://tfl.gov.uk/corporate/transparency/freedom-of-information/foi-request-detail?referenceId=FOI-0660-1718>.

Appendix

A. Moving Average Model Python Source Code

```
# 24 Hour Moving Average Model

# Import required libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import mean_squared_error

print("Trailing Moving Average for Marlebone Road")

# Read dataset
data = pd.read_csv('MaryleboneRoad_May_13_17_NO2.csv')
x = data['NO2'].values

# model variables
# window, in hours, default: 24
window = 24

actual = [x[i] for i in range((len(x) - 24*30))] #Take actual values
test = [x[i] for i in range((len(x) - 24*30), len(x))] #Take test
values, last 30 days
prediction = list() # Make an empty list for predictions

# Predict next 30 days by calculating the average
for t in range(len(test)):
    length = len(actual)
    rolling_mean = np.mean([actual[i] for i in range(length - window,
length)])

    obsv = test[t]
    prediction.append(rolling_mean)
    actual.append(obsv)
    print('predicted=%f, expected=%f' % (rolling_mean, obsv))

# Calculate and print MSE
error = mean_squared_error(test, prediction)
print('Test MSE: %.3f' % error)

# Save results into a csv file so we can create nice graphs later
df = pd.DataFrame({"Actual" : test, "Predicted" : prediction})
df.to_csv("movingavg_results.csv", index=False)

plt.plot(prediction, color = 'blue', label = 'Predicted')
plt.plot(test, color = 'red', label = 'Actual')
plt.legend(loc="upper left")
plt.show()
```

B. ARIMA Model Python Source Code

```
# ARIMA model

# Import required libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.tsa.arima_model import ARIMA
from pandas.core import datetools

print("ARIMA model for Marylebone Road")

# Read dataset
data = pd.read_csv('MaryleboneRoad_May_13_17_NO2.csv')

# Split the data for running the ARIMA model and validaion for later
# period (in hours) : default is 30 days
period = 30 * 24
split = len(data) - period
dataset = data[0:split]
validation = data[split:]
dataset.to_csv('dataset_arima.csv')
validation.to_csv('validation_arima.csv')

# function to create a differenced series (subtract obsv from the same day
a year ago)
# default interval is 1 year
# this is to remove any seasonal differences
def season(dataset, interval = 24*365):
    new = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        new.append(value)
    return np.array(new)

# function to invert a differenced series
def inv_season(history, yhat, interval = 24*365):
    return yhat + history[-interval]

# take only the relevant column
x = dataset.values[:,1]
y = validation.values[:,1]
differenced = season(x)

# fit ARIMA model
model = ARIMA(differenced, order = (7,0,1))
model_fit = model.fit(dis = 0)

# multi-step out-of-sample forecast
step = 24*30
forecast = model_fit.forecast(steps = step)[0]

# Invert all results for usable data
history = [x for x in x]
hour = 1
for yhat in forecast:
    inverted = inv_season(history, yhat)
    print('Hour %d: %f' % (hour, inverted))
    history.append(inverted)
```

```

    hour += 1

# resplit to only get the relevant hours
split = len(history) - step
predicted_final = history[split:]

# Calculate MSE
print("Test MSE : ", np.mean((predicted_final - y)**2))

# Save results into a csv file so we can create nice graphs later
df = pd.DataFrame({"Actual" : y, "Predicted" : predicted_final})
df.to_csv("results_arima.csv", index=False)

# Plot a quick graph so we can see if everything's all right
plt.plot(predicted_final, color = 'blue', label = 'Predicted')
plt.plot(y, color = 'red', label = 'Actual')
plt.title("Predicted against Actual for One Month")
plt.legend(loc="upper left")
plt.show()

```

C. RNN Model Python Source Code

```
# Recurrent Neural Network Model

# Import required libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

print("RNN model for Marylebone Road")

# Read dataset
df = pd.read_csv('MaryleboneRoad_May_13_17.csv')

# Store Date and NO2 columns in numpy arrays
dates = df['Date'].as_matrix()
no2_lvl = df['NO2'].as_matrix()

# Split the data for training and testing
# period: How many hours before to split the data (default is 31*24 (one
month))
period = 31*24

train = no2_lvl[0:len(no2_lvl)-period]
valid = no2_lvl[len(no2_lvl)-period:len(no2_lvl)]

# Normalise
train_min = min(train)
train_max = max(train)
train_normalised = (train - train_min) / (train_max - train_min)
plt.plot(train_normalised)
plt.show()

# Initialise graph with proper parameters
n_inputs = 1 # number of nodes in input layer
n_hidden = 100 # number of nodes in hidden layer
n_outputs = 1 # number of nodes in output layer

# Initialise batch parameters
batch_size = 50 # number of training samples to consider during each
training instance
n_steps = 12 # number of steps in every input Sequence

# Initialise optimiser parameters
n_iterations = 10000 # number of training iterations to execute
learning_rate = 0.001 # training learning rate

# Construct computational graph
tf.reset_default_graph()
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_hidden,
activation=tf.nn.relu)
rnn_outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_hidden])
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])

loss = tf.reduce_mean(tf.square(outputs - y))
```

```

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()

saver = tf.train.Saver()

# Generate batches of training inputs and outputs
def next_batch(input_sequence, batch_size, n_steps):
    i_first = 0
    i_last = len(input_sequence)
    i_starts = np.random.randint(i_first, high=i_last-n_steps,
size=(batch_size, 1))
    i_sequences = i_starts + np.arange(0, n_steps + 1)
    flat_i_sequences = np.ravel(i_sequences[:, :])
    flat_sequences = input_sequence[flat_i_sequences]
    sequences = flat_sequences.reshape(batch_size, -1)
    return sequences[:, :-1].reshape(-1, n_steps, 1), sequences[:,
1:].reshape(-1, n_steps, 1)

#Train the RNN
with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(train_normalised, batch_size,
n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 1000 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    saver.save(sess, "./Marylebone_RNN")

#predictions
train_normalised_working = train_normalised
predictions_normalised = np.array([])

with tf.Session() as sess:
    saver.restore(sess, "./Marylebone_RNN")
    for pred in range(period):
        flat_X_new =
train_normalised_working[len(train_normalised_working)-
n_steps:len(train_normalised_working)]
        X_new = flat_X_new.reshape(1, -1).reshape(-1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_new})
        predictions_normalised = np.append(predictions_normalised,
[y_pred[0, n_steps-1, 0]], axis=0)
        train_normalised_working = np.append(train_normalised,
predictions_normalised, axis=0)

#Denormalise predictions
predictions = predictions_normalised * (train_max - train_min) + train_min

no2_lvl_pred = np.array([train[len(train)-1]])
for pred in range(period):
    no2_lvl_pred = np.append(no2_lvl_pred, [no2_lvl_pred[pred] +
no2_lvl_pred[pred] * (predictions[pred] / 100)], axis=0)
no2_lvl_pred = np.delete(no2_lvl_pred, 0)

# Plot a quick graph so we can see if eveything's all right
plt.plot(no2_lvl_pred, color = 'blue', label = 'Predicted')

```

```
plt.plot(valid, color = 'red', label = 'Actual')
plt.title("Predicted against Actual for One Month")
plt.legend(loc="upper left")
plt.show()
```


D. Seq2seq Model Python Source Code

```
# Seq2seq model

# Import required libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from tensorflow.contrib import rnn
from tensorflow.python.ops import variable_scope
from tensorflow.python.framework import dtypes
import tensorflow as tf
import copy
import os

# Read dataset
df = pd.read_csv('MaryleboneRoad_May_13_17.csv')

# Split the data for training and testing
# period: How many hours before to split the data (default is 31*24
(one month))
period = 31*24
train = df.iloc[:(-period), :].copy()
test = df.iloc[-period:, :].copy()

# Copy useful columns from the dataset for use in modelling
# Change the range in var X_train, X_test for features
# Change the array in y_train and y_test for feature
# Available features: Hour, NO, NO2, NOX, O3, PM2.5, PM10_TEOM, PM10,
CO, SO2, East, West, TMP, WDIR, WSPD
# ['Hour'], ['NO'], ['NOX'], ['O3'], ['PM2.5'], ['PM10_TEOM'],
['PM10'], ['CO'], ['SO2'], ['East'], ['West'], ['TMP'], ['WDIR'], ['WSPD']
X_train = train.loc[:, ['NO']].values.copy()
X_test = test.loc[:, ['NO']].values.copy()
y_train = train['NO'].values.copy().reshape(-1, 1)
y_test = test['NO'].values.copy().reshape(-1, 1)

# Normalise (z-transform) X, to be able to compare features
for i in range(X_train.shape[1]):
    x_mean = X_train[:, i].mean()
    x_std = X_train[:, i].std()
    X_train[:, i] = (X_train[:, i] - x_mean) / x_std
    X_test[:, i] = (X_test[:, i] - x_mean) / x_std

# Normalise (z-transform) y, to be able to compare with features
y_mean = y_train.mean()
y_std = y_train.std()
y_train = (y_train - y_mean) / y_std
y_test = (y_test - y_mean) / y_std

# Sequence length paramters
input_seq_len = 30
output_seq_len = 5

# Function to randomly generate training samples from the data
def generate_train_samples(x = X_train, y = y_train, batch_size = 10,
input_seq_len = input_seq_len, output_seq_len = output_seq_len):
    total_start_points = len(x) - input_seq_len - output_seq_len
```

```

        start_x_idx = np.random.choice(range(total_start_points),
batch_size, replace = False)

        input_batch_idx = [list(range(i, i+input_seq_len)) for i in
start_x_idx]
        input_seq = np.take(x, input_batch_idx, axis = 0)

        output_batch_idx = [list(range(i+input_seq_len,
i+input_seq_len+output_seq_len)) for i in start_x_idx]
        output_seq = np.take(y, output_batch_idx, axis = 0)

        return input_seq, output_seq # in shape: (batch_size, time_steps,
feature_dim)

# Function to generate test samples
def generate_test_samples(x = X_test, y = y_test, input_seq_len =
input_seq_len, output_seq_len = output_seq_len):
    total_samples = x.shape[0]

    input_batch_idx = [list(range(i, i+input_seq_len)) for i in
range((total_samples-input_seq_len-output_seq_len))]
    input_seq = np.take(x, input_batch_idx, axis = 0)

    output_batch_idx = [list(range(i+input_seq_len,
i+input_seq_len+output_seq_len)) for i in range((total_samples-
input_seq_len-output_seq_len))]
    output_seq = np.take(y, output_batch_idx, axis = 0)

    return input_seq, output_seq

# Generate training samples for features and labels
x, y = generate_train_samples()

# Generate training samples for features and labels
test_x, test_y = generate_test_samples()

# Parameters for graph building
learning_rate = 0.01
lambda_l2_reg = 0.003
# LSTM cell size
hidden_dim = 64
# Number of features, based on columns copied
input_dim = X_train.shape[1]
# Number of labels, based on columns copied
output_dim = y_train.shape[1]
# Number of stacked LSTM layers
num_stacked_layers = 2
# Parameter to prevent gradient explosion
GRADIENT_CLIPPING = 2.5

# Function for tensorflow graph building using RNNs
# Uses parameter feed_previous: False for training, True for testing
def build_graph(feed_previous = False):

    # Clear default graph stack and resets
    tf.reset_default_graph()

    # Set tf Variable for steps
    global_step = tf.Variable(
        initial_value=0,
        name="global_step",

```

```

        trainable=False,
        collections=[tf.GraphKeys.GLOBAL_STEP,
tf.GraphKeys.GLOBAL_VARIABLES])

    weights = {
        'out': tf.get_variable('Weights_out', \
                                shape = [hidden_dim, output_dim], \
                                dtype = tf.float32, \
                                initializer =
tf.truncated_normal_initializer()),
    }
    biases = {
        'out': tf.get_variable('Biases_out', \
                                shape = [output_dim], \
                                dtype = tf.float32, \
                                initializer =
tf.constant_initializer(0.)),
    }

    # Define ops to create a seq2seq model
    with tf.variable_scope('Seq2seq'):
        # Encoder: inputs
        enc_inp = [
            tf.placeholder(tf.float32, shape=(None, input_dim),
name="inp_{}".format(t))
                for t in range(input_seq_len)
        ]

        # Decoder: target outputs
        target_seq = [
            tf.placeholder(tf.float32, shape=(None, output_dim),
name="y".format(t))
                for t in range(output_seq_len)
        ]

        # Feed "GO" token to the decoder
        # If dec_inp are fed into decoder as inputs, this is 'guided'
training; otherwise only the
        # first element will be fed as decoder input which is then
'unguided'
        dec_inp = [ tf.zeros_like(target_seq[0], dtype=tf.float32,
name="GO") ] + target_seq[:-1]

        # Define ops for a LSTM Cell
        with tf.variable_scope('LSTMCell'):
            cells = []
            for i in range(num_stacked_layers):
                with tf.variable_scope('RNN_{}'.format(i)):
                    cells.append(tf.contrib.rnn.LSTMCell(hidden_dim))
            cell = tf.contrib.rnn.MultiRNNCell(cells)

        # From tensorflow seq2seq repo
        #
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/leg
acy\_seq2seq/python/ops/seq2seq.py
        def _rnn_decoder(decoder_inputs,
                        initial_state,
                        cell,
                        loop_function=None,
                        scope=None):
            """RNN decoder for the sequence-to-sequence model.

```

```

        Args:
            decoder_inputs: A list of 2D Tensors [batch_size x
input_size].
            initial_state: 2D Tensor with shape [batch_size x
cell.state_size].
            cell: rnn_cell.RNNCell defining the cell function and size.
            loop_function: If not None, this function will be applied
to the i-th output
                        in order to generate the i+1-st input, and decoder_inputs
will be ignored,
                        except for the first element ("GO" symbol). This can be
used for decoding,
                        but also for training to emulate
http://arxiv.org/abs/1506.03099.
            Signature -- loop_function(prev, i) = next
                * prev is a 2D Tensor of shape [batch_size x
output_size],
                * i is an integer, the step number (when advanced
control is needed),
                * next is a 2D Tensor of shape [batch_size x
input_size].
            scope: VariableScope for the created subgraph; defaults to
"rnn_decoder".
        Returns:
            A tuple of the form (outputs, state), where:
                outputs: A list of the same length as decoder_inputs of
2D Tensors with
                        shape [batch_size x output_size] containing generated
outputs.
                state: The state of each cell at the final time-step.
                        It is a 2D Tensor of shape [batch_size x
cell.state_size].
                        (Note that in some cases, like basic RNN cell or GRU
cell, outputs and
                        states can be the same. They are different for LSTM
cells though.)
    """
    with variable_scope.variable_scope(scope or "rnn_decoder"):
        state = initial_state
        outputs = []
        prev = None
        for i, inp in enumerate(decoder_inputs):
            if loop_function is not None and prev is not None:
                with variable_scope.variable_scope("loop_function",
reuse=True):
                    inp = loop_function(prev, i)
            if i > 0:
                variable_scope.get_variable_scope().reuse_variables()
            output, state = cell(inp, state)
            outputs.append(output)
            if loop_function is not None:
                prev = output
        return outputs, state

# From tensorflow seq2seq repo
#
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/leg
acy\_seq2seq/python/ops/seq2seq.py
    def _basic_rnn_seq2seq(encoder_inputs,
                           decoder_inputs,
                           cell,

```

```

        feed_previous,
        dtype=dtypes.float32,
        scope=None):
    """Basic RNN sequence-to-sequence model.
    This model first runs an RNN to encode encoder_inputs into a
state vector,
    then runs decoder, initialized with the last encoder state,
on decoder_inputs.
    Encoder and decoder use the same RNN cell type, but don't
share parameters.
    Args:
        encoder_inputs: A list of 2D Tensors [batch_size x
input_size].
        decoder_inputs: A list of 2D Tensors [batch_size x
input_size].
        feed_previous: Boolean; if True, only the first of
decoder_inputs will be
            used (the "GO" symbol), all other inputs will be
generated by the previous
            decoder output using _loop_function below. If False,
decoder_inputs are used
            as given (the standard decoder case).
        dtype: The dtype of the initial state of the RNN cell
(default: tf.float32).
        scope: VariableScope for the created subgraph; default:
"basic_rnn_seq2seq".
    Returns:
        A tuple of the form (outputs, state), where:
            outputs: A list of the same length as decoder_inputs of
2D Tensors with
                shape [batch_size x output_size] containing the
generated outputs.
            state: The state of each decoder cell in the final time-
step.
                It is a 2D Tensor of shape [batch_size x
cell.state_size].
    """
    with variable_scope.variable_scope(scope or
"basic_rnn_seq2seq"):
        enc_cell = copy.deepcopy(cell)
        _, enc_state = rnn.static_rnn(enc_cell, encoder_inputs,
dtype=dtype)
        if feed_previous:
            return _rnn_decoder(decoder_inputs, enc_state, cell,
_loop_function)
        else:
            return _rnn_decoder(decoder_inputs, enc_state, cell)

    def _loop_function(prev, _):
        """Naive implementation of loop function for _rnn_decoder.
Transform prev from
        dimension [batch_size x hidden_dim] to [batch_size x
output_dim], which will be
        used as decoder input of next time step """
        return tf.matmul(prev, weights['out']) + biases['out']

    dec_outputs, dec_memory = _basic_rnn_seq2seq(
        enc_inp,
        dec_inp,
        cell,
        feed_previous = feed_previous

```

```

    )

    reshaped_outputs = [tf.matmul(i, weights['out']) +
biases['out'] for i in dec_outputs]

    # Training loss
    with tf.variable_scope('Loss'):
        # Least Square Error (L2) loss
        output_loss = 0
        for _y, _Y in zip(reshaped_outputs, target_seq):
            output_loss += tf.reduce_mean(tf.pow(_y - _Y, 2))

        # L2 regularization for weights and biases
        reg_loss = 0
        for tf_var in tf.trainable_variables():
            if 'Biases_' in tf_var.name or 'Weights_' in tf_var.name:
                reg_loss += tf.reduce_mean(tf.nn.l2_loss(tf_var))

        loss = output_loss + lambda_l2_reg * reg_loss

    # Optimiser
    with tf.variable_scope('Optimizer'):
        optimizer = tf.contrib.layers.optimize_loss(
            loss=loss,
            learning_rate=learning_rate,
            global_step=global_step,
            optimizer='Adam',
            clip_gradients=GRADIENT_CLIPPING)

    saver = tf.train.Saver

    return dict(
        enc_inp = enc_inp,
        target_seq = target_seq,
        train_op = optimizer,
        loss=loss,
        saver = saver,
        reshaped_outputs = reshaped_outputs,
    )

# Training parameters
total_interactions = 100
batch_size = 16

# Create a training model
rnn_model = build_graph(feed_previous=False)

# Add ops to save and restore tf variables
saver = tf.train.Saver()

# Create a training tf session
init = tf.global_variables_initializer()
with tf.Session() as sess:

    sess.run(init)

    # Print training losses to see if training is working as intended
    print("Training losses: ")
    for i in range(total_interactions):
        batch_input, batch_output =
generate_train_samples(batch_size=batch_size)

```

```

        feed_dict = {rnn_model['enc_inp'][t]: batch_input[:,t] for t in
range(input_seq_len)}
        feed_dict.update({rnn_model['target_seq'][t]: batch_output[:,t]
for t in range(output_seq_len)})
        _, loss_t = sess.run([rnn_model['train_op'],
rnn_model['loss']], feed_dict)
        print(loss_t)

    # Save
    temp_saver = rnn_model['saver']()
    save_path = temp_saver.save(sess, os.path.join('results/',
'Smart_City_Prediction'))

    # Print location
    print("Checkpoint saved at: ", save_path)

    # Create a testing model
    rnn_model = build_graph(feed_previous=True)

    # Create a testing tf session
    init = tf.global_variables_initializer()
    with tf.Session() as sess:

        sess.run(init)

        # Load
        saver = rnn_model['saver']().restore(sess,
os.path.join('results/', 'Smart_City_Prediction'))

        # Calculate and later print MSE
        feed_dict = {rnn_model['enc_inp'][t]: test_x[:, t, :] for t in
range(input_seq_len)} # batch prediction
        feed_dict.update({rnn_model['target_seq'][t]:
np.zeros([test_x.shape[0], output_dim], dtype=np.float32) for t in
range(output_seq_len)})
        final_preds = sess.run(rnn_model['reshaped_outputs'], feed_dict)

        final_preds = [np.expand_dims(pred, 1) for pred in final_preds]
        final_preds = np.concatenate(final_preds, axis = 1)
        print()
        print("Test MSE : ", np.mean((final_preds - test_y)**2))

    # Concatenate results into an array
    actual_y = np.concatenate([test_y[i].reshape(-1) for i in range(0,
test_y.shape[0], 5)], axis = 0)
    predicted_y = np.concatenate([final_preds[i].reshape(-1) for i in
range(0, final_preds.shape[0], 5)], axis = 0)

    # Save results into a csv file so we can create nice graphs later
    df = pd.DataFrame({"Actual" : actual_y, "Predicted" : predicted_y})
    df.to_csv("results.csv", index=False)

    # Plot a quick graph so we can see if everything's all right
    plt.plot(predicted_y, color = 'blue', label = 'Predicted')
    plt.plot(actual_y, color = 'red', label = 'Actual')
    plt.title("Predicted against Actual for One Month")
    plt.legend(loc="upper left")
    plt.show()

```