UNIVERSITY OF GRANADA

DEPARTMENT OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE

Deep learning for prediction of environmental pollution in London's underground

Author:
David Alberto Martín Vela

Supervisors:
Miguel Molina-Solana
Rossella Arcucci

Submitted in partial fulfillment of the requirements for the Double Bachelor's degree in Computer Science and Mathematics of University of Granada

Academic course 2020-2021

Abstract

Your abstract.
TODO: ¿?

Acknowledgments

Professor Miguel Molina-Solana has helped me immensely with his corrections and advice in writing this document, allowing me to clarify my thoughts when writing this document and teaching me how to write clearly in a scientist style and develop the project in the right direction.
I am grateful to Prof. Kumar and his collaborators at University of Surrey (UK) who have generously shared with us the pollution data we used in this work. They and Rossella Arcucci (my second supervisor) are partly funded by the EPSRC grant EP/T003189/1 *Health assessment across biological length scales for personal pollution exposure and its mitigation (INHALE)*.

Thanks also to my co-workers for their understanding. They have allowed me to combine finishing my degree with working as a Software Developer Engineer.

Lastly, a special thanks to my friends Alex, Rafa and Abel for always helping me with their time series training whenever I have a question related to the topic.

Contents

Chapter 1

Introduction

Environmental pollution is the cause of numerous premature deaths and other associated diseases. With the rapid development of urbanization and industrialization, many developing countries are suffering from heavy air pollution and many governments and citizens have expressed increasing concern regarding air pollution because it affects human health and sustainable development worldwide.

When one thinks of public transport, what comes to mind is a healthy environment. But for Londoners the reality is that public transport does not mean pollution-free, in fact it often means the exact opposite. Recent studies reveal high levels of pollution on London Underground [Smi+20] around **15 times** greater than above ground. Concentrations of fine particles ($PM_{2.5}$) which are linked to several health problems, were many times greater than in other modes of transport (such as cycling, driving and taking a bus) or even those in other subway systems around the world (such as New York, Sydney, Seoul or Barcelona). In fact, those Londoners traveling by car, who contribute to the pollution in the street, are breathing far less harmful pollutants than people using the "Tube". [1]

As a starting point we have two studies in London that contextualize the problem. The first one aimed at finding potential relationships between income deprivation and pollutant levels on the basis that people with low income often experience higher exposures to air pollutants [RKHZ17] and the second one which find the determinants of personal exposure concentrations of commuters pollutants, black carbon (BC), ultra fine particle number concentrations (PNCs), and particulate matter, PM_1 ($PM \leq 1\mu m$), $PM_{2.5}$ ($PM \leq 2.5\mu m$) and PM_{10} ($PM \leq 10\mu m$) investigating in different travel modes [Riv+17].

Surprisingly, there are not many studies focused on **predicting** the concentration of polluting particles in this context. This fact inspired us to investigate that issue and this project is a step in that direction. This document is divided into different chapters.

[1] Slang name for the London Underground because the tunnels for some of the lines are round tubes running through the ground.

After an initial **Introduction** chapter, we will contextualize the problem in the **Background** with a review of previous work on modelling air pollution in transport / metro, along with various techniques that have been used so far. In these previous articles different datasets have been collected by monitoring pollution particles in different transport micro environments. Some of these authors have kindly shared with us a novel dataset related to one particular station in the London Underground network. This is the dataset that we use in our study.

In the third chapter, the densest chapter, **Methods**, the methods used to develop the work and achieve the objectives will be explained. We used the **London public transport API** [Tfl] to obtain datasets related to train schedules and station crowding, which will be used for comparison purposes with the original pollution time series. We will describe how to apply preprocessing, clean and data scaling techniques (feature engineering) to our datasets. Finally, we will explain several methods to compare time series and predict future values gathering expertise from [GBC16].

In the fourth chapter, **Experiments**, we will visualize our data with **exploratory data analysis**, which exposes trends, patterns, and relationships that are not readily apparent. We present the different experiments we performed and what is pursued with them using the methods described before which mainly contain results of time series analysis, comparison between our different datasets and pollution prediction.

Finally, we include the results obtained and their implications in **Conclusions and further work**. Here we will also check if we have achieved the objectives and expectations and describe potential paths for further development.

For the development of this document, knowledge about machine learning, multivariate statistics and time series have been needed. The technical part has been done with **Python** using **docker** environments for dependencies in **Jupyter Notebooks** that allows to create and share documents that contain live code. You can find the code of the experiments in a Github repository. However, this repository is restricted due to the confidentiality of the data, so alternatively, **we have created a web page where you can find all Jupyter notebooks used for the experiments**, `[sl.ugr.es/tfgdavidmv1996]`.

The main sources for the writing of this document were taken from [L21̈].

Main goals and results achieved

The initial goals of this Bachelors' thesis were:

1. Analyse and extract information from a given dataset (feature engineering).

2. Collect and compile other datasets.

3. Use a public API to collect data.
4. Compare time series.
5. Create different deep learning models and use them for predicting the concentration of pollutant particles in the London Underground.
6. Use clustering techniques to find stations that behave similarly.
7. Implement a web application that provides the most optimal time to use public transport (understanding as optimal the lowest possible level of contamination).

This project had an expected extension of 18 ECTS. As each ECTS credit amounts to 25 hours of work, the project should involve no less than 450 hours of work.

1.1 Summary

Brief Summary

In this document, we use different data analysis and prediction techniques to draw conclusions about environmental pollution in the London underground (specifically in one station from which pollution data has been collected and shared with us).

The second chapter describes the background of our project and the motivations to pursue the modeling and prediction of pollution in the London underground.

In the third chapter, we obtain, preprocess and clean data related to pollution in one station of the London underground. We also describe several methods to compare between pollution data, train schedules and crowding levels of the station. We highlight these last two datasets since we obtained them using the London public transport API. We also present some methods to predict.

Then, in the fourth chapter, we begin by describing the structure of our data by making an initial visualization. After that, and using the methods above, different experiments are carried out in order to predict air pollution. Comparison algorithms will also be applied to our time series with the objective of finding relationships and similarities.

Finally, we will draw conclusions from the obtained results, indicate if the objectives have been fully, partially or not achieved, and suggest possible clear ways of further development of the work.

Keywords: Air pollution, timeseries comparison, timeseries analysis, London public transport API, pollution forecasting.

Resumen extendido en español

Abordando el tema de la contaminación de manera global, nos encontramos en una situación donde muchos países sufren de una elevada contaminación ambiental, principalmente aquellos que cuentan con un fuerte proceso de industrialización sin precedentes, lo cuál ha propiciado un mayor aumento en la preocupación de la sociedad, alcanzando su debate a todos los sectores de la comunidad. En algunas ocasiones esta contaminación se aprecia a simple vista, pero en otras situaciones uno no se da cuenta de que indirectamente es una causa de numerosas muertes y otras enfermedades asociadas provocadas por este aumento de contaminación.

Cuando uno piensa en transporte público, lo ideal sería que a nuestra cabeza viniera la idea de un medio sostenible que apueste por energías limpias y renovables, desgraciadamente la realidad es mucho más distinta. Por ejemplo, en la ciudad de Londres esto implica exactamente lo contrario. Recientes estudios revelan que los niveles de contaminación en el metro de Londres alcanza hasta **15 veces** un mayor valor que en la superficie normal [Smi+20]. La cantidad de partículas $PM_{2.5}$, la cuál esta relacionada con muchos problemas respiratorios bastante graves, es muy elevada en comparación con otros medios de transporte como puede ser una bicicleta, autobús e incluso el coche. Esta elevada diferencia también se mantiene si comparamos el metro de Londres con el mismo medio de transporte de otras ciudades alrededor del mundo como son Nueva York, Sydney, Seoul e incluso Barcelona. Lo que es más, alguien que utilice el coche, contribuyendo así a la contaminación ambiental, respira aire menos contaminado que aquellos que usan el metro de Londres.

Como punto de partida inicial de nuestro trabajo contamos con dos estudios que nos han servido para contextualizar nuestro problema. Estos estudios han sido realizados por los mismos investigadores que nos han proporcionado datos de contaminación en el metro de Londres para abordar nuestro proyecto. Un primer estudio busca encontrar si hay algún tipo de relación entre los ingresos salariales y los niveles de contaminación, pudiendo darse el caso que la gente con bajos ingresos estuvieran más expuestos a la contaminación del aire [RKHZ17]. Por otro lado, el segundo estudio busca encontrar cuales son los factores determinantes de las concentraciones de contaminación en distintos medios de transporte, pudiendo ser desde el viento hasta la velocidad del medio entre distintos tipos de partículas de contaminación como son black carbon (BC), partículas de número de concentración (PNC), medida de la cantidad de partículas en el aire y partículas en suspensión, las cuáles son indicadores de contaminación urbana, seguido de un número que representa su tamaño en micras, PM_1 ($PM \leq 1\mu m$), $PM_{2.5}$ ($PM \leq 2.5\mu m$) t PM_{10} ($PM \leq 10\mu m$) [Riv+17].

Hasta ahora, al menos partiendo de estos dos estudios iniciales, no se han realizado muchos estudios que aborden el problema con el objetivo de **predecir** la contaminación (teniendo en cuenta como medida de contaminación la cantidad de particulas perjudiciales que puedan encontrarse en el aire). Los beneficios de predecir la contaminación son innumerables. Por ejemplo, en nuestro caso, si se pudiera llegar a predecir la contaminación en el transporte público a corto plazo se podría

proporcionar a la gente medios para saber cuál sería el tiempo donde se encuentra menos concentración de contaminación de aire en el ambiente, lo que supondría estar menos expuesto a partículas perjudiciales para la salud, por ello predecir la contaminación será uno nuestros objetivos principales.

Para complementar esto, se busca encontrar si existe algún tipo de relación entre estos valores de contaminación y otros valores como pueden ser si justo en el momento de un pico de contaminación pasa un tren o incluso el número de personas que se encuentran en la estación en ese momento. Teniendo esto en cuenta hemos organizado nuestro trabajo en distintos capítulos. En concreto:

Contando esta **Introducción** como el primer capítulo donde hacemos un comentario global del trabajo, en el capítulo siguiente, **Background** (Antecedentes), profundizaremos más en los artículos comentados anteriormente para contextualizar el problema, revisaremos el trabajo anterior donde mencionaremos las técnicas utilizadas hasta ahora que consideramos más relevantes, las que tienen que ver en gran parte con el problema de predicción que se plantea aquí. En estos artículos anteriores, se han recogido distintos datos de contaminación en el aire a través de distintos medios de transporte mediante distintas rutas donde se buscaba alcanzar la mayor diversidad posible. Estos datos corresponden a valores de partículas de contamiación que se consideran perjudiciales para la salud, no vamos a entrar en detalle sobre como han sido recogidos ni que tipo de artefactos se han utilizado. Lo que principalmente nos interesa es que de datos de contaminación, algunos autores de esos mismos artículos nos han proporcionado amablemente un conjunto de datos relacionado con una estación de particular del metro de Londres, en concreto, la estación de South Kensington. Este conjunto de datos consiste en valores por minuto de la partícula PM_{10} durante una semana entre el 4 de Octubre de 2020 y el 11 de octubre de 2020. La partícula PM_{10} (Particulate Matter) se suele utilizar como indicador de contaminación, corresponde a partículas que se encuentran de cierto tamaño en micras, en este caso 10, algunos ejemplos pueden ser polvo o humo.

En el capítulo tres, **Methods** (Métodos), explicaremos los principales métodos que usamos en el trabajo con la finalidad de alcanzar los objetivos propuestos. Primero, para estudiar si existen relaciones entre nuestros datos de contaminación, horarios de trenes y la cantidad de gente que se encuentra en la estación. Utilizamos la API del transporte público de Londres para obtener horarios de trenes que pasan por la estación donde han sido recolectados estos datos de contaminación y cantidad de conexiones wifi en la misma estación, lo que nos proporciona una idea aproximada de la gente que puede encontrase en la estación. Explicaremos como hemos utilizado la API y tras obtener estos datos con fines comparativos, aplicamos distintas técnicas para preprocesar y limpiar nuestros datos lo cuál eliminará posibles errores o irregularidades que encontremos. Es importante tratar con el conjunto de datos de esta manera debido a que las técnicas de comparación y/o predicción son muy sensibles a este tipo de irregularidades, por eso escalar y regular nuestros datos es algo muy positivo. También hablaremos de series de tiempo (**time series**) ya que

nuestros datos corresponden con este tipo de secuencia de datos, valores que dependen del tiempo. Veremos cómo estudiar y analizar series de tiempo ya que series con ciertas propiedades proporcionan son más deseables respecto a otras. Por ejemplo, series que tengan la propiedad de estacionalidad implican que sus propiedades, concretamente su variación, no cambia en función del tiempo, lo cuál tiene importantes repercusiones a la hora de predecir. Finalmente se explicaran métodos para **comparar** nuestros datos obtenidos buscando correlaciones y similitudes. Como algoritmo destacamos **Dynamic Time Warping**, que nos permitirá medir y comparar dos frecuencias (series de tiempo) tanto en tiempo como en espacio. Este algoritmo es comúnmente utilizado para reconocimiento de voces, ya que permite distinguir si dos personas hablan a distintas veclodiades. También veremos maneras de **predecir** la contaminación, en particular, mediante modelos de series temporales que utilizen regresiones aprovechando distintas propiedades de una serie temporal, junto a técnicas de deep learning para predecir tanto un valor como un conjunto de valores teniendo en cuenta valores anteriores.

En el capítulo cuatro, **Experiments** (Experimentos), se aplicarán los métodos definidos anteriormente. Inicialmente se realizará una visualización (**Exploratory Data Analysis**) para hacernos una idea de la estructura de nuestros datos, esta visualización contendrá cierto nivel de análisis ya que nos ayudará a exponer tendencias, patrones y relaciones que no son evidentes a simple vista. Seguidamente se realizará, mediante lo visto en el capítulo anterior, comparación de nuestros datos con la finalidad de encontrar correlaciones de los mismos. Por ejemplo, incialmente no sabemos en que plataforma de la estación se han tomado nuestros valores de contaminación, pero tal vez si comparamos estos valores con los horarios de trenes en ambas direcciones, podemos saber en que plataforma (dirección este o dirección oeste) han sido tomados estos datos. Finalmente se experimentará con distintos métodos de predicción con la finalidad de predecir la contaminación de un valor o de varios valores, esta predicción estará condicionada por los datos y las propiedades de nuestra serie de tiempo, como ya nos podemos imaginar.

Finalmente, en el capítulo cinco, **Conclusions and further work**, extraemos resultados relevantes y comentaremos sus implicaciones. Veremos si hemos alcanzado los objetivos propuestos y las expectaciones iniciales. El tema de la contaminación ambiental ofrece infinidad de posibilidades, concretamente, la predicción de contaminación es algo recurrente con gran importancia a día de hoy. Analizar a tiempo real la contaminación en el medio es un problema dificil en la que influyen muchísimos factores. Veremos lo que hemos conseguido con nuestros experimentos, tanto de comparación de nuestros conjuntos de datos como de predicción de contaminación, y daremos lugar a más posibilidades que pudieran surgir de esta primera toma de contacto.

Palabras clave: Contaminación ambiental en el aire, análisis de series temporales, API del transporte público de Londres, comparación de series temporales, previsión de contaminación.

Chapter 2

Background

Air pollution is considered a major threat to human health because of its link to an increased mortality and loss of disability-adjusted life years. Combustion emissions, especially particles in various size ranges, are suspected to be particularly harmful. In this section we describe **two** articles dealing with the problem of air pollution in London's transport microenvironments from different approaches. But first of all, some definitions on pollution particles to make the reading clearer.

- **PM** stands for **particulate matter** [Pm] (also called particle pollution), the term for a mixture of solid particles and liquid droplets found in the air.

  Some particles, such as dust, dirt, soot, or smoke, are large or dark enough to be seen with the naked eye. Others are so small they can only be detected using an electron microscope. These particles come in many sizes and shapes and can be made up of hundreds of different chemicals.

  Some are emitted directly from a source, such as construction sites, unpaved roads, fields, smokestacks or fires. Most particles form in the atmosphere as a result of complex reactions of chemicals such as sulfur dioxide and nitrogen oxides, which are pollutants emitted from power plants, industries and automobiles.

  Particulate matter contains microscopic solids or liquid droplets that are so small that they can be inhaled and cause serious health problems. Some particles smaller than 10 micrometers in diameter can get deep into one's lungs and some may even get into one's bloodstream. Of these, particles less than 2.5 micrometers (μm) in diameter, also known as fine particles or $PM_{2.5}$, pose the greatest risk to health.

- **Black carbon (BC)** is the sooty black material emitted from gas and diesel engines, coal-fired power plants, and other sources that burn fossil fuel [Bc]. It comprises a significant portion of **particulate matter or PM**, which is an air pollutant.

- Particle number concentrations is the total number of particles per unit volume

of air. In this case we are dealing with ultrafine particles (with an aerodynamic diameter of $0.1\mu m$ or less), which are present at high concentrations near busy roadways, and are associated with markers of cardiovascular and respiratory disease risk.

Going back to the articles, both are related to air pollution and have some authors in common, both were supported by the UK's Economic and Social Research Council [grant number ES/N011481/1]

The first one, **Exposure to air pollutants during commuting in London: Are there inequalities among different socio-economic groups?** [RKHZ17] aimed at finding potential relationships between income deprivation and pollutant levels on the basis that people with low income often experience higher exposures to air pollutants. The study was carried out in Greater London (the broad area including and surrounding the City of London) and assesses the inequalities in exposure to air pollutants during commuting using real time personal measurements, thus providing a precise input of exposure concentrations. Data were collected using different measurement instruments to monitor (for practical reasons and because of their potential health effect) PM_1, $PM_{2.5}$, PM_{10}, BC and PNC for typical commutes by car, bus and underground from 4 London areas with different levels of income deprivation (from most to least deprived). Monitoring networks only provide a partial insight in personal exposure since this differs greatly with activity, location and time spent on each activity. Comparison between income areas in each mode of transport shows that global PM concentrations were dominated by the very high concentrations in the underground. Among transport modes PM concentrations were significantly highest in the underground ($PM_{2.5} = 34.5\mu gm^{-3}$) for all fractions, followed by the bus ($PM_{2.5} = 13.9\mu gm^{-3}$) and being lowest in the car ($PM_{2.5} = 7.3\mu gm^{-3}$).

According to the authors, no relationship was found between income deprivation and pollutant concentrations but the work identified the main drivers of exposure during commuting suggesting that differences between transport modes are a stronger influence. We highlight some statistical models used to analyze the differences among means.

- PM and PNC concentrations vary across the groups with a statistically significant difference (**Kruskal-Wallis test** [MN10], $p = 0.05$). Concentration at the origin of all pollutants varies across income groups with a statistically significant difference (**Kruskal-Wallis test**, $p = 0.05$) which points to a difference in the background concentrations.

- From the pairwise **Dunn's test** (significant at $p = 0.05$), the second most deprived zone emerged with significantly lower PNCs unlike the other three groups that were not significantly different from each other. **Dunn's test** is the appropriate nonparametric pairwise multiplecomparison procedure when a **Kruskal–Wallis** test is rejected. [Dun64]. The **Dunn's test** showed in many cases that the differences among some groups are not statistically significant.

- Fe influence (iron) is demonstrated by the correlation between BC and all PM fractions, with an $r > 0.85$ **(Pearson)** in the underground measurements, but only $r \leq 0.40$ for the car and bus measurements.

As a future work, additional studies are needed to evaluate exposure and deprivation at individual level for all the London population, in order to further explore the inequalities in exposure to air pollutants during commuting. It is also proposed to incorporate the results from this study into predictive models of exposure during commuting, what motivates one of our main objectives.

The second study, **Determinants of black carbon, particle mass and number concentrations in London transport microenvironments** [Riv+17] attempted to find the determinants of personal exposure concentrations of commuters pollutants, black carbon (BC), ultra fine particle number concentrations (PNC), and particulate matter (PM) investigating in different travel modes. This study contributed to the understanding of factors determining the exposures while traveling in different transport modes to assess the determinants of personal concentrations of particles during commuting through regression modeling. In the first study [RKHZ17] potential factors affecting the concentrations in the different transport microenvironments are identified, while the focus in this work is to identify and quantify the statistically significant predictor variables in order to build an explanatory model of the concentrations faced for travels by car, bus, underground and walking.

The correlation for BC concentrations for the car mode is moderate, but weak for PNC and especially for the PM fractions. The validation of the concentration in bus trips is made based on only four observations, and in some cases, it is driven by a simple data point. Moreover, the reported flat slopes indicate that the model is not able to capture the variability of the actual concentrations

The non-normality of the concentration data requires the use of non-parametric models (such as the Generalised Linear Models, GLM) or the decimal log transformation of the concentrations

- **Linear regression** models with analysis of covariance **(ANCOVA)**, which allows the input of categorical and continuous variables, was performed to assess the determinants of the mean concentration faced while travelling by car, bus, underground or walking. Also they performed a linear regression of measured $PM_{2.5}$ and PM_{10} concentrations during car, bus and walking trips against two different sets of ambient $PM_{2.5}$ and PM_{10} data. Ambient PM concentrations at a high temporal resolution from a single monitoring station explained a higher variation of concentrations in car trips than daily averages spatially matched, while the contrary was observed for bus trips.

- Concentrations during commuting times might be affected by more than just one variable, therefore **multivariate regression** models were performed assessing the effect of wind speed together with other possible predictor variables.

- For multiple regression analysis, for the ease of comparison, the adjusted R^2 from the multivariate models (ratio of the sum of squared residuals of the corresponding variable to the total sum of squares) for the entire model is presented.

Much of the observed variation in exposure remains unexplained and there are some inconsistencies across different studies. Further research is required to understand the factors that explain the variability in the exposures during commuting. Unravelling the relative roles of determinant factors is key for developing successful strategies for air quality management in transport microenvironments. Models for air pollutants during commuting are required to explore their potential health effects or possible environmental injustices in population-based studies and can be incorporated into larger models assessing the daily exposure at an individual level. The COVID-19 pandemic has stressed the necessity of understanding and caring about what commuters breathe while traveling. This strengthens the idea (suggested by the study) that additional variables should be analysed, for instance:

- Direction of the wind, either parallel or perpendicular, and even the wind speed.
- The distance to the centre of the street for walking mode.
- Traffic intensity and composition, travel speeds, road size, type of fuels

The idea is to include this type of variables in future studies in order to explain remaining unexplained components of the variation of concentrations in commuting exposures.

In some cases, the models were able to explain a large component of the variation in concentrations (especially in the underground trips), and this information can be used to reduce personal exposures of London commuters as well as to extrapolate the estimates of exposure at a population level for other investigations such as epidemiological studies. Understanding the variation in concentration could contribute to more efficient and inclusive air quality policies in urban centres and better urban planning. A policy relevant addition to this study would be to measure the personal exposure of cyclists for different routes, given Transport for London's ambitions to develop cycling as an integral part of London's transport system.

Chapter 3

Methods

In this chapter, we will explain the architecture employed for the project, how to use London public transport API to build time series, two time series for train arrivals and departures at a specific station, and another time series for crowding levels of the station. As we aim to align and combine all these series, we will thus study different methods to compare time series. Additionally, and as a first step, we will carry out some feature engineering to clean and process our datasets. As a final objective, we seek to create predictive models of contamination both from real and synthetic data.

Accurately modeling pollution processes is an extremely complicated task due to the different factors affecting temporal trends and spatial distributions, which include air pollutant emissions and deposition, and weather conditions. Traditional modeling and forecasting techniques require the explicit identification and weight of those factors. In contrast, **Deep learning** algorithms can extract representative features without prior knowledge and may lead to a good performance for pollution predictions, including in the London underground.

The rest of this chapter is organised as follows: in Section [3.1] we will describe the software platform where we have developed all the experiments with the different technologies used. Next, in Section [3.2], we will show the Transport for London Unified API, and how we will use it to build new datasets to study if there is some kind of relationship between the pollution measured on the station platform and the passage of trains, specifically, series of train schedules and crowding level of the station. After seeing how to build these timeseries, in Section [3.3] we will see how to preprocess and clean our datasets. In Section [3.4], we will see statistical tests that will allow us to deduce properties of timeseries, timeseries with certain properties are easier to compare and predict. Finally, in Section [3.5] we will show the Dynamic Time Warping algorithm (that we will use to compare time series), and in Section [3.6] the prediction models that we will use to make this pollution prediction (specifically regression models and deep learning).

3.1 Developing environment

Python is an open-source, general-purpose high-level programming language. We chose Python because, besides been the most popular programming language data scientists use nowadays[1], it is a powerful computational tool to solve complicated tasks in the fields of finance, econometrics, economics, data science, and machine learning. Therefore, it is a language for programming all kinds of applications that was designed to be easy to use and fun. It is also the perfect option for this type of work because it supports a large number of scientific libraries.

Python can be used in different types of environments. In our case we have chosen to use Jupyter Notebook, a client-based interactive web application that allows users to create and share codes, equations, visualizations, as well as text. Whether to analyse a collection of written text, creating music or art or to develop engineering concepts, Jupyter Notebook can combine codes and explanations with the interactivity of the application. This makes it a handy tool for data scientists for streamlining end to end data science workflows.

In our case and as a personal preference, we are going to work using a docker images [`https://docs.docker.com/get-started/overview/`]. Docker is an open platform that enables one to separate the applications from the infrastructure providing the ability to run an application in a isolated environment called a container. In our case we will use a **Python docker image**, which will automatically load all the dependencies and packages that the project needs in a container that docker will run. We will not even need to have the specific version of python installed locally, we will only have to load the docker image and docker will be in charge of creating the development environment with everything you need.

This option has been chosen because the version management of our modules and even python itself is trivial in this way. This avoids that, for example, if all the methods and experiments carried out on another machine were reproduced, it could give rise to problems and even different results of the experiments due to possible different versions. Also the ease of making the entire development environment work in this way is very simple, because the only requirement is to use docker.

We use the visual studio code editor with an extension to work with these docker containers [`https://code.visualstudio.com/docs/remote/containers`]. Likewise, although jupyter notebook is a web interface, we have used a extension so that the visual studio editor can reproduce the notebook.

Long story short, the tools we have used to develop this project are:

- Python v3.6

[1] More information about this in the following url `https://www.jetbrains.com/lp/python-developers-survey-2020/`

- Jupyter notebook
- A Docker image with all the necessary dependencies

But the only requirement is to use docker. The docker image will be a default image of python version 3.6 taken from the official Microsoft Github repository[2] as an example of how to use containers in visual studio code[3].

The only thing that we configure in this default image will be that when docker creates the container where to work, it takes into account all the dependencies and modules necessary for it to install them, this is done by creating a file (`requeriments.txt`) with all the necessary modules and telling docker to install the dependencies found in the file.

In all code developed in this work we have applied the good Python practices and the coding style (one of many) established by its creator, Guido van Rossum in PEP 8 - Style Guide for Python Code. This Style Guide for Python Code can be found in the official python webpage given by the Python Software Foundation[4].

All the technical work done during the project is stored in the **project's Github repository** [`https://github.com/daviduster/deep-learning-london-pollution`], but due to the confidentiality of the data, this repository has restricted access. Anyway, anyone interested can find all the experiments and notebooks on [`sl.ugr.es/tfgdavidmv1996`], a **webpage created exclusively to present the notebooks**. For the deployment of the website, we have also added a continuous deployment (CD) process. We have automated that any change made in the repository will cause the website to be automatically updated. For example, suppose we change some of the Jupyter notebooks, save these changes and push them to the Github repository where it is located. In that case, the website will be automatically updated thanks to the CD process.
The page is hosted on the **Netlify** [5] service, where an index leads to the different notebooks used to write the memory. The different notebooks found on the site are as follows:

- **Vanilla notebook**, the essential notebook where you can find the not-specific stuff (e.g. visualisations, use of Transport For London API, application of statistical tests, analysis of time series, etc.)
- **Dynamic Time Warping (DTW)**. This notebook contains everything related to the comparison of time series, visualisations and the study of the correlation between series. Finally, it concludes by applying the Dynamic Time Warping algorithm.

[2] `https://github.com/microsoft/vscode-remote-try-python`
[3] `https://code.visualstudio.com/docs/remote/containers-tutorial`
[4] `https://www.python.org/dev/peps/pep-0008/`
[5] Netlify is a cloud computing company that offers hosting and serverless backend services for web applications and static websites.

- **ARIMA model**. In this notebook, we build the ARIMA forecasting model. It also searches for the best order by brute force and subsequent visualisation, analysis and prediction attempts with the model.

- **Deep Learning**. Timeseries forecasting using deep learning models from the TensorFlow python module. In particular, it contains the Linear, Dense, Convolutional Neural Network and Recurrent Neural Network (LSTM) models both for predicting one pollution value and several pollution values. It also has prediction visualisations and graphs to evaluate performance between models.

3.2 Transport for London Unified API

Transport for London (TfL) is a local government body responsible for most of the transport network in London. TfL have been a leader amongst Transport and Government departments in the provision of free and open data to the public, and actively encourage the use of data by 3rd party developers across multiple application domains, with a data subscriber database of $+5000$ registered application developers and organisations.

With the use and integration of TfL's Open data, developers have produced a wide and varied selection of mobile and desktop applications, spanning the fields of travel and trip planning to historic city data analysis and mining. Some recent studies have even used TfL's data to detect traffic patterns in real time [GFMS21].

As an early adopter of the Open Data initiative (particularly within transport), TfL has provided Open Data since 2007. All this information has been unified in the **Transport for London Unified API**.

In this section we will explain what the Transport for London Unified API consists of and how we have worked with it to build new datasets. We remind the reader here that we aim to study if there is some kind of relationship between the pollution measured at the station platform (South Kensington Underground Station) and the passage of trains. For that, we need train schedules in inbound and outbound directions together with crowding levels, all these datasets in the same station where we have our pollution data. The API will not directly provide us with the data we are looking for, with the information provided by the API we will build the desired datasets of train schedules and crowding levels.

As the API is somewhat complex, we will do a general review of how to use it, and then we will describe the functionalities that we have used mainly to fulfill our objective.

TfL's unified API aims to simplify accessing the key public information across all modes of transport, so to obtain information related to London transport, the use of this API seems one of the best solutions on a technical level.

What's the Unified API?

The Tfl unified **API** (Application Programming Interface) brings together data across all modes of transport into a single RESTful API[6].

This API provides access to the most highly requested real time and status information across all the modes of transport, specifically (and useful for our use case) for the London underground, in a single and consistent way.
The API supports all the data requirements of the TfL website. Every data-driven aspect of the website (including maps) is powered by the unified API.
Some of the multi-modal core datasets included and available to developers can be find in the official API documentation[7]:

- Journey Planning (current and future)
- Status (current and future)
- Disruptions (current) and Planned works (future)
- Arrival/departure predictions (instant and websockets)
- Timetables
- Embarkation points and facilities
- Routes and lines (topology and geographical)
- Fares

Additionally, the API supports an extensive capability for looking up and matching locations by name, postcode etc. It also includes cycle hire data.
Other datasets are also available for Cabwise, providing locations of registered taxi firms and WebCAT, which includes modelling information on transport, such as travel times between locations.

These different datasets can be obtained through calls to different endpoints of the API. The API also has an elaborate documentation where most of the typical cases of use of the same can be found, and if necessary, for more technical guidance there is a forum TfL Forum where people who are used to using the API and even developers answer questions, and the TfL Digital Blog; I have used both references during the course of this work.

[6] An API is a software intermediary that allows two applications to talk to each other. More information on what a RESTful API is can be obtained at `https://www.redhat.com/en/topics/api/what-is-a-rest-api`

[7] `https://tfl.gov.uk/info-for/open-data-users/unified-api`

What we pursue using the API

Figure [3.1] shows a map of London with the Piccadilly line superimposed. Figure [3.2] only shows the line with all its stations.

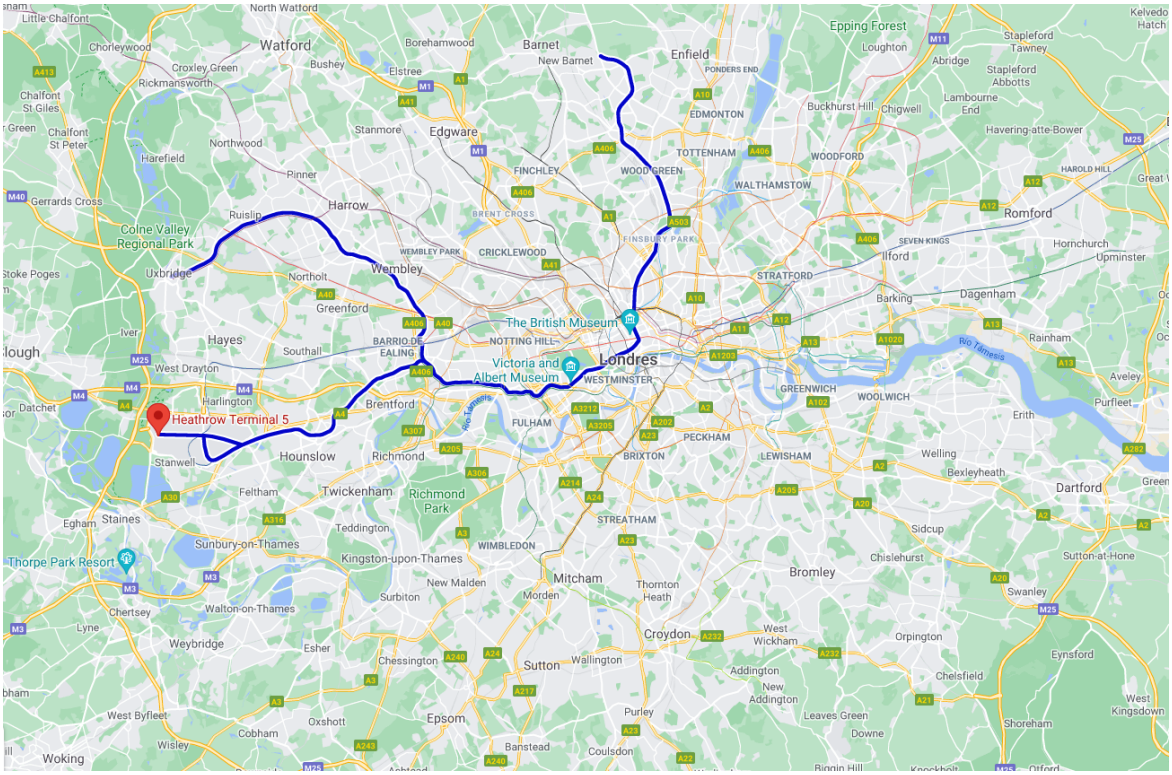


Figure 3.1: London map with Piccadilly line superimposed

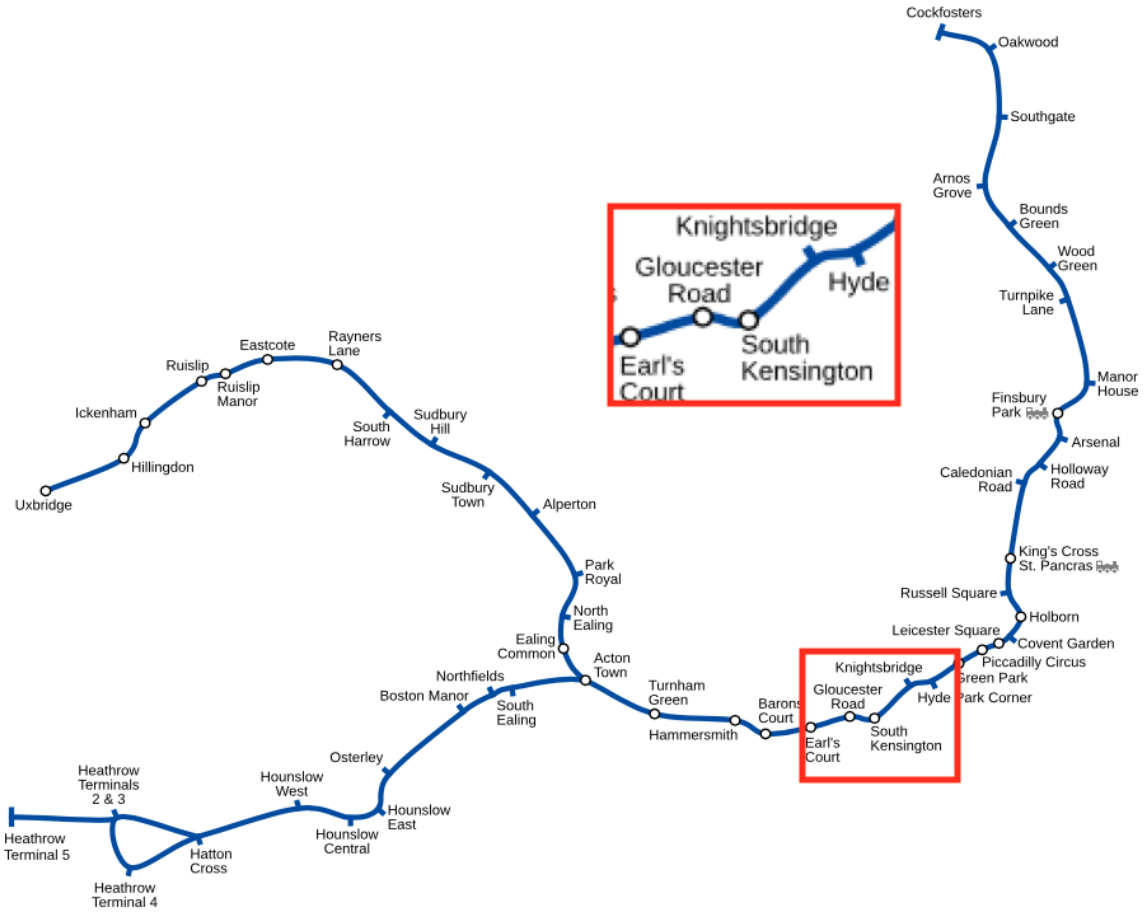


Figure 3.2: Piccadilly line

The Piccadilly line is a deep level London Underground line running from the north to the west of London. We use TfL's portal[8] to obtain more information about the line, as well as to plan routes. As we have commented previously, everything that is found in the TfL portal is powered by calls to the API, demonstrating the potential that this tool has. Our air quality time series corresponds to air pollution measurements (of the PM_{10}, urban pollution indicator particle smaller than $10\mu m$) located in the **South Kensington** station of this line.

We want to study if there is a relationship between the pollution measured on the station of South Kensington and the passage of trains on the same station, for this proposal we seek to obtain the train schedule in both directions (inbound and outbound) and information on crowding levels of the station using the TfL API.
As an added difficulty, South Kensington station, the station we have the data from, is currently under renovation works from February 2021 to March 2022. As we can see in the official [TfL webpage].

The API will provide us with the time it takes a train to get from one station to the next one (later we will see how we manipulate this information to obtain train

[8] `https://tfl.gov.uk/tube/route/piccadilly/`

schedules), but of course, since the South Kensington station is closed, that means we do not know how long it takes for a train to arrive from the previous station to this one since the trains will not stop, so we will not know exactly when a train pass through the station. What is certain is that the train schedule is the same always, and therefore we are going to assume that the train schedule when the data was collected is the same as it is today, but as currently no trains pass anymore for the station since is closed, we will approximate this calculation.

Looking at the Piccadilly line (Figure [3.2]), the neighbouring stations to South Kensington are Gloucester Road and Knightsbridge. If we know the time it takes for a train to go from Gloucester Road to Knightsbridge (or viceversa), we will assume that the time it takes to get to South Kensington is half the time, and in this way we will have an approximate time of when a train passes.

As an extra, and because the API allows it, we also use data related to the crowding of the station, with the same purpose as the previous series, to compare and search for a correlation with the original series, so we will have, for a given station, contamination data, trains passing through, and number of people in it. The intuitive hypothesis is that all this information will be related.

Using the unified API

The API is designed to be as simple as possible in its usage, and is freely available to all. The unified API is designed for applications to use it in real time and at high volume. Many of the calls we make will be answered with real time data. The default response format is `JSON`, but developers can also request XML if preferred.

`JSON` is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types. It is a very common data format, with a diverse range of applications, such as serving as a replacement for `XML` in `AJAX` systems. Listing [3.1] shows an example of what a `JSON` looks like.

```
myObj = {
  "name":"John",
  "age":30,
  "cars": {
    "car1":"Ford",
    "car2":"BMW",
    "car3":"Fiat"
  }
 }
```

Listing 3.1: JSON example

When the data emerges from the API, it is uniformly consistent in output and structure. The core benefit for this approach is that with the API acting as a facade, the logic and processes behind creating the API and merging the datasets are abstracted away from the user.

First of all, we start from the main page of the API[9]. This site is the developer portal for Transport for London's OpenAPI. Developers are encouraged to use it to view documentation and generate subscription keys to access the API.

Access to the API is free to all, and one can easily try the endpoints of the API in the browser or in any terminal without any type of authorization. The only disadvantage of using the API in this way, without any further login, is that there is a limit of 50 request per minute.

In our case this is more than enough since we do not need to make many calls per minute since the data we are looking for, train schedules, will not change, so we don't have to constantly update it. A similar argument applies to the crowding level case.

However, this limitation can be easily extended by registering [`https://api-portal.tfl.gov.uk/signup`]. General registered users are able to make up to 500 requests per min.

Registering will provide us with an application ID and Key, which we can add as query parameters to our requests to avoid the limit of unregistered users. Figure [3.3] shows how we can check for the keys.

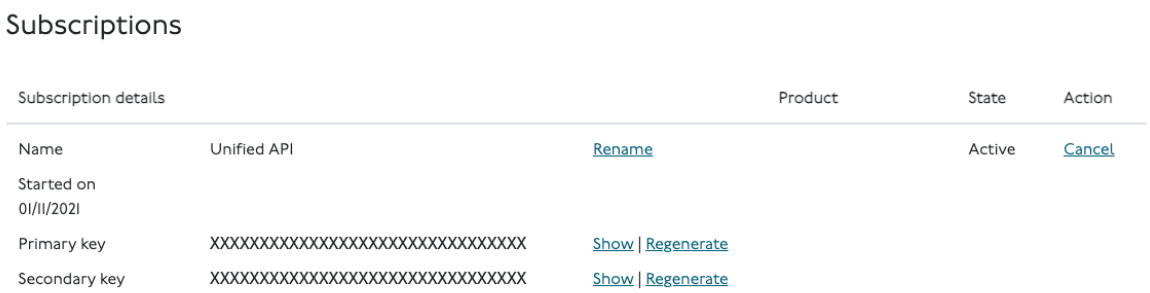


Figure 3.3: Example of tokens

Endpoints description

In this section we are going to explain the endpoints that have been used, then we will see how we have automated the call of these endpoints and manipulated the information in python.
We have already seen what the API consists of, and the "bureaucratic" needs to use the API. Append our unique `app_id` and `app_key` as querystring parameters let us to get higher rate limits than anonymous users.

As said before, the unified API presents all the data that is semantically similar for

[9]Main page of the TfL API `https://api.tfl.gov.uk/`

each mode of transport in the same format and consistent structures. This enables you to write once, and access all of the same types of data across all the modes of transport quickly, making multi-mode application development easier.

This sounds good, but how do we get this information? Being a restful API, it provides us with a series of endpoints to call to obtain information.

An endpoint is one end of a communication channel. When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL[10] of a server or service. Each endpoint is the location from which APIs can access the resources they need to carry out their function.

APIs work using "requests" and "responses". When an API requests information from a web application or web server, it will receive a response. The place in an API to which requests are sent and where a resource lives, is called an endpoint.
The API portal is sufficiently well documented regarding the available endpoints; it makes available a swagger[11] and even a collection in postman[12]
In the API, the core identifiers for all stations and platforms have been normalised to the UK national Naptan standard. This standard is an identification scheme that is supported by the UK Department for Transport (DfT) nationally, allowing the API to integrate data from transport authorities outside of London. The complexity of mapping between multiple identification systems used within TfL is thus hidden from consumers of the API.

StopPoint

So first, it makes sense to see how to obtains this Naptan codes from the lines we need. The way to do it is using the `StopPoint` endpoint, that search `StopPoint` by their common name, the endpoint in the [`StopPoint` Swagger documentation]. For getting the Naptan code for the South Kensington Underground Station, we can call this URL endpoint with that name and the mode as tube, as we said before, the endpoint is just to call an URL adding the headers of keys. To call the endpoint, we can use the same browser or anything that makes http requests, such as `curl`[13]. The same swagger provides an example of how to call the endpoint, Listing [3.2].

```
1 GET https://api.tfl.gov.uk/StopPoint/Search?query=South%20Kensington
     %20Underground%20Station&modes=tube HTTP/1.1
2
3 Cache-Control: no-cache
```

Listing 3.2: GET `StopPoint Naptan code`

[10]URL stands for Uniform Resource Locator. A URL is nothing more than the address of a given unique resource on the Web.

[11]Swagger is an Interface Description Language for describing RESTful APIs expressed using JSON.

[12]Postman is a popular API client that makes it easy for developers to create, share, test and document APIs.

[13]tool for transferring data using network protocols, the name stands for "Client URL"

It's input parameters are in Table [3.1].

Name	Type	Description
query	string	The query string, case-insensitive.
modes	array	An optional, parameter separated list of the modes to filter by
faresOnly	boolean	True to only return stations in that have Fares data available for single fares to another station.
maxResults	integer	Format - int32. An optional result limit,
lines	array	An optional, parameter separated list of the lines to filter by

Table 3.1: `StopPoint/Search` parameters

The endpoint makes a query of the requested string and returns matches, we will receive the following `JSON` response, Listing [3.3].

```
{
    "$type": "Tfl.Api.Presentation.Entities.SearchResponse, Tfl.Api.
   Presentation.Entities",
    "query": "South Kensington Underground Station",
    "total": 1,
    "matches": [{
        "$type": "Tfl.Api.Presentation.Entities.MatchedStop, Tfl.Api.
   Presentation.Entities",
        "icsId": "1000212",
        "topMostParentId": "940GZZLUSKS",
        "modes": ["tube", "bus"],
        "zone": "1",
        "id": "940GZZLUSKS",
        "name": "South Kensington Underground Station",
        "lat": 51.494094,
        "lon": -0.174138
    }]
}
```

Listing 3.3: South Kensington StopPoint response

To use the application ID and key obtained when registering (although it will not be included for convenience, since in our case its not really important), there is an example in the Listing [3.4].

```
GET https://api.tfl.gov.uk/StopPoint/Search?query=South%20Kensington
   %20Underground%20Station&modes=tube&app_id={{app_id}}&app_key={{
   app_key}} HTTP/1.1

Cache-Control: no-cache
```

Listing 3.4: Example adding credential keys

Looking at the response, the only value of real interest is `"id":"940GZZLUSKS"`, which will be the identifying Naptan value of that station that is used in other endpoints.

Although now the entire response is shown, the following endpoints will return much larger responses, thus we will show them in a more compressed and reduced way, focusing only in the parts that we need.
Now that we know how to obtain the Naptan code of the stations, the next step will be to obtain the train schedules of the Piccadilly line; from them, we will look for those trains that pass through the station of South Kensington, the station where the measurements were taken.

Timetable

Let's see the other two endpoints that we will use; proceeding as before, we will explain the endpoints and we will see an example of use. The endpoint `Line` relating to Line and similar services, has two `GET` requests about getting timetables. The first one gets the timetable for a specified station on the give line with specified destination. Listing [3.5].

```
GET https://api.tfl.gov.uk/Line/{id}/Timetable/{fromStopPointId}/to/{
    toStopPointId} HTTP/1.1

Cache-Control: no-cache
```

Listing 3.5: Get timetable from to

Its associated request parameters are in Table [3.2].

Name	Type	Description
fromStopPointId	string	The originating station's stop point id (station Naptan code e.g. 940GZZLUSKS, you can use /StopPoint/Search/{query} endpoint to find a stop point id from a station name)
id	string	A single line id e.g. piccadilly
toStopPointId	string	The destination stations's Naptan code

Table 3.2: Parameters of `Line/id/Timetable/{fromStopPointId}/to/{toStopPointId}`

Presenting it in the same way, the second endpoint gets the timetable for a specified station on the give line. It is the same as before but using instead `fromStopPointId`. Listing [3.6]. The request parameters are in Table [3.3].

```
GET https://api.tfl.gov.uk/Line/{id}/Timetable/{fromStopPointId} HTTP
    /1.1

Cache-Control: no-cache
```

Listing 3.6: Get timetable from to Id

Name	Type	Description
fromStopPointId	string	The originating station's stop point id (station Naptan code e.g. 940GZZLUSKS, you can use /StopPoint/Search/{query} endpoint to find a stop point id from a station name)
id	string	A single line id e.g. piccadilly

Table 3.3: Parameters of `Line/id/Timetable/{fromStopPointId}`

Although the names of the endpoints indicate respectively that they provide the timetables between two ids or from an id to the end of the line, empirically testing we see that the parameter `toStopPointId` is irrelevant, the endpoint will return the entire timetable of the line. The only thing that affects the response is Naptan Id of the line from which it starts.

We will use the first timetable endpoint, since the `toStopPointId` parameter gives us a good "stop and reference" criterion, when we manipulate the response with Python. Let's see an example, let's remember the map of the piccadilly line in the Figure [3.2], we will call the endpoint from **Hyde Park Corner Underground Station** (`940GZZLUHPC`) to **Gloucester Road Underground Station** (`940GZZLUGTR`). We obtain the Naptan code of both stations and make the call to the browser. Listing [3.7].

```
GET https://api.tfl.gov.uk/Line/piccadilly/Timetable/940GZZLUHPC/to
    /940GZZLUGTR HTTP/1.1

Cache-Control: no-cache
```

Listing 3.7: Request timetable from Hyde Park Corner to Gloucester Road

We compressed the response in the Listing [3.8] with dots [...] due to its large size.

```
{
  "$type": "Tfl.Api.Presentation.Entities.TimetableResponse, Tfl.Api.
    Presentation.Entities",
  "lineId": "piccadilly",
  "lineName": "Piccadilly",
  "direction": "inbound",
  "stations": [...],
  "stops": [...]
  "timetable": [...]
}
```

Listing 3.8: Response timetable from Hyde Park Corner to Gloucester Road

It must be said that the API documentation is not very descriptive regarding what the parameters of the responses of some endpoints consist of. In this case it also returns confusing and redundant information, but we managed ourselves around this without problem.

In this case the direction is `inbound`, let's focus on the response whose name is `timetable`, Listing [3.9].

```
"timetable": {
  "$type": "Tfl.Api.Presentation.Entities.Timetable, Tfl.Api.
Presentation.Entities",
  "departureStopId": "940GZZLUHPC",
  "routes": [
    {
      "$type": "Tfl.Api.Presentation.Entities.TimetableRoute, Tfl.
Api.Presentation.Entities",
      "stationIntervals": [...],
      "schedules": [...]
  }
```

Listing 3.9: timetable

As we have previously commented, both endpoints described on timetables are the same, the only thing that influences is the `fromStopPointId`, in the case of the `departureStopId`, the station from the `timetable` start.
Let's go deeper, we show now what is inside `stationIntervals`, Listing [3.10] and `schedules` inside `routes`, Listing [3.11].

```
  "routes": [
    {
      "$type": "Tfl.Api.Presentation.Entities.TimetableRoute, Tfl.
Api.Presentation.Entities",
      "stationIntervals": [
        {
          "$type": "Tfl.Api.Presentation.Entities.StationInterval,
Tfl.Api.Presentation.Entities",
          "id": "0",
          "intervals": [
            {
              "$type": "Tfl.Api.Presentation.Entities.Interval, Tfl
.Api.Presentation.Entities",
              "stopId": "940GZZLUKNB",
              "timeToArrival": 1.0
            },
            {
              "$type": "Tfl.Api.Presentation.Entities.Interval, Tfl
.Api.Presentation.Entities",
              "stopId": "940GZZLUGTR",
              "timeToArrival": 6.0
            },
            {
              "$type": "Tfl.Api.Presentation.Entities.Interval, Tfl
.Api.Presentation.Entities",
              "stopId": "940GZZLUECT",
              "timeToArrival": 7.0
            },
            ...
```

Listing 3.10: `stationIntervals`

This part is quite interesting, on the one hand, we have `stationIntervals` (see Listing [3.10]), which consists of a `JSON` containing a list of pairs of `ids` (from 0 to

the maximum number of routes) and an array of intervals where each element of the vector intervals contains a `stopIds` and `timeToArrival` to that specific station `stopId`.

Long story short, we have the different routes that depart from the station with the `"departureStopId": "940GZZLUHPC"`, and the time it takes to get to the next stations (in minutes, `timeToArrival`) together with the station id, `stopIds`. We also realize that the station that we use as a destination is merely decorative, it only serves to mark the direction (in this case inbound), since the response also returns the time it takes to reach all the successive stations.

For example, for the id `"id": "0"`, we have the sequence of stops `940GZZLUKNB` (**Knightsbridge**, 1 minute to arrive from **Hyde Park**), `940GZZLUGTR` (6 minutes to arrive from **Hyde Park**). The `timeToArrival` always represents the time it takes from the starting station, not between stops.
Going back to the Piccadilly line (Figure [3.2]) between **Knightsbridge** and **Gloucester Road** lies **South Kensington** station. In the Listing [3.10], the station South Kensington should appear as the next one, but it is not (the first Naptan code shoud be South Kensington, but is `940GZZLUKNB`, Knightsbridge, the next one). This is due to South Kensington being closed for the Piccadilly line trains, as we have previously indicated.

```
        "schedules": [
          {
            "$type": "Tfl.Api.Presentation.Entities.Schedule, Tfl.Api
   .Presentation.Entities",
            "name": "Monday - Friday",
            "knownJourneys": [
              {
                "$type": "Tfl.Api.Presentation.Entities.KnownJourney,
    Tfl.Api.Presentation.Entities",
                "hour": "5",
                "minute": "50",
                "intervalId": 2
              },
              {
                "$type": "Tfl.Api.Presentation.Entities.KnownJourney,
    Tfl.Api.Presentation.Entities",
                "hour": "5",
                "minute": "56",
                "intervalId": 9
              },
              ...
```

Listing 3.11: `schedules`

On the other hand, `schedules`, Listing [3.11] contains three sections representing a whole week, `"Monday - Friday"`, `"Saturdays and Public Holidays"` and `"Sunday"`. For each section of the week, the parameter `"knownJourneys"` returns a vector whose elements are pairs of hours of train departures from the departure station together with the `"intervalId"` of `"stationIntervals"` mentioned above,

then with this we will know the departure time of the train from the origin station, and the different route and how long it takes to get to the different stops on the line until the end. We can see this visually in Figure [3.4].

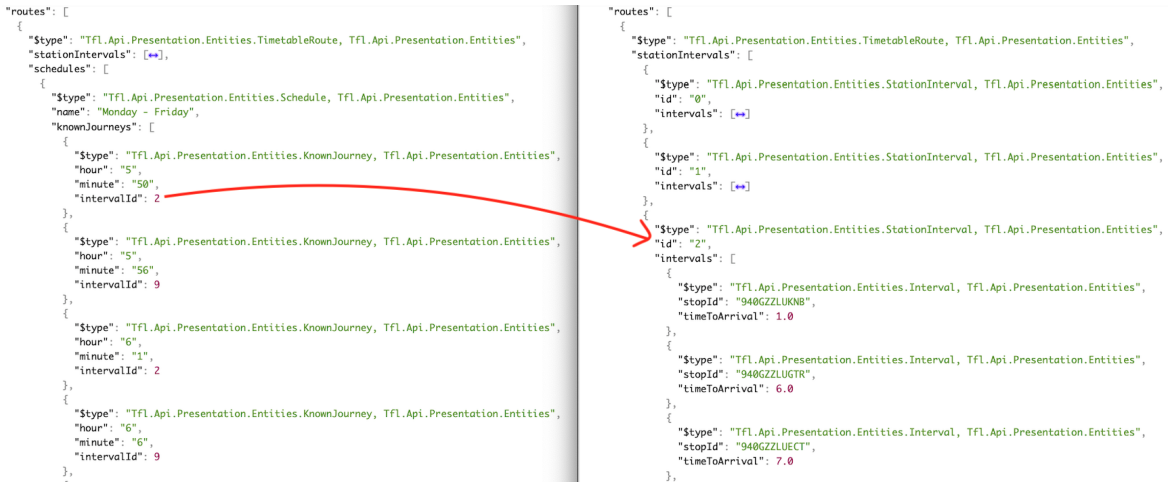


Figure 3.4: Relation between `schedules` and `stationIntervals`

The main idea with this endpoint will be, knowing the departure time of the trains from a previous station (inbound and outbound directions) to the South Kensington target station, and knowing how long it takes to reach the successive following stations, use an interpolation to obtain the approximate times of the trains that pass through the South Kensington station and generate a time series to compare with the original series.

Crowding information

The last endpoint we are going to use, [`Crowding`], Listing [3.12], gives information about crowding levels within TfL Stations.

```
1 GET https://api.tfl.gov.uk/crowding/{Naptan} HTTP/1.1
2
3 Cache-Control: no-cache
```

Listing 3.12: Crowding information for Naptan code

This endpoint only supports a single argument, the Naptan code identifier. South Kensington is only closed to the Piccadilly line while the escalators down to those platforms are replaced. The station is still open, and Circle and District line trains still stop at platforms 1 and 2.
Even if the station was closed, crowding information would still be available despite the lack of passengers, as the crowding API currently uses historical WiFi connection

data[14]. Then we can get the crowding information, which is based on the station's WiFi connections, from the South Kensington station, Listing [3.13], to obtain more information that we can use for comparative purposes with the original series, and the one that we create in the next section with the previous endpoint timetable.

```
GET https://api.tfl.gov.uk/crowding/940GZZLUSKS HTTP/1.1

Cache-Control: no-cache
```

Listing 3.13: Crowding information for Naptan code

The response from the Listing [3.14] consists of the different days of the week reduced using a 3 letter acronym (`MON` means Monday and so on), and different timebands of 15 minutes with a relative percentage based on the Wifi connections mentioned before which is not based on real time, but on connection history.

```
{
  "naptan": "940GZZLUSKS",
  "daysOfWeek": [
    {
      "dayOfWeek": "WED",
      "amPeakTimeBand": "07:30-09:30",
      "pmPeakTimeBand": "16:15-18:15",
      "timeBands": [
        {
          "timeBand": "00:00-00:15",
          "percentageOfBaseLine": 0.0019
        },
        {
          "timeBand": "00:15-00:30",
          "percentageOfBaseLine": 0.0019
        }, ...
        ]
    },
    {
      "dayOfWeek": "THU",
      "amPeakTimeBand": "07:30-09:30",
      "pmPeakTimeBand": "16:15-18:15",
      "timeBands": [
        {
          "timeBand": "00:00-00:15",
          "percentageOfBaseLine": 0.0029
        },
        {
          "timeBand": "00:15-00:30",
          "percentageOfBaseLine": 0.0029
        }, ...
```

Listing 3.14: Crowding information for Naptan code

[14]According to information from the forum and [New WiFi data API] TfL does, however, plan on releasing near real time data in the summer of 2021, so at that point, the crowding info would likely reflect any closures.

3.3 Exploratory Data Analysis

Basically, all machine learning algorithms use some input data to create outputs. This input data comprise features, which are usually in the form of structured columns. Algorithms require features with some specific characteristic to work properly. Here, the need for feature engineering arises. According to a survey in Forbes, data scientists spend 80% of their time on data preparation [For]. This metric is very impressive and shows the importance of feature engineering in data science.

As explained in [Kaggle][15], the goal of feature engineering is simply to make your data better suited to the problem at hand. Consider "apparent temperature" measures like the heat index and the wind chill. These quantities attempt to measure the perceived temperature to humans based on air temperature, humidity, and wind speed, things which we can measure directly. You could think of an apparent temperature as the result of a kind of feature engineering, an attempt to make the observed data more relevant to what we actually care about: how it actually feels outside!
Some techniques might work better with some algorithms or datasets, while some of them might be beneficial in all cases. In our case, because our original pollution dataset has only one column about PM_{10}, (and the others have been built manually) we focus on **data preprocessing**, imputation (missing values), handling outliers and scaling.
We will mainly use Pandas and Numpy python libraries. Let's take a glance at the pollution data we have. In the Listing [3.15] are the first/last rows from the file that has been provided to us.

```
data = pd.read_excel("../data/Underground_PM10 Data.xlsx", names = ["
    date_time", "PM10 [ug/m3]"])
```

Listing 3.15: Read pollution data

We can find the result in the following Table [3.4],

date_time	PM10 [ug/m3]
NaN	NaN
04Oct2020 (Sunday)	NaN
date & time	PM10 [ug/m3]
04/10/2020 00:00:00	36.4
04/10/2020 00:01:00	57
...	
Mean	49.8326
STDEV	38.8362
Median	41.45
Min	4.4
Max	212.2

Table 3.4: Reading pollution data

[15]Kaggle is a community of data scientists and machine learning practitioners by Google

Our pollution data has the PM particle data for 1 week, from Sunday, October 4, 2020 to Saturday, October 11, 2020, all in tick data (value per minute).
Each day begins with a header that indicates the day in a string (not numerical) followed by all the values per minute collected that day, finally the day ends showing the mean, median, variance, minimum and maximum value collected that day.
We will proceed to clean our data by eliminating this head and these values that are in the middle, simplifying it to two columns, a first column would be the index that would correspond to the tick data and a second column the value of the particle at that moment. In addition, we see that we have many **Nan** (Not a number) values, which represent undefined values in the dataset, we proceed to eliminate them directly.
The result of the dataset cleaning can be viewed in Table [3.5].

date_time	PM10 [ug/m3]
2020-10-04 00:00:00	36.4
2020-10-04 00:01:00	57.0
2020-10-04 00:02:00	25.3
2020-10-04 00:03:00	66.3
2020-10-04 00:04:00	44.6
...	
2020-10-10 23:55:00	90.6
2020-10-10 23:56:00	45.7
2020-10-10 23:57:00	45.3
2020-10-10 23:58:00	67.0
2020-10-10 23:59:00	67.6

Table 3.5: Pollution data after cleaning

These dates has been transformed into a python [datetime] type and the frequency set on minutes; this is useful due to the versatility and methods it provides.
Next, look at the statistics of the dataset in Table [3.6].

	PM10 [ug/m3]
count	10048.000000
mean	50.910938
std	40.926312
min	0.900000
25%	16.400000
50%	42.300000
75%	76.000000
max	756.100000

Table 3.6: Pollution data statistics

One thing that should stand out is the number of values, we have one week (10080 minutes) of data but only 10048 minutes, so we have missing values.

Also the maximum value is suspiciously large (according to the Environment Protection Authority, which gives an idea of which are common values, we confirm that the value of 756,10 is too large), so it must be an erroneous value. So it is clear that the pollution dataset has "outliers", we will deal with these irregularities. We accept the low values since they can be perfectly understandable, values before dawn where not even trains pass, only test trains that do not carry people.

Let's make an initial visualization of our data distinguishing the days of the week in different colors (Figure [3.5]). This color convention will be used from now on to associate a particular color with each day.

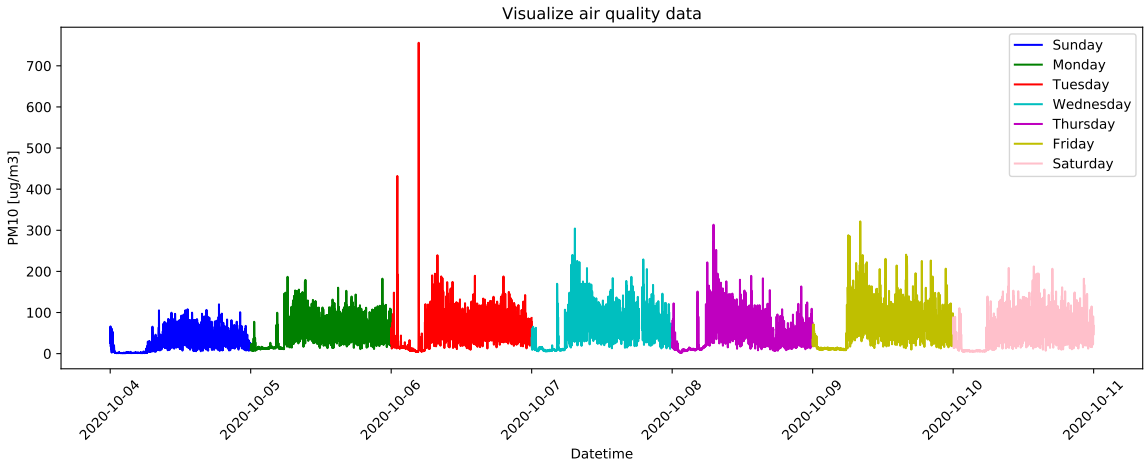


Figure 3.5: Visualize air quality data

Visually it is clearer what we have just commented, the existence of outliers (on Tuesday) in our dataset, which we will proceed to deal with next.

Missing values

Missing values are one of the most common problems you can encounter when you try to prepare your data for machine learning. The reason for the missing values might be human errors, interruptions in the data flow, privacy concerns, and so on. Whatever the reason is, missing values affect the performance of the machine learning models.

Some machine learning platforms automatically drop the rows which include missing values in the model training phase and it decreases the model performance because of the reduced training size. On the other hand, most of the algorithms do not accept datasets with missing values and gives an error. The most simple solution to the missing values is to drop the rows or the entire column. We already dropped all our missing values (Nan) so now we want to replace them for more realistic options.

As we have commented previously, a week has 10,080 minutes while we have 10,048 values, so we have 32 minutes missing. We can locate the missing values in Figure [3.6].

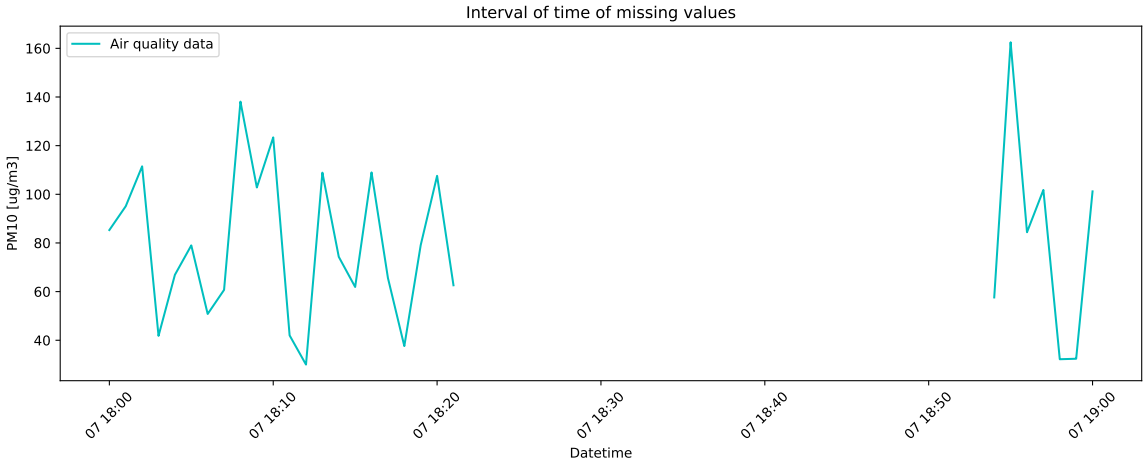


Figure 3.6: Interval of missing values

We see that the values are concentrated in a very specific interval, from `2020-10-07 18:22:00` to `2020-10-07 18:53:00`, these 32 missing values may be due to some kind of technical problem with the tool used to collect data or even some kind of human error. If null/NaNs are present, first identify why the data is missing and if NaNs mean anything. Missing values can be filled by interpolation, forward-fill or backward-fill depending on the data and context. We also need to be sure that null does not mean 0, which is acceptable but has modeling implications. It's important to understand how the data was generated (manual entry, ERP system), any transformations, assumptions were made before providing the data. The most interesting ones we use are the following:

- "ffill" stands for "forward fill" and will propagate last valid observation forward.
- interpolate, fill NaN values using an interpolation method, by default linear.
- And finally, using rolling windows calculation. We will calculate the mean/-median using a period of 33 minutes (since we have 32 missing values) and replace the missing values with the mean/median of the previous/next 33 values. In a very simple words we take a window size of k at a time and perform some desired mathematical operation on it.

We try these different ways to replace our missing values and visualize them in Figure [3.7] where we show the interval of missing values together with the possible substitutions techniques used.

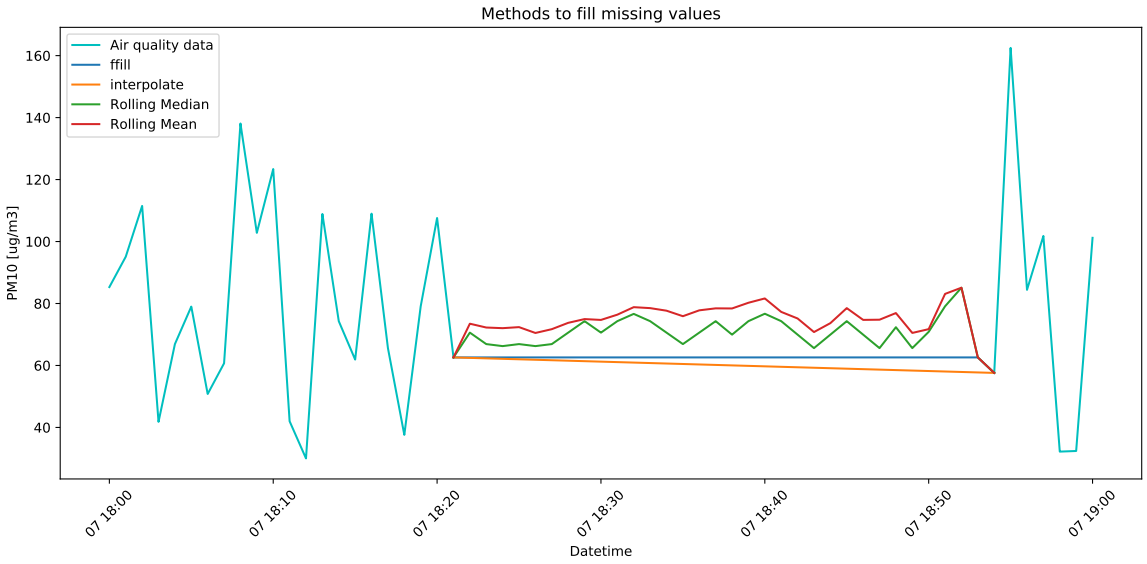


Figure 3.7: Plot replacement of missing values using different techniques

Finally we will stick with the rolling median technique because it is the one that more closely resemble the original signal, with peaks, also because moving median is a bit better than moving average for some applications (for example, it is less sensitive to outliers). It will work if you have a very short spike (preferably shorter than the median/average sample size). However, if you have a large spike, then taking the median will not help eliminating the spikes. Let's visualize the day which has missing values after replace them (Figure [3.8]).

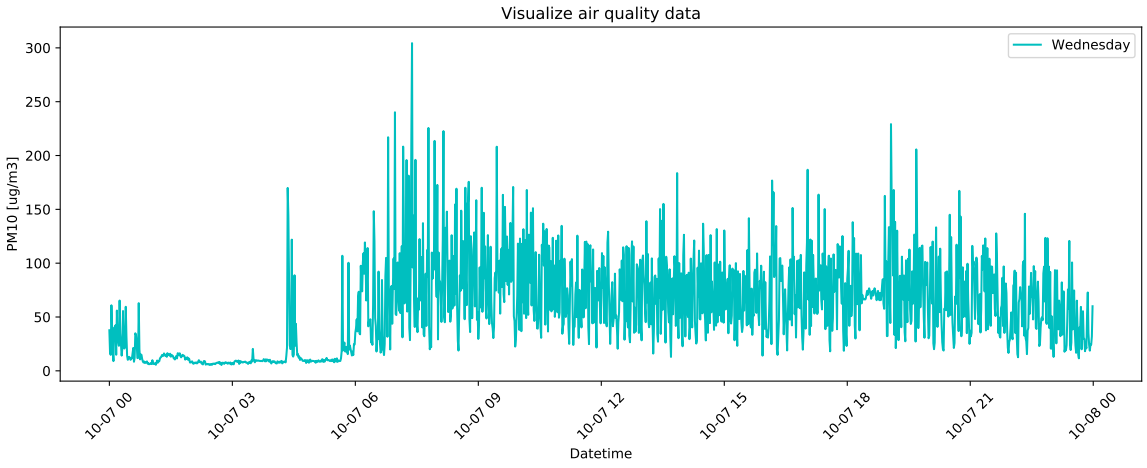


Figure 3.8: Plot day which has missing values after replace them

so we have replaced the values with more realistic ones.

Handling outliers

An outlier is an observation of a data point that lies an abnormal distance from other values in a given population. Outliers tell us how the data values differ significantly from the overall perspective.

Generally, outliers affect statistical results while doing the EDA process. A quick example is the Mean and Mode of a given dataset, as outliers can make then unrepresentative of the real data. Also correlation coefficients are highly sensitive to outliers. Since they measure the strength of a linear relationship between two variables, the relationship dependent of the data and correlation is a non-resistant measure and the correlation coefficient is strongly affected by outliers. While "Outlier Detection" is a topic in itself, in forecasting context we want to treat outliers before the data is used for something.

Before mentioning how we will handle outliers, I want to state that the best way to detect outliers is often the visual inspection. All other statistical methodologies are open to making mistakes, whereas visualizing the outliers gives a chance to take a decision with high precision. If we go back to the general plot of our air quality data (Figure [3.5]), some outliers can be spotted with the naked eye.

Statistical methodologies are less precise (as mentioned), but on the other hand, they are faster. There are many different ways of handling outliers, some of them are:

- Winsorization: Use Box and whiskers and clip the values that exceed 1 & 99th percentile (not preferred)
- Use residual standard deviation and compare against observed values (preferred but cannot do a priori)
- Use moving average to check spikes/troughs (iterative and not robust)

Another important reason to pay close attention to outliers is that we will choose the appropriate error metric based on that. There are many error metrics used to assess accuracy of forecasts, e.g. MAE, MSE, RMSE, %MAPE, %sMAPE. If outliers are present, using RMSE is not recommended because squaring the error at the outlier value can inflate the RMSE. In that case model should be selected/assessed using %MAPE or %sMAPE. In our case we will build a low pass filter. For instance, a moving average is a filter, and can be applied in a trend/noise decomposition framework:

$$T_i = \frac{1}{n} \sum_{k=0}^{n-1} x_{i-k}$$

$$N_i = x_i - T_i$$

where T_i in out case represent the moving average/median and when the noise component is "too large" it indicates an outlier. We set a threshold PM_{10} value (of

180) and if the difference between the value and the moving average in a specific point is above this threshold, we set that value as a outlier (if $threshold < N_i$ we set the value x_i as an outlier). This filter implementation is very similar to an outlier test known as [Grubbs] which is used to detect a single outlier in a univariate dataset that follows an approximately normal distribution. Here, I'm using the moving window as a sample in Grubbs test.

The drawback of this method is, that it requires to specify a model for the data first and then look at the deviation from that model. We could have also fit a polynomial to the data instead of the moving average, but without the time influence, they work quite fine actually.

Figure [3.9] shows the outliers in our dataset detected with the low pass filter. We marked the outliers with black points in the original plot. The threshold we have used provides quite affordable results, at least visually it detects the obvious outliers that are seen with the naked eye.

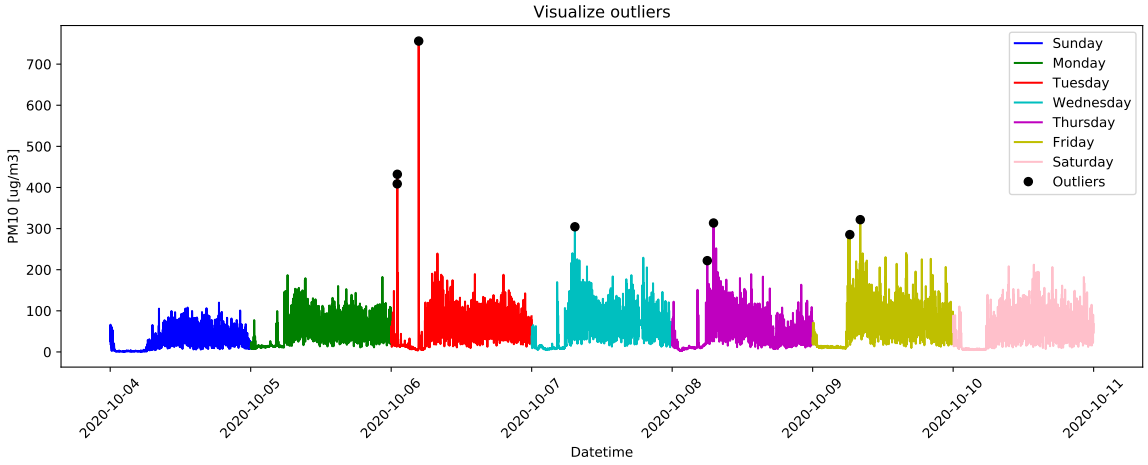


Figure 3.9: Plot outliers low pass filter

In particular, the original outlier values are as follows in the Table [3.7], column `original_PM10_[ug/m3]_values`. In a similar way to how we have proceeded with the missing values, we proceed to substitute these original outlier values with more realistic ones, again using the rolling median, because comparing to the Moving average, it is less sensitive to outliers. Again, Table [3.7], column `updated_PM10_[ug/m3]_values`.

date_time	original_PM10_[ug/m3]_values	updated_PM10_[ug/m3]_values
2020-10-06 01:02:00	408.8	24.1
2020-10-06 01:03:00	432.1	143.7
2020-10-06 04:42:00	756.1	21.3
2020-10-07 07:23:00	304.4	100.0
2020-10-08 06:00:00	222.0	28.8
2020-10-08 07:04:00	313.6	132.5
2020-10-09 06:21:00	285.4	65.2
2020-10-09 08:08:00	321.7	129.8

Table 3.7: Original outliers detected with the low pass filter and updated values

These values do make sense, since it is perfectly normal that we have such low values in the last hours of the night compared to hours where public transport begins at the beginning of each day.
Finally, we can show our air quality dataset cleaned of outliers and missing values in Figure [3.10].

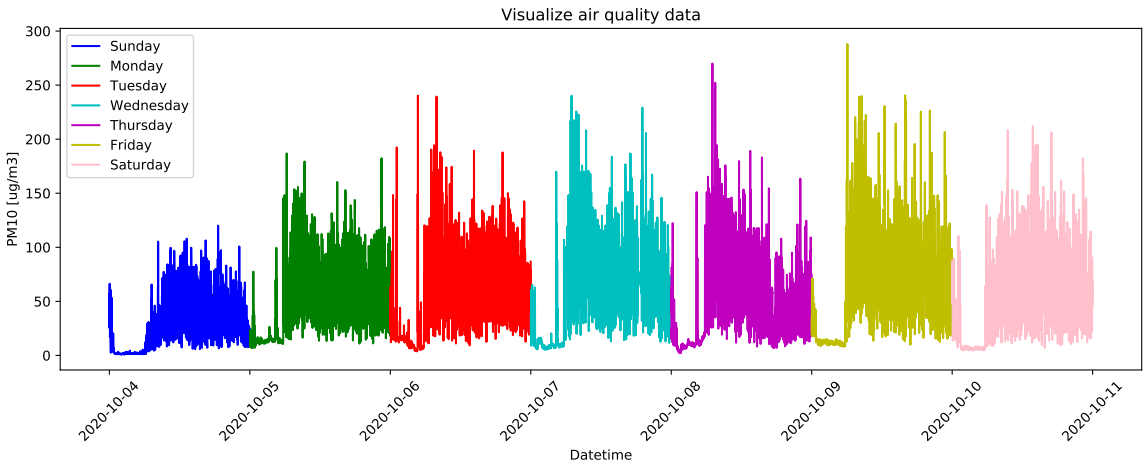


Figure 3.10: Plot air quality without outliers

Scaling

In most cases, the numerical features of the dataset do not have a certain range and they differ from each other. In real life, it is nonsense to expect age and income columns to have the same range and be directly comparable. But from the machine learning point of view, how can these two columns be compared?

Scaling solves this problem. Continuous features become identical in terms of the range, after a scaling process. This process is not mandatory for many algorithms, but it might be still nice to apply. In our case we will apply Normalization.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Normalization (or min-max normalization) consists on scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This transformation does not change the distribution of the feature and due to the decreased standard deviations, the effects of the outliers increases. Therefore, before normalization, it is recommended to handle the outliers, as we did. The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

We apply this type of normalization to two of our datasets, air quality data, Figure [3.11] and crowding levels, Figure [3.12]. The others will not be necessary because they are already between 0 and 1. This can be achieved using [`MinMaxScaler`] from `scikit-learn` Python module.

The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

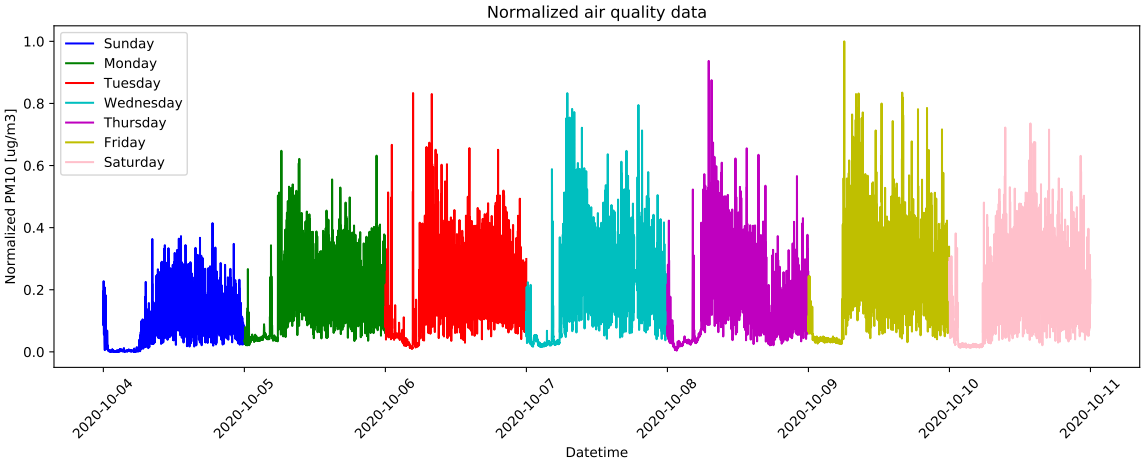


Figure 3.11: Plot air quality data normalized

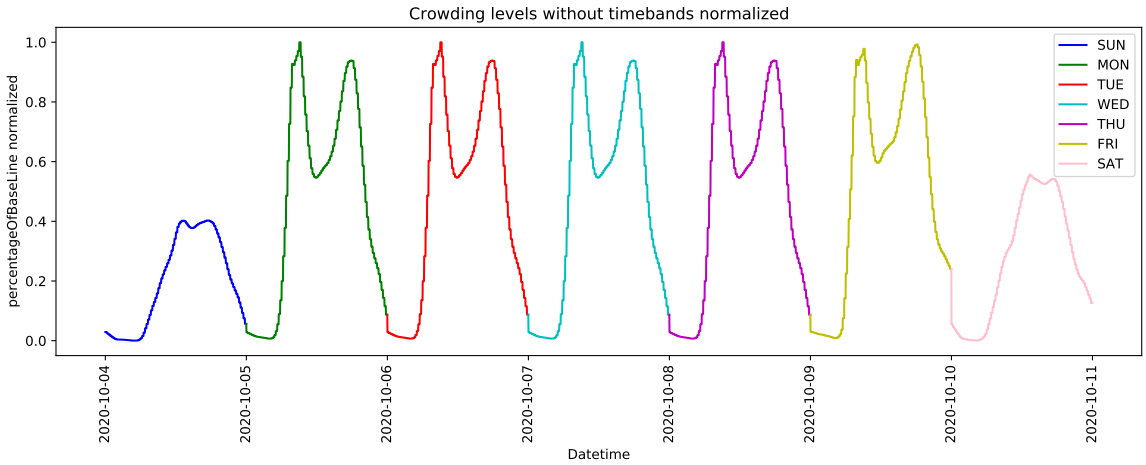


Figure 3.12: Plot crowding levels data normalized

3.4 Time series analysis

In this section, we will see statistical ways to analyze time series, in particular, how to decompose a series as a combination of level, trend, cycle, seasonality, and noise components, a process that is done to better understand the series and increase forecasting possibilities. Analyze and visualize time series is important because it is a necessary step to create forecasting of the data. On the other hand, we will show statistical tests that we will use to extract properties of our time series, in particular, tests that have to do with stationarity, normality, and autocorrelation of the residuals. The consequences and utilities of whether the series has these properties or not will be explained below.

Decomposition

Time series analysis involves understanding various aspects and definitions of the inherent nature of the series so that you are better informed to create meaningful and accurate forecasts.
Any time series can be split into base level, trend, cycle, seasonality and noise. These components are defined as follows:

- The level is average value in the series
- A trend (T_t at time t) is observed when there is an increasing or decreasing slope watched within the time series. The trend reflects the long-term progression of the series and don't have to be linear.
- The cycle (C_t at time t), reflects repeated but non-periodic fluctuations
- Seasonality (S_t at time t) are cycles that repeat regularly over time

- (R_t at time t) The noise is variation in the series, it describes random, irregular influences. Sometimes called the residuals.

It is highly important to know how to identify any seasonal patterns, trends, cycles and stationarity of the series to understand what type of model we can use to make a forecast or prediction, this is made by deconstructing the time series through the statistical task called "decomposition". To know more about this decomposition, we have consulted the following book reference [HA21]. The trend is usually combined with the cycle into a single trend-cycle component (sometimes called the trend for simplicity). It is mandatory for a time series to have level and noise, but not to have trend and/or seasonality. A time series may not have a distinct seasonality but have a trend. The opposite can also be true.
Each observation in the series can be expressed as either a sum (additive model) or a product (multiplicative model) of the components; let's go a little deeper into these two types of decompositions.
Let x_t be the data, if we assume an additive decomposition, then we can write

$$x_t = T_t + S_t + C_t + R_t$$

The additive decomposition is linear where changes over time are consistently made by the same amount, also a linear trend is a straight line. Alternatively, a multiplicative decomposition would be written as

$$x_t = T_t * S_t * C_t * R_t$$

The multiplicative decomposition is no linear, (like exponential). Changes increase or decrease over time and the trend is a curved line.
The additive decomposition is the most appropriate if the magnitude of the seasonal fluctuations, or the variation around the trend-cycle, does not vary with the level of the time series. When the variation in the seasonal pattern, or the variation around the trend-cycle, appears to be proportional to the level of the time series, then a multiplicative decomposition is more appropriate. Multiplicative decompositions are common with economic time series.
Decomposition is a very useful process in time series analysis, apart from visually allowing one to see the trend and properties such as seasonality, one can make forecasts using the different decomposed series, calculating the future values for each component separately and then adding them up to obtain a prediction. The serial problem in that case is to find the best model for each of the components.

Statistical tests

Statistical tests calculates a test statistic, a number that describes how much the relationship between variables in your test differs from the null hypothesis of no association. Statistical tests then computes a p-value to estimate the likelihood of seeing the difference specified by the test statistic if the null hypothesis of no association is true.

You can infer a statistically significant association between the predictor and outcome variables if the test statistic's value is more extreme than the null hypothesis's statistic. On the other hand, if the test statistic's value is less extreme than the null hypothesis's, you can conclude that there is no statistically significant association between the predictor and outcome variables.
Concerning time series analysis, the use of these statistical tests will allow us to deduce properties of our time series that will provide information on how to approach the prediction problem. In particular, the Ljung Box test to check if the residuals of the time series are correlated or not. Secondly, the Augmented Dicky Fuller test to check if our series is stationary (a series is stationary when the mean and variance are constant over time which makes it easier to predict, if it is not stationary transformations will be applied to make it stationary). And finally, Jarque Bera to study the normality of the distribution, although this can be appreciated visually by making a figure, the test will confirm this information with certainty.
We will use the implementation of these tests provided by the Python `[scipy]` module that provides classes and functions for conducting statistical tests.

Box-Ljung Test

The Ljung Box test (or just the Box test) [NIS12] is a statistical test to check the absence of serial autocorrelation, up to a specified lag k.
The test determines whether or not errors are independent and identically distributed (iid) or whether there is something more behind them; whether or not the autocorrelations for the errors or residuals are non zero. Essentially, it is a test of lack of fit: if the autocorrelations of the residuals are very small, we say that the model does not show 'significant lack of fit'.
We will use the Ljung Box test to check if the residuals of the time series are correlated or not.
The null hypothesis of the Box Ljung Test, H_0, is that our model does not show lack of fit (or in simple terms—the model is just fine). The alternate hypothesis, H_a, is just that the model does show a lack of fit.
The Ljung Box test calculate the statistic Q for a time series Y of length n.

$$Q(m) = n(n+2)\sum_{k=1}^{m}\frac{\hat{r}_k^2}{n-k}$$

where $\hat{r}_k$ is the estimated autocorrelation of the series at lag k, and m is the number of lags being tested. A significant p-value in this test rejects the null hypothesis that the time series is not autocorrelated. Again, the Box-Ljung test rejects the null hypothesis (indicating that the model has significant lack of fit) if

$$Q > \chi^2_{1-\alpha,\,h}$$

where $\chi^2_{1-\alpha,\,h}$ is the chi-square distribution table value with h degrees of freedom and significance level α. If the residuals are correlated, we can perform transformations to see if it stabilizes the variance. It's also an indication that we may need to use exogenous variables to fully explain the time series behaviour or use higher order

models. Ljung Box test tests the residuals as a group. Some residuals may have significant lag but as a group, we want to make sure they are uncorrelated.

The `scipy` method we will use can be found in the following url [`https://www.statsmodels.org/stable/generated/statsmodels.stats.diagnostic.acorr_ljungbox.html`]

Augmented Dicky Fuller test

A stationary time series data is one whose properties do not depend on the time, a more dense definition can be found in Section [3.6.1]. Everything related to the following definitions can be found in the reference [ERS96]. First, let's explain what an unit root tests is. Let x_t be

$$x_t = \alpha x_{t-1} + \beta y_e + \epsilon$$

where x_t is the value of the time series at time t, and y_e is an exogenous variable. We say there is an unit root if in the upper equation exists $\alpha = 1$. An unit root is a characteristic of time series that makes it not stationary.
After explain what an unit root test is, Dicky Fuller is an unit root test that tests the null hypothesis of $\alpha = 1$ in the next equation

$$x_t = c + \beta t + \alpha x_{t-1} + \phi \Delta X_{t-1} + \epsilon$$

where x_{t-1} means the lag of 1 value of the time serie, and $\phi \Delta X_{t-1}$ the first difference of the time serie at time $t - 1$.
Similar to the unit test, the coefficient of $\alpha = 1$ means the presence of a unit root. If not rejected, the series is no stationary.
Finally, Augmented Dicky Fuller test is an augmented version of the Dickey Fuller test for a larger time series that expands the Dickey Fuller test equation to include high order regression.
To summarize, because the null hypothesis implies the presence of a unit root, $\alpha = 1$, the obtained p-value must be smaller than the significance level, 0.05, in order to reject the null hypothesis. As a result, the time series is said to be stationary.
The facility with which a time series may be decomposed and anticipated using statistical techniques is determined by its stationarity. The majority of time series (like stocks and forex) are not stationary, which is one of the key reasons why forecasting is so difficult.

The `scipy` method we will use can be found in the following url [`https://www.statsmodels.org/stable/generated/statsmodels.tsa.stattools.adfuller.html`]

Jarque Bera test

The Jarque-Bera Test, a sort of Lagrange multiplier test, is a statistical test for normality in which the skewness and kurtosis of data are compared to see if they follow

a normal distribution. Because it is not a time series-specific test, it can be used to a variety of data sets. [Jar11]
A normal distribution has a skew of zero (completely symmetrical around the mean) and a kurtosis of three, which indicates how much data is in the tails and how "peaked" the distribution is. In order to run the test, you don't need to know the data's mean or standard deviation.
The formula for the Jarque-Bera test statistic is:

$$JB = \frac{n}{6}\left(S^2 + \frac{1}{4}(K-3)^2\right)$$

where n is the sample size, S is the sample skewness coefficient and K is the kurtosis coefficient. The test's null hypothesis is that the data is normally distributed, and the alternative hypothesis is that the data is not normally distributed.

In general, a large JB value indicates that the data are not normally distributed. For example, in the `scipy` method that we will use from `statsmodels`, [`https://www.statsmodels.org/stable/generated/statsmodels.stats.stattools.jarque_bera.html`] , a result of 1 means that the null hypothesis has been rejected at the 5% significance level. In other words, the data does not come from a normal distribution. A value of 0 indicates the data is normally distributed. A small p-value means that you can reject the null hypothesis that the data is normally distributed.
As a last point, normality is not a requirement for forecasting. It's data stationarity that's the issue. Forecasting can be done even if residuals are not normally distributed (as determined by the Jarque Bera Test, for example). Forecasting should not be done if the series is determined to be non-stationary (for example, using the Augmented Dickey Fuller Test), because such forecasts would be unreliable.

3.5 Comparing time series

This section will see different ways to measure the relationship between time series, from more basic methods such as studying Pearson's correlation coefficient to a more complex algorithm such as Dynamic Time Warping to calculate the similarity between two temporal sequences.

Although there are many ways to compare time series, calculating the Pearson correlation coefficient and studying the cross-correlation is an excellent way to begin. The Pearson correlation coefficient shows the linear relationship between variables. The cross-correlation is similar to the Pearson correlation coefficient. Still, it is used to measure the similarity of two series based on the lag of one concerning the other. After that, we proceed to use the **Dynamic Time Warping (DTW)** algorithm, one of the strategies for detecting similarity between two temporal sequences that may change in speed. For example, even if one person was walking faster than the other or accelerations and decelerations throughout the observation, DTW could discern similarities in walking. The best advantage of this algorithm is that it can also deal

with time series of different lengths [DR14]. We will go into more detail about this algorithm in the next section.

3.5.1 Dynamic Time Warping

Dynamic time warping compares the amplitude of the first series at one time with the amplitude of the second series with the times around the time of the previous series, for example, two seconds or so. Time series are compared calculating the Euclidean distance (or other distances) between the different values. So series with similar shape and values will have lower distance values [SC78].
Let's consider two time series, DTW is a measure of similarity between these two time series. Let be $x = (x_0, \ldots, x_{n-1})$ and $y = (y_0, \ldots, y_{m-1})$ the two series with lengths n and m where all the elements are supposed to be in he same dimensional space. Then, DTW can be formulated as the following optimization problem:

$$DTW(x, y) = \min_{\pi} \sqrt{\sum_{(i,j)\in\pi} d(x_i, y_j)^2}$$

where $\pi = [\pi_0, \ldots, \pi_K]$ is a path that satisfies

- list of index tuples $\pi_k = (i_k, j_k)$ with $0 \leq i_k < n$ and $0 \leq j_k < m$
- $\pi_0 = (0, 0)$ and $\pi_K = (n-1, m-1)$
- $\forall k > 0, \pi_k = (i_k, j_k)$ is similar to $\pi_{k-1} = (i_{k-1}, j_{k-1})$ in the following way:
  - $i_{k-1} \leq i_k \leq i_{k-1} + 1$
  - $j_{k-1} \leq j_k \leq j_{k-1} + 1$

This could be visualized as a matrix where each of the sides are the values of each of the series, and the values of the matrix is the corresponding value of the distance chosen in the respective values of the position of the matrix, we calculate the minimum path of this matrix (warping path).

A DTW distance of zero implies that the time series are very similar.

A path can be regarded as an alignment of the series such that Euclidean distance between aligned time series is the minimum. The matrix stores the distance values.

Actually, the DTW optimization problem is set for the Euclidean distance but can be used for a different metric.

There are several libraries to implement this algorithm, the basic implementation of DTW can be found in pseudocode in the Listing [3.16]. The naive implementation has time and space complexity of $O(m * n)$ where m and n are the lengths of the compared time series whose distance is to be calculated.

```
def dtw(x, y):
    # Initialization
    for i = 1..n
        for j = 1..m
            C[i, j] = inf

    C[0, 0] = 0.

   # Main loop
   for i = 1..n
        for j = 1..m
            dist = d(x_i, y_j) ** 2
            C[i, j] = dist + min(C[i-1, j], C[i, j-1], C[i-1, j-1])

   return sqrt(C[n, m])
```

Listing 3.16: Basic DTW implementation

We will use Python modules that provide us with a better implementation and efficiency, specifically, two python modules.

A first module, taken from the Github repository [`https://github.com/pierre-rouanet/dtw`], is interesting because the DTW implementation is optimized in the way it calculates distances. The speed for calculating the DTW between a large number of coefficients is slow, so using a method to calculate distances taken from the module `scipy` [`https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html`] to compute distance between each pair of the two collections of inputs will significantly improve the algorithm efficiency.

And a second module, that can be found in the Github repository [`https://github.com/wannesm/dtaidistance`] that we have chosen since it allows us to make visualizations together with the graphs of the time series that are compared.
Although we can use different distances, in our case we will use the Euclidean distance to perform the calculations. There are many modules that allow us to apply this algorithm, in our case we have chosen, prioritizing efficiency and visualization. Of course both modules compute the DTW distance measures between all sequences in a list of sequences. The compared series containing a distance closer to zero, will be more similar

3.6 Prediction models

Forecasting is the next step in the process, and it involves predicting the series' future values. In this section, we will look at different models to try to predict pollution data. We will see ARIMA models and Deep Learning.

Using the ARIMA model, one can forecast a time series using the past series values. We choose the ARIMA model because other statistical models, such as exponential smoothing or simple linear regression, are less versatile than ARIMA models. In reality, specific exponential models are ARIMA models in disguise. Forecasting is difficult

in general. In practice, very complicated models perform well on in-sample forecasts but not so well in the real world compared to simpler models. ARIMA models fall somewhere in the centre, being simple enough to avoid overfitting while also being flexible enough to capture some of the sorts of correlations found in data.

On the other hand, while linear approaches have dominated time series forecasting because they are generally understood and effective on many more straightforward forecasting issues, deep learning neural networks can automatically learn arbitrary complex input-output mappings and handle numerous inputs and outputs. In consequence, Deep learning have successfully been applied to address time series forecasting problems, which is a very important topic in air quality [LBCGR21].

3.6.1 ARIMA model

There's a particular collection of strategies and methods especially well suited for predicting the value of a variable concurring to time, and therefore, ideal for time series. The following book has been used for the definitions presented in this section [HA21]. ARIMA is an acronym that stands for **A**uto-**R**egressive **I**ntegrated **M**oving **A**verage. Is a statistical model for analyzing and forecasting time series data. It is really simplified in terms of using it, but is really powerful.

Briefly, the key words of the model:

- AR: Auto-Regressive. Uses the relationship between an observation and some number of lagged observations. The variable of interest is forecasted using a linear combination of past values of the variable. Autoregression means that it is a regression of the variable against itself. An autoregressive model of order p can be written as

  $$x_t = c + \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \cdots + \alpha_p x_{t-p} + \epsilon_t$$

  where x_t are the values of the series of time t, x_{t-p} the effect of "shifting" the data back p periods[16], ϵ the noise and α_i for $i \in \{1, \ldots, p\}$. We refer to this as an $AR(p)$ model of order p. There is a restriction of autoregressive models to only be used in stationary data.

- I: Integrated. Use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) to make the time series stationary.

- MA: Moving Average. The dependency between an observation and a residual error from the moving average is applied to lagged observations. It uses past forecast errors in a regression-like model.

  $$x_t = c + \epsilon_t + \phi_1 \epsilon_{t-1} + \phi_2 \epsilon_{t-2} + \cdots + \phi_q \epsilon_{t-q}$$

  This is called a moving average model of order q, $MA(q)$.

[16] For example, for monthly data, considering the same month last year would be y_{t-12}

Before we introduce ARIMA models, we must first discuss the concept of stationarity and the technique of differencing time series.

Stationarity

A stationary time series data is one whose properties do not depend on the time. Thus, time series with trends, or with seasonality, are not stationary.
On the other hand, for stationarity, it does not matter when the series is observed as it should look similar at any point in time. This is a core concept of the ARIMA methods, as only stationary processes can be modeled using ARIMA.
Some cases can be confusing —a time series with cyclic behaviour (but with no trend or seasonality) is stationary. This is because the cycles are not of a fixed length, so before we observe the series we cannot be sure where the peaks and troughs of the cycles will be.
In general, a stationary time series will have no predictable patterns in the long-term, it means that the time series constant mean, variance and autocorrelation are insignificant. The stationarity of a time series is important because most statistical forecasting methods are designed to work on a stationary time series (as is the case of ARIMA). It is possible to make nearly any time series stationary by applying a suitable transformation to convert a non-stationary series to stationary.
There are different ways to check whether a time series is stationary or no stationary. Some of them can be:

- Visualize the data and check if there are any obvious trends or seasonality.
- Use periods or random partitions of the data and check for noticeable or significant differences with statistics variables like the mean or the variance.
- Unit root tests are tests for stationarity, like the Augmented Dicky Fuller test, explained in the Section [3.4]. The null hypothesis is defined as the presence of a unit root, and the alternative hypothesis is either stationarity.

Differencing

One way to make a non-stationary time series stationary is to compute the differences between consecutive observations. This is known as differencing.
Differencing can help stabilise the mean of a time series by removing changes in the level of a time series, and therefore reduce or eliminate trend and seasonality.
The Auto Correlation Function (ACF)[17] is also useful for identifying non-stationary time series. For a stationary time series, the autocorrelation will go down to zero quickly, while the autocorrelation of non-stationary data decreases slowly.

Finally, going back to our model, combining differencing with autoregression and a moving average model results in the ARIMA model. This model can be written as

$$x_t^{'} = c + \alpha_1 x_{t-1}^{'} + \cdots + \alpha_p x_{t-p}^{'} + \phi_1 \epsilon_{t-1} + \cdots + \phi_q \epsilon_{t-q} + \epsilon_t$$

[17]is the correlation of the series with its previous values, more on this coming up.

where $x_t^{'}$ is the differenced series (maybe more than one time). We call this an **ARIMA(p,d,q)** model ,where

- p is the order of the autoregressive part
- d the degree of the differencing involved
- q order of the moving average part

Particular cases of ARIMA models:

- ARIMA(0,0,0) = White noise
- ARIMA(0,1,0) with no constant = Random walk
- ARIMA(0,1,0) with a constant = Random walk with drift
- ARIMA(p,0,0) = Autoregression
- ARIMA(0,0,q) = Moving average

Selecting appropriate values for p, d and q is difficult. In simple cases, the **Auto Correlation Function (ACF)** and the **Partial Auto Correlation Function (PACF)** are used in most circumstances.

The appropriate number of **MA** terms is determined using the **ACF** (the correlation between present data and observations from all previous points in time, as stated previously), where the number of terms determines the order of the model.

On the other hand, the **Partial Auto Correlation Function (PACF)**, a subset of the Auto Correlation Function (ACF), expresses the correlation between observations taken at two points in time while allowing for any effect from additional data points. PACF can be used to figure out how many terms to employ in the **AR** model. Again the model's order is determined by the number of terms. Let us explain how to obtain the order for the ACF. The same reasoning applies to the PACF.

When we display both graphs, we'll have vertical lines matching each lag. The height of each spike represents the lag's autocorrelation function value, which ranges from -1 to 1. Because this is the autocorrelation between each term and itself, the autocorrelation with lag zero always equals 1.

Horizontal lines will also be seen on the graph (significance threshold). Statistical significance is assigned to each line that rises above or falls below the dashed lines. This indicates that the spike has a value that differs significantly from zero. Again, Autocorrelation is evident when a point is significantly distinct from zero and Autocorrelation is refuted by a spike that is near zero.

The number of values exceeding these horizontal lines (so having statistical significance) will correspond to the order of the model, which will be **MA** or **AR**, depending

on whether **ACF** or **PACF** is used.

In our case, it is somewhat more complex, as we have a week of tick data, data per minute, each lag will correspond to 1 minute, then we would have to study autocorrelation values between the 1440 minutes that each of the seven days of the week has.

Therefore, although it is good to know the above, in our case, we will find which are the best values that fit our model by brute force, using the **Akaike Information Criteria (AIC)**.

Akaike Information Criteria (AIC)

The Akaike Information Criteria (AIC) [BA03] is a frequently used statistical model assessment. It is calculated as:

$$AIC = 2K - 2ln(L)$$

where:

- **K** is the number of model parameters.
- And $ln(L)$ the log-likelihood of the model. The higher the number, the better the fit.

It essentially combines the goodness of fit and the model's simplicity/parsimony into a single metric. Without going into too much detail, when comparing two ARIMA models, the one with the lower AIC is generally "better". Of course, there are no absolutes, and all measures have disadvantages, because ARIMA models are more general than linear models, AIC appears to be a reasonable measure in this case. When a plot has trends, the ACF progressively falls as the lags expand. The lower the value for AIC, the better the fit of the model. The absolute value of the AIC value is not essential. It can be positive or negative.
Also, as the last comment, again, when ACF or PACF is displayed, if lags are frequently outside of the pair of horizontal lines, the trends are considered non-stationary.

3.6.2 Artifical Intelligence, Machine Learning and Deep learning

This section will look at different deep learning models that we will apply to predict pollution. That review will be mostly based on [GBC16]. But first of all, I will make a small introduction about some terms related to this topic.
Thinking about a few years ago, I am sure many of you will remember when there was a significant shift to the first mobile world, everyone had desktop websites. These desktop websites never work on mobile devices, and then there was a vast industry that did work on mobile phones. Otherwise, you would lose business. In my opinion, the same thing is going to happen with AI, and I am pretty sure we are going to move to an AI approach, so every industry will in some way be affected by

AI in the not-too-distant future.

But what is Artificial Intelligence, Machine Learning, and Deep learning?

Artificial Intelligence (AI)

Artificial Intelligence (AI) is the science of making things bright. It is defined as "Human intelligence exhibited by machines."

Essentially this is the science of making things bright or, more formally, human intelligence exhibited by machines. Now, this term is an extensive term, and, right now, we are in a position where we develop systems that are in the realm of narrow AI.

Now narrow AI means that we are creating systems that are as good or better than a human expert could be in one or more areas, for example, object recognition like the medical industry where doctors are trying to scan for brain tumours, and in the past, you would get back this very grainy image of a brain scan, and sometimes they would miss the signs of a brain tumour in that scan.

So now, using AI, they can be more accurate in pointing out regions in the image that may be more likely to have the tumour in so the doctor can see it and recognize it without looking at it.

Machine learning (ML)

Machine learning (ML) is an approach to achieve AI. ML involves teaching a computer to recognize patterns by example rather than programming it with specific rules. ML can find these patterns within data. Essentially, ML is about creating algorithms that learn complex functions from data and make predictions on it.
The critical thing that's very important here is that ML can reuse these systems, so if I train a system that can recognize cats, we can use the same system to identify dogs if we train it with dogs without changing any of the actual code, which makes it a potent tool.

So, for example, to make a spam filter, we would have lots of conditional statements to check if the email contains possible spam words, but this is not very scalable because this is a constant battle between programmer and spammer unmaintainable (Listing [3.17]).

We can use machine learning to do this, so instead of explicitly define the rules, we mark an email as spam, and the machine learning model will take all of tens of thousands of millions of examples of spam and will figure out mathematically what words contribute to spam itself and then mark it (Listing [3.18]). So that saves us a lot of time in our efforts and allows us to do more important things.

```
1 if email contains "You won 129837198417 bitcoins!"
2     then spam is true;
```

```
if email contains ...
```

Listing 3.17: Traditional programming

```
Train model with spam emails;
Try to classify some emails;
Change self to reduce errors;
repeat;
```

Listing 3.18: Machine learning programs

Deep Learning (DL)

Finally, **Deep learning (DL)** is a technique for implementing Machine Learning. Deep learning is part of a more prominent family of **artificial neural network**-based machine learning methods.

The most crucial difference between deep learning and traditional machine learning is the performance as the scale of data increases. When the data is small, deep learning algorithms don't perform that well. This difference is because deep learning algorithms need a large amount of data to understand it perfectly because deep learning learns patterns of patterns.

Before explaining the algorithms, we want to provide an easy-to-understand and moderate introduction to deep learning that doesn't rely significantly on math or theoretical concepts, let's look at a few primary concepts.

TODO: Quiero explicar los modelos sin entrar en mucho detalle.

[https://developer.nvidia.com/blog/deep-learning-nutshell-core-concepts/]

Artifical Neural Network. An **artificial neural network** takes some input data. It changes it by computing a weighted sum over the inputs and then applying a non-linear (or linear) function to this transformation to arrive at an intermediate state. The transformative function is often referred to as a **unit**, while the three processes above compose a **layer**. Examples of activation functions in the Figure [3.13].

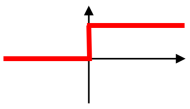
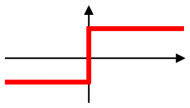
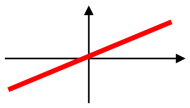
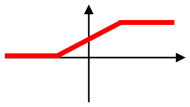
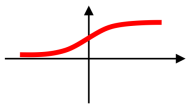
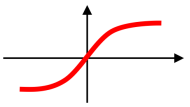
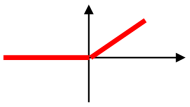
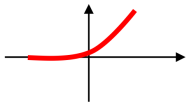
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Figure 3.13: Activation Functions for Artificial Neural Networks [by Sebastian Raschka]

A **layer** is a container that accepts weighted input, changes it using a collection of primarily non-linear functions, and then delivers the transformed values to the next layer as output. A layer is usually uniform, meaning it only has one sort of activation function so that the layer can easily compare it to other network components. The initial and last layers refer to input and output layers, respectively, while the layers in between are called hidden layers.

The artificial neural network learns numerous layers of non-linear information through repetition of these stages, which it then integrates into a final layer to create a forecast.

The neural network learns by generating an error signal that evaluates the difference between the network's predictions and the desired values and then using that

signal to adjust the weights (or parameters) to improve prediction accuracy.

Convolution. Convolution is a mathematical process that defines how to combine two functions or pieces of information according to a set of rules. The convolution kernel and the input data combine to generate altered input data. Convolution frequently regards as a filter, with the kernel filtering the input for specific types of information.

In physics and mathematics, the convolution theorem[18] establishes a link between the spatial and time domains and the frequency domain. Fourier transformations are used to illustrate this bridge. (when using a Fourier transform on the input, then the convolution operation is simplified significantly, integration becomes mere multiplication, for more information an Nvidia implementation[19].

Let us now look at the different models that we will apply to our problem of predicting pollution.

Linear model. Inserting a linear transformation between the input and output is the simplest trainable model we use (linear activation function from Figure [3.13]). In this situation, a time step's result is solely dependent on that step. The layer exclusively uses a linear transformation to transform the data last axis on batches of expansive windows.

When used in this manner, the model generates a series of independent predictions at successive time steps. At each time step, there are no interactions between the projections. Linear models have the advantage of being generally straightforward to interpret. You can see the weights set to each input by pulling out the layer's weights.

Dense model. Overall, dense models are similar to linear models, but the result passes through a non-linear activation function. We can see now that dense models can be reduced back to linear models if we use a linear activation function. Stacking linear layers (or, here, dense layers but with linear activation) will be redundant. Dense layers have a unique nonlinearity property that allows them to mimic any mathematical function. They are, however, constrained in the sense that we always get the same output vector for the same input vector. They are unable to identify time repetition or produce various responses in response to the same stimulus. In our case, Dense models, can only be executed on input windows of exactly this shape. In particular, we comment on the most commonly used one, the piecewise function ReLU (Rectified Linaer Unit, Figure [3.13]). So that, if the input is positive, it outputs the information directly; else, it outputs zero. Because a model that utilizes it

[18] See `https://en.wikipedia.org/wiki/Convolution_theorem`

[19] See developer.nvidia.com/blog/accelerate-machine-learning-cudnn-deep-neural-network-library

is quicker to train and generally produces higher performance, it has become the default activation function for many types of neural networks. Convolutional models fix this problem.

Convolution neural network (CNN). Convolutional layers (see Convolution definition [3.6.2]) filter inputs for relevant information in a convolutional neural network. These convolutional layers contain learnt parameters, allowing these filters to automatically modify to extract the most helpful information for the job at hand.

The convolutional layer applies to a sliding window of inputs, and the convolutional model can be run on inputs of any length.

Recurrent neural network (RNN). Recurrent neural networks (RNNs) are a type of neural network that excels at modelling sequence data like time series and natural language.

An RNN layer iterates over the timesteps of a sequence using a loop while keeping an internal state that encodes information about the timesteps it has observed so far. Even though this would not function precisely in practice, we could acquire close to a good result. The important thing here is not that we got the correct answer, but that we can train recurrent neural networks to learn very respective outputs for any input sequence, which is helpful.

We will use an RNN layer called Long Short Term Memory (LSTM).

Without going into too much detail, a linear unit with a self-connection and a constant weight of 1.0 is used in Long Short Term Memory (LSTM) units. Using this enables for the indefinite preservation of a value (ahead pass) or gradient (reverse pass) that flows into this self-recurrent unit (inputs or errors multiplied by 1.0 still have the same value. Thus, the previous time step's output or mistake is the same as the next step's output) so that the deal or gradient may be recovered precisely at the time step when needed most. Thus, the memory cell, a self-recurrent unit, provides a kind of memory that can store information from dozens of time steps ago.

LSTM is extremely useful for various tasks; for example, an LSTM unit can retain information from the previous paragraph and apply it to a sentence in the current section.

To dive deeper into LSTM and make sense of the whole architecture [Gre+17] is a fantastic reference.

Keras and Data Windowing

We will use implementations of these deep learning algorithms using the Python language and the powerful `[Keras]` library, where we will use as a reference the book

written by the creator of `Keras` and Google AI researcher François Chollet [Cho17], the book builds your understanding through intuitive explanations and practical examples.

Based on a window of successive samples from the data, the models will make a set of predictions. We will create a range of models (including Linear, DNN, CNN, and RNN models) and use them to forecast for single and multiple timesteps. [`https://towardsdatascience.com/ml-approaches-for-time-series-4d44722e48fe`]

The format of our windows will be input data, offset and label. So, for example, to do single-step forecasting 60 minutes into the future, given 60 minutes of history, we will define a window whose input will be 60 minutes, with an offset of 60 minutes, to predict the next label (we will try to predict the value at minute 121). Of course, the offset is optional. We see an example of what a window of our air quality dataset (PM_{10}) with the input of one day (1440 minutes) would look like to predict the next value (minute 1441).

In general, our visualisation format will be similar to that of Figure [3.14], the first graph with a global visualisation and a second graph enlarged (In the vertical axis, you can see minutes in which it zooms).

Although we will always show our pollution data for all our predictions and models, we will take into account in addition to pollution data, train schedules and crowding levels. When possible, we will show the weight of each of the datasets in the prediction obtained.

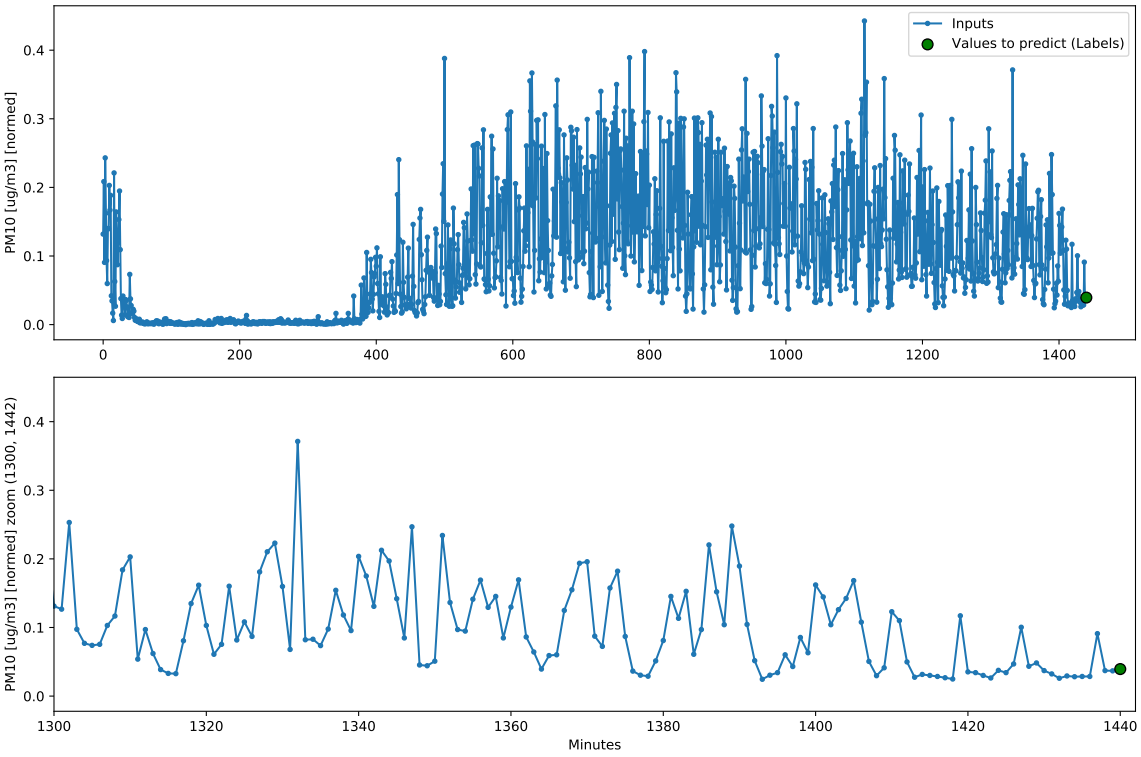


Figure 3.14: Windows of 1440 minutes and one final prediction

Chapter 4

Experimentation

In this chapter, we will apply the different methods seen in the previous chapter. We will see what can be achieved with the experiments and how they have been used to fulfill the proposed objectives.

Initially, we will do statistical visualizations and summaries to better understand our pollution dataset content in a section called Exploratory Data Analysis. In this section, we will also see how to decompose our time series, calculate and show the autocorrelation function and partial autocorrelation function of the time series along with stationarity and normality using statistical tests. These are properties of the time series that will allow us to perform further forecasting of the pollution data.

Next, we will use the London public transport API to construct time series, namely inbound and outbound train schedules along with station crowding levels. We will perform an experiment on correlating pollution peaks in train schedules, as well as crowding levels. We will also apply the Dynamic Time Warping algorithm to compare time series.

Finally, we create our models for pollution prediction, both the ARIMA model and different deep learning algorithms.

4.1 Exploratory data analysis

The following critical step in any data science workflow is to utilize **Exploratory Data Analysis (EDA)** to better know our data. For time arrangement, this is often particularly imperative, as we need to be able to recognize any seasonal patterns, trends, and stationarity of our series to assist us to get what sort of model we should be using to forecast into the future.

We have already done EDA over our data in the previous chapter (Section of Exploratory Data Analysis [3.3]). Data cleaning and preprocessing is usually a part of EDA.

The pollution dataset consists of pollution data collected on the Piccadilly line of the London underground, specifically at the South Kensington underground station. The pollution data consists of the value of the PM_{10} particle[1]. This information is collected in the interval from October 4, 2020 to October 11, 2020, in a sample of values per minute. At first glance at a general visualization of this week, it makes sense to comment on the obvious, in rush hours on the metro, such as in the morning, pollution peaks are seen during working hours, specifically in hours of entry and exit from work, while on weekends these values gradually decrease. So let's see more visualizations.

Table [4.1] shows some basic statistical details of our entire contamination data set like percentile, mean, std and so on.

	PM10 [ug/m3]
count	10080.000000
mean	50.736860
std	39.601088
min	0.900000
25%	16.400000
50%	42.500000
75%	75.900000
max	288.100000

Table 4.1: Air quality general descriptive statistics

We also distinguish this general basic statistical analysis by one each day on Table [4.9]. As we have commented previously, we see that the highest values are found during the week, while when Saturday and Sunday arrive, they decrease. We also see that after having replaced the missing values of our dataset, each day has 1440 minutes (24 hours), as it should be.
Let's visualize our dataset, with scaling pollution air quality data[2] (Figure [4.1]) and the original dataset without scaling (Figure [4.2]).
Is clear that there are no null values, outliers or duplicate values in the series (we have dealt with these problems before). The air quality time series is continuous tick data (value per minute). There are prediction models that can work with no continuous data. In our case, we require the data to be continuous since we will use after an ARIMA model. As a commentary, to deal with a series that does not have continuous we can add dummy data or use interpolation until it has continuous data. Since all contamination values are positive, which is normal since a negative contamination value is meaningless, we can show this on both sides of the pollution axis to emphasize if there is any growth, Figure [4.4].
We also see the sum of values of the PM_{10} particle each day (Figure [4.5]). That is,

[1]We recall that the PM_{10} particle corresponds to small solid or liquid particles of dust, ash, soot, metallic particles, cement or pollen, dispersed in the atmosphere

[2]We have already normalized our contamination data between 0 and 1 in Section 3.3

Table 4.2: 2020-10-04 (Sunday)

	PM10 [ug/m3]
count	1440.000000
mean	27.954236
std	24.547730
min	0.900000
25%	5.900000
50%	22.200000
75%	44.200000
max	120.000000

Table 4.3: 2020-10-05 (Monday)

	PM10 [ug/m3]
count	1440.000000
mean	51.965694
std	34.425089
min	6.900000
25%	20.300000
50%	45.550000
75%	76.800000
max	186.900000

Table 4.4: 2020-10-06 (Tuesday)

	PM10 [ug/m3]
count	1440.000000
mean	55.700972
std	38.715517
min	3.800000
25%	23.300000
50%	48.400000
75%	80.125000
max	240.300000

Table 4.5: 2020-10-07 (Wednesday)

	PM10 [ug/m3]
count	1440.000000
mean	58.592326
std	42.175218
min	5.400000
25%	21.275000
50%	52.650000
75%	87.350000
max	240.200000

Table 4.6: 2020-10-08 (Thursday)

	PM10 [ug/m3]
count	1440.000000
mean	49.090000
std	39.326911
min	2.200000
25%	16.200000
50%	39.200000
75%	73.650000
max	270.000000

Table 4.7: 2020-10-09 (Friday)

	PM10 [ug/m3]
count	1440.000000
mean	62.022153
std	46.106185
min	8.100000
25%	21.025000
50%	54.950000
75%	89.775000
max	288.100000

Table 4.8: 2020-10-10 (Saturday)

	PM10 [ug/m3]
count	1440.000000
mean	49.832639
std	38.836177
min	4.400000
25%	16.800000
50%	41.450000
75%	75.500000
max	212.200000

Table 4.9: Air quality descriptive statistics of each day

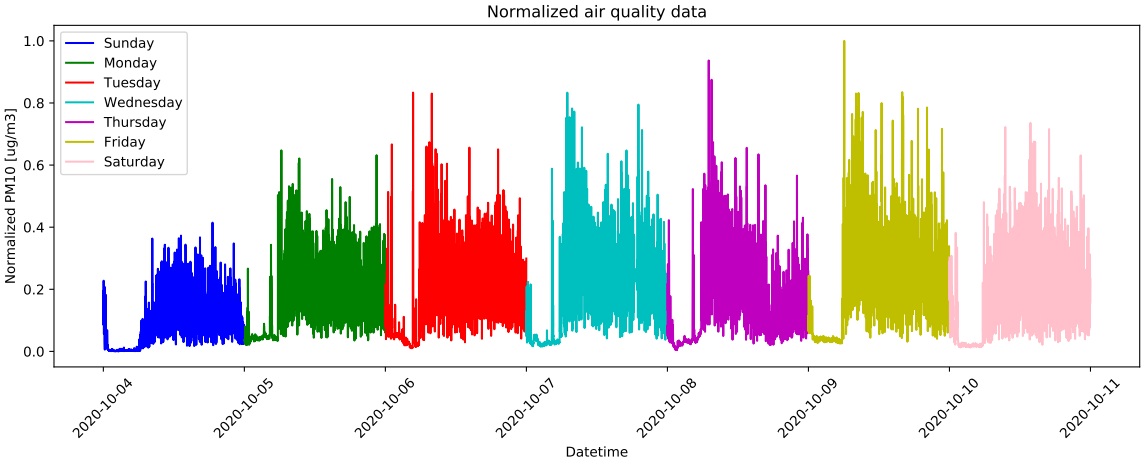


Figure 4.1: Plot pollution air quality data normalized

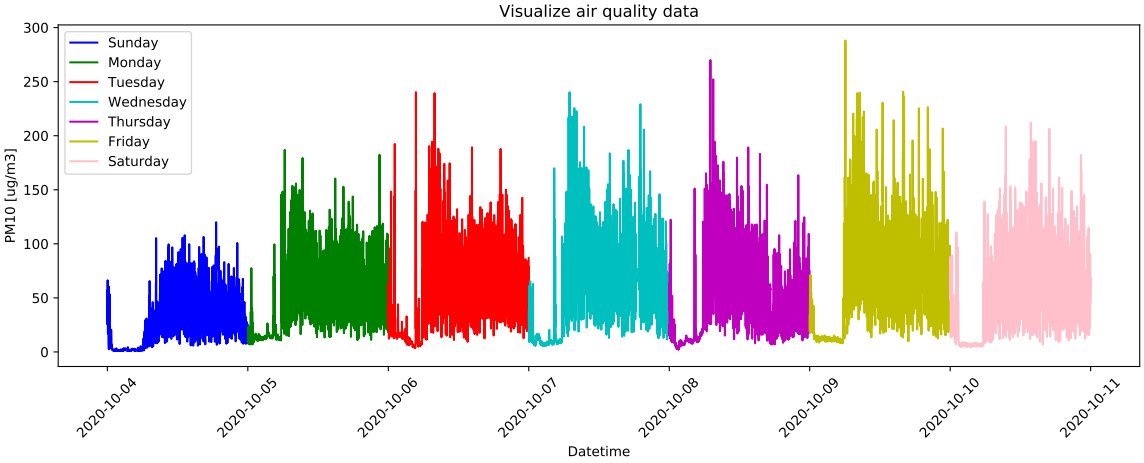


Figure 4.2: Plot original pollution air quality dataset

we can visualize the days that contain the highest amount of pollution (Friday and Saturday), while Sunday would be the day that has the least pollution, up to half the amount of pollution.
A boxplot is a standardized way of displaying the dataset based on a five-number summary: the minimum, the maximum, the sample median, and the first and third quartiles, lets plot it with the air quality of each day in the Figure [4.6].
Mix it up next with a stripplot, it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.
A heat map (or heatmap) is a data visualization technique that shows magnitude of a phenomenon as color in two dimensions. The variation in color may be by hue or intensity, giving obvious visual cues to the reader about how the phenomenon is clustered or varies over space.

In our case, the colors that we will use will be blue to indicate a low value and

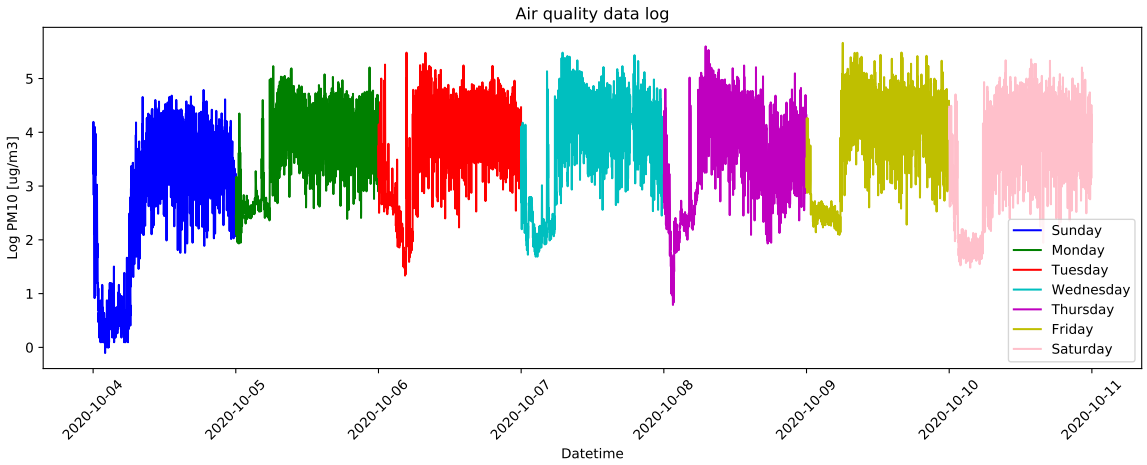


Figure 4.3: Plot air quality data logarithmic

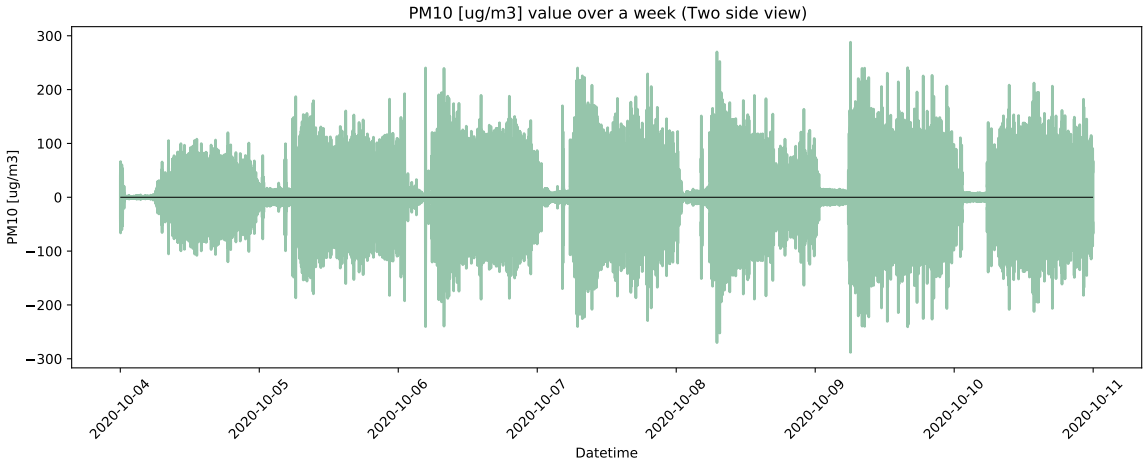


Figure 4.4: Plot air quality data two view

red to indicate a higher value of PM_{10}. We show two types of heatmaps, one to compare days and hours in Figure [4.8], and another to compare days and minutes (Figure [4.9](. In both heatmaps, we can see that in the early hours of the morning, there is hardly any pollution since no trains are passing through.
In this case two heatmaps feels redundant but when the series is long, heatmap can reveal more patterns.

Normality

If the data is skewed, normalizing the data before building a model can be appropriate. The distribution of the air quality data is shown in Figure [4.10]; we compare it with the distribution of each of the days, plotting a univariate distribution of observations. Normally distributed data is not a requirement for forecasting and does not necessarily improve point forecast accuracy, but it can help to stabilise the variance and thus the prediction [HA21].

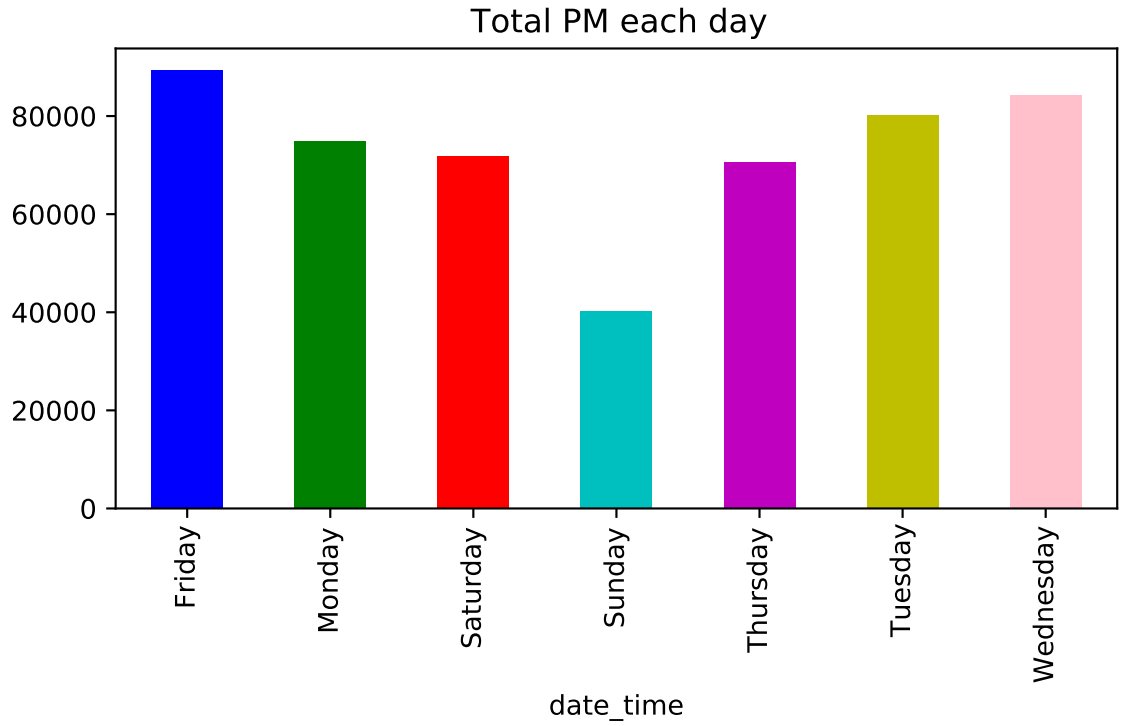


Figure 4.5: Count of PM_{10} each day

Plotting a QQ plot (see Figure [4.11]) also give us the same information, when looking at sorted values on the y-axis and (approximate) expected quantiles on the x-axis, we can identify from how the values in some section of the plot differ locally from an overall linear trend by seeing whether the values are more or less concentrated than the theoretical distribution would suppose in that section of a plot.
The values are more spread out than supposed at these quantiles. To have a complete study of normality, let's apply Jarque Bera Test. Listing [4.1] confirm the data does not follow the Normal distribution.

```
>is_normal = jb(air_quality)[1]
>print(f"p value:{is_normal}", ", Series is Normal" if is_normal
   >0.05 else ", Series is Non-Normal")
p value:0.0 , Series is Non-Normal
```

Listing 4.1: Jarque Bera Test

and our series does not follow the normal distribution, as we were already visualizing.

Auto Correlation Function (ACF)

Until now, we have been using the contamination series without normalizing. From now on, to prevent the sensitivity of the measurements, we will use the normalized series that we previously scaled in Section [3.3].

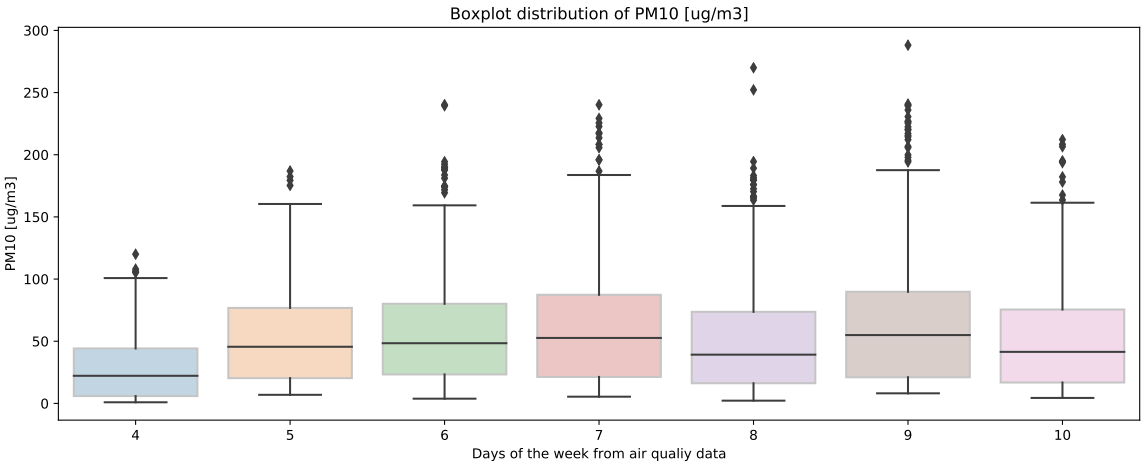


Figure 4.6: Boxplot air quality data

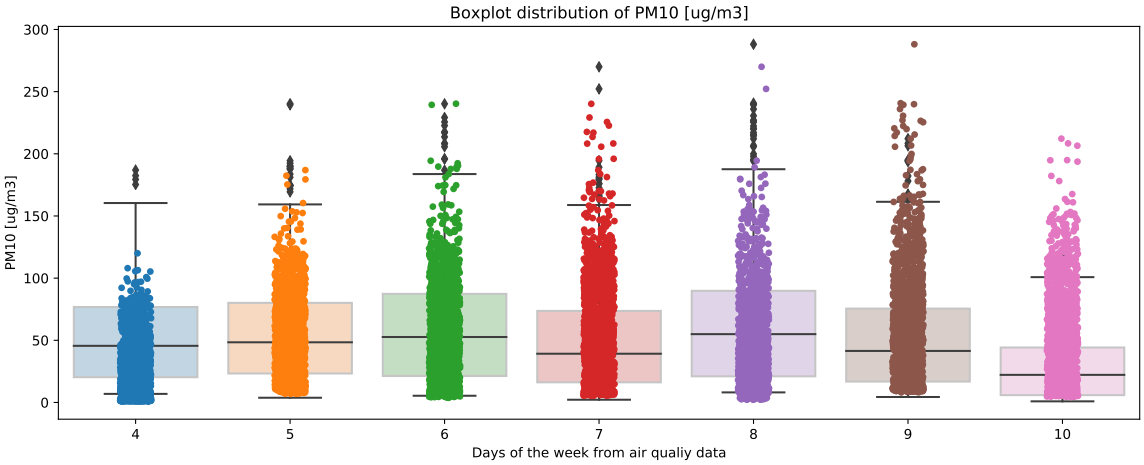


Figure 4.7: Stripplot air quality data

Let's check the Auto Correlation Function now. As we said before in other sections, the Auto Correlation Function (ACF), as explained in section [3.4], is nothing but the correlation of the series with its previous values (more on this coming up). It is also useful for identifying non-stationary time series. For a stationary time series, the Auto Correlation will go down to zero quickly, while the Auto Correlation of non-stationary data decreases slowly. However, apart of showing the Auto Correlation we will use a statistical test to check stationarity.

In Figure [4.12], we see the Auto Correlation Function of the pollution data.
As we can see, the autocorrelation plot goes to zero, so we can think that our serie is stationary. We also see the horizontal lines that imply significance threshold, every value that exceeds that horizontal line has Statistical significance, which influences the order of the ARIMA model, as we explained in the ARIMA section. This is the reason why we will use brute force for the order of our model, and it is very complex to see which is the right order when we have so many values, more than 10000 lags.

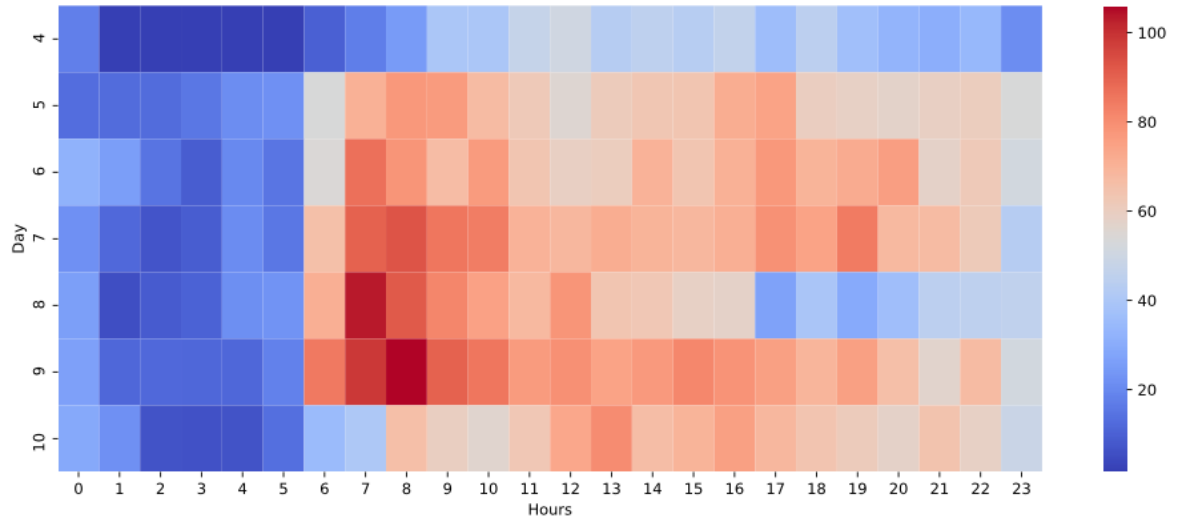


Figure 4.8: Heatmap air quality data by hour

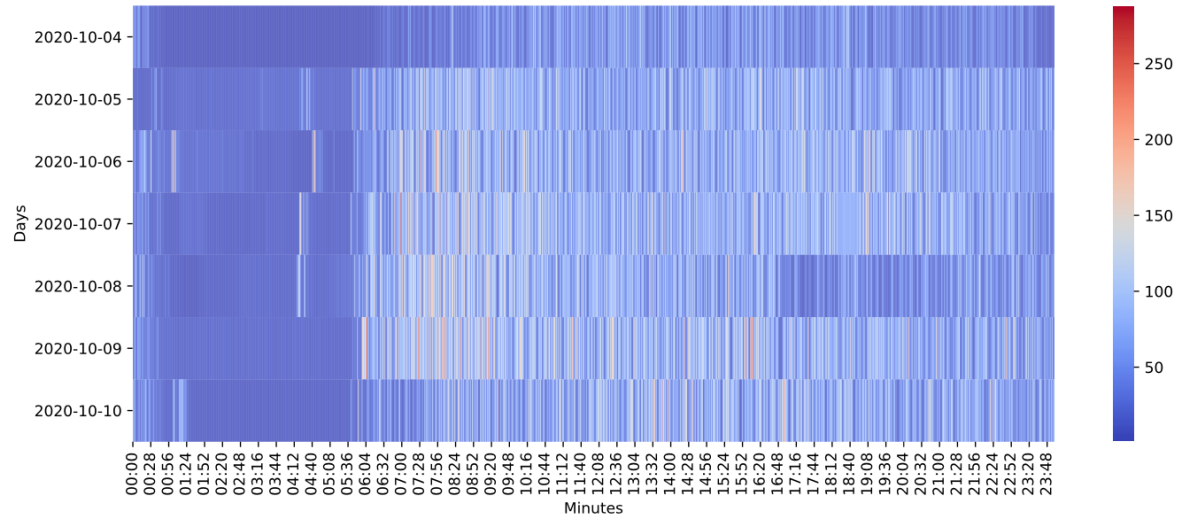


Figure 4.9: Heatmap air quality data by minute

Parcial Auto Correlation Function (PACF)

Let's also see the Parcial Auto Correlation Function (PACF). As we saw in a previous section [3.6.1]. PACF is useful because it helps to indicate the order of the AR model. We see the Auto Correlation Function and Parcial Auto Correlation Function (PACF) with 60 lags (1 hour-60 minutes), Figure [4.13] and 1440 lags (24 hours-1440 minutes), Figure [4.14].
Now let's look at the stationarity of the series.

Stationarity

As we explain before in the methods Section [3.6.1], a stationary series is one whose values are not a function of time. We are going to test two ways to determine whether a given time series is stationary.

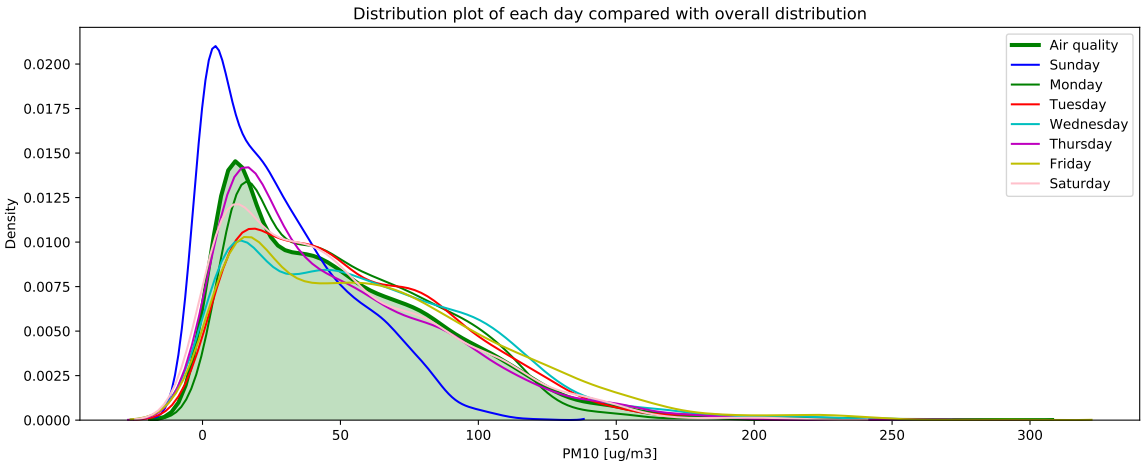


Figure 4.10: Plot air quality general distribution and days

- Seeing the Auto Correlation Function of the series, which we have seen that as it progressively decreases, it indicates stationarity.
- Augmented Dickey Fuller Test from Section (ADF) [3.4]: The time series is considered stationary if the p-value is low (according to the null hypothesis) and the critical values at 1%, 5%, 10% confidence intervals are as close as possible to the Augmented Dickey Fuller Test Statistics.
- By viewing the rolling statistics, the rolling mean and rolling standard deviation. If the rolling mean and rolling standard deviation remain constant with time, then the time series is stationary.

The difference between average and rolling average, a 5-day rolling average would average out the pollution data for the first five days as the first data point. The next data point would drop the earliest price, add the price on day six and take the average.
A stationary series does not have any persistent autocorrelation, so the predictors (lags of the series) in the forecasting models will be nearly independent. The rolling statistics are plotted with the original pollution data in Figure [4.15]. As we can see, the rolling mean and rolling standard deviation does not increase with time. Therefore, we can conclude that the time series is stationary.

Checking this out with the Augmented Dickey Fuller Test give us the stationarity result in Listing [4.2]. Because the p-value is below the threshold of 0.05 and the ADF Statistic is close to the critical values, the time series is stationary.

```
ADF Statistic: -4.045761095984308
p-value: 0.001188965577467865
Critical Values:
  1%: -3.431001491304817
  5%: -2.861827920528921
  10%: -2.5669232549582546
```

Listing 4.2: Augmented Dickey Fuller Test

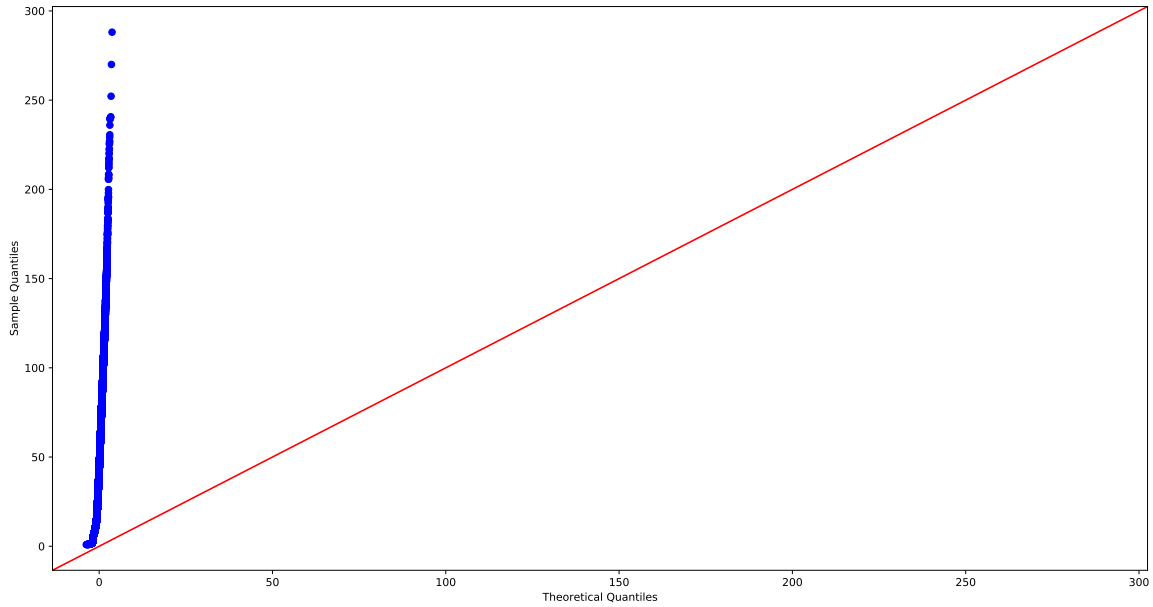


Figure 4.11: QQ plot of air quality data

There are multiple transformations that we can apply to a time series to render it stationary. However, as we have proved, in our case it will not be necessary since the series is stationary.

Decomposition

We will decompose the air quality time series into trend, seasonal and residuals as explained in Section [3.4]. Since we have to decompose in a specific period, we will choose 60 minutes (Figure [4.16]) and 1440 minutes (Figure [4.17]). With the second period we can see an almost linear upward trend, and also that the seasonal pattern is consistent.
The residuals are what is left after fitting the trend and seasonal components to the observed data. The residuals are an element that we cannot describe. The residuals should be i.i.d (i.e uncorrelated). If the residuals show a pattern, it suggests that some structural information is still missing. Our residuals have a wavy pattern, which is not desirable.

Let's run the Ljung Box test for the different periods to see if they are all independent and identically distributed as a group (i.i.d) because we do not want to detect any patterns in the residuals, such as waves or a downward slope. If the Ljung Box test shows a p-value above 0.05, the residuals as a group are white noise, so uncorrelated. We applied Ljung Box for both periods, 60 minutes (Listing [4.3]) and 1440 minutes (Listing [4.4]).

```
1 Ljung Box period 60, p value: 2.983385412207894e-47 , Residuals are
    correlated
```

Listing 4.3: Box-Ljung Test period 60

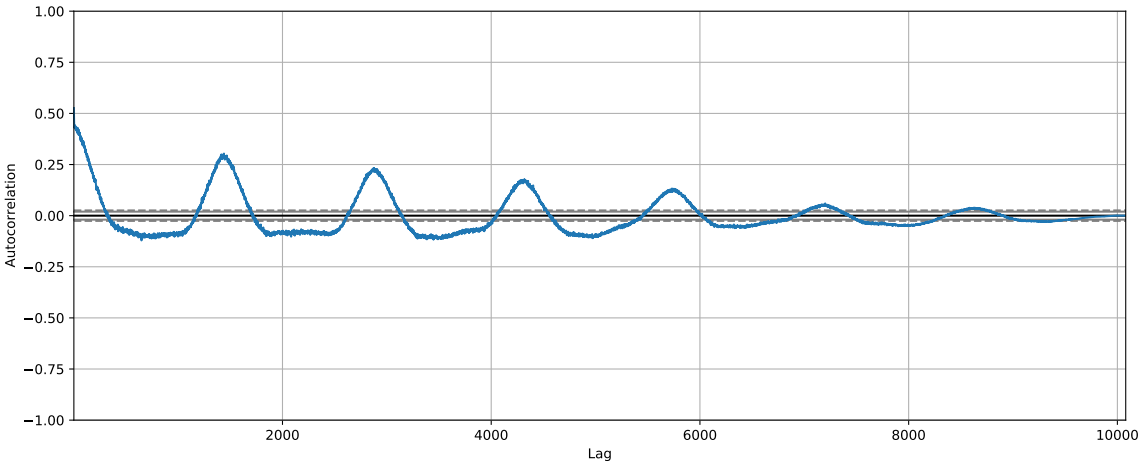


Figure 4.12: Auto Correlation Function pollution data

```
Ljung Box period 1400, p value: 5.954235481736826e-206 , Residuals
    are correlated
```

Listing 4.4: Box-Ljung Test period 1440

Residuals are correlated. If the residuals are correlated, transformations can be used to see if the variance is stabilized. It is also a sign that we might need to utilize an exogenous variable or higher-order models to completely explain the time-series behavior.

4.2 Building time series with the TfL API

The main objective is to use the TfL API to create time series to compare with our original pollution data and draw conclusions from the comparison. We will create three time series, two series correspond to the hours in which the trains pass through the station where the original air quality data were taken, South Kensington, in both directions, inbound and outbound, and a third series that will be related to crowding levels in the station, it will correspond to a relative percentage based on the Wifi connections mentioned before, not based on real time, but on connection history.

Once we have the hours of the week where the train passes through the station, the series that we will create will have the same times as the pollution data, only that instead of having the pollution value marked with the PM_{10} particle, we will mark with a one if the train passes through the station at that time or with a 0 if not, in both directions. In the case of crowding levels, we saw in a previous Section [3.2] how to use the `Crowding` endpoint; Table [4.11] shows the crowding data with timebands, we will "extend" the 15-minute timebands to match the original series (in minutes) by repeating a value 15 times.

We will build these series by manipulating the responses received with the endpoints

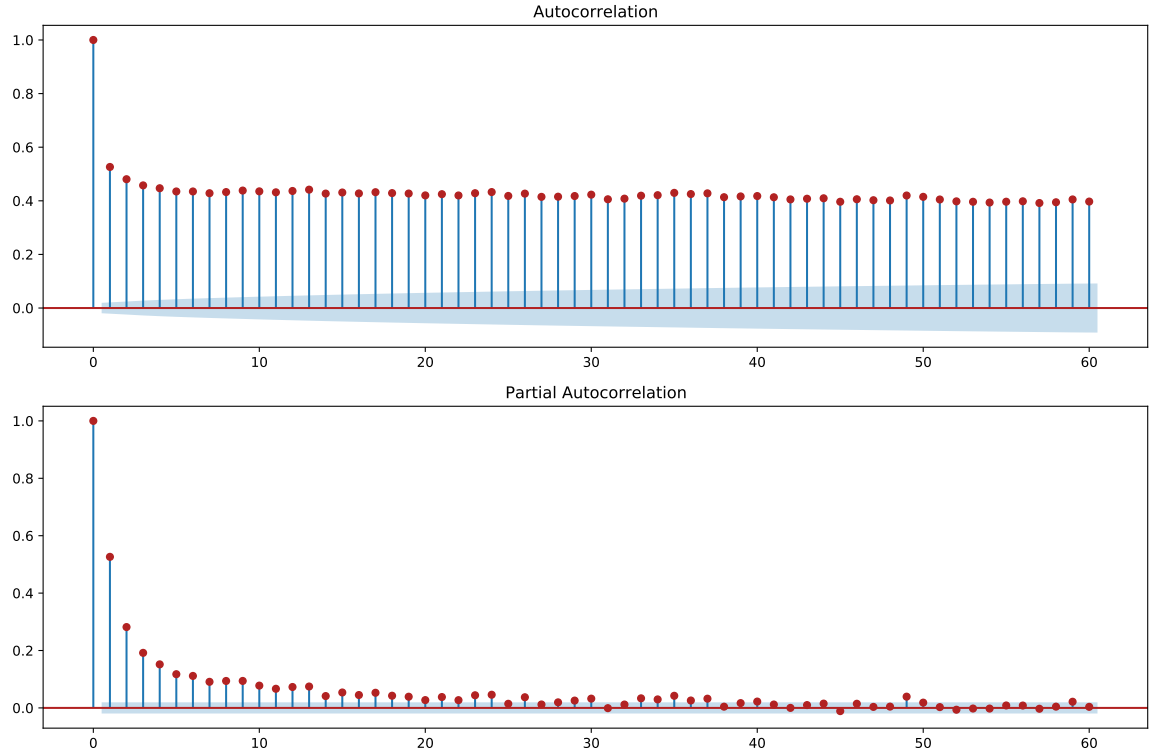


Figure 4.13: ACF and PACF with a period of 60 minutes

calls shown above from the TfL API. We created a collection of small Python functions which make common patterns shorter and easier, a file called `api_tfl_utils.py`. We will create methods to use the TfL API in Python, for which they will require the Naptan id as an argument, so first, we create a method that correspond to the `StopPoint` endpoint (Listing [3.2]), the method is `search_id_by_query`, Listing [4.5], the method will make a query and returns the Naptan Code of the query.

```
def search_id_by_query(name):
    """
    Search StopPoints by their common name

    Endpoint: https://api.tfl.gov.uk/StopPoint/Search/{query}

        Parameters
    ----------
    name : str
        The query string, case-insensitive. Leading and trailing
        wildcards are applied automatically.

    Returns
    -------
    matches_dict
        a dict with the stop points and the naptan id code
    """
```

Listing 4.5: `search_id_by_query` method

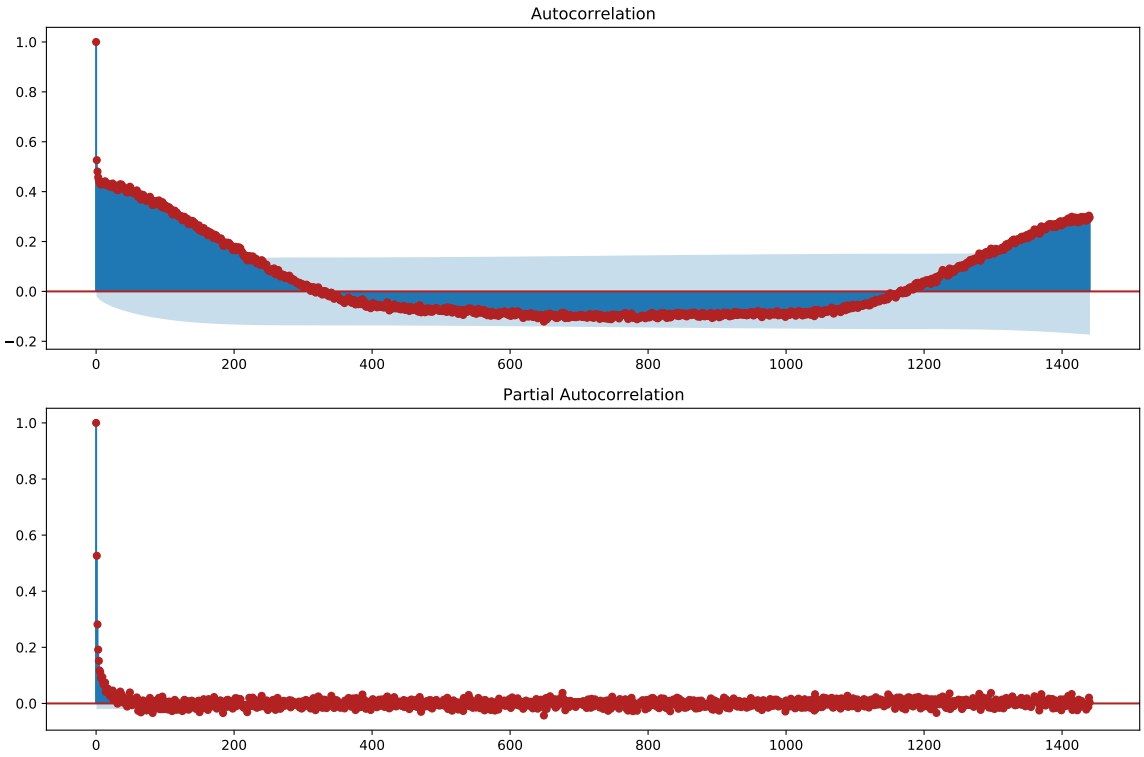


Figure 4.14: ACF and PACF with a period of 1440 minutes

An example of usage for get the Naptan Id of South Kensington Underground Station can be found in Listing [4.6].

```
> search_id_by_query("South Kensington Underground Station")
{'South Kensington Underground Station': '940GZZLUSKS'}
```

Listing 4.6: `search_id_by_query` response

The answer is a key value dictionary data structure, so we can access to the Naptan code `940GZZLUSKS` with the key `South Kensington Underground Station`. Now that we can obtain the Naptan code in a more comfortable way, we proceed with the other methods.

4.2.1 Inbound time series

Both inbound and outbound series are built in the same way, the only thing that changes are the stations that are set as departure / arrival, which establishes the direction.
We create `get_timetable` method from Listing [4.7], which consists of using the timetables endpoint explained before in Listing [3.5] and manipulating the response to obtain the hours when the train passes through the station, this is possible since in the response we find the hours when the train departs from the departure station, and how long it takes to get to the successive stations of the line.

```
def get_timetable(line, from_id, to_id):
    """
```

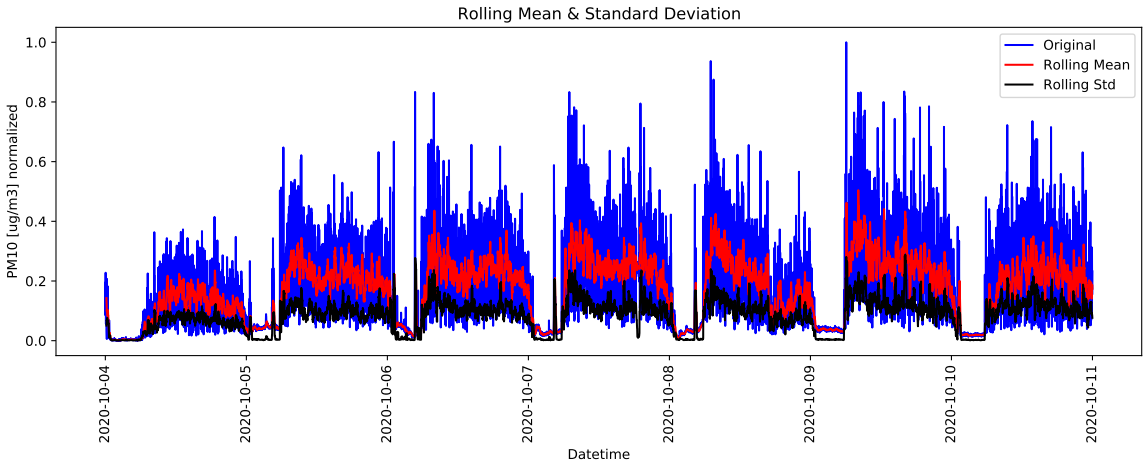


Figure 4.15: Stationarity of the air quality series

```
    Gets the timetable for a specified station on the give line with
specified destination

    Endpoint: https://api.tfl.gov.uk/Line/{id}/Timetable/{
fromStopPointId}/to/{toStopPointId}

    Parameters
    ----------
    line : str
        A single line id e.g. victoria, piccadilly ...
    from_id : str
        The originating station's stop point id
        (station naptan code e.g. 940GZZLUASL, you can use
        /StopPoint/Search/{query} endpoint to find a stop point id
from a station name)
    to_id : str
        The destination stations's Naptan code

    Returns
    -------
    matches_dict
        a dict with different days of week and the hours the train
passes for
        the station to_id
    """
```

Listing 4.7: `get_timetable` method

The biggest difficulty that we have found for the creation of this method is:

- As we have commented in the description of the timetable endpoint, the destination station only serves in a "decorative" way, it only helps us to know the direction. However, we use it as a stopping point.
- South Kensington station is under renovation works, so the exact time it takes to get from the departure station does not appear.

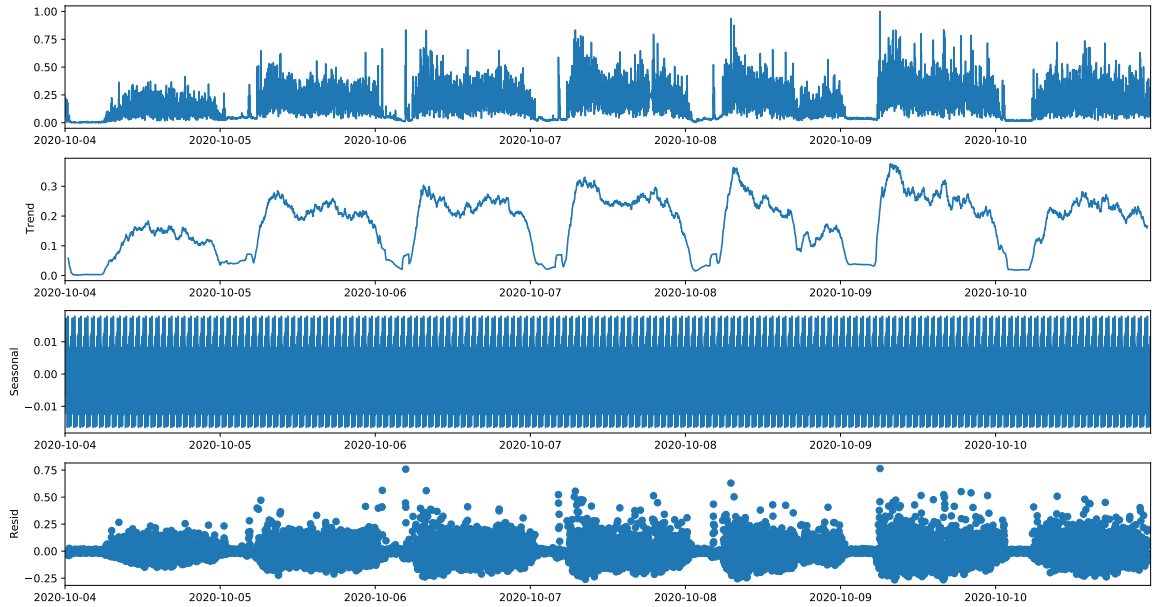


Figure 4.16: Decomposition with period of 60 minutes

To fix this, although it is not exact, we use the station before and after South Kensington, and assume that the time it takes to get to South Kensington is the average, we save the times when it leaves the departure station plus the time we assume it takes to get to South Kensington, half the time it takes to get to the next station. We can find an example of execution and response below in Listing [4.8].

Let's take a look to the Piccadilly line, in Figure [3.2], from top to bottom, the inbound direction, the station before South Kensington is Knightsbridge Underground Station, while the one after is Gloucester Road Underground Station. As the intermediate station, South Kensington is not there, so the response to the API endpoint will be the time it takes from the one after to the one before, so we apply the average assuming when it would have passed through South Kensington.

```
>from_id = search_id_by_query("Knightsbridge Underground Station")
>to_id = search_id_by_query("Gloucester Road Underground Station")
>line = "piccadilly"
>get_timetable(line, from_id["Knightsbridge Underground Station"],
   to_id["Gloucester Road Underground Station"])
{'Monday - Friday': [datetime.time(0, 1),
  datetime.time(0, 6),
  datetime.time(0, 11),
  datetime.time(0, 16),
  datetime.time(0, 21),
  datetime.time(0, 26),
  datetime.time(0, 31),
  datetime.time(0, 36),
  datetime.time(0, 42),
  datetime.time(5, 56),
  ...
   'Saturdays and Public Holidays': [datetime.time(0, 1),
```

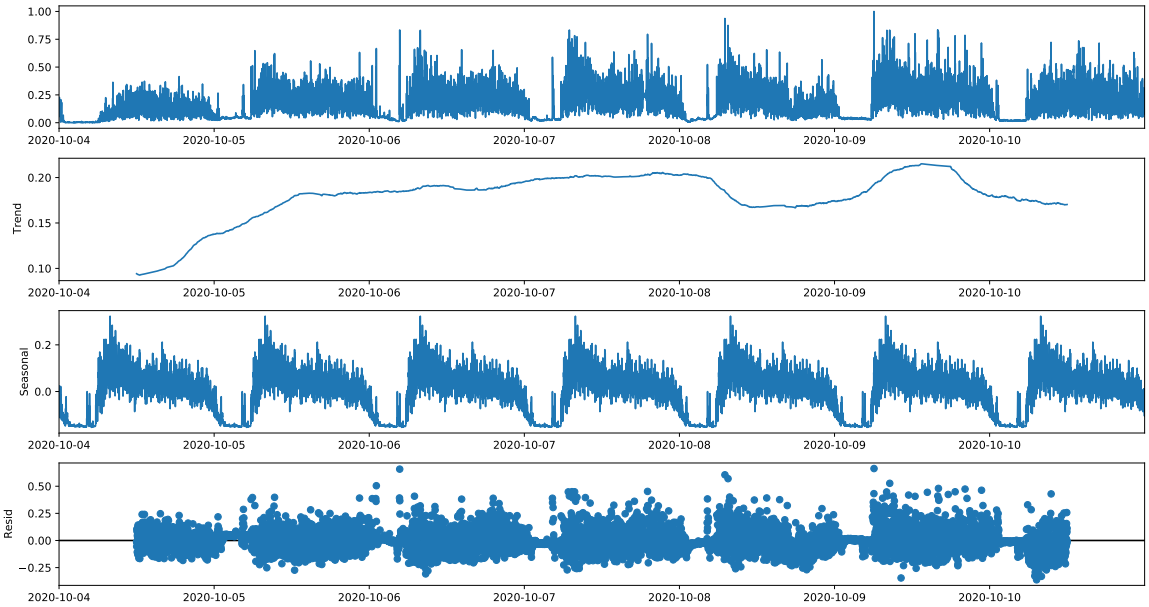


Figure 4.17: Decomposition with period of 1440 minutes

```
  datetime.time(0, 6),
  datetime.time(0, 11),
  datetime.time(0, 16),
  datetime.time(0, 21),
  ...
   'Sunday': [datetime.time(5, 56),
  datetime.time(6, 8),
  datetime.time(6, 20),
  datetime.time(6, 32),
```

Listing 4.8: `get_timetable` response Inbound direction

Using `search_id_by_query` from Listing [4.5] we obtain the Naptan codes of the stations we need, then we use them as parameters in `get_timetable` like in Listing [4.7]. For example, `from_id ["Knightsbridge Underground Station"]` is the string Naptan code `"940GZZLUKNB"`, the departure station before South Kensington in Inbound direction, and `to_id["Gloucester Road Underground Station"]` is the string Naptan code `"940GZZLUKNB"`.

We have used a datetime data type for convenience for manipulating dates and times. Once we have the hours in which a train passes through the station, using the index of the original series, tick data from Sunday, October 4,2020 to Saturday, October 11, 2020, (value per minute). We create a series that corresponds to the same time interval, but we mark with a 1 if the time is in the hours that the train passes and with a 0 otherwise. The series would have the format of the following table [4.10]
Visually it would be represented with vertical lines with a maximum value of 1 that indicate that the train passed through the station at that moment. For example, we make a small plot in an interval of day 5 (Monday), we choose a considerably

date_time	hours_train_passes
2020-10-05 00:00:00	0
2020-10-05 00:01:00	1
2020-10-05 00:02:00	0
2020-10-05 00:03:00	0
2020-10-05 00:04:00	0

Table 4.10: Inbound time serie

small interval since if not, the lines will overlap. Then, we can find a graph between `2020-10-05 00:15:00` and `2020-10-05 08:00:00`, Figure [4.18].

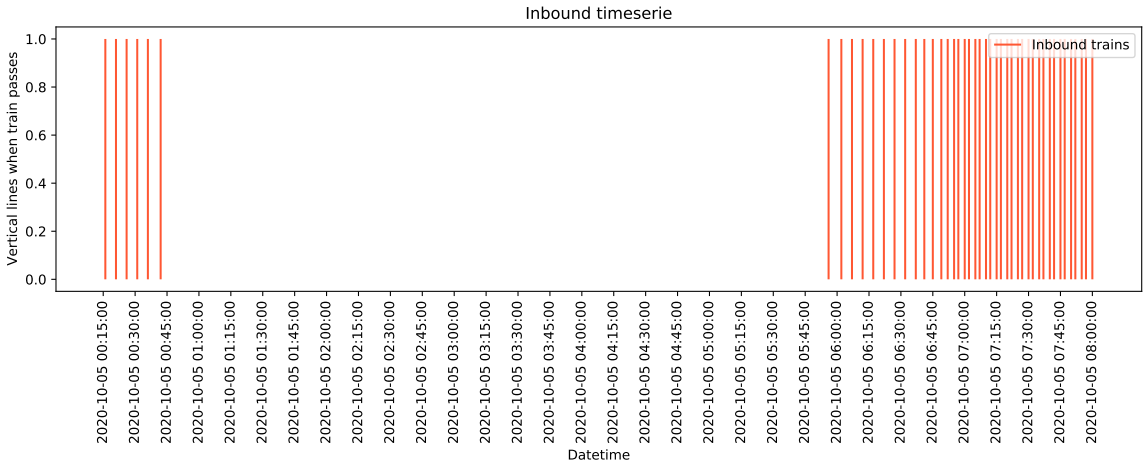


Figure 4.18: Inbound time serie

4.2.2 Outbound time serie

Similarly to how we have obtained the inbound series, again using the method `get_timetable` from Listing [4.7], we obtain the hours that he spends on the train in the opposite direction, from Gloucester Road to Knightsbridge, we see an example of execution can be found in Listing [4.9].

```
>from_id = search_id_by_query("Gloucester Road Underground Station")
>to_id = search_id_by_query("Knightsbridge Underground Station")
>line = "piccadilly"
>get_timetable(line, from_id["Gloucester Road Underground Station"],
   to_id["Knightsbridge Underground Station"])
{'Monday - Friday': [datetime.time(0, 3),
  datetime.time(0, 7),
  datetime.time(0, 11),
  datetime.time(0, 15),
  datetime.time(0, 19),
  datetime.time(0, 24),
```

```
  datetime.time(0, 28),
  datetime.time(5, 45),
  ...
  'Saturdays and Public Holidays': [datetime.time(0, 3),
  datetime.time(0, 7),
  datetime.time(0, 11),
  datetime.time(0, 15),
  datetime.time(0, 19),
  datetime.time(0, 24),
  datetime.time(0, 28),
  datetime.time(5, 45),
  ...
  'Sunday': [datetime.time(0, 1),
  datetime.time(0, 11),
  datetime.time(5, 45),
  datetime.time(5, 58),
  datetime.time(6, 10),
```

Listing 4.9: `get_timetable` response Outbound direction

Again, a visual representation of the series in the outbound direction in the same small interval shown in the series in the inbound direction between `2020-10-05 00:15:00` and `2020-10-05 08:00:00`, Figure [4.19].

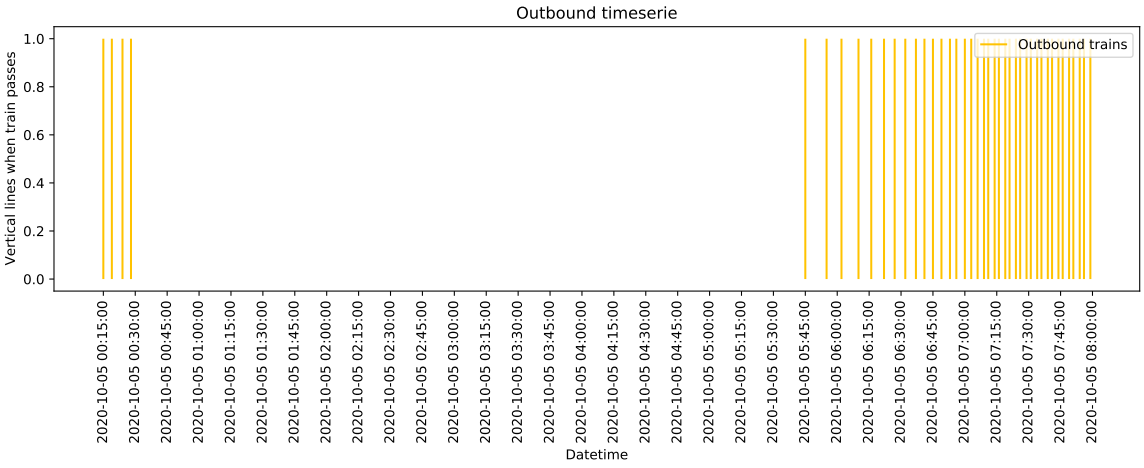


Figure 4.19: Outbound time serie

Again we observe that the early morning hours are empty due to the fact that no trains pass through during the official schedule, although in our pollution data, there may be some kind of value since it may be possible that some maintenance trains pass through during the morning hours.

4.2.3 Crowding time series

The last time series we build has to do with information about crowding levels within TfL Stations. Although the South Kensington station is under renovation, the end-point provides us with a historical value related to Wifi serial connections in the

station, and also although the line is not available the station is still open for possible transfers so the value is useful. First, we implement a method `get_crowding` that provides us with the result of calling the crowding endpoint (Listing [3.12]).

```
def get_crowding(naptan_id):
    """
    Information about crowding levels within Tfl Stations

    Endpoint: https://api.tfl.gov.uk/crowding/{Naptan}

    Parameters
    ----------
    naptan_id : str
        Naptan code

    Returns
    -------
    dict
        a dict with the timebands, and the percentageOfBaseLine of
   each day of week
        in the format MON, SUN ....
    """
```

Listing 4.10: `get_crowding` method

The result of this method will be a key value dictionary where for each day of the week an array will correspond that contains the crowding value in 15-minute intervals from 00:00 to the end of the day, as we have seen in the response of the endpoint in Listing [3.14], we see an example of execution below in Listing [4.11].

```
>crowding_id = search_id_by_query("South Kensington Underground
   Station")
>print(crowding_id)
{'South Kensington Underground Station': '940GZZLUSKS'}
>get_crowding(crowding_id["South Kensington Underground Station"])
{'WED': [0.006521003725331947,
  0.006084644046126459,
  0.005659829651076312 6,
  0.005252116680552391,
  ...
  'TUE': [0.006521003725331947,
  0.006084644046126459,
  0.005659829651076312 6,
  0.005252116680552391,
```

Listing 4.11: `get_crowding` response

With this answer we build the crowding with timebands dataframe, we show some first values of Table [4.11].
We can graphically represent the values of the Table in Figure [4.20]. In this case, the values for Monday, Thursday and Wednesday coincide, so they overlap.
Now the idea is to create another series with the same times as our original air quality series, we will extend these timebands by repeating each value 15 times, and we will correspond it according to the day of the week. The value of Wednesday

timebands	WED	TUE	SAT	THU	MON	SUN	FRI
00:00-00:15	0.006521	0.006521	0.011147	0.006521	0.006521	0.006715	0.006906
00:15-00:30	0.006085	0.006085	0.009700	0.006085	0.006085	0.005987	0.006641
00:30-00:45	0.005660	0.005660	0.008332	0.005660	0.005660	0.005288	0.006384
00:45-01:00	0.005252	0.005252	0.007040	0.005252	0.005252	0.004629	0.006117
01:00-01:15	0.004858	0.004858	0.005771	0.004858	0.004858	0.003995	0.005820

Table 4.11: Crowding with timebands of 15 minutes and separate days of the week

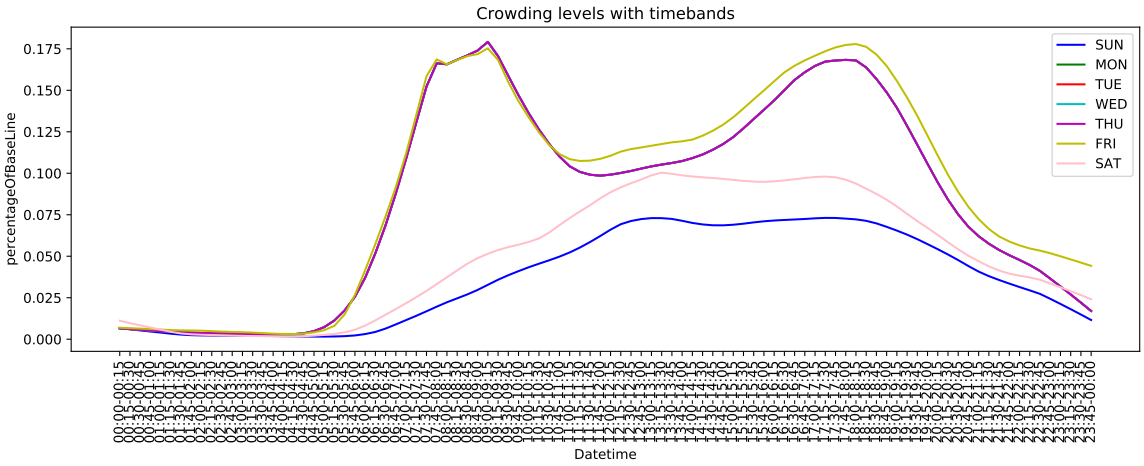


Figure 4.20: Crowding levels with timebands

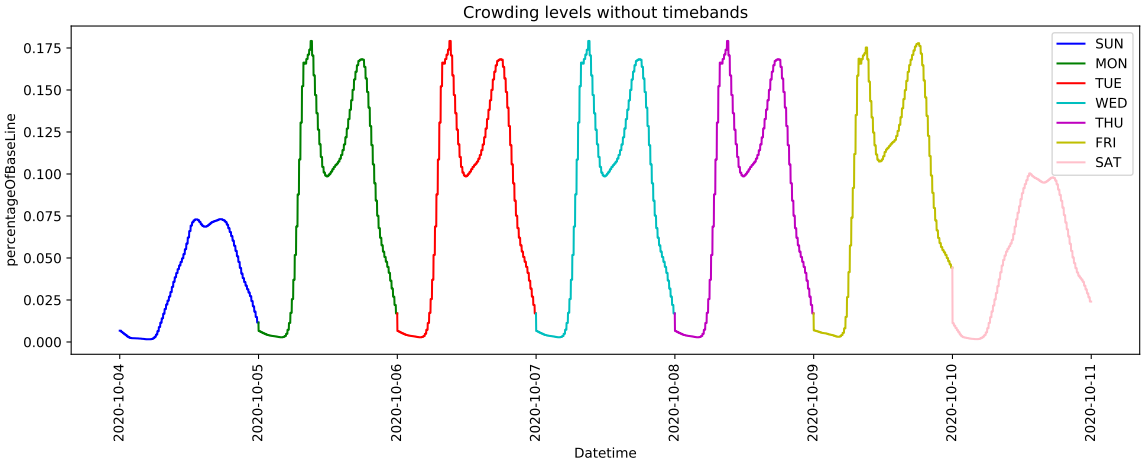


Figure 4.21: Crowding levels without timebands

between `00:00-00:15` is `0.006521`, so this will become 15 values from `2020-10-07 00:00:00` to `2020-10-07 00:14:00` of `0.006521`.
In the figure, we see how Table [4.12] would look with the commented format for the first values of `2020-10-05`. Figure [4.21] represents our final times series for the crowding levels of the South Kensignton station as we have represented our original air quality series.

date_time	crowding_information
2020-10-07 00:00:00	0.006521
2020-10-07 00:01:00	0.006521
2020-10-07 00:02:00	0.006521
2020-10-07 00:03:00	0.006521
2020-10-07 00:04:00	0.006521
...	...
2020-10-07 00:14:00	0.006521
2020-10-07 00:15:00	0.006085

Table 4.12: Crowding without timebands

Finally, we obtained three different series from the original one that we were given to compare, two series where the time at which the train passes through the target station, South Kensington (approximated time), and another series that contains crowding information levels in the same station.

4.3 Comparison of time series

Now we will experiment on correlating peaks in pollution with trains arriving at the station and crowding data. But, first, let's plot some visualization comparing our data, pollution information, crowding level and train departure and arrival peaks.

Again, for a more comfortable and comparative visualization, we will choose a limited interval, such as a few hours at the beginning of the 5th, Monday. We use 8 hours from the beginning of the day (From 00:00 to 08:00). The data displayed will be normalized between 0 and 1, as already explained in a previous section. We see the pollution data with the trains in the inbound direction, Figure [4.22]. Also, with the trains in the outbound direction, Figure [4.23], and with both peaks of trains in the inbound and outbound demand at the same time Figure [4.24].

We also visualize pollution data with normalized crowding levels in the same previous interval , Figure [4.25], and all of the above in a graph, Figure [4.26].

To have a more global and general vision, we show this last graph where we visualize all our datasets in a larger interval, in two days, Figure [4.27] and throughout the week, Figure [4.28]. As we have mentioned, we see that it is not easy to visualize the times when the trains pass in the inbound and outbound direction because the lines overlap even if we make them as thin as possible. Still, it is good to have a more general visualization since the sections with no trains passing can be seen.

Also, although we do not have train schedules at those times, small pollution peaks are perfectly affordable since test trains can also pass at different times outside the usual and established plan.

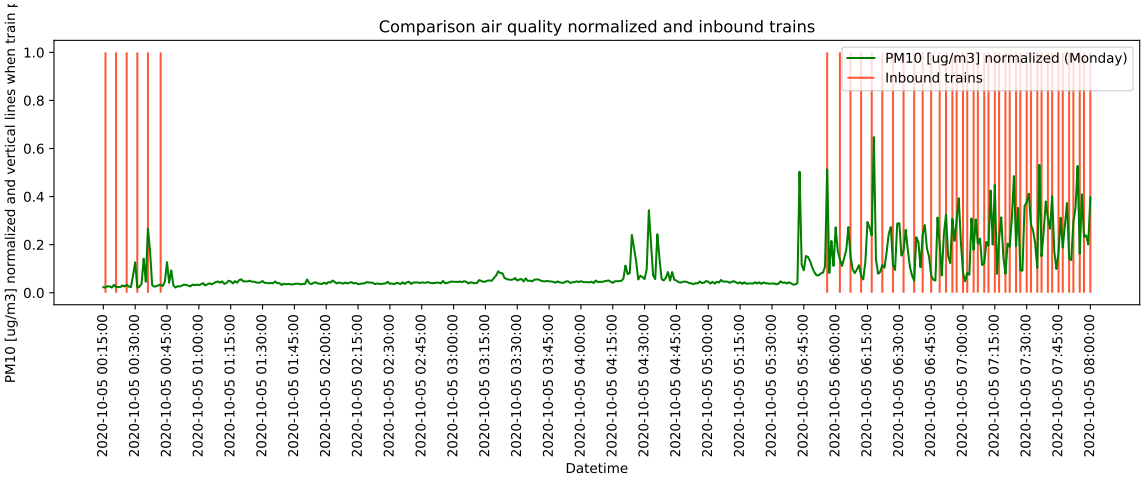


Figure 4.22: Air quality with inbound trains

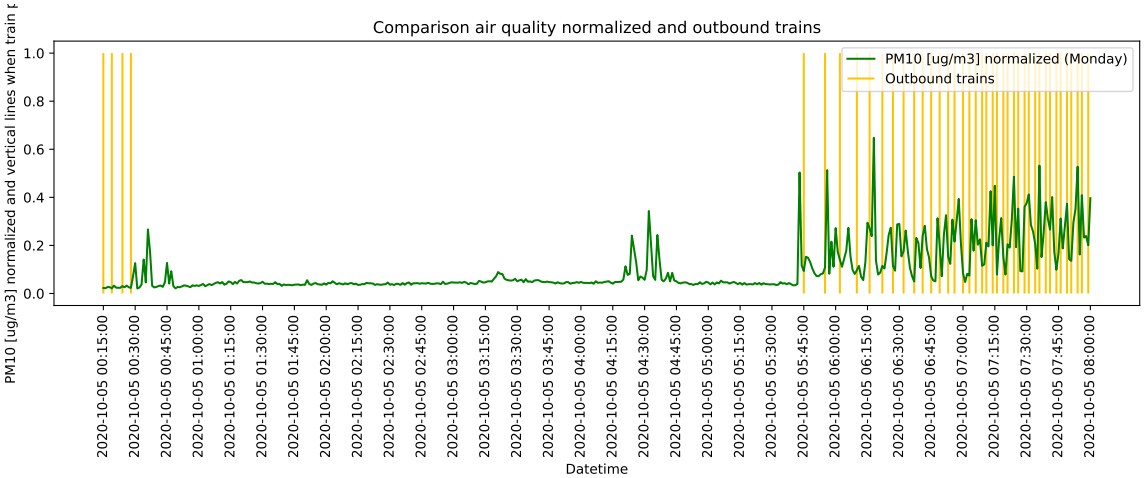


Figure 4.23: Air quality with outbound trains

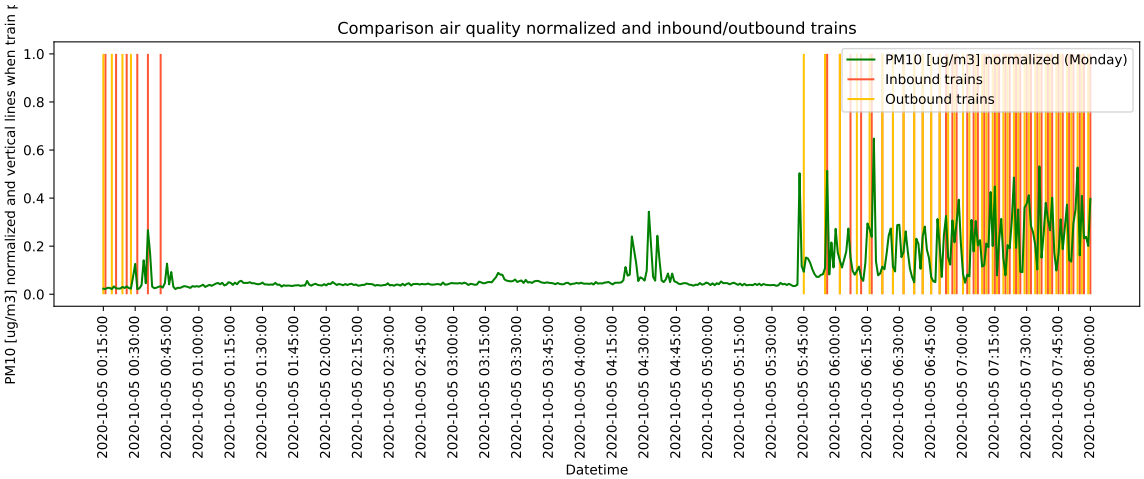


Figure 4.24: Air quality with inbound and outbound trains

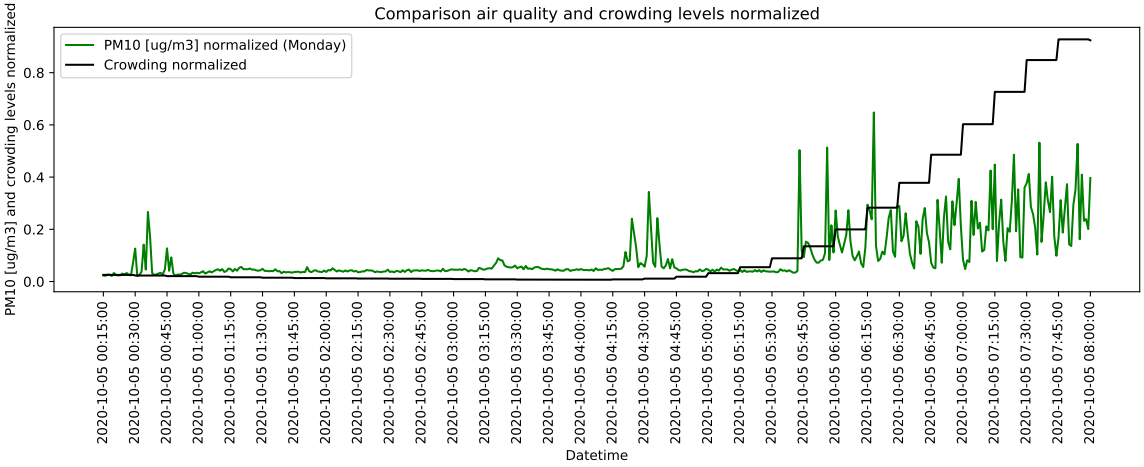


Figure 4.25: Air quality with crowding levels

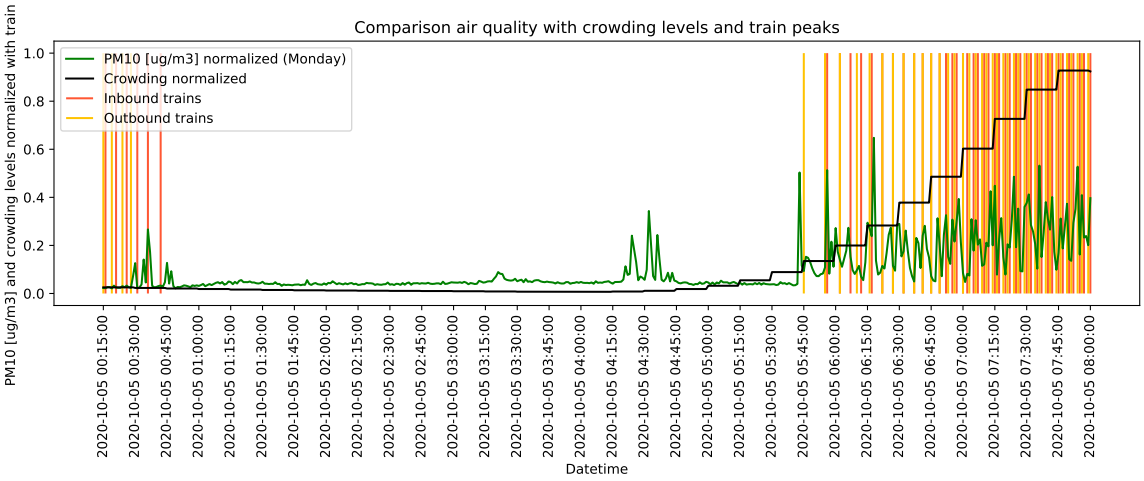


Figure 4.26: Air quality with crowding levels, inbound and outbound trains

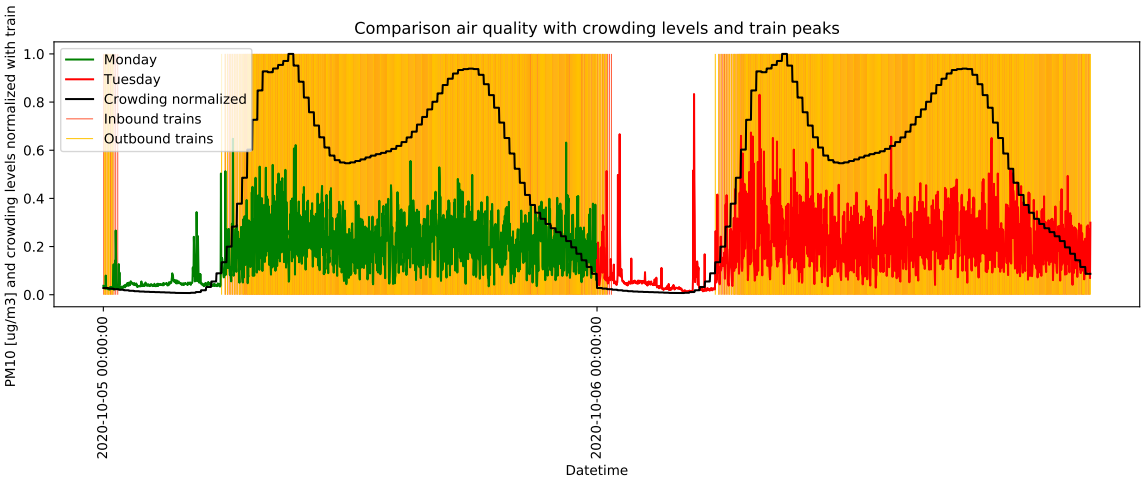


Figure 4.27: Air quality with crowding levels, inbound and outbound trains in two days

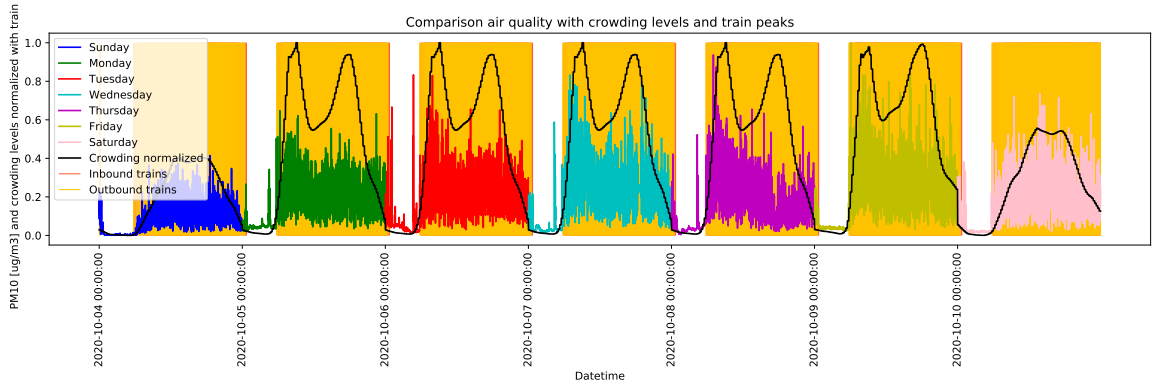


Figure 4.28: Air quality with crowding levels, inbound and outbound trains in the whole week

4.3.1 Cross-correlation

After making visualizations of our different datasets, we will study the cross-correlation of our time series. First, Pearson's correlation expands into cross-correlation. Cross-correlation aims to establish a relationship between the lags of two time series when comparing them.

The first step in detecting cross-correlation between two time series is to ensure that both are stationary (as we know, have a constant mean, variance, and autocorrelation). This step is significant since a correlation seeks to determine a linear relationship between two variables. We already checked that before with the ADFuller test, Listing [4.2].

The Pearson correlation determines how two continuous series vary over time and expresses the linear relationship as a number ranging from -1 (negatively correlated) to 0 (not correlated) to 1 (perfectly correlated).

Several factors can influence Pearson's correlation coefficient. For example, outliers can distort correlation estimation findings and lead to the assumption that the data is homoscedastic (variance of your information is homogenous across the data range). We have dealt with this when preprocessing and scaling datasets.

We calculate the Pearson correlation between all the time series. This information is showed in Figure [4.29]. We have added some sets with a delay of more or less a minute since we have commented on some occasions that the pollution measurement does not have to be instantaneous just when the train arrives or leaves the station. The delay of more or less one minute corresponds to the same series moved one minute to the right (if added) or one minute to the left (if subtracted). In Figure [4.29], the name of the datasets correspond to the time series, followed by + or - minutes, if some delay has been applied. For example, `inbound_times_shift_+1_minutes` indicates the series of inbound trains shifted 1 minute to the right.

The Pearson correlation, once again, is a metric of global synchronization that boils down the relationship between two time series to a single number. Using Pearson correlation, there is a way to look at moment-to-moment local synchronization.

One method is to measure the Pearson correlation in a tiny area of the series and

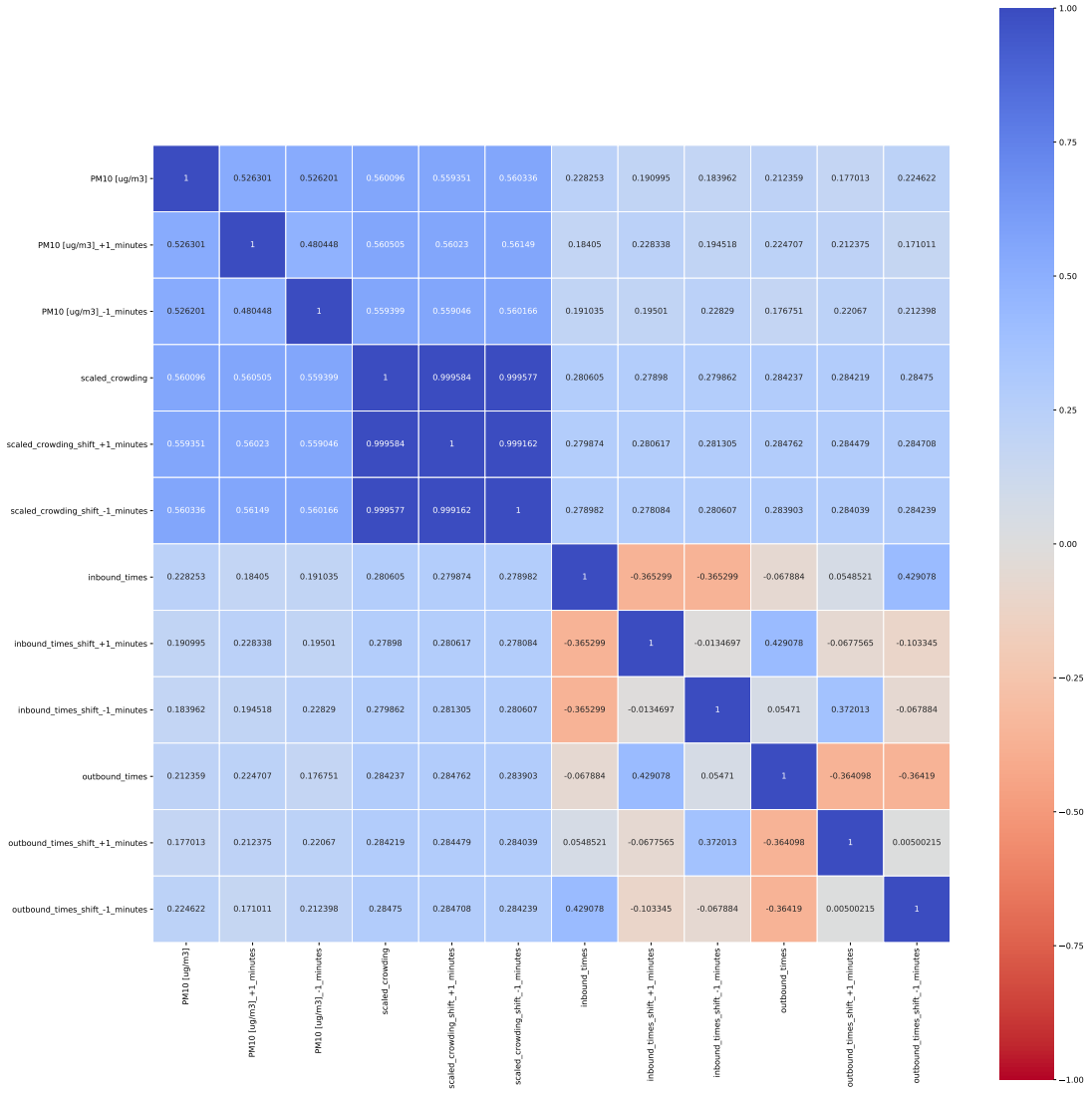


Figure 4.29: Heatmap Pearson correlation

then repeat the process with a rolling window until the entire series is covered. This method is subjective because it requires arbitrarily defining the window size in which you'd like to repeat the procedure. In this case, we use a window size of 60 minutes and plot the moment-to-moment synchrony in the bottom figure of the pollution data, inbound, outbound times and crowding (Figure [4.30]).
We see that there are gaps at the beginning of each day, it makes sense since there will be null values due to the lack of pollution, people and trains.

4.3.2 Dynamic Time Warping

As we explained in the Section [3.5.1], Dynamic time warping is used as a similarity measure between temporal sequences. We will experiment with the different

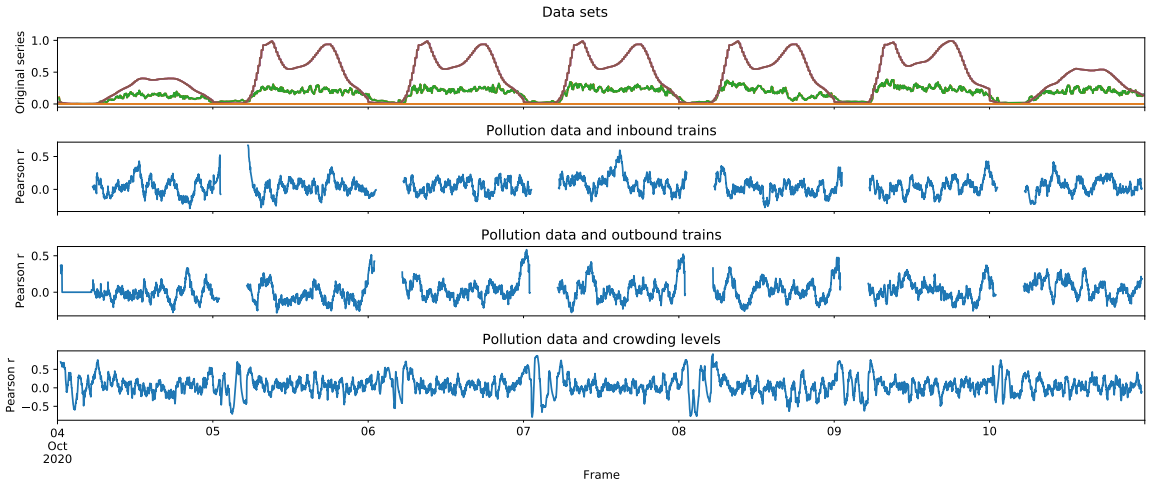


Figure 4.30: Local correlation in interval of 60 minutes

Python modules and see the algorithm's results between our other time series, pollution data, train schedules, and crowding levels.

First, let's test the Python module with the optimized version of DTW due to better distance calculation. Then, from the Github repository [`https://github.com/pierre-rouanet/dtw`], we applied the accelerated version of the algorithm between our pollution data and the inbound train schedule. Result of the accumulated cost matrix and the shortest path (in white) found in Figure [4.31] where the value of the axes correspond to seconds.
Although it is an optimized version of the algorithm, as we have seen in the section where we explained DTW, its efficiency is still the multiplication of the maximum number of values of each compared time series. We have added a parameter to quantify the time (in seconds) that the algorithm takes to run. In the case of Figure [4.32], it takes 570 seconds$\sim$10 minutes, a relatively large amount.

Therefore, in the future, we will apply DTW in a shorter time interval, such as two days. However, remember that the default distance is calculated in this DTW implementation is the Euclidean distance power of two.
We will apply this optimized algorithm in the two-day interval between our pollution data with train schedules in the inbound direction (again) (Figure [4.33]), outbound direction (Figure [4.34]) and crowding levels (Figure [4.35]). We see that compared to the case where we applied DTW to the whole week, the time it takes to use the algorithm has decreased considerably, not even reaching one minute (in the worst case, 49 seconds). We also see that the distance between pollution data and crowding levels is undoubtedly the smallest, followed by trains in the outbound direction, and finally trains in the inbound order, the latter two with a difference in the DTW result of fewer than 5 units. So earlier Crowding levels data is matched with the synchrony of later pollution PM_{10} data. This is how one can determine the similarity of two different time series via dynamic time warping.
We will now use the second module, which you can found in the Github repository

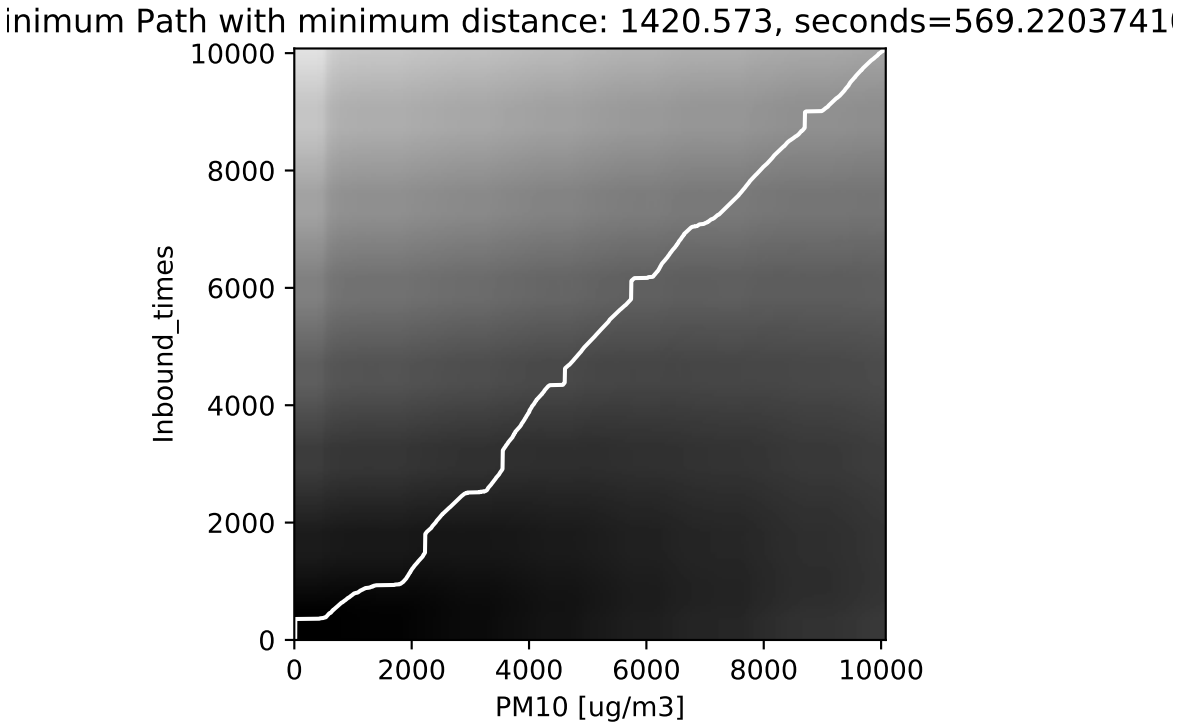


Figure 4.31: Optimized DTW one week

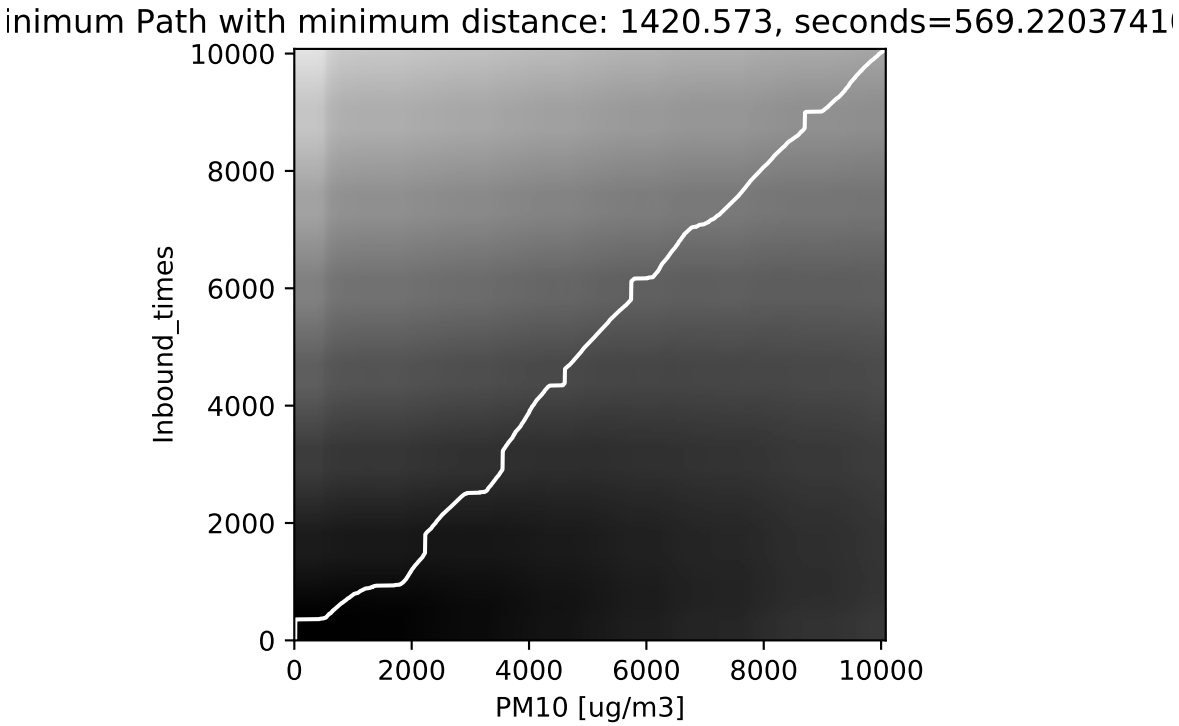


Figure 4.32: Optimized DTW pollution data with inbound trains one week

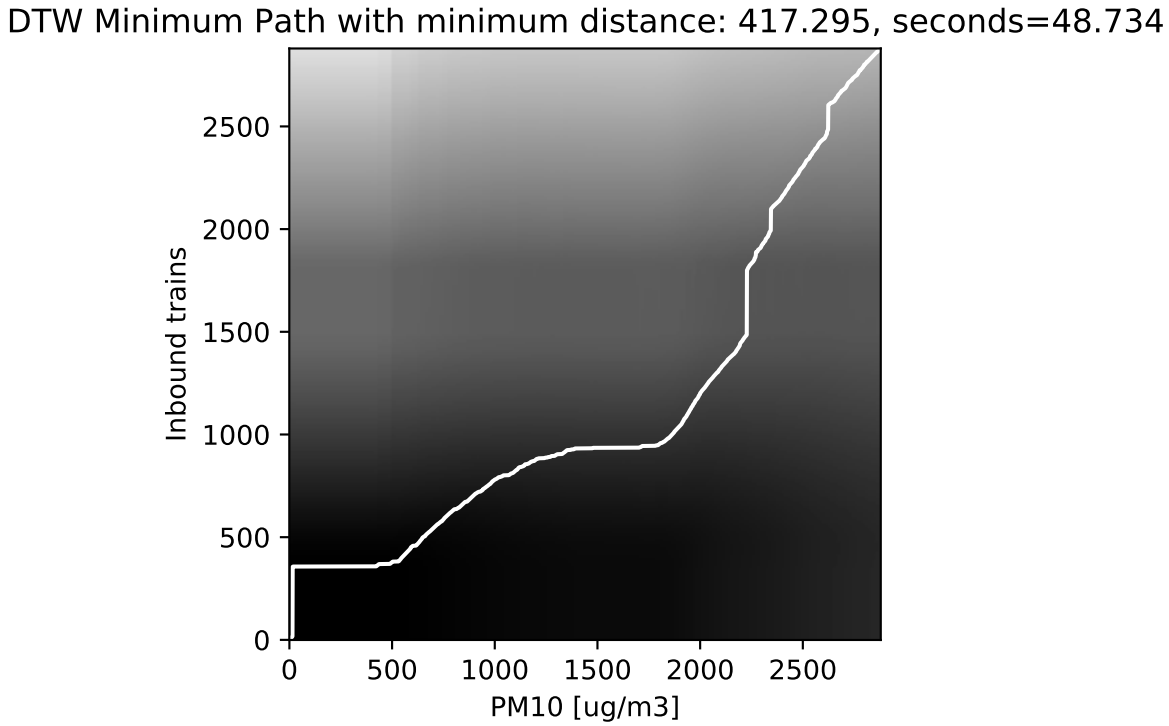


Figure 4.33: Optimized DTW pollution data with inbound trains two days

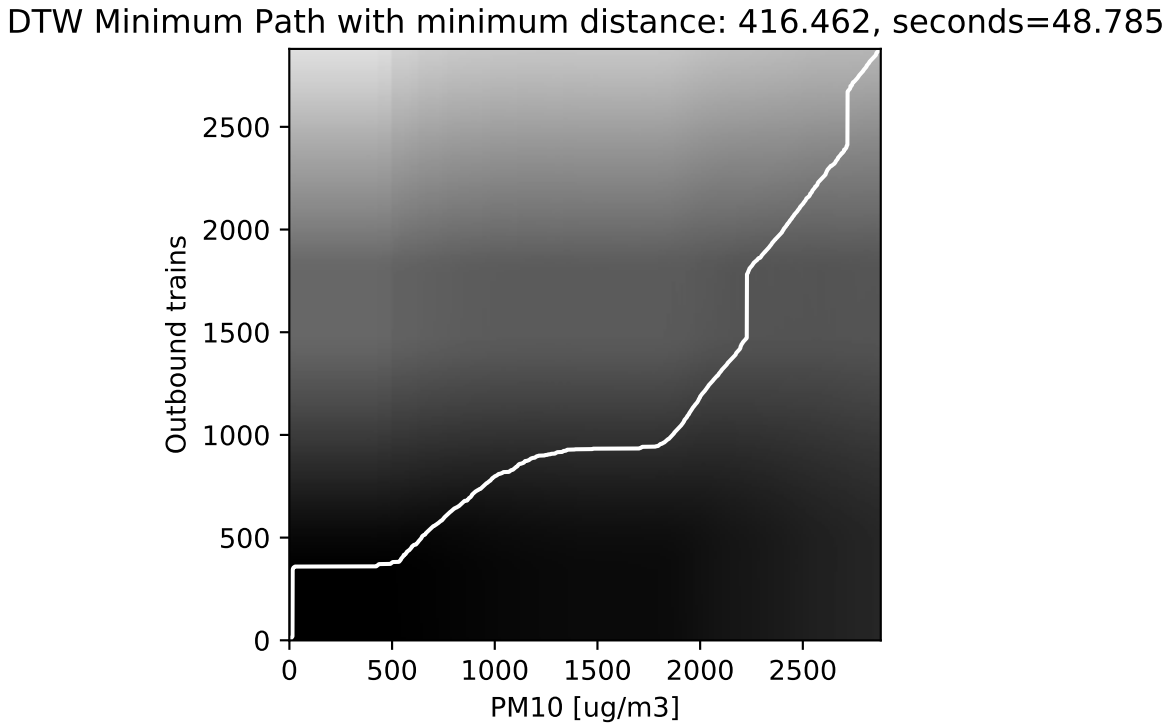


Figure 4.34: Optimized DTW pollution data with outbound trains two days

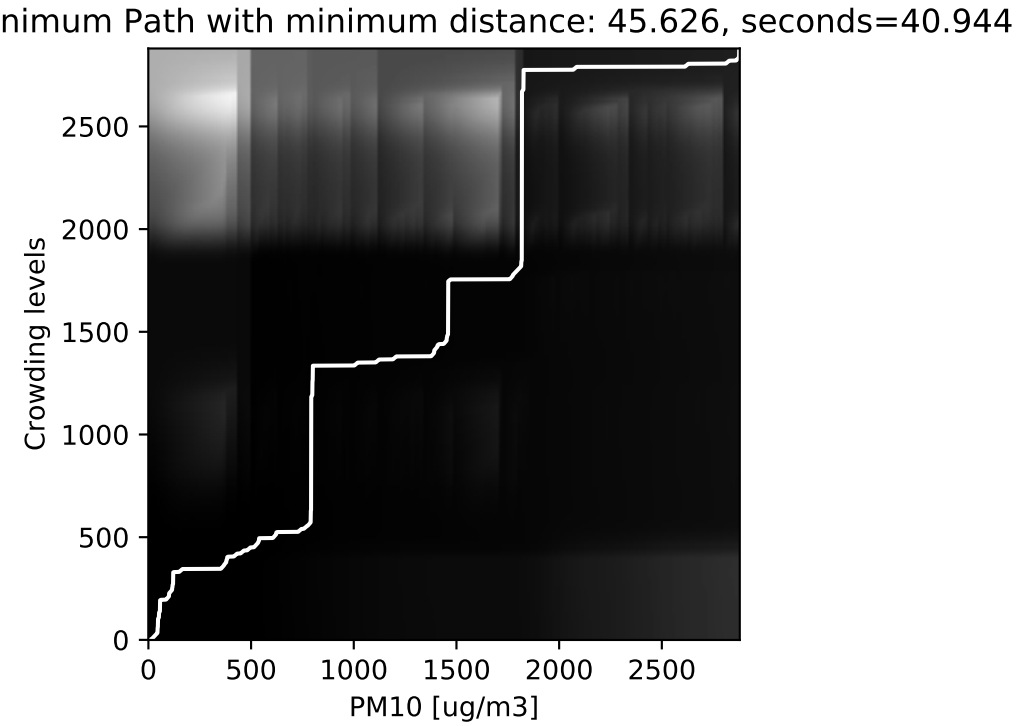


Figure 4.35: Optimized DTW pollution data with crowding trains two days

[https://github.com/wannesm/dtaidistance] whose strong point we had already mentioned was the simultaneous display of the data and the warping path of DTW.

In this case, to add some diversity, we will apply DTW between pollution data and crowding level to the whole week since we have seen that it was the case that was more synchronous with the previous module. This is shown in Figure [4.36].

The results for applying DTW to pollution data and crowding levels, as in the previous module, are outstanding. Again, the indices correspond to seconds. In this case, the distance shown is the actual Euclidean distance, not the previous module that used the squared Euclidean distance.

Although we have not commented on it, in this case, the performance is much worse than in the previous point. However, visually it is more pleasant as it allows comparison with the graphs of the series, for example; in the last Figure [4.36] where we compare pollution data and crowding level of the whole week, we see that the warping path (the red line) is transported horizontally, (that is, there are many minimum values in a row) occur between day and day when there are hardly pollution values (because no trains pass) or importance of people in the station (because the station is closed).

By the same reasoning as above, we will also reduce the interval of applying DTW with this module.

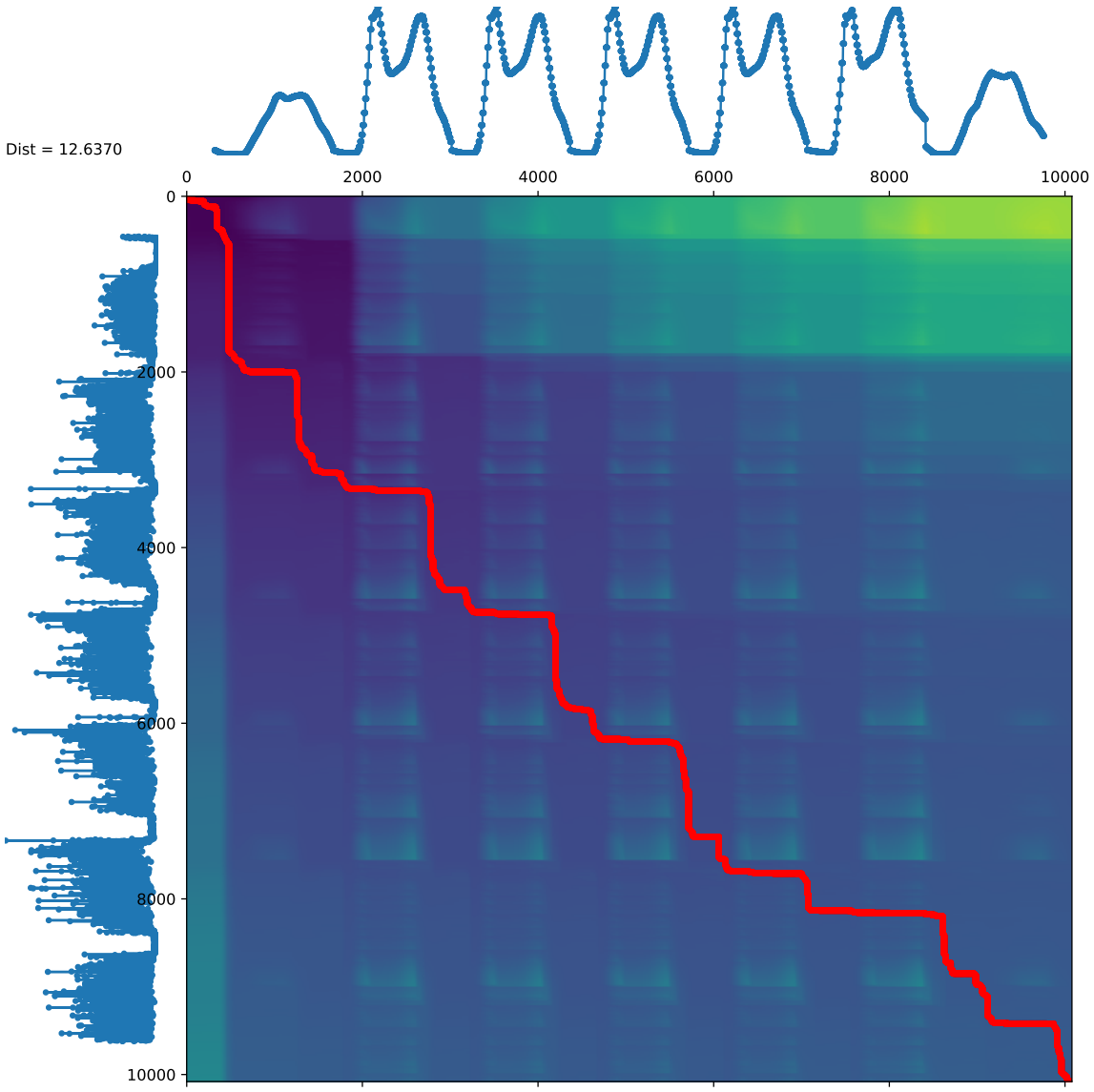


Figure 4.36: DTW pollution data and crowding levels with plots in one week

We show DTW with two-day plots of pollution data with crowding levels in Figure [4.37], with inbound trains in Figure [4.38], and with outbound trains in Figure [4.39].

The different distance values obtained in this module compared to the previous one are because the calculated distance during the last module is the squared Euclidean distance. In contrast, in this module, the actual Euclidean distance (l2 norm) has been used.

We also see that in Figures [4.38] and [4.39], we have problems visualising the graphs of our train schedules (where we marked the time value if the train passed

and with 0 if it did not pass). As they are vertical lines, if we choose considerable intervals, it is expected that they overlap, and it isn't easy to see them.

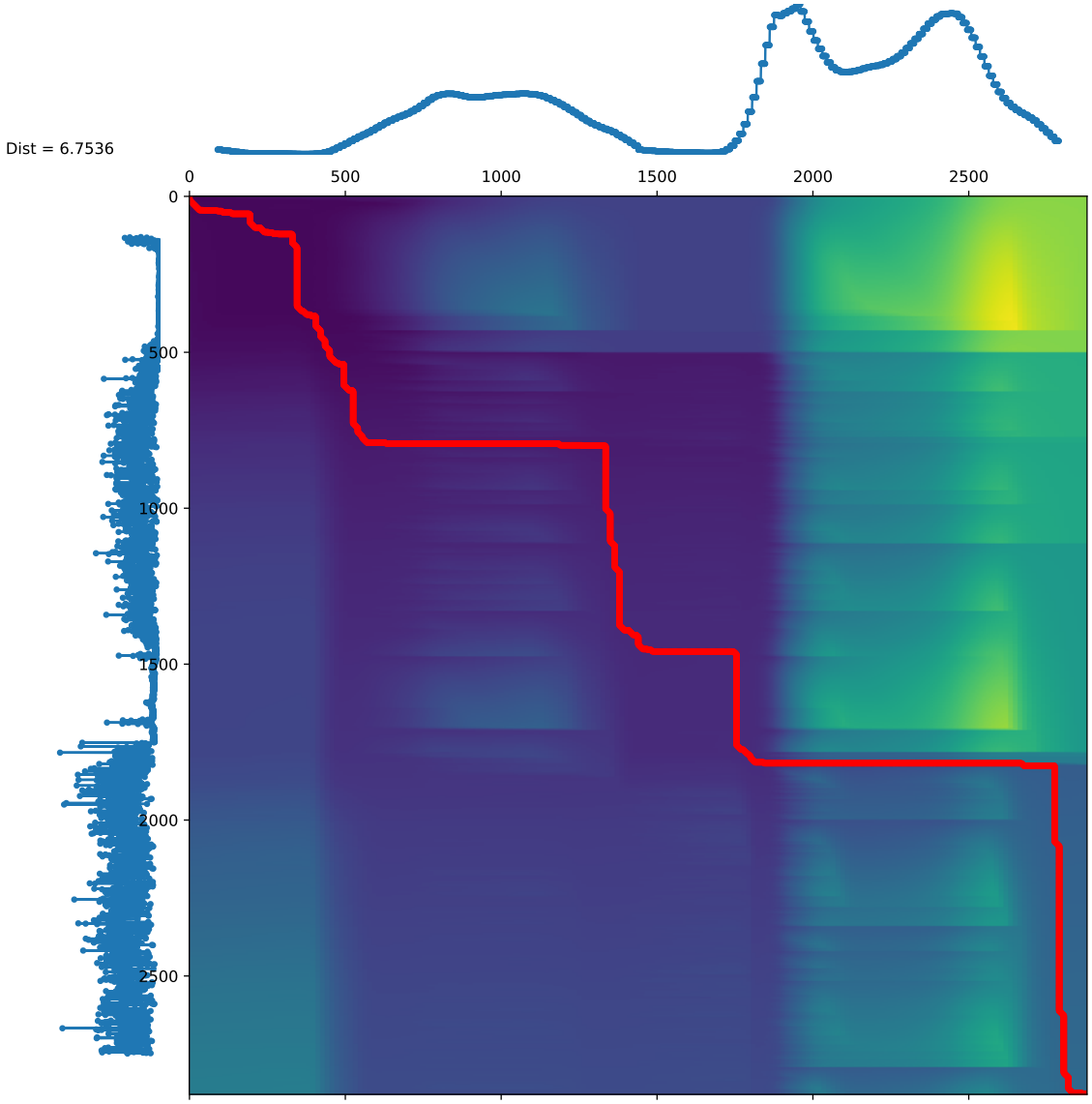


Figure 4.37: DTW pollution data and crowding levels. Only two days are shown.

Again, as in the first DTW Python module, the pollution time series is most in sync with station crowding levels, followed by train schedules in the outbound direction, and finally, train schedules in the inbound order.

4.4 Forecasting

In this section of experiments, we will focus on our models for forecasting pollution data.

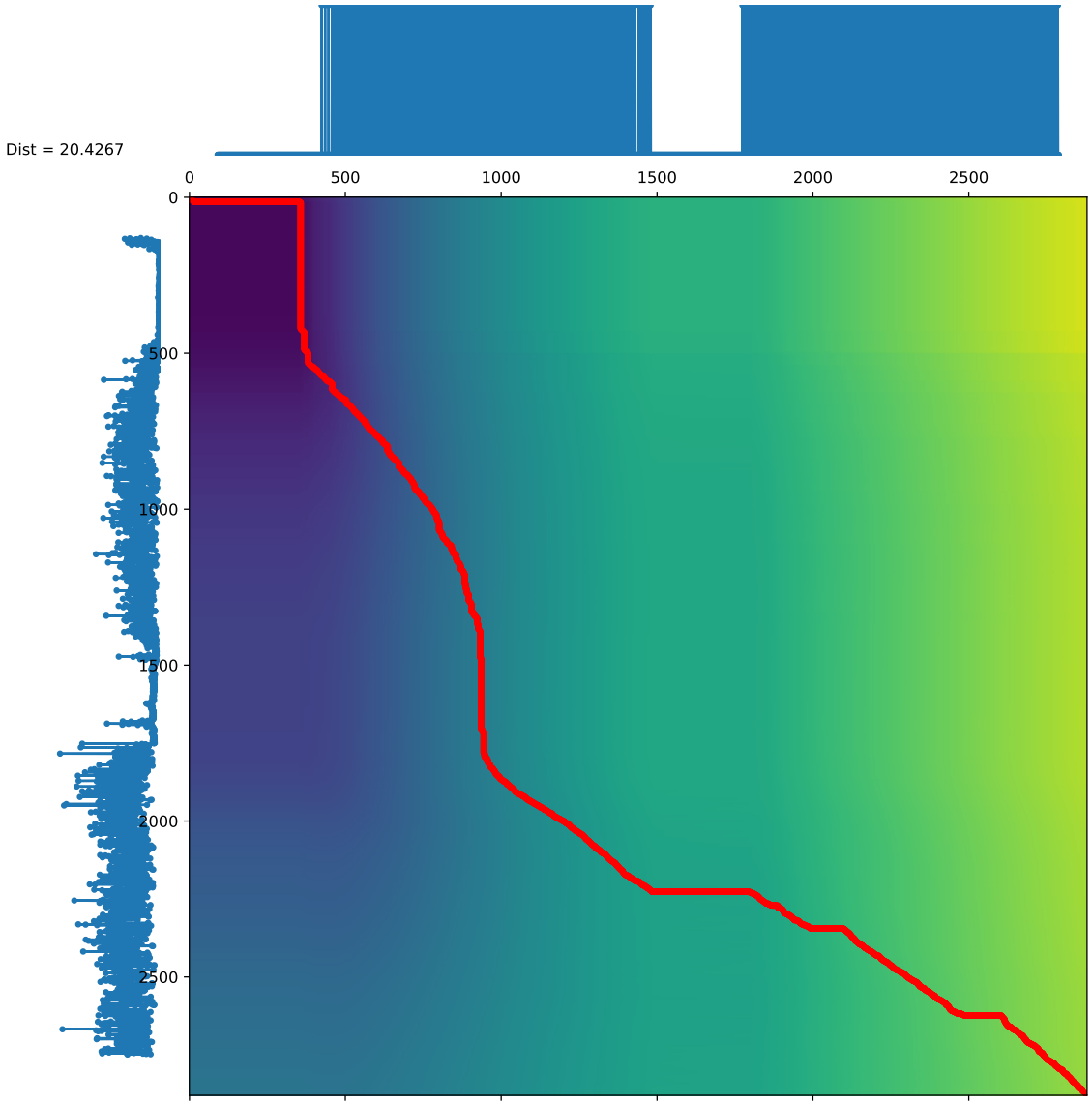


Figure 4.38: DTW pollution data and inbound trains. Only two days are shown.

Forecasting a time series has a lot of commercial applications. The methodology and concepts behind time series forecasting are relevant in any industry, but it does not have to be for commercial purposes. For example, it can be for health improvement purposes; the benefits of predicting air pollution are evident in this case.

A time series can now are separated into two forms of forecasting.

- Univariate Time Series Forecasting is when you utilize only the initial values of a time series to predict its future values.
- Multi-Variate Time Series Forecasting is when you employ predictors other than the series (also known as exogenous variables) to forecast.

We focus on a particular type of forecasting method called ARIMA modelling. We

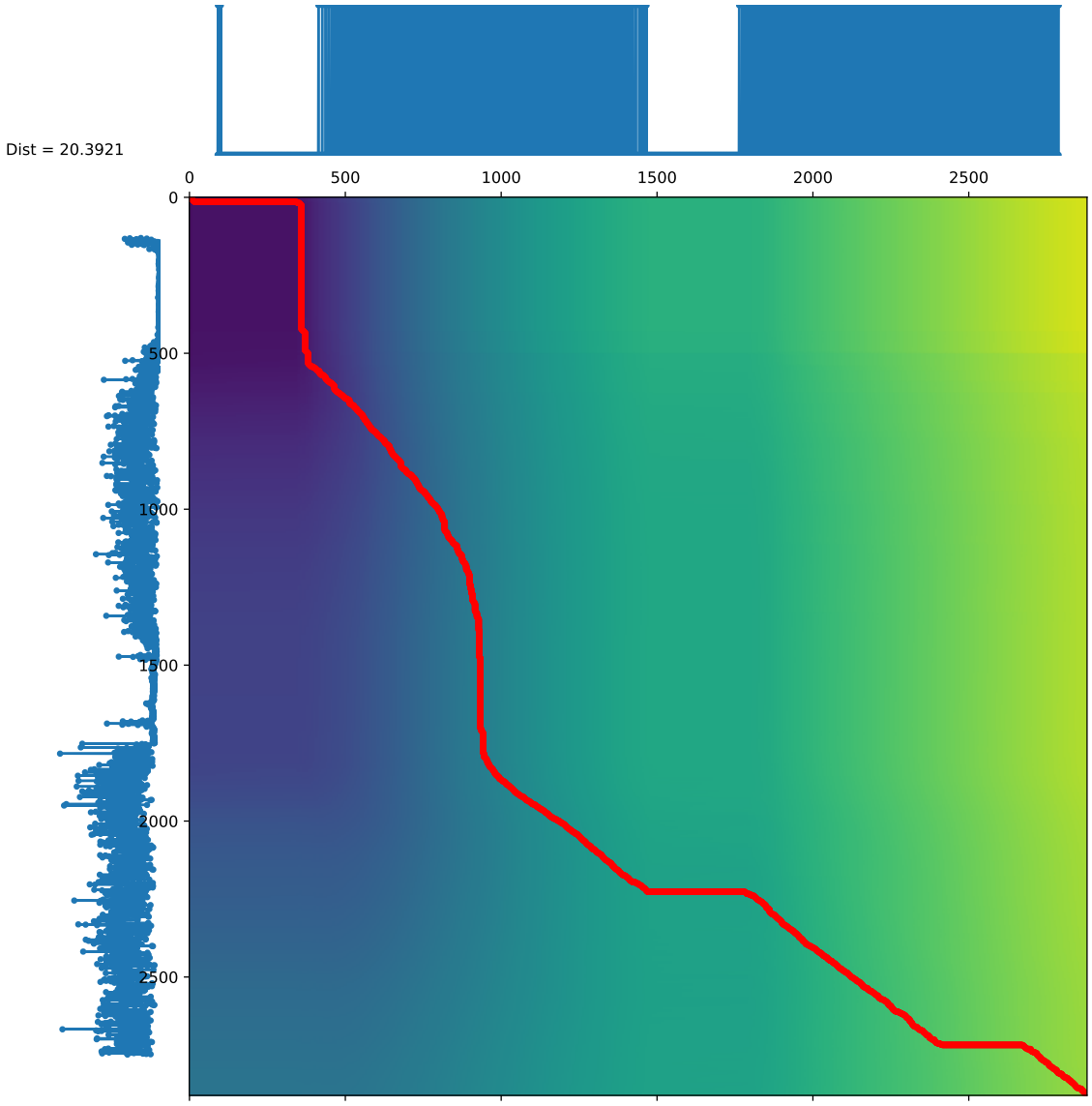


Figure 4.39: DTW pollution data and outbound trains. Only two days are shown.

will first build this ARIMA model. Then we will apply the different Deep learning algorithms explained in the methods section. Finally, we will make a few other neural models, including Convolutional and Recurrent Neural Networks (CNNs and RNNs).

4.4.1 ARIMA model

We recall that the **ARIMA(p,d,q)** model has the following parameters:

- **p** is the order of the autoregressive part
- **d** the degree of the differencing involved
- **q** order of the moving average part

The first thing we have to do to build our ARIMA model is to find the correct order of the model parameters. In a previous Section [4.1], we have seen that our series is stationary, both visually and statistically, using the ADF test. Therefore, the value of **d** is the minimum number of differences needed to make the series stationary. And if the time series is already stationary, then d = 0.

Next, what are the **p** and **q** terms?
p is the order of the "Auto-Regressive" (AR) term. The number of lags to use as predictors. And **q** is the order of the "Moving Average" (MA) term. The number of lagged forecast errors should go into the ARIMA Model. The next step is to identify if the model needs any AR terms.

We have seen that we initially take the order of the AR term to be equal to as many lags as cross the significance limit in the PACF graph. Similarly, we choose the order of the MA term to be the number of lags that cross the boundary in ACF. We have previously visualised the Auto Correlation Function and Partial Auto Correlation Function (PACF) with 60 lags (1 hour-60 minutes), Figure [4.13] and 1440 lags (24 hours-1440 minutes), Figure [4.14] of our pollution data. We see that in this case, it is challenging to choose order by visualising both functions.

Other statistical programming languages, like R, offer automated solutions to this problem, but they have not yet ported to Python. Using the Akaike Information Criteria to identify the optimum model values (AIC), we will utilise a "grid search" to iteratively examine alternative parameter combinations. We fit a new ARIMA model with the `ARIMA()` function from the `[statsmodels]` package for various parameters and assess its overall quality. Our optimal set of parameters will be the one that gives the best performance for our criteria of interest after we've explored the complete landscape of parameters.

As a result of our brute-force run, Table [4.13] shows the five best parameter configurations ordered by AIC value from least to most significant for the different ARIMA models of order **(p, d, q)**.

Order (p,d,q)	AIC
(2, 0, 2)	-17019.440024
(3, 0, 1)	-17019.307341
(3, 0, 2)	-17017.125990
(2, 0, 3)	-17015.260911
(3, 0, 3)	-17014.089525

Table 4.13: Best ARIMA parameters by AIC.

In application, one computes AIC for each candidate model and selects the model with the smallest value of AIC.

Referring to [BA03], usually, AIC is positive; however, it can be shifted by an additive constant, and some shifts can result in negative values of AIC. It is not the absolute size of the AIC value. The relative values over the set of models considered, particularly the differences between AIC values, are essential.

We found the collection of parameters that best fit our time series data using grid search and the AIC value. Our corresponding ARIMA model would have the properties summarised in Listing [4.12].

```
                               SARIMAX Results
==========================================================================================
Dep. Variable:           PM10 [ug/m3]   No. Observations:                 10080
Model:                ARIMA(2, 0, 2)   Log Likelihood                8514.720
Date:                 Tue, 28 Jun 2021   AIC                          -17019.440
Time:                         14:12:17   BIC                          -16983.350
Sample:                     10-04-2020   HQIC                         -17007.228
                          - 10-10-2020
Covariance Type:                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.4333      0.051     28.202      0.000       1.334       1.533
ar.L2         -0.4336      0.051     -8.534      0.000      -0.533      -0.334
ma.L1         -1.2455      0.053    -23.444      0.000      -1.350      -1.141
ma.L2          0.2778      0.050      5.565      0.000       0.180       0.376
sigma2         0.0108      0.000    100.691      0.000       0.011       0.011
===================================================================================
Ljung-Box (L1) (Q):                   0.02   Jarque-Bera (JB):              6079.67
Prob(Q):                              0.90   Prob(JB):                         0.00
Heteroskedasticity (H):               1.82   Skew:                             1.11
Prob(H) (two-sided):                  0.00   Kurtosis:                         6.09
===================================================================================
```

Listing 4.12: ARIMA Summary

The summary from Listing [4.12] contains a lot of data, but we'll concentrate on the table of coefficients. The `coef` column displays each feature's relevance (weight) and how it affects the time series. The `P>|z|` column tells us how important each feature weight is.

We found the collection of parameters that best fit our time series data using grid search and the AIC value. Therefore, our corresponding ARIMA model would have the properties summarised in the Listing. When fitting ARIMA models (or any other model for that matter), we should do model diagnostics to confirm that none of the model's assumptions broke. Those diagnostics are shown in Figure [4.40].
Our primary concern is that our model's residuals are uncorrelated and normally distributed with a zero-mean distribution. If the seasonal ARIMA model does not meet these criteria, it can likely be improved further.

Based on Figure [4.40], our model diagnostics imply that the model residuals are normally distributed, analysing the result:

- In the top right plot, we see that the orange KDE line follows closely with the N(0,1) green line (where N(0,1)) is the standard notation for a normal distribution with mean 0 and standard deviation of 1). A good indication that the residuals are normally distributed.

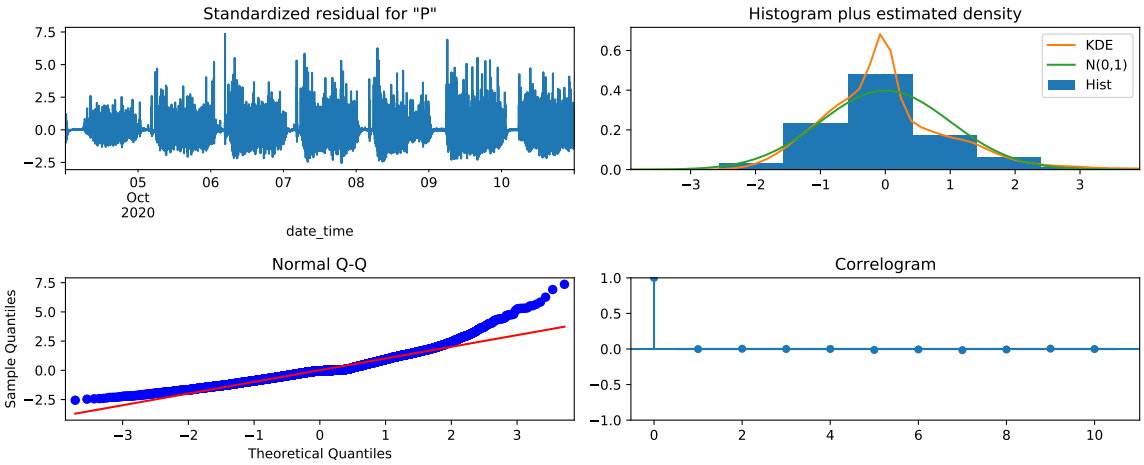


Figure 4.40: ARIMA model diagnostics

- The ordered distribution of residuals (blue dots) follows the linear trend of samples drawn from a standard normal distribution with N in the QQ plot on the bottom left (0, 1). This fact indicates that the residuals are normally distributed once more.
- There is no clear seasonality in the residuals over time (top left plot). The autocorrelation (i.e. correlogram) plot on the bottom right confirms this, indicating that the time series residuals exhibit minimal correlation with lagged copies of themselves.

These findings encourage us to believe that our model provides a good match, which could aid us in understanding our time series data and forecasting future values. However, although we have a good match, the model might tweak certain factors in our ARIMA model to improve it.

For our time series, we've created a model that we may now use to make forecasts. We begin by comparing predicted values to actual time series values to determine the accuracy of our predictions. In Figure [4.41], we make one-step-ahead forecasts, which means that forecasts at each point base on the entire history up to that point. We also plot the actual and predictable values of the PM_{10} time series.
Overall, our forecasts are pretty close to the actual numbers.
It's also helpful to be able to quantify how accurate our forecasts are; for that, we will utilize the **MSE** (Mean Squared Mistake), which sums up our forecasts' average error. Then, we square the result for each anticipated value based on its distance from the exact value. When computing the overall mean, the data must be squared so that positive and negative differences do not cancel each other out. For this case, `The Mean Squared Error of our forecasts is 0.01`. It's close to zero, so it's shallow. An MSE of 0 indicates that the estimator is perfectly accurate in predicting parameter measurements, an ideal scenario but not always attainable.

Let's try using only data from the time series up to a particular point and then gen-

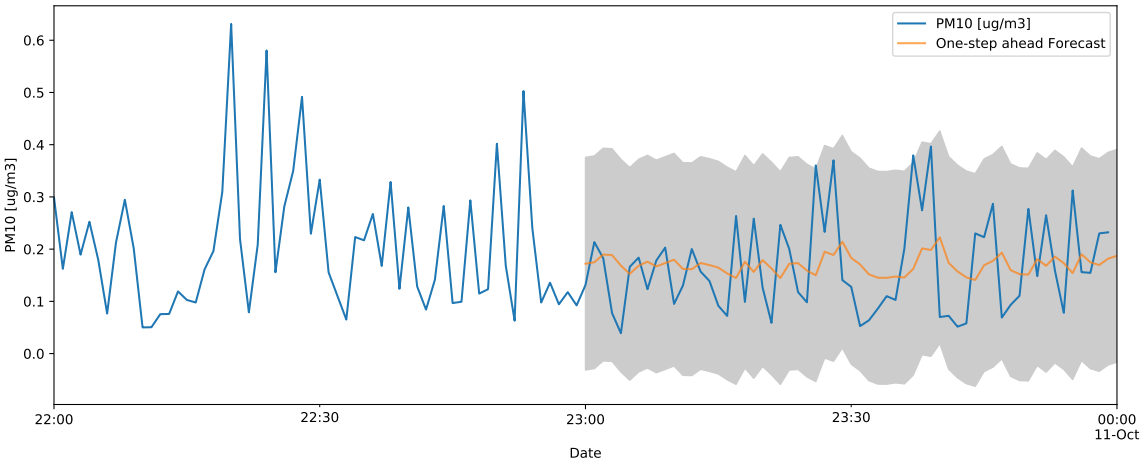


Figure 4.41: ARIMA forecasting one-step-ahead

erating forecasts using values from previously projected time points. We start computing the dynamic forecasts and confidence intervals from `2020-10-10 23:00:00`, the result of forecasting are in the Figure [4.42].

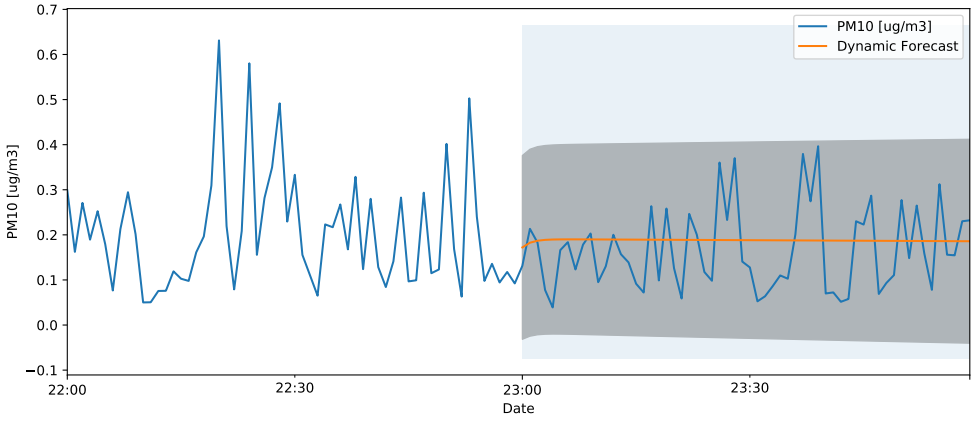


Figure 4.42: ARIMA generating forecast from particular point

Once again, we quantify the predictive performance of our forecasts by computing the MSE, `The Mean Squared Error of our forecasts is 0.01`. Often a flat forecast is, in fact, better than non-trivial ARIMA. [HA21]
We have a computation problem since you have a lag of 1440 minutes. If we probably use the maximum likelihood estimator (conditional) to initialize, it has to solve a 1440 linear system plus multiplying more than 1440 matrices by each other.
So it's pretty challenging to use ARIMA here. The best way would be to go with ARIMAX to create a dummy variable every 1440, 0 otherwise.
In the final step of this Subsection, we forecast future unseen values. Figure [4.43] shows this prediction for several steps.
Likely, the basic ARIMA model is too "blunt" for such high-frequency data. For exam-

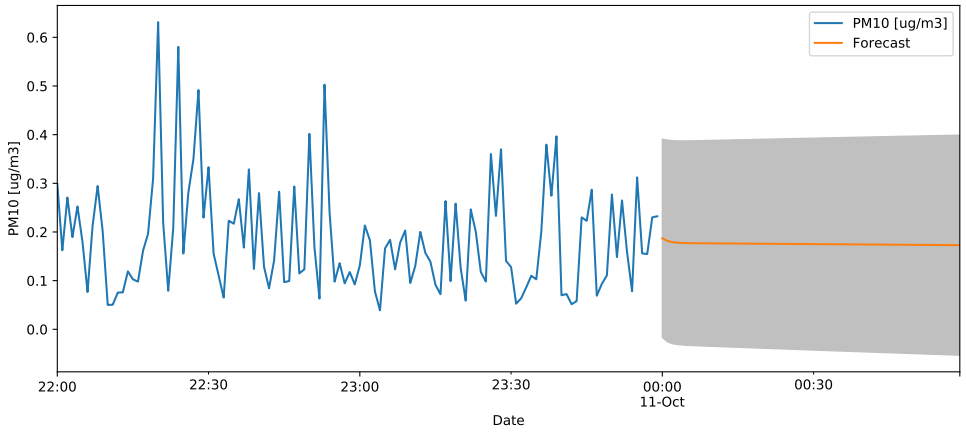


Figure 4.43: ARIMA forecasting future

ple, there are possibilities of very long seasonal patterns (e.g. weekly or daily data) that are not very easy under the basic model. In addition, there can be calendar effects that the model must take into account. The high-frequency data make the autocorrelations challenging to interpret and make it appear that large numbers of lags are required to fit the model well. Unfortunately, the best way to proceed can probably only be determined by looking at and thinking about the data.

4.4.2 Deep Learning

In this part, we will build different models explained previously in the methods section of Deep learning [3.6.2], including Linear, Dense, Convolutional and Recurrent Neural Networks (CNNs and RNNs), particularly LSTM. Then, to apply the models, we will divide our prediction into predicting one pollution value (Forecast for a single pollution value) and predicting several pollution values considering previous values (Forecast multiple pollution values).

Before applying the models, our data sets must be inspected, cleaned and preprocessed (subtract the mean and divide by the standard deviation of each feature). The normalisation of data generally accelerates learning and leads to faster convergence. And before proceeding with this normalisation, we have split our datasets in 70% for training, 20% for validation and 10% for testing. We use the training set to train neural networks, the validation set for early stopping (stopping training when the error in the validation data set increases, as this is a sign of overfitting in the training data set) and the test set to provide an impartial evaluation of a final model fit on the training data set.

Splitting the data like this ensures that the validation/testing results are more realistic because they base on data collected after the model has trained. Normalisation applies to the training set after splitting the data set so that the models do not have access to the values of the validation and test sets. We see our standardised training

set in Table [4.14], pollution data, train schedules and crowding level.

PM10 [ug/m3]	crowding_information	outbound_times	inbound_times
0.131921	0.028991	0.0	0.0
0.208473	0.028991	1.0	0.0
0.090673	0.028991	0.0	0.0
0.243032	0.028991	0.0	0.0
0.162393	0.028991	0.0	0.0

Table 4.14: Train dataset

Forecast for a single pollution value

First of all, it is a good idea to have a performance **Baseline** before developing a trainable model so that you can compare it to later, more advanced models.

The task is to predict pollution one minute in the future, given the current value of all our datasets (train schedules and crowding level). The current values include the current pollution data.

Start with a model that returns the current pollution level as the prediction; "No change" seems like a good choice. Because pollution fluctuates slowly, this is a reasonable baseline. Naturally, if you predict pollution later in the future, this baseline will perform less well. When you plot the predictions of the baseline model (Figure [4.44]), you can see that it is just the labels pushed right by one minute:

- The blue "Input" line shows the input pollution data at each time minute.
- The green "Labels" dots show the target prediction value. These dots show at the prediction time, not the input time.
- The red "Predictions" stars are the model's prediction's for each output time step. If the model were predicting perfectly, the predictions would land directly on the "labels".

The next model we apply is the **Linear Model**. We remember that it just insert linear transformation between the input and output from the Linear Model description in the previous Deep learning Section [3.6.2]. In Figure [4.45] we show the plot of an example prediction. Note how the forecast is better than just returning the successive input pollution in many cases, but in some instances, it is worse.
Linear models have the advantage of being generally straightforward to interpret. For example, we can see the weights allocated to each input by pulling out the layer's importance (weights) in Figure [4.46]. We see that in this case, train timetables have been almost irrelevant.
It's worth assessing the performance of more profound, more powerful single input step models before applying models that operate on multiple time-steps. The next

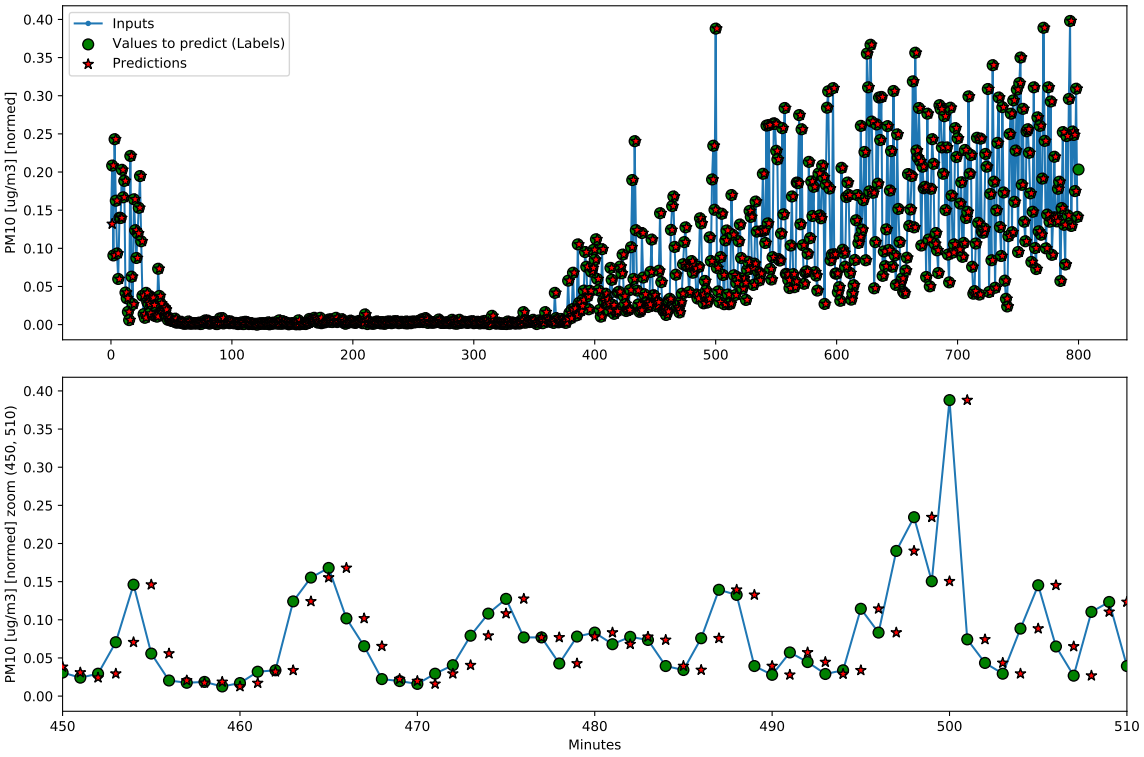


Figure 4.44: Baseline model

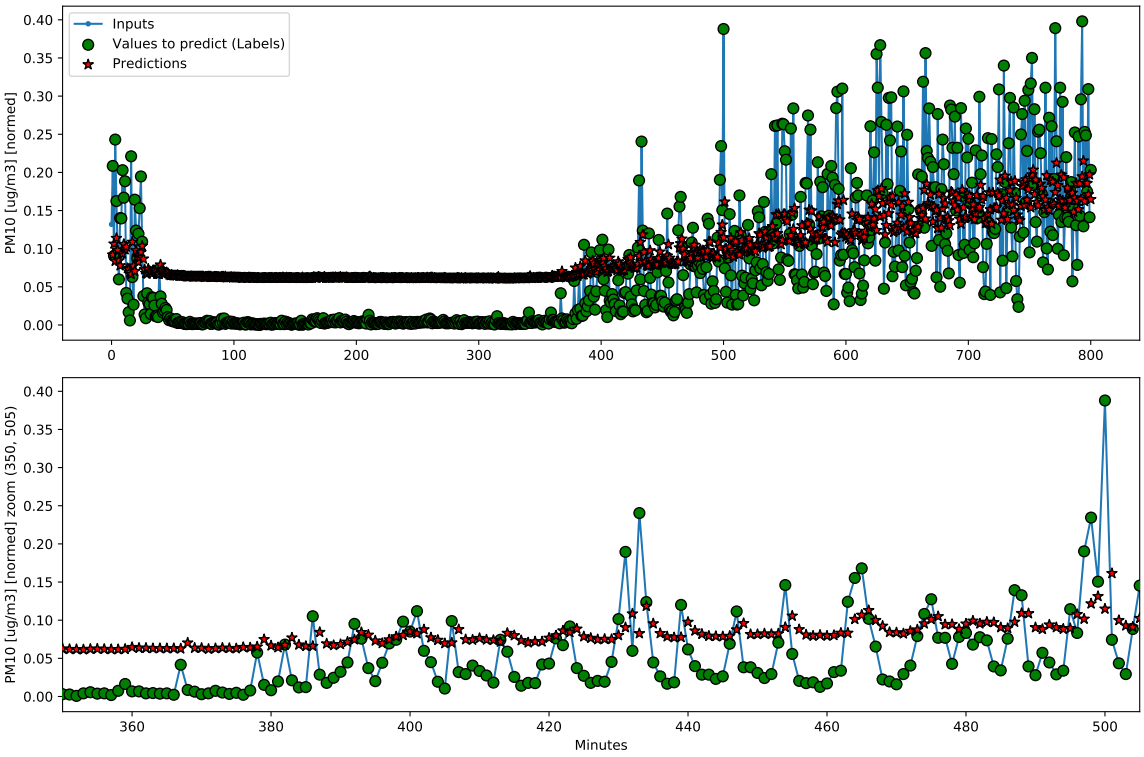


Figure 4.45: Linear model

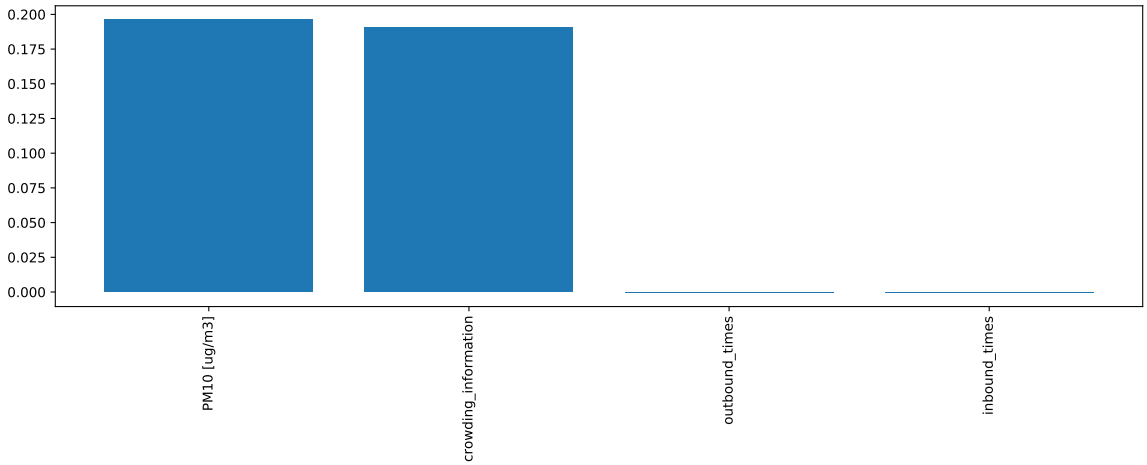


Figure 4.46: Linear weights

model is similar to the previous linear model. However, it stacks many **Dense layers** from Chapter [3.6.2], with ReLU activation functions (Figure [3.13]) between the input and the output. To produce a single output, the model will need many time steps as input. We will create a window that will make batches of the 800 pollution input data and 1 minute of the label. See the result in the Figure [4.47].

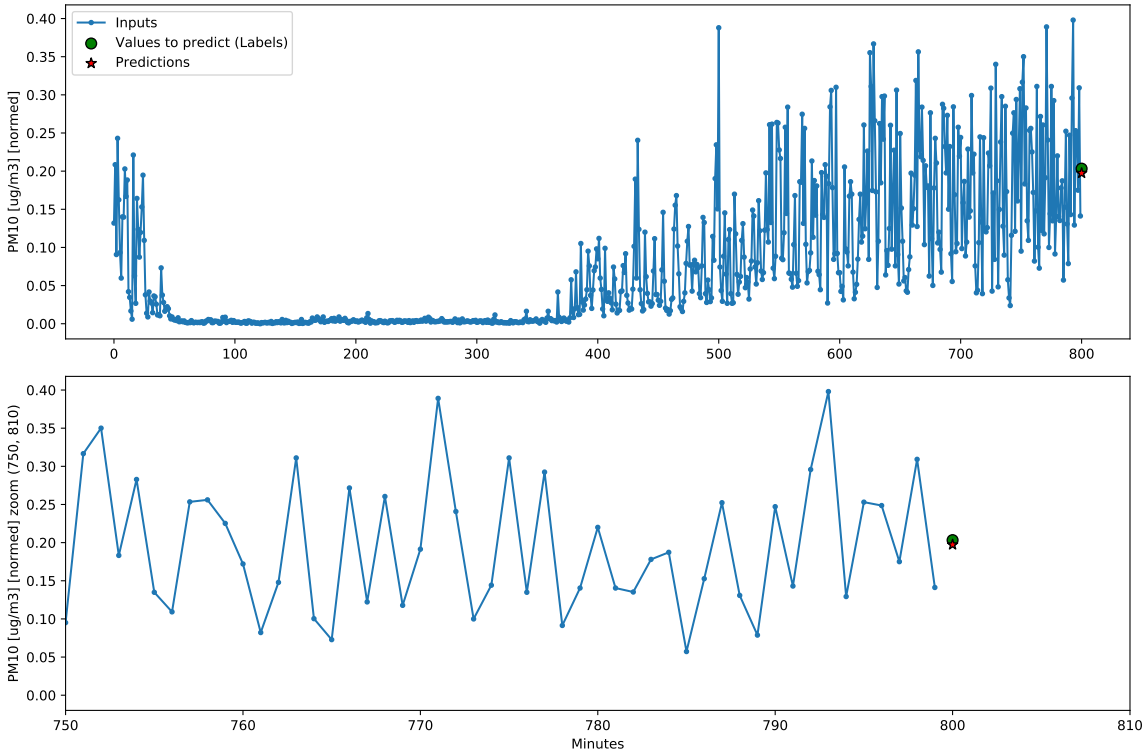


Figure 4.47: Dense model many layers

The main drawback of this approach is that the model can only be run on input windows of this particular size, but the following convolutional models fix this problem.

For the **Convolution Neural Network (CNN)** described in Section [3.6.2], each prediction is also fed by a convolution layer, which takes several time steps as input. We rewrite the previous dense model but adding this convolution layer. Now we plot the model prediction in Figure [4.48]. Note the previous ∼800 input minutes before the first prediction. All the subsequent predictions are based on these preceding minutes.

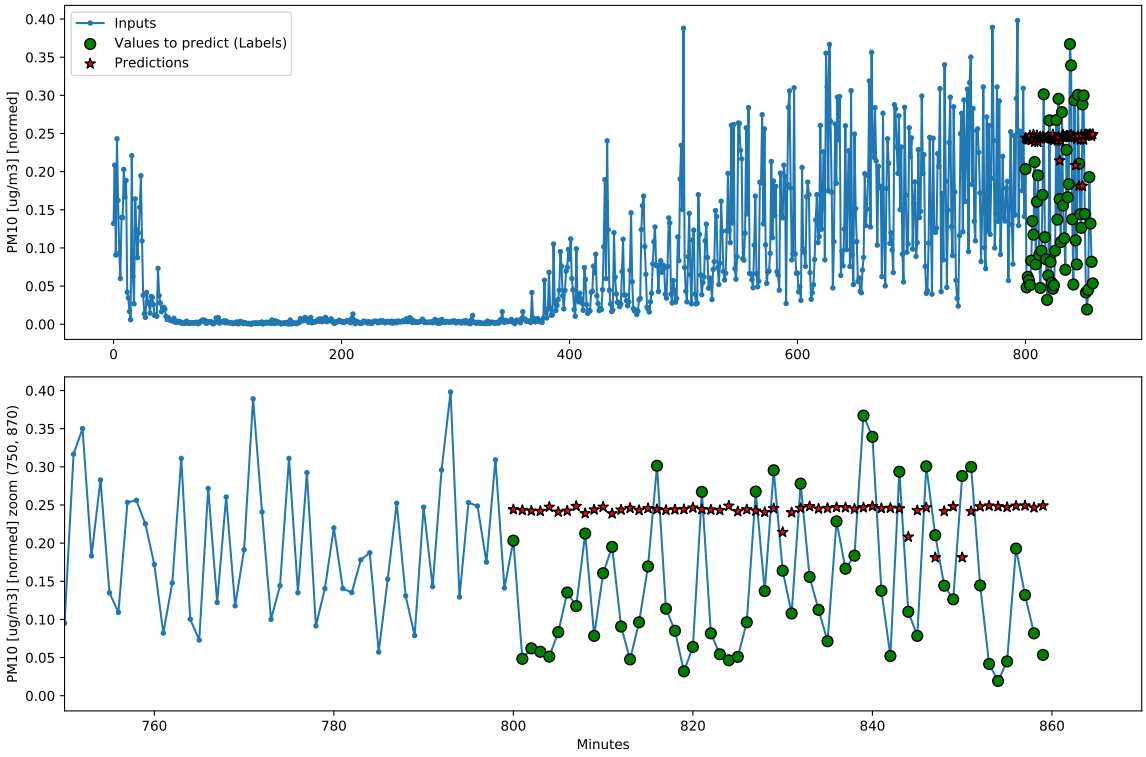


Figure 4.48: Convolutional neural network model

Again, the distinction between the convolutional and dense models is that the convolutional model can be used with any length of the input.
Finally, the **Recurrent Neural Network (RNN)** described in Section [3.6.2] is a sort of neural network that works well with time series data. RNNs go through a time series step by step, keeping an internal state from one stage to the next. We will use an RNN layer called Long Short Term Memory (LSTM), where the layer returns output for each input, which helps stack RNN layers and train a model on many timesteps at the same time. For example, in Figure [4.49], we show the model trained on 24h of data at a time.
We are finally checking the Mean Absolute Error (MAE)[3] performance in the validation and test sets. This is shown in Figure [4.50], with the exact values in Listing [4.13].

[3]MAE measures the average magnitude of the errors in a set of predictions without considering their direction. It is the average over the test/validation sample of the absolute differences between forecast and actual observation where all individual differences have equal weight.

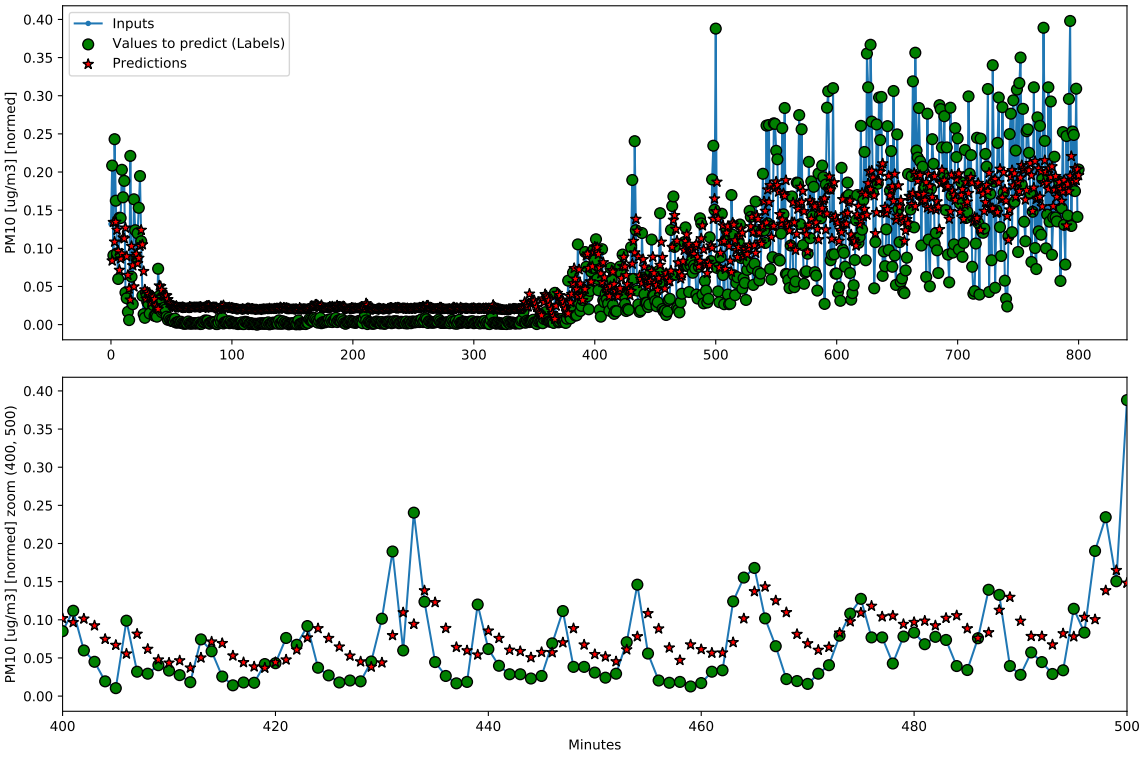


Figure 4.49: LSTM neural network model

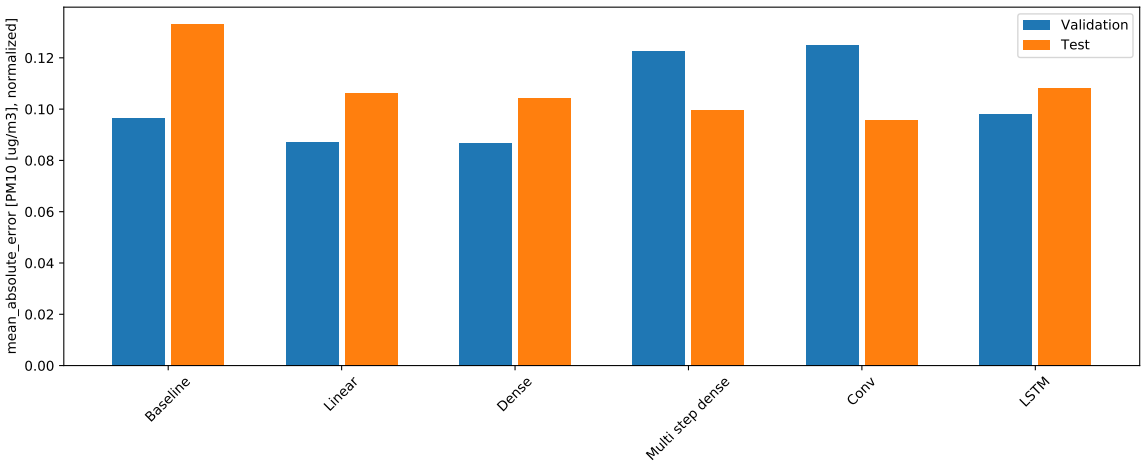


Figure 4.50: Performance by MAE single models

```
Baseline     : 0.1331
Linear       : 0.1062
Dense        : 0.1044
Multi step dense: 0.0998
Conv         : 0.0958
```

```
6 LSTM          : 0.1082
```

Listing 4.13: MAE in test set exact values

Forecast multiple pollution values

We used the prior models to predict one minute into the future in single time steps. We will now look at how to extend these models to predict multiple future values. First, the model must be taught to forecast a wide range of future discounts. Unlike a single value prediction model, which predicts only one future value, a multiple values model predicts a sequence of future pollution values.

The model can approach this in two ways:

- One shot predictions are those that predict a sequence of values of the time series at once.
- Auto regressive predictions are when the model makes only single value predictions, and the output is supplied back into the model as input.

In this case, all of the models will predict all of the features for all output time steps. We use minute samples to train the multi-step model. For example, the models will learn to forecast multiple minutes (approx 1 hour) of the future from almost a day of values in the past.
Figure [4.51] is a window example of what we want to predict from the dataset.
Again, as we did when forecasting a single value, we build a **Baseline** model to have as a reference. In the Figure [4.52], we visualise the Baseline model with the labels of the values we want to predict and the baseline prediction values. In this case, we repeat the last input minute for the required number of values we wish to forecast as a baseline prediction.

One-Shot models A one-shot model (the model predicts the entire sequence in a single step) is one complex approach to our multiple forecasting problem. Let's look at the above models in this way.
Although a simple **Linear model** based on the most recent input time step outperforms both baselines, it is underpowered. With a linear projection, the model must forecast 1 hour from a single input time step. Thus, it can only capture a low-dimensional slice of behaviour, which is determined by the time of day and year (see Figure [4.53]). The **linear model** gains greater power when dense layers attached between the input and output for the **Dense model**; see Figure [4.54].
In this situation, the **Convolutional Neural Network** model produces predictions based on fixed previous values data, resulting in better results than the dense model because it can see how things change over time, (see Figure [4.55]).
If the history of inputs is relevant to the model's predictions, a **Recurrent Neural Network model** can learn to use it.

In the particular case of LSTM, the model only has to create output at the last step

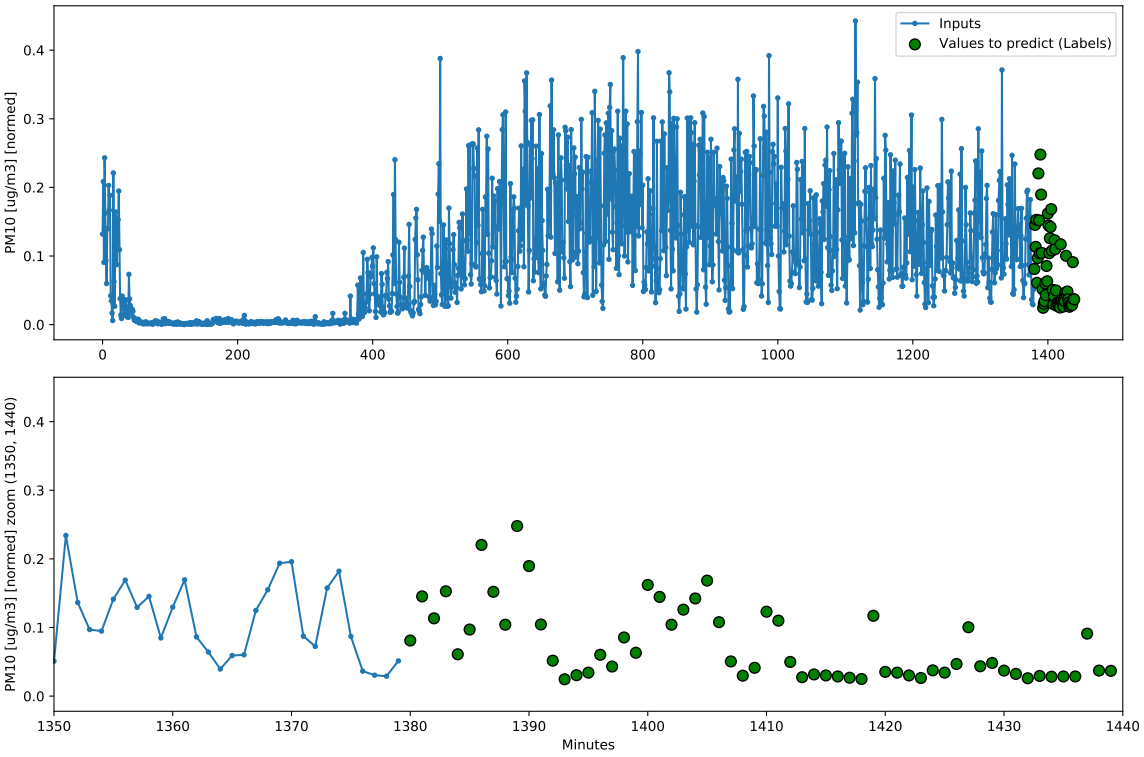


Figure 4.51: Data window of multiple values to predict

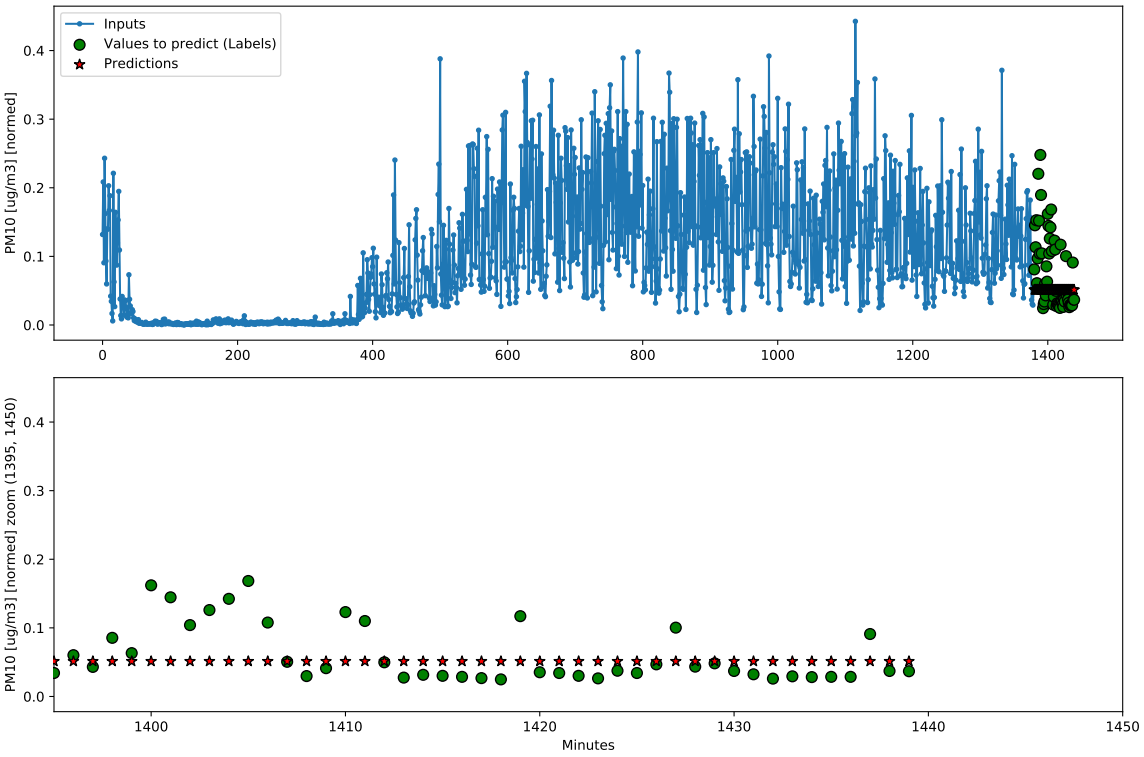


Figure 4.52: Baseline for multiple forecasting

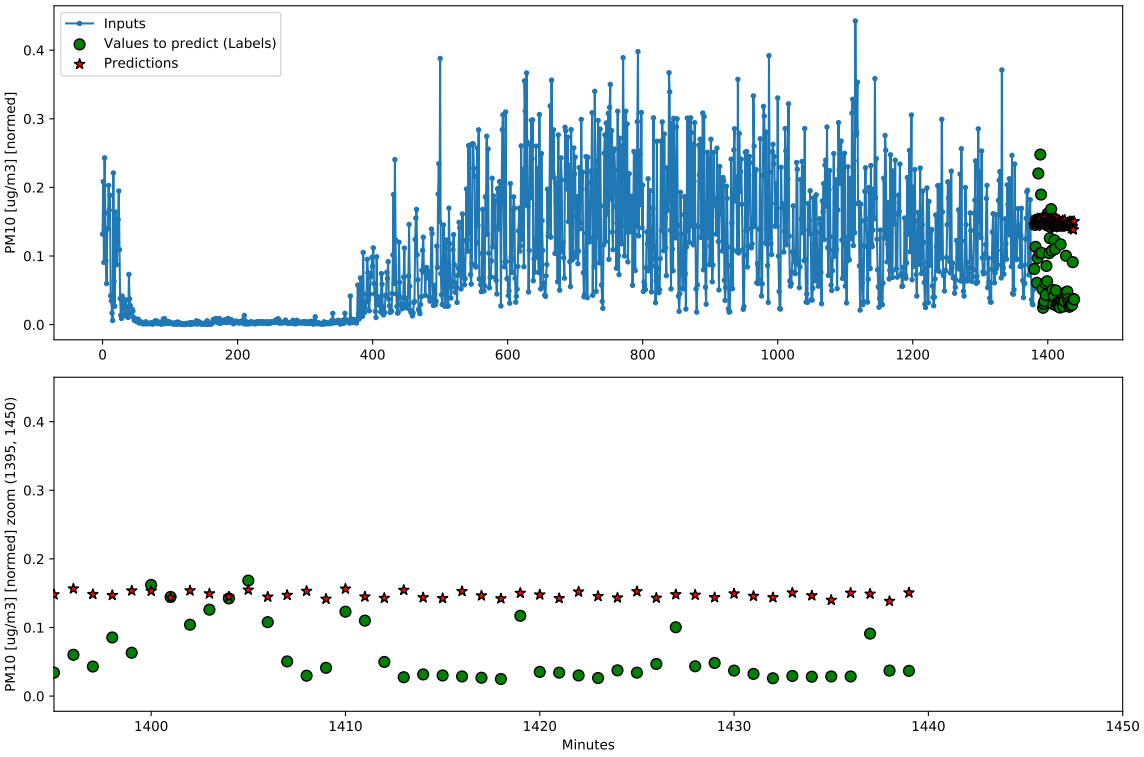


Figure 4.53: Linear model for multiple forecasting

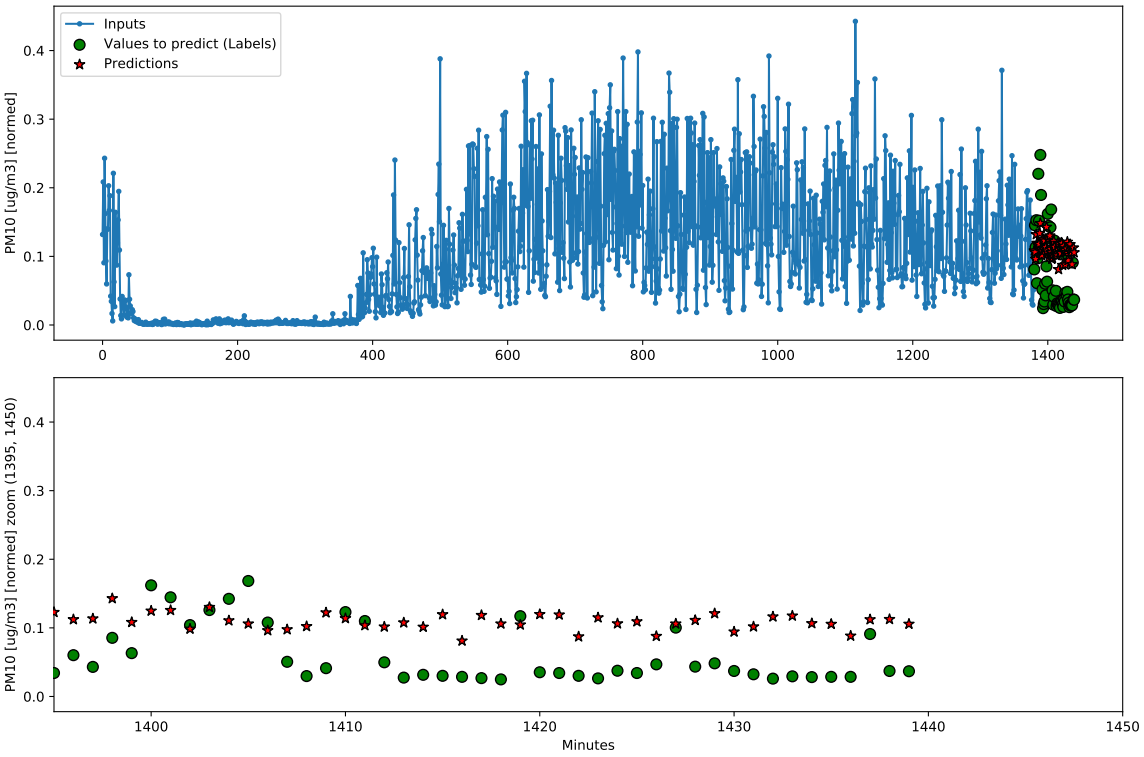


Figure 4.54: Dense model for multiple forecasting

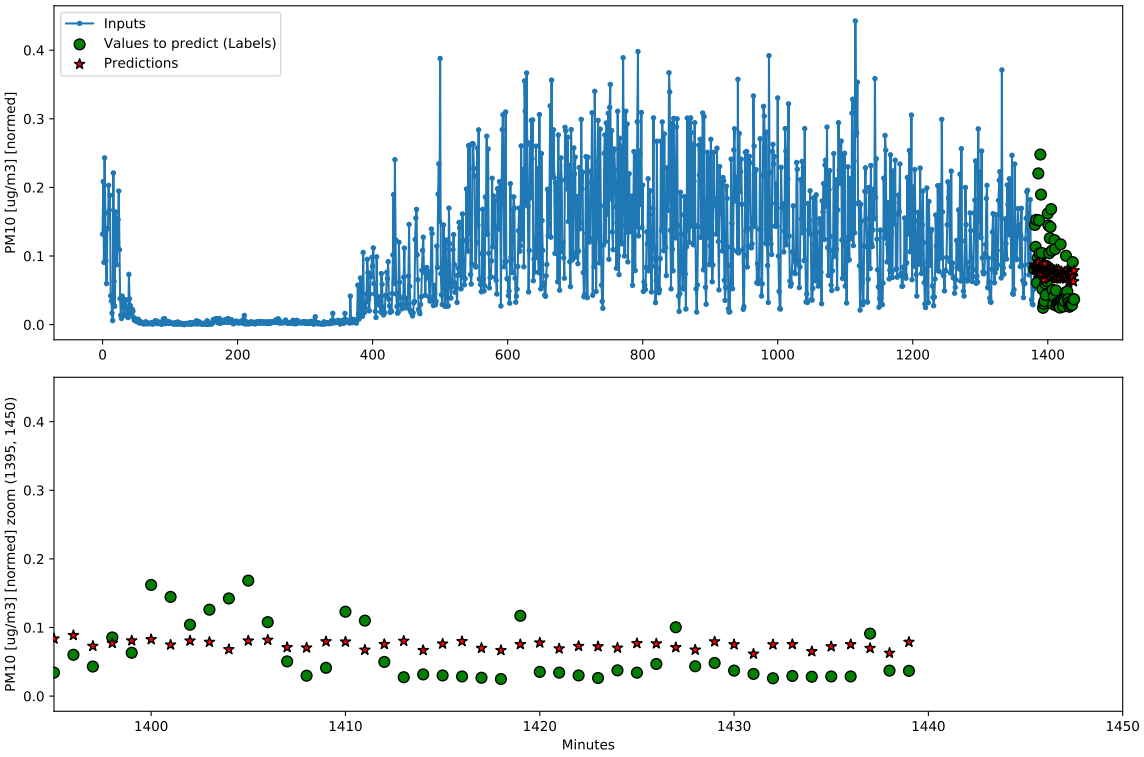


Figure 4.55: Convolutional model for multiple forecasting

in this one-shot format. The outcome that has to make is because the layer only delivers the final timestep's result, giving the model time to warm up its internal state before producing a single prediction.

The model will accumulate an internal state for one day of minutes before making a single prediction for the next hour.

Autoregressive model All of the models shown above forecast the whole output sequence in one step.

In some circumstances, breaking down this forecast into individual time steps may be beneficial to the model. The output of each model can then be fed back into itself at each phase, allowing for predictions based on the preceding one.

This type of model has the distinct advantage of being able to provide an output of varied lengths.

We can run any of the models previously used to predict one or multiple pollution values in an **Autoregressive** feedback loop. Still, we will develop one that is going to be trained to do so.

We create an autoregressive RNN model (any model designed to output a single

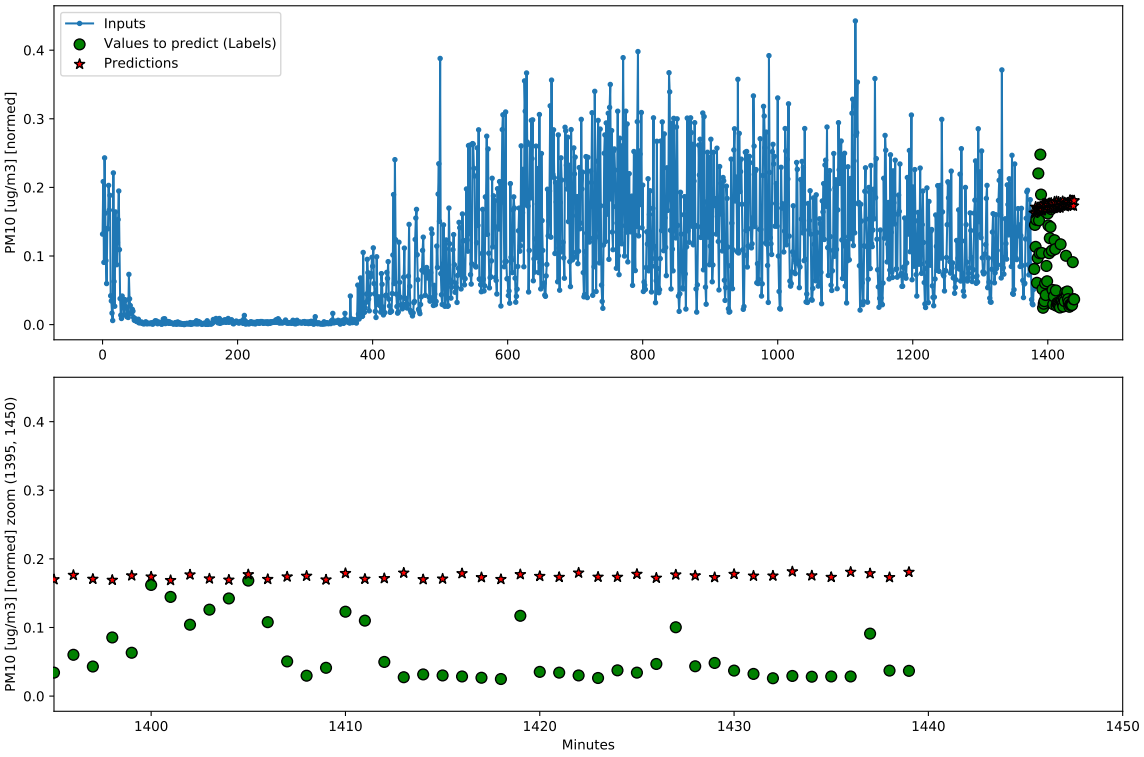


Figure 4.56: LSTM model for multiple forecasting

prediction value could apply this building pattern).

The model will have the same fundamental structure as the LSTM model with a single prediction value. However, because the model must manage the inputs for each step manually in this situation, it uses the single time step interface directly for the lower level. A warmup method is required for this model to initialize its internal state based on the inputs. This state will capture the essential parts of the input history once the model has been trained. The method returns a single value prediction as well as the LSTM's internal state. With the state of the RNN and a starting prediction, we iterate the model, giving the predictions back as input at each step. See the result in the Figure [4.57].
Finally, we show in Figure [4.58] a performance analysis of our multiple values forecast. (`Last` means the last baseline, the baseline for the multiple forecasting)

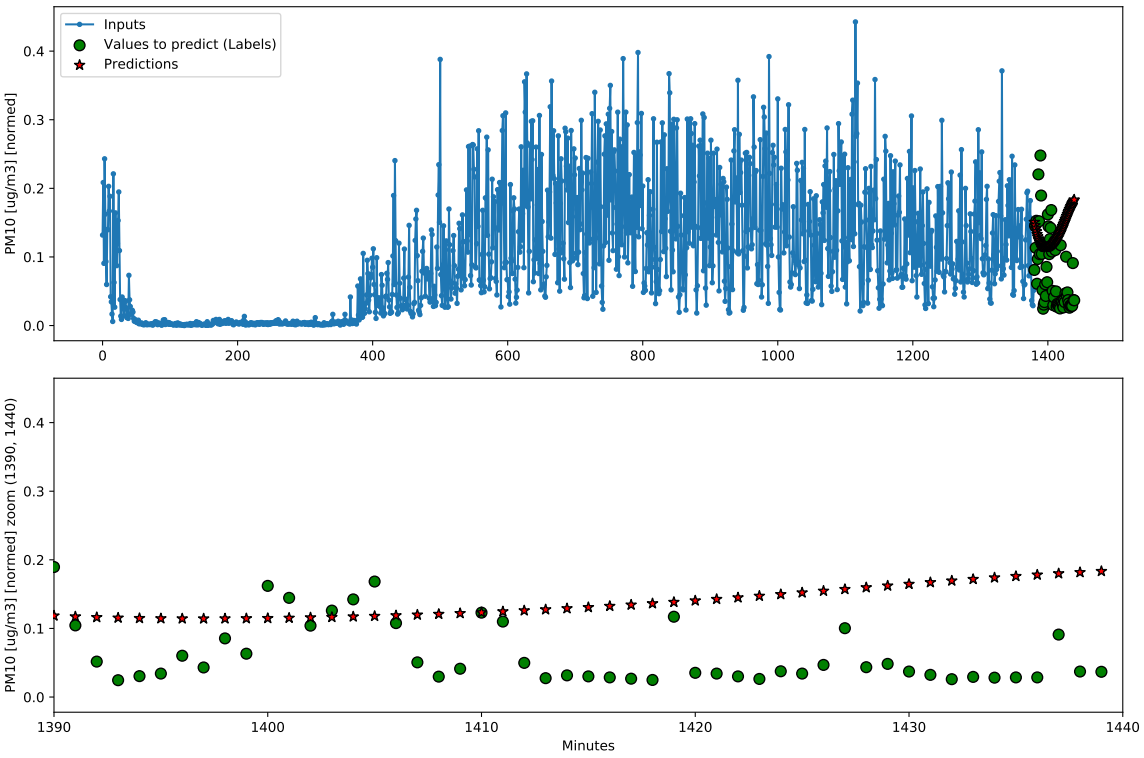


Figure 4.57: Regressive LSTM model for multiple forecasting

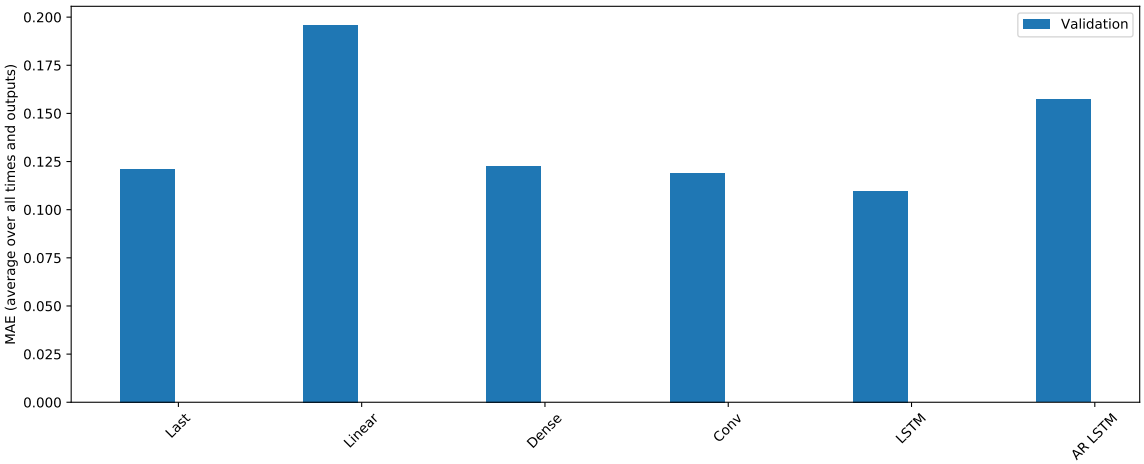


Figure 4.58: Multiple forecasting performance

Chapter 5

Conclusions and futher work

This chapter includes all the conclusions we have drawn from the applied experiments, both professionally and academically. We will see if we have achieved the objectives presented at the beginning of the work (even if only partially), and we will see straightforward ways of further development to continue the work.

We recall that we have worked with a dataset provided generously by some of the researchers from the studies discussed in Chapter [2] (Background). The pollution dataset consisted of PM_{10} particulate matter contamination data from the South Kensington station in the Piccadilly line in the London underground. This dataset contains one week of data in minute values, from 4 October 2020 (Sunday) to 11 October 2020 (Sunday).

5.1 Time and resources devoted to the project

This project had an expected extension of 18 ECTS. As each ECTS credit amounts to 25 hours of work, the project should involve no less than 450 hours of work which we have indeed fulfilled. In the Figure, we present Table [5.1], where we show a Gantt chart[1] of the project's development, where the months where each part has been developed are shown in orange. Although, in general, we have not distinguished between research and development time, learning by doing has been our primary methodology. Undoubtedly, the part that we have spent the most time on has been the TfL API due to creating specific methods to build the time series.

In the Github repository, where all the technical content of the memory is stored, it is also possible to visualise the ongoing work through the commits made. In total, 81 commits [2] have been made. Figure [5.2]. TODO: Update this

[1] type of bar chart that illustrates a project schedule

[2] The definition of commit can be found in the following Glossary `https://docs.github.com/en/get-started/quickstart/github-glossary`

Figure 5.1: Gantt chart

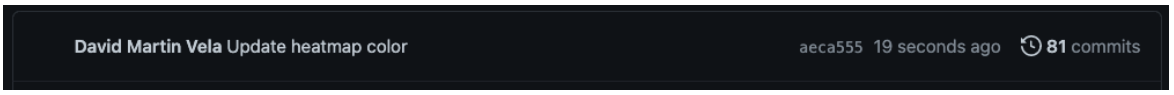


Figure 5.2: Github commits

5.2 TfL API

Initially, we started from wanting to compare correlating pollution peaks in train schedule and crowding levels. However, it seems an apparent relationship to think that the train timetable and the number of people in the station when the train passes has anything to do with the pollution value. To make this comparison, we first had to obtain train timetables and people at the station. To get these new datasets for comparative purposes (although later on, we have used them for prediction purposes), we used the Transport for London API (TfL API), a tool that we have seen has many potentials. We have designed an algorithm to manipulate the API's endpoints' responses to generate these datasets of train times and people in the station.

Overall, we have managed to build the train schedules and crowding level datasets. Still, with the handicap that we already mentioned when we constructed the datasets, the South Kensington station, which corresponds to the pollution data, is unfortunately closed and under construction. Moreover, one of the shortcomings of the TfL API is that there is no way to get historical data (it would have been easier if we could have known the train timetable when we collected the data). The API only returns factual time information. We then assumed that the train schedule when we collected the data was the same as it is now, and since the station is closed, we approximated when the train passed through South Kensington station by looking at how long it takes from the previous station to the next. Similar reasoning applies to the crowding levels of the station. Thus, we obtain the dataset in real time and condition to the station closure (remember that these data are from WiFi connections in the station, even if no trains pass through the station, it is still open for people).

As a comment on the API, we have already seen that it is a tool with great potential. For example, it would be easy to build an application to consult real time train arrivals. However, the most significant disadvantage we have experienced would be

the documentation and the format of the answers. In many cases, it is pretty confusing, and I think that a lot of information is irrelevant or simply not practical.

For example, we are referring to the timetable endpoint (Methods Section [3.2]), we see that according to the description of the endpoint in the API documentation, In the endpoint request could indicate the destination station, but in practice, the response ignored this value. Similarly, with the crowding endpoint (Methods Section [3.2]), there are some values in the response that it is unclear what they are. Even if we ask this kind of thing in the API forum, people don't have these things. Here we show an example where we ask something regarding a parameter of the crowding endpoint response in the TfL forum regarding this fact [`https://techforum.tfl.gov.uk/t/get-crowding-information/1766`]. Again, when generating the train timetable, we have taken an approximation of arrival time. For example, if we want to know when the train passes South Kensington, if from the previous station to the next station the train takes 10 minutes, we have assumed that it will take 5 minutes to South Kensington. Unfortunately, take time like this is generally not true, as the distance from South Kensington to the previous or next station is not the same. An improvement could be to approximate by interpolation.

Also pending is the possibility of publishing a Python package with the various methods and algorithms we have created to generate the series of train timetables and crowding levels without timebands for a specific station.

5.3 Time series analysis

After generating these datasets, we have seen different techniques to analyse time series, empirically, visually, or through statistical tests. This time series analysis provides us with properties that allow us to understand our dataset better and, in some cases, facilitate the creation of predictive models, as is the case of stationarity, Experiments Section [4.1]. We have tried to do a complete analysis and visualisation, but we could have added more. For example, we have seen how to differentiate a time series so that it is stationary. In the same way, we could have seen how to detrending a time series and even deseasonalise it. We could also have seen some statistical tests of the same to see if there is seasonability. It would also have been advantageous (and even more so in our particular case) to know if one time series helps forecast another. It could have prevented us from using unnecessary series and information in our prediction. Again, a statistical test could have done this, as the Granger Causality test.

5.4 Time series comparison

Talking about the time series comparison, we have seen a direct correlation between all our datasets, taking minutes of delay. We did this because it makes sense to think that pollution displacement is not instantaneous. For example, recalling Figure

heatmap [4.29], the relationship between our pollution data and station crowding levels is evident with more or more minor delays. The more delay we consider in the crowding levels and the more the pollution data time is advanced, the more the correlation increases. However, this increase is negligible because it is a minimal amount.

Overall, the Pearson correlation is an excellent place to start because it makes computing global and local synchronisation relatively simple. However, this still doesn't give us any information on signal dynamics, such as which signal appears first. For this reason, we decided to use the Dynamic Time Warping (DTW) algorithm to compare time series that may vary in speed and may be of different time series. As we have seen in the section on experiments where we applied the DTW algorithm [4.3.2], we have used two libraries with different algorithm implementations, one more optimised and one with more visual content.

We have seen that the performance is also costly, even for the optimised one. In the methods Section regarding DTW [3.5.1], we have already seen that the efficiency was the product of the length of both series compared. We, therefore, had to simplify our experimental interval and focus only on two days.

Regarding the results of applying the algorithm, as we suspected from studying the correlation between our pollution data and crowding levels, the minimum path showed a convex line. So earlier pollution data is matched with the synchrony of later crowding levels data. On the other hand, the result of comparing pollution data between train schedules in both inbound and outbound directions is quite similar and much worse than reaching crowding levels.

Excellent addition to this comparison study could have been to vary the length of the series, or even compare the same time series in different periods, for example, in our pollution data series, perhaps reaching a Monday with a Friday. Finally, we may measure instantaneous phase synchrony if we have time series data that we believe has oscillating properties. This measure, like DTW, measures moment-to-moment synchrony between two signals, allowing us to assess if two time series are in phase (moving up and down together) or out of phase, which could have been helpful to compare peaks of pollution with mounts of people.

5.5 ARIMA model

Commenting on our ARIMA forecasting model Experiments Section [4.4.1], our prediction is good if we use historical data to predict future data, but concerning forecasting future values, our prediction is a horizontal line. This result is not at all surprising.

Because in our case, we have high-frequency data, and the basic ARIMA will not know the difference between a weekday and a weekend, so it probably won't give

you valuable results. To create a model correctly, we should consider the periodicity, seven days 10800 minutes, 1440 minutes each day. We will have a computation problem since we have lags of 1440 minutes. So we probably use maximum likelihood estimator (conditional) to initialize it has to solve a 1440 linear system plus multiplying more than 1440 matrices by each other) So it's pretty challenging to use ARIMA here.

There is something called SARIMA (Seasonal Autoregressive Integrated Moving Average). SARIMA would be helpful if we were dealing with daily data points, but it is not very suitable for minute data either. So, in my opinion, the best way would be to go with ARIMA to create a dummy variable that is one every 1440, 0 otherwise.

In forecasting, you very often find that elementary methods, like the overall mean, the naive random walk (last observation used as a prediction), seasonal random walk and Single exponential smoothing outperform more complex methods. Another way of approaching our problem might have been to start with more straightforward methods and evaluate benchmarks against the results. From [HA21], "Some forecasting methods are straightforward and surprisingly effective".

Another good addition would have been to use the R programming language with the `forecast` module, which provides a method called `auto.arima` that returns the best ARIMA model according to either AIC, AICc or BIC value avoiding the work of searching empirically or theoretically for the order of the model. [`https://www.rdocumentation.org/packages/forecast/versions/8.15/topics/auto.arima`].

5.6 Deep learning

Finally, the models have provided us with good results concerning the latest experiments with deep learning models. Because of the complicated data models, training is exceedingly costly. We have achieved results but at the cost of a lack of interpretability and efficiency except for some models (e.g. the Linear model, where we can see the weights, Figure [4.46]). Recalling the Figure for the MAE of the validation and training set forecasting a single value [4.50] and Listing [4.13], simpler models such as Linear or Dense provide better results at both the validation set and test set level. At the same time, in particular, convolutional neural networks perform better in the test set but not in the validation set. Although visually, the model does not seem to work very well in our dataset. Figure [4.48] shows an example of a prediction, where the result is practically a straight line compared to other models. The chosen data windowing model for developing the models is quite successful in our case, as predicting a value or several values in the future is trivial (add more labels to forecasting). On the other hand, the benefits from a dense model to convolutional and recurrent models are negligible (if at all), whereas the autoregressive model performed significantly worse. So these more complex ways might not be worth it for this topic, but there's no way to know unless you try, and these models could be helpful for this situation.

Unfortunately, we could not complete the objective to perform clustering to find stations that behave similarly, as we mainly had pollution data from only one station, South Kensington. It would be interesting to have at least two weeks of data to compare days of the week and even to predict. The main disadvantage of deep learning is that it requires a large amount of data to work. However, even if we have two weeks of pollution data from different stations, it will give us enough freedom to explore and investigate pollution prediction. In the absence of data, we could also consider generating another week of data manually by copying the one we already have and introducing noise or some random component so that they are not the same. We were also unable to complete the objective of creating a website that applied the most optimal model for predicting pollution. Mainly due to the above argument, we only have one week of pollution data from one station. Therefore, our prediction was quite limited to that. The approach would be to use the versatility of deep learning model inputs to learn with more stations and implement the model on a web page. It can provide results such as predicting which station will have less pollution value at a specific time.

The project has been exciting. Learning and using an API is something that will always be useful in the development world. In addition to learning to use the API, we have created different datasets to predict pollution. The problem of predicting pollution is complex and depends on many factors. Most of the current models that try to solve this problem are much more complicated than those we have used in this project and consider many more variables such as wind, humidity and temperature. Nevertheless, as a first contact to apply predictive models to a real problem, it has been a very satisfactory experience.

Bibliography

[Dun64] Olive Jean Dunn. "Multiple Comparisons Using Rank Sums". In: *Technometrics* 6.3 (1964), pp. 241–252. DOI: 10.1080/00401706.1964.10490181.

[SC78] H. Sakoe and S. Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (1978), pp. 43–49. DOI: 10.1109/TASSP.1978.1163055.

[ERS96] Graham Elliott, Thomas J. Rothenberg, and James H. Stock. "Efficient Tests for an Autoregressive Unit Root". In: *Econometrica* 64.4 (1996), pp. 813–836. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/2171846.

[BA03] Burnham and Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer Science and Business Media, 2003.

[MN10] Patrick E. McKight and Julius Najab. "Kruskal-Wallis Test". In: *The Corsini Encyclopedia of Psychology*. American Cancer Society, 2010, pp. 1–1. ISBN: 9780470479216. DOI: https://doi.org/10.1002/9780470479216.corpsy0491.

[Jar11] Carlos M. Jarque. "Jarque-Bera Test". In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 701–702. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_319. URL: https://doi.org/10.1007/978-3-642-04898-2_319.

[NIS12] NIST/SEMATECH. *e-Handbook of Statistical Methods*. https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4481.htm. 2012.

[DR14] Natalia Díaz-Rodríguez. "Handling real-world context awareness, uncertainty and vagueness in real time human activity tracking and recognition with a fuzzy ontology based hybrid method". In: (2014). DOI: 10.3390/s141018131.

[For] *Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says*. 2016. URL: https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Cho17] François Chollet. *Deep Learning with Python*. Manning, 2017.

[Gre+17] Klaus Greff et al. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017), 2222–2232. ISSN: 2162-2388. DOI: 10.1109/tnnls.2016.2582924. URL: http://dx.doi.org/10.1109/TNNLS.2016.2582924.

[RKHZ17] Ioar Rivas, Prashant Kumar, and Alex Hagen-Zanker. "Exposure to air pollutants during commuting in London: Are there inequalities among different socio-economic groups?" In: *Environment International* 101 (2017), pp. 143–157. ISSN: 0160-4120. DOI: 10.1016/j.envint.2017.01.019.

[Riv+17] Ioar Rivas et al. "Determinants of black carbon, particle mass and number concentrations in London transport microenvironments". In: *Atmospheric Environment* 161 (2017), pp. 247–262. ISSN: 1352-2310. DOI: 10.1016/j.atmosenv.2017.05.004.

[Smi+20] J.D. Smith et al. "PM2.5 on the London Underground". In: *Environment International* 134 (2020), p. 105188. ISSN: 0160-4120. DOI: 10.1016/j.envint.2019.105188.

[GFMS21] Huan Min Gan, Senaka Fernando, and Miguel Molina-Solana. "Scalable object detection pipeline for traffic cameras: Application to Tfl JamCams". In: *Expert Systems with Applications* 182 (2021). DOI: 10.1016/j.eswa.2021.115154.

[HA21] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice, 3rd edition, OTexts: Melbourne, Australia. OTexts.com/fpp3.* https://otexts.com/fpp3/. 2021.

[LBCGR21] Pedro Lara-Benítez, Manuel Carranza-García, and José C. Riquelme. "An Experimental Review on Deep Learning Architectures for Time Series Forecasting". In: *International Journal of Neural Systems* 31.03 (2021), p. 2130001. ISSN: 1793-6462. DOI: 10.1142/s0129065721300011. URL: http://dx.doi.org/10.1142/S0129065721300011.

[Lö21] Gábor Lövei. *Writing and Publishing Scientific Papers: A Primer for the Non-English Speaker*. 2021. ISBN: 978-1-80064-091-7. URL: https://www.openbookpublishers.com/product/1272.

[Pm] *Particulate Matter Basics*. 2021. URL: https://www.epa.gov/pm-pollution/particulate-matter-pm-basics.

[Tfl] *TfL Unified API*. 2021. URL: https://tfl.gov.uk/info-for/open-data-users/unified-api.

[Bc] *What is black carbon?* 2021. URL: https://www.ccacoalition.org/en/slcps/black-carbon.