CMP6200 — A3: Dissertation

Traffic Management Within Smart Cities of the Future:

The Case of Traffic Accident Prediction and Prevention

Alexander Samuel Roberts

BSc (Hons) Computer and Data Science

Student No. 21158995

Submitted: July 2024

Word count: 10,843

Supervisor: Emmett Cooper, MSc, PGDip, BSc, FHEA.

BIRMINGHAM CITY University

School of Computing and Digital Technology

Faculty of Computing, Engineering and the Built Environment

Birmingham City University

Abstract

This paper explores how recent advances in Artificial Intelligence and the increased abundance of open-source datasets can be leveraged for helping predict and possibly prevent road traffic accidents within Smart Cities, specifically Greater London as a use case. This also includes investigating how could such processes be automated and made more predictable using Machine Learning Operations (MLOps). Furthermore, this also involves examining Machine Learning and Deep Learning techniques for constructing a Long Short-Term Memory neural network.

This paper concluded that currently there is not enough open-source data available with enough granularity to support precise and reliable accident likelihood prediction.

Acknowledgements

I would like to take this opportunity to express my sincere and utmost gratitude and respect to my project supervisor, Emmett Cooper, for their steadfast support and guidance they have given me throughout this project endeavour. His suggestions and well-versed knowledge gave me new insight and direction with how to develop and shape this project. His unwavering support he has provided me throughout this project has been remarkable and irreplaceable; truly outstanding.

Contents

Glossary

RTA	Road Traffic Accident
TM	Traffic Management
SC	Smart City
ML	ML
DL	Deep Learning
EDA	Exploratory Data Analysis
LSTM	Long Short-Term Memory

List of Tables

List of Figures

1 Introduction

This report will include the project reasoning and objectives, a review of relevant literature, system design methodology and associated implementation, an evaluation of the implementation, and finally the conclusions from the findings and possible future work.

1.1 Background Information

The Smart City (SC) model encompasses a suite of objectives and practices mostly aimed at utilising technology to reduce or eliminate problems the populace of a city face, with common examples including crime; air pollution; health problems; social problems; and traffic management issues (Gassmann, Bohm and Palmie, 2019, p. 3). In essence, the SC model aims at improving the quality of life within cities.

Traffic Management (TM) is an important topic for revolutionising cities and urban areas. This is one of the indispensable objectives of the SC model. This is primarily due to the higher number of road users within a more condensed area — the more road users there are, the more likely traffic issues such as accidents, traffic delays and standstills, and road logistical chaos will occur. This is why it is more important to advance TM within cities rather than more rural areas. TM can be defined as the process of directing vehicles and pedestrians so that they avoid or are less affected by a form of disruption (TBF Traffic, 2023). Therefore, the better the TM plans and procedures, the less likely a disruption will impact road users and pedestrians.

1.2 Problem Definition

Road traffic accidents (RTAs) are one of the most common incidents emergency services and road authorities face today. They often lead to devastating consequences for all parties involved, but can also indirectly affect numerous other people and services. Due to the higher population density within cities, these issues become more apparent and problematic.

This project's origin came about from a discussion between this report's author and supervisor when discussing key issues in road transport and logistics today.

1.3 Scope

The intended scope of this project is to research various aspects and implications of RTAs within SC, and subsequently construct a solution which leverages relevant data, ML and DL techniques to predict the likelihood of accidents using live information. The solution should also include a way to present predictions and relevant data to key stakeholders.

1.3.1 Scope limitations

As indicated, this project coverage will be limited to SC because of two factors:

1. It would not be practical to construct a predictive model on a larger scale (such as that of county or nation-wide level) with the resources at hand in this project.
2. SC collect vastly larger amounts of data, particularly for transport, compared to more rural areas. Performing reliable data analysis and constructing predictive models requires huge amounts of data.

Other limitations include:

- The reliability and quality of data available relating to RTAs, as this project will only utilise open-source data.
- The integration of existing technologies in SCs, as this would immensely increase system complexity.
- Possibly privacy considerations, depending on what data is found and to what extent it can be considered anonymous.

1.4 Rationale

In Greater London in 2022, there were 23,465 reported RTAs, with 102 people tragically being killed; 3,859 people being seriously injured and 23,246 people being slightly injured (Transport for London, 2023). As indicated by the statistics, RTAs even in a well-known and developed city such as Greater London are still frequent, and therefore dangerous. Despite these facts, there has not been a tremendous amount of research regarding RTA

prediction and prevention, with even fewer practical solutions. This report will help expand this field.

Every RTA can cause significant disruption to motorists and businesses, with hours of delays in some cases. From a business perspective, not only is this highly inconvenient for workers trying to get to and from work, but it could also lead to financial losses, or a reputation hit for certain businesses (such as those pertaining to transport and delivery services).

Investigating ways to improve TM procedures through live accident prediction and prevention in cities would likely help develop the SC model further.

1.5 Project Aims and Objectives

The aim of this project is to help develop traffic management and road safety procedures as part of the SC model.

The objectives of this project are as listed below:

1. Produce a literature review that will explore the domain and scope of this project.
2. Collect and analyse opinions on road safety and traffic management within cities from at least five road users by a questionnaire or interview as part of primary research.
    (a) The participants should have substantial experience driving within cities. This data can be used to identify what traffic factors are important from a road user's perspective.
3. Establish requirements that must be fulfilled so that the implementation meets project objectives and necessary functionalities to contribute a higher standard of application quality. These requirements will be decided using results from the literature review in Objective 1.
4. Design and plan the features and what data is to be utilised for a ML model that can evaluate traffic accident probabilities to a high enough accuracy as possible. Moreover, this includes creating pipelines to help automate the model creation and

making predictions processes. The features and data to be used will be determined through the literature review in Objective 1 and researched datasets.

5. Design and plan the features for a web app that uses an application programming interface (API) to illustrate the results from the ML model in Objective 4, and show accident prevention and traffic management recommendations.

6. Implement the ML model, pipelines, web app and web API using the designs, ensuring they adhere to the requirements in Objective 3. The technologies and tools to achieve this will be chosen through further research and prototyping.

7. Evaluate the effectiveness of the ML model, pipelines, web app and web API together (in terms of compatibility) and separately to make sure each aspect meets the requirements set out in Objective 3. Subsequently, draw conclusions and next steps for refining the system further to better meet requirements.

8. Refine the ML model, pipelines, web app and web API using software refinement methodologies, which will be identified through further research and prototyping.

2 Literature Review

2.1 Literature Search Methodology

2.1.1 Themes

The table below illustrates the themes.

Table 2.1: Report themes.

Theme	**Explanation for the necessity of the theme**	**Keywords**
Traffic management	Current and previous traffic management procedures and guidelines can be used selectively and appropriately when considering possible actions that can be taken in response to certain likelihoods of accidents or after an accident has occurred.	Traffic management, traffic disruptions, traffic diversions.
Road safety	It is critical to gain an appreciation of road safety regulations and efforts made to improve safety so that more informed decisions can be made when progressing the project.	Road safety regulations, speed limits, driver awareness, driver environments.
RTA mitigation strategies	Investigating RTA prevention guidelines can be further utilised for the project implementation for providing ideal ways to reduce a RTA likelihood, or possibly how the severity of the accident could be reduced.	RTA severity, accident prevention, accident contributing factors.
Accident likelihood prediction models	The benefits and downsides of previous accident prediction models will likely enable more informed decisions on what approaches should be considered during the implementation stages.	Machine learning, deep learning, neural networks.

2.1.2 Data collection

In 2024, a literature search will be conducted with the aim of gathering research material namely related to traffic management and RTAs. This time period for this will be between 1990-2023.

2.1.2.1 Search terms

Search term to be used in the literature search were derived thematically based on the themes identified: ("road traffic management" OR "road traffic control" OR "road security" OR "road safety management" OR "traffic accident prevention" OR "accident factors" OR "accident prediction models"). Additional filter phrases: ("English" AND "anywhere"). 'anywhere' in this case is about having the database engine gather results which match the search term or derivatives of it from anywhere in the research material, including the title; abstract; or full-body text.

2.1.2.2 Databases to be searched

The databases to be searched are identified to be appropriate for finding research material related to this report's topic and themes:

- Association for Computing Machinary Digital Library (2023)
- Directory for Open Access Journals (2023)
- Institute of Electrical and Electronics Engineers Xplorer (2023)
- Science Direct (Elsevier, 2023a)
- Scopus (Elsevier, 2023b)
- Web of Science (Clarivate, 2023)

2.1.2.3 The searchable databases

The searchable databases that can be used for finding further research material are (Birmingham City University, 2023) Digital Library and Google Scholar (Google, 2023). Both are publicly available for use.

2.1.3 Selection criteria for inclusion of research material in the current review of traffic management and RTAs

Research material which meet the following criteria should be focused on:

- Includes an abstract or introduction.
- Includes relevant studies or empirical evidence for justifications.
- Is not unfairly critical or has ambiguous reasoning.
- Is dated between January 2010 to December 2023, as this is when smart cities started to excel in popularity in part due to Cisco Systems and International Business Machines (IBM) (Augusto, 2021, p. 4).

2.2 Results

Table 2.2: Initial literature search results (without inclusion criteria).

Search phrase	ACMDL	DOAJ (articles)	IEEE Xplorer	Science Direct	Scopus	Web of Science	Google Scholar	BCU Library	Total (including duplicates)
Road traffic management	77	2,158	161	434	456	270	approx. 8,270	379	approx. 12,205
Road traffic control	36	35	199	449	312	349	approx. 11,400	1,068	approx. 13,849
Road security	19	19	34	152	132	81	approx. 3,060	397	approx. 3,894
Road safety management	5	62	37	389	271	192	approx. 5,620	245	approx. 6,821
Traffic accident prevention	10	24	18	113	142	70	approx. 1,940	70	approx. 2,387
Accident factors	13	36	27	527	310	152	approx. 4,650	146	approx. 5,861
Accident prediction	81	113	193	744	884	573	approx. 11,800	666	approx. 15,054

2.2.1 Review

2.2.1.1 Traffic management

Souza et al. (2017) states that in today's world fast mobility is essential, but older traffic infrastructure and a stark rise in the number of active vehicles greatly hinders this initiative. Zagidullin (2017) concurs this; particularly during large scale events, such as sports competitions in cities. To attempt to rectify this, the concept of Intelligent Transport Systems came about, with the main intent of working in conjunction with Smart Cities (Pop and Proștean, 2018).

A more recent study by Jurczenia and Rak (2022) discusses how recent advancements in technology have made technological solutions a far more viable option for managing traffic within cities, as it is impractical to replace current road infrastructure due to the scale, time, and complexity of such a task. However, although Ouallane, Bahnasse et al. (2022) agrees with this initiative to a large extent, they explain how optimal traffic management is dependent on many other factors as well, including local transport and socio-economic aspects. Ouallane, Bahnasse et al. (2022) has more authors compared to Jurczenia and Rak (2022), implying Ouallane, Bahnasse et al. (2022) has a more balanced and informed perspective.

Ouallane, Bakali et al. (2022) explains how there are two major causes of traffic congestion: recurrent and non-recurrent. The former concerns regular traffic that usually occurs in the same place and time. The latter concerns special occurrences which take place without warning, such as road traffic accidents.

2.2.1.2 Road safety

Road safety management is defined by Jähi et al. (2012) as an area where local authorities and governments aim to reduce road crashes and their severity. Road safety has improved globally over recent years due to a variety of reasons and driving factors, including revised research methods into the causes of accidents and (new) government-backed initiatives to reduce the after-effects of road incidents (Wegman et al., 2015). Alfonsi et al. (2016) has similar findings, but additionally recognises that there is not enough support from governments and local authorities to improve road safety significantly.

Alfonsi et al. (2016) describes how the topic area itself still needs further development — which is evidenced particularly through quantifiable statistics. Wegman (2017) expounds traditional road safety, such as educating human behaviour; safer infrastructure; and improved vehicle safety, are still widely considered the centre of discussion for road safety. But argues road safety systems are too reliant on participating actors (e.g., drivers, riders and pedestrians) performing their role properly, which contrasts to other transport systems such as those in the rail and aviation sectors. Despite providing logical arguments to support this, Wegman (2017) does not evidence why the rail and aviation sectors are like the way he describes.

Recent applications of technological advancements have enabled better data collection of road statistics and monitoring (Torbaghan et al., 2022). Furthermore, they discuss how further technology-related solutions require more real-life data. However, Perri and Vaiana (2022) maintain a different focus. They spotlight how certain (minor) changes to road infrastructure can help encourage drivers to make more safe decisions. However, this has not been evidenced sufficiently to prove this would be the case for most drivers.

2.2.1.3 RTA mitigation strategies

Regan, Lee and Victor (2013) explain how an individual driver's behaviour and/or condition can have a significant impact on an accident occurring, such as not being fit to drive or getting distracted while driving. This is agreed by Soehodho (2017) who completed a study about Indonesia traffic conditions and traffic accident factors, but they highlighted that road infrastructure and public transportation can be just as important as a driver's manner of driving. Celesti et al. (2018) is of similar opinion to both sources but takes a more hybrid approach where better road infrastructure through the use of sensors and the Internet of Things can be better leveraged to better inform drivers of hazards and distractions.

In more recent years, Kaneko et al. (2021) echoes the summarised points of Regan, Lee and Victor (2013) but features how on average, more elderly drivers and those with cognitive decline are more prone to making driving mistakes than other age categories. Furthermore, other surveys carried out, such as the government backed survey completed

by Nzuchi, Ngoma and Meshi (2022), illustrate how poor road experience and/or attitude can increase the chances of being involved in a RTA. In addition, another study by Boua, Kouabenan and Belhaj (2022) concluded that drivers who feel a high sense of road risks but a lack of precaution towards these risks make them more prone to RTAs.

There are numerous ways to reduce the likelihood of road crashes, but many of the sources screened in this literature review implied an individual driver's behaviour and experience contribute the most. Hence, the more experience a driver has or the more they comply with good safety practices, the less likely they are to experience an accident due to their own manner of driving.

2.2.1.4 Accident likelihood prediction models

Ren et al. (2017) utilises Deep Learning, specifically the Long-Short Term Memory (LSTM) method, to construct a model that outputs the likelihood of a road traffic accident occurring by providing a risk level output. Yu et al. (2021) discusses how between machine learning and deep learning approaches; deep learning methods have gained more attention in recent years due to their ability to overcome some traditional machine learning limitations. Yu et al. (2021) utilises a Graph Convolutional Network (another type of neural network) for their experiment. Lin et al. (2021) supports this and proposes the use of other Deep Neural Networks (DNNs) such as Convolutional Neural Networks (CNNs) for predicting accidents and their severity. Although Yu et al. (2021) is a more recent source, their experiment trialled by them was related to railway data in the United States, compared to Ren et al. (2017) which used road traffic data in Beijing, China. Therefore, the technique proposed by Ren et al. (2017) could be considered more transferable for investigating other road traffic information. Tian and S. Zhang (2022) highlight the various considerations that would be needed when considering how to plan out the model-making process, mainly for deep learning, especially some pre-processing steps.

Other researchers have contrastingly used machine learning compared to deep neural networks. Monsefi et al. (2023) proposes a supervised classification multimodel architecture to be used for a multi-source dataset, where parts of the dataset are encoded then aggregated and subsequently used to predict the likelihood of an accident occurring for

a specific area. Ribeiro, Nicolau and Santos (2023) also utilise machine learning. But neither of the sources explicitly describe why machine learning is used compared to deep learning — potentially due to them being conference proceedings. Ahmed et al. (2023) clarifies the use of machine learning as it is more explainable compared to neural networks; the processes of neural networks are much more difficult to explain and evaluate due to the nature of how they work.

2.2.1.5 Evaluation metrics

There are different evaluation metrics depending on the (machine learning or deep learning) approach taken. Mohanta et al. (2022) for example suggests common evaluation metrics for classification problems are accuracy score, sensitivity, specificity, precision and F1-score. Furthermore, Torre et al. (2019) discusses some example evaluation metrics for determining the goodness of fit of solutions to regression problems include R-squared coefficient of determination (R2), root mean squared error (RMSE) and Pearson's Chi-squared test. Machine learning and deep learning share a number of evaluation metrics, however, there are some exclusive deep learning techniques such as cross entropy and cross-k function (Yuan, Zhou and T. Yang, 2018). Typically, good evaluation metrics results indicate an accurate and likely representative trained model (Liang et al., 2018). Kanakala, Mohan and Reddy (2023) agrees with this view and states that using evaluation metrics are a standard way of determining different weightings and bias between different machine learning (including deep learning) algorithms, and their development processes, for specific cases. However, Zhou et al. (2020) explains that using evaluation metrics alone is not advisable as this can give a false sense of optimism that the model is representative of the data and a good solution to the task or issue. In addition, confirming whether a model is both representative and a good solution requires a number of considerations, including the model suitability for the desired end goal.

2.2.2 Primary research of traffic management and road safety within Smart Cities

Further research can be conducted for Sections 2.2.1.1 and 2.2.1.2 to help better explore first-hand experiences and attitudes towards traffic management and road safety. This would likely help encourage more informed decisions in the implementation methodology. A straightforward and effective approach of collecting such information is through a questionnaire. A questionnaire has been constructed with the target audience criteria; see Appendix A. In total, the questionnaire had 12 total participants.

2.2.3 Theory

The information researched in Sections 2.2.1.1, 2.2.1.2, 2.2.1.3 and 2.2.1.4 provides a basis for seeking relevant data that could be leveraged to construct a machine learning or deep learning model that can estimate the likelihood of a road traffic accident occurring in an urban area. This data can be categorised into three levels: the readiness to drive of the driver and their vehicle condition (level 1), what factors outside of an individual's vehicle circumstances exist (e.g., road words, traffic flow, speed limit, etc.) (level 2) and what environmental factors exist (e.g., if it is raining, the road will become wet) (level 3). The figure below illustrates some of the varying factors that contribute to a road traffic accident — which have been derived thematically and using findings from the literature review. The figure was created using JGraph Ltd (2023).

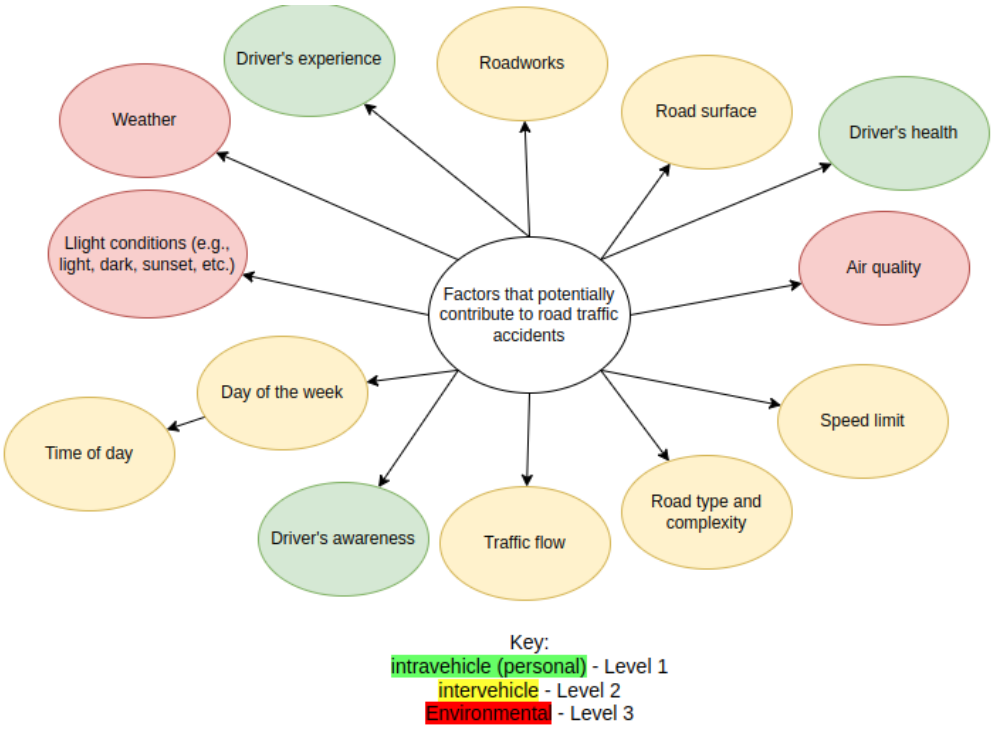


Figure 2.1: Varying factors that potentially contribute to road traffic accidents.

2.2.3.1 Main dataset identification

When finding the relevant data, it is important it is related to the same target area. For example, traffic flow data for Birmingham, England, would not be useful if the rest of the data is for Wolverhampton, England, as they are two different locations. Fortunately, a lot of open access and useful data are available for an appropriate and convenient location, Greater London. Although Greater London is classed as a county rather than a city, the Smart City model is currently being applied across this county, so it would still be acceptable to use — it can be viewed a one enormous city in a sense. Transport for London (2024a) and Greater London Authority (2024) have huge amounts of open access data and information available regarding Greater London. A collection of datasets (in the form of semi-structured comma separated values files) that are extremely applicable to this report are accessible at Transport for London (2024b), under the 'Collision data extracts' section (which is visualised in the figure below for convenience). These datasets cover data between and including 2005-2023.

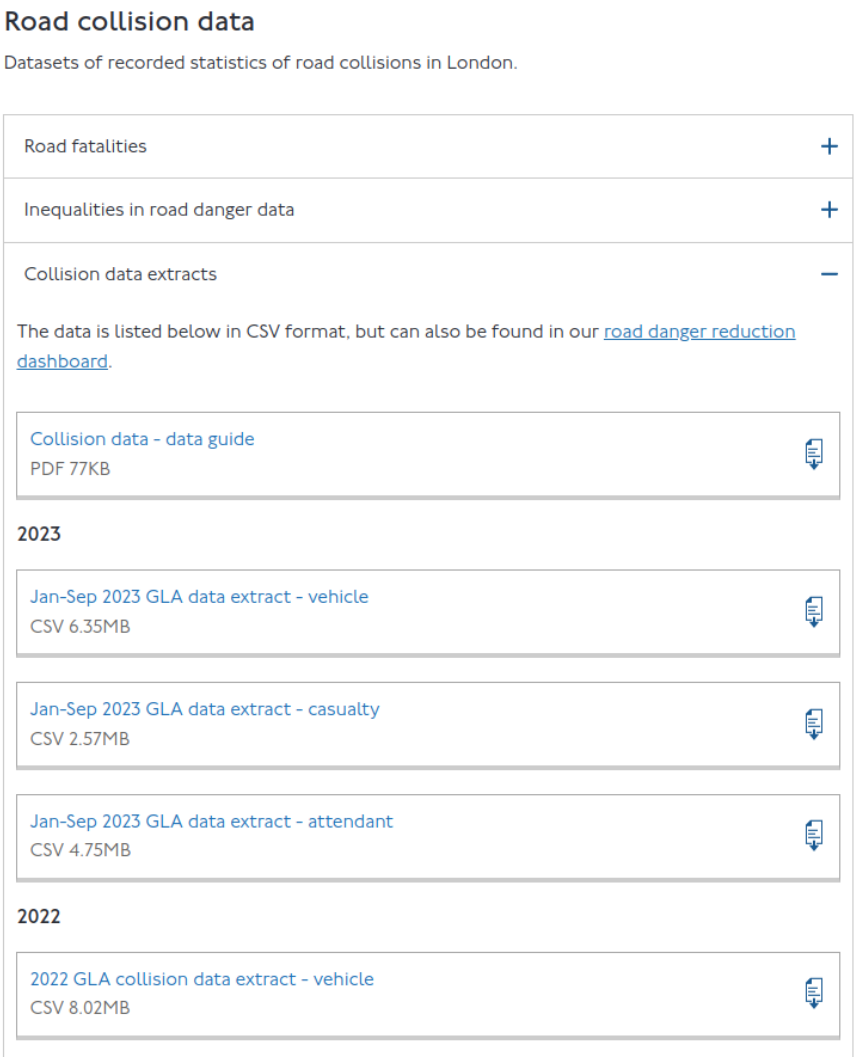


Figure 2.2: Transport for London recorded RTAs datasets.

Source: Transport for London (2024b).

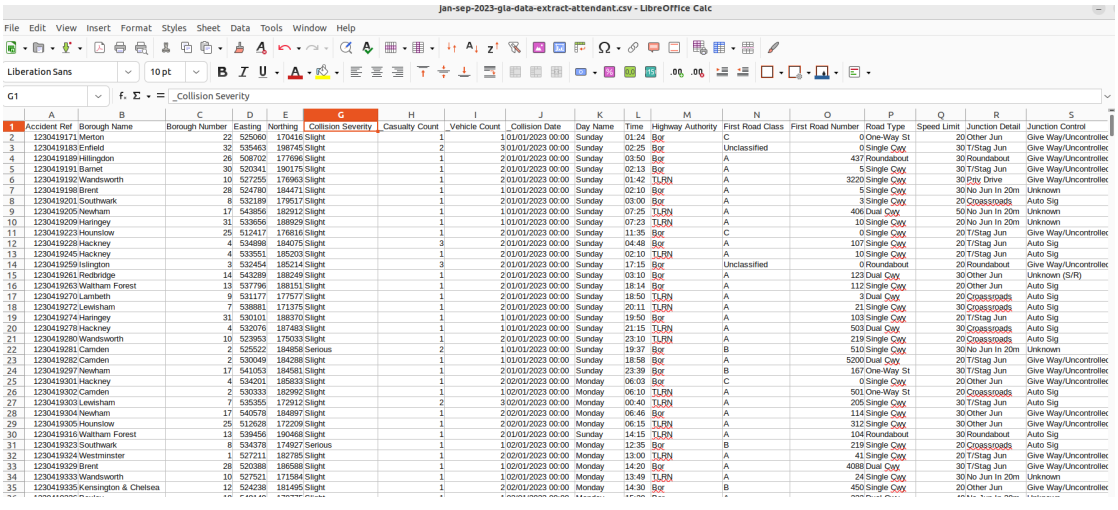
After comparing the data contained within the three types of denoted category of files - 'vehicle', 'casualty' and 'attendant' - to the factors in Figure 2.1, the most relevant file category type is 'attendant'. To summarise:

- 'vehicle' files concern finer details of vehicles and driver intentions for specific accidents.
- 'casualty' files focus on the statistics and details of casualties of RTAs.
- 'attendant' files cover more generalised factors regarding particular RTAs; e.g., en-

vironmental conditions such as light levels and weather details, carriageway characteristics such as speed limits, etc.

The 'vehicle' files would not be too useful for machine learning purposes due to the lack of generalisable information. Furthermore, the 'casualty' files would suffer from the same issue, where there is no identifiable way to convert the specific information to generalisable patterns. The more detailed analysis of the 'attendant' files will be discussed in Section 3.3.1.

Although the 'attendant' files contain data directly relating to a number of factors that contribute to RTAs, it does not contain data for other key factors such as traffic flow information at the time of an accident, or the condition of the road surface (e.g., were potholes or damaged pavement present). Unfortunately, there is not enough detailed open source data available that cover other conditions adequately. This is due to reasons such as those related to privacy of individual information, a lack of data, a lack of processing of such data, and/or the cost of using more fine-grained data. For instance, Ordnance Survey (2024) has a vivid range of in-depth transport-related data, but it is proprietary information, so getting this data could be relatively very costly. Figure 2.3 shows a sample of the data in an 'attendant' file using LibreOffice Calc (The Document Foundation, 2022).



	A	B	C	D	E	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Accident Ref	Borough Name	Borough Number	Easting	Northing	Collision Severity	Casualty Count	Vehicle Count	Collision Date	Day Name	Time	Highway Authority	First Road Class	First Road Number	Road Type	Speed Limit	Junction Detail	Junction Control
2	1230419171	Merton	22	525060	170416	Slight	1	1	01/01/2023 00:00	Sunday	01:24	Bor	C	0	One-Way St	20	Other Jun	Give Way/Uncontrolle
3	1230419183	Enfield	32	535463	198745	Slight	2	3	01/01/2023 00:00	Sunday	02:25	Bor	Unclassified	0	Single Cwy	30	T/Stag Jun	Give Way/Uncontrolle
4	1230419189	Hillingdon	26	508702	177696	Slight	1	2	01/01/2023 00:00	Sunday	03:50	Bor	A	437	Roundabout	30	Roundabout	Give Way/Uncontrolle
5	1230419191	Barnet	30	520341	190175	Slight	1	2	01/01/2023 00:00	Sunday	02:13	Bor	A	5	Single Cwy	30	T/Stag Jun	Give Way/Uncontrolle
6	1230419192	Wandsworth	10	527255	176963	Slight	1	2	01/01/2023 00:00	Sunday	01:42	TLRN	A	3220	Single Cwy	30	Priv Drive	Give Way/Uncontrolle
7	1230419198	Brent	28	524780	184471	Slight	1	1	01/01/2023 00:00	Sunday	02:10	Bor	A	5	Single Cwy	30	No Jun In 20m	Unknown
8	1230419201	Southwark	8	532189	179517	Slight	1	2	01/01/2023 00:00	Sunday	03:00	Bor	A	3	Single Cwy	20	Crossroads	Auto Sig
9	1230419205	Newham	17	543856	182912	Slight	1	1	01/01/2023 00:00	Sunday	07:25	TLRN	A	406	Dual Cwy	50	No Jun In 20m	Unknown
10	1230419209	Haringey	31	533656	188929	Slight	1	1	01/01/2023 00:00	Sunday	07:23	TLRN	A	10	Single Cwy	20	No Jun In 20m	Unknown
11	1230419223	Hounslow	25	512417	176816	Slight	1	2	01/01/2023 00:00	Sunday	11:35	Bor	C	0	Single Cwy	20	T/Stag Jun	Give Way/Uncontrolle
12	1230419228	Hackney	4	534898	184075	Slight	3	2	01/01/2023 00:00	Sunday	04:48	Bor	A	107	Single Cwy	20	T/Stag Jun	Auto Sig
13	1230419245	Hackney	4	533551	185203	Slight	1	2	01/01/2023 00:00	Sunday	02:10	TLRN	A	10	Single Cwy	20	T/Stag Jun	Auto Sig
14	1230419259	Islington	3	532454	185214	Slight	3	2	01/01/2023 00:00	Sunday	17:15	Bor	Unclassified	0	Roundabout	20	Roundabout	Give Way/Uncontrolle
15	1230419261	Redbridge	14	543289	188249	Slight	1	2	01/01/2023 00:00	Sunday	03:10	Bor	A	123	Dual Cwy	30	Other Jun	Unknown (S/R)
16	1230419263	Waltham Forest	13	537796	188151	Slight	1	2	01/01/2023 00:00	Sunday	18:14	Bor	A	112	Single Cwy	20	Other Jun	Auto Sig
17	1230419270	Lambeth	9	531177	177577	Slight	1	2	01/01/2023 00:00	Sunday	18:50	TLRN	A	3	Dual Cwy	20	Crossroads	Auto Sig
18	1230419272	Lewisham	7	538881	171375	Slight	1	2	01/01/2023 00:00	Sunday	20:11	TLRN	A	21	Single Cwy	30	Crossroads	Auto Sig
19	1230419274	Haringey	31	530101	188370	Slight	1	1	01/01/2023 00:00	Sunday	19:50	Bor	A	103	Single Cwy	20	T/Stag Jun	Auto Sig
20	1230419278	Hackney	4	532076	187483	Slight	1	1	01/01/2023 00:00	Sunday	21:15	TLRN	A	503	Dual Cwy	30	Crossroads	Auto Sig
21	1230419280	Wandsworth	10	523953	175033	Slight	1	2	01/01/2023 00:00	Sunday	23:10	TLRN	A	219	Single Cwy	20	Crossroads	Auto Sig
22	1230419281	Camden	2	525522	184858	Serious	2	1	01/01/2023 00:00	Sunday	19:37	Bor	B	510	Single Cwy	30	No Jun In 20m	Unknown
23	1230419282	Camden	2	530049	184288	Slight	1	1	01/01/2023 00:00	Sunday	18:58	Bor	A	5200	Dual Cwy	20	T/Stag Jun	Give Way/Uncontrolle
24	1230419297	Newham	17	541053	184581	Slight	1	2	01/01/2023 00:00	Sunday	23:39	Bor	B	167	One-Way St	30	T/Stag Jun	Give Way/Uncontrolle
25	1230419301	Hackney	4	534201	185833	Slight	1	2	02/01/2023 00:00	Monday	06:03	Bor	C	0	Single Cwy	20	Other Jun	Give Way/Uncontrolle
26	1230419302	Camden	2	530333	182992	Slight	1	1	02/01/2023 00:00	Monday	06:10	TLRN	A	501	One-Way St	20	Crossroads	Auto Sig
27	1230419303	Lewisham	7	535355	172912	Slight	2	3	02/01/2023 00:00	Monday	00:40	TLRN	A	205	Single Cwy	30	T/Stag Jun	Auto Sig
28	1230419304	Newham	17	540578	184897	Slight	1	2	02/01/2023 00:00	Monday	06:46	Bor	A	114	Single Cwy	30	Other Jun	Give Way/Uncontrolle
29	1230419305	Hounslow	25	512628	172209	Slight	1	2	02/01/2023 00:00	Monday	06:15	TLRN	A	312	Single Cwy	30	Other Jun	Give Way/Uncontrolle
30	1230419316	Waltham Forest	13	539456	190468	Slight	1	2	01/01/2023 00:00	Sunday	14:15	TLRN	A	104	Roundabout	30	Roundabout	Auto Sig
31	1230419323	Southwark	8	534378	174927	Serious	1	1	02/01/2023 00:00	Monday	12:35	Bor	B	219	Single Cwy	20	Crossroads	Auto Sig
32	1230419324	Westminster	1	527211	182785	Slight	1	2	02/01/2023 00:00	Monday	13:00	TLRN	A	41	Single Cwy	20	T/Stag Jun	Give Way/Uncontrolle
33	1230419329	Brent	28	520388	186588	Slight	1	1	02/01/2023 00:00	Monday	14:20	Bor	A	4088	Dual Cwy	30	T/Stag Jun	Give Way/Uncontrolle
34	1230419333	Wandsworth	10	527521	171584	Slight	1	1	02/01/2023 00:00	Monday	13:49	TLRN	A	24	Single Cwy	30	No Jun In 20m	Unknown
35	1230419335	Kensington & Chelsea	12	524238	181495	Slight	1	2	02/01/2023 00:00	Monday	14:30	Bor	B	450	Single Cwy	20	Other Jun	Give Way/Uncontrolle

Figure 2.3: Sample of TfL 'attendant' files; recorded RTA data in Greater London.

Additional datasets will be identified in Section 3.3.1.

2.2.3.2 The use of machine learning and/or deep learning

There was a mixture of using ML and DL methods identified in Section 2.2.1.4. Although DL offers an opportunity to discover more complex patterns and trends in the data, it comes with the drawbacks of being much more difficult to explain and interpret compared to ML. This refers to the Explainable AI concept.

The datasets identified in Section 2.2.3.1 can be judged as time-series datasets because the time in which records are denoted has significance for the respective recorded values. For example, there could be more accidents during "rush hours" compared to the rest of the day, possibly because of more hurried drivers on the road at once. Moreover, the objective of creating a predictive model in this project is to predict a future event (the likelihood of a RTA) based on past information. For DL, Recurrent Neural Networks (RNNs) are often used with time-series data, specifically Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GNU) networks. LSTM networks are often favoured over GNU and other RNNs for time-series data, mainly because they greatly diminish the issues of the Exploding Gradient and Vanishing Gradient problems. For this project, a LSTM neural network will be used to construct the predictive model mainly due to its capabilities for handling time-series information and discovering more complex patterns in the data.

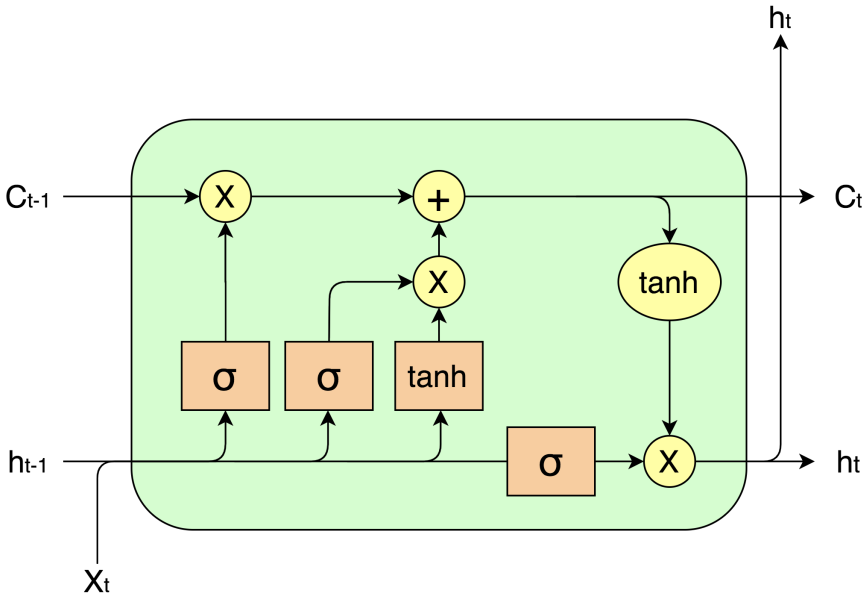


Figure 2.4: Visual representation of a LSTM unit.

Source: Cheng (2020).

2.3 Summary

This literature review explored the context and circumstances of traffic management and road safety, particularly in urban areas. This will help make more informed decisions in the next stage when putting the implementation into its target setting. In addition, accident mitigation strategies and previous RTA prediction models were investigated to provide a stronger basis for making implementation direction choices. For instance, considering the complexity of this project task and possible data, the prediction models researched will be extremely helpful for determining which processes are more effective than others. Finally, some possible evaluation metrics were reviewed, which will enable a stronger understanding of how to better test the implementation prediction models.

3 Implementation Design and Methods

This section will discuss and explain an overall plan, with a basis formed of the findings from the review of the literature, that can be used to successfully implement this project. This will be broken down into subsections including the methodology, limitations and options, the design specification, a concept solution, testing strategies, design and development, testing, and a final summary.

3.1 Methodology

According to Ghena and Ghiculescu (2023) there are two popular types of project management methodologies: the Waterfall method and the Agile method. The Waterfall method is a traditional and linear approach in which each stage of the project must be completed before moving onto the next stage. The Agile method is a more flexible and iterative approach where a project is broken down into several dynamic stages known as "sprints". The Waterfall method is more useful for projects where there is a clear and well-defined project scope and a set timeframe, while the Agile method is more useful for business environments where customer needs can change, and better collaboration and communication is needed.

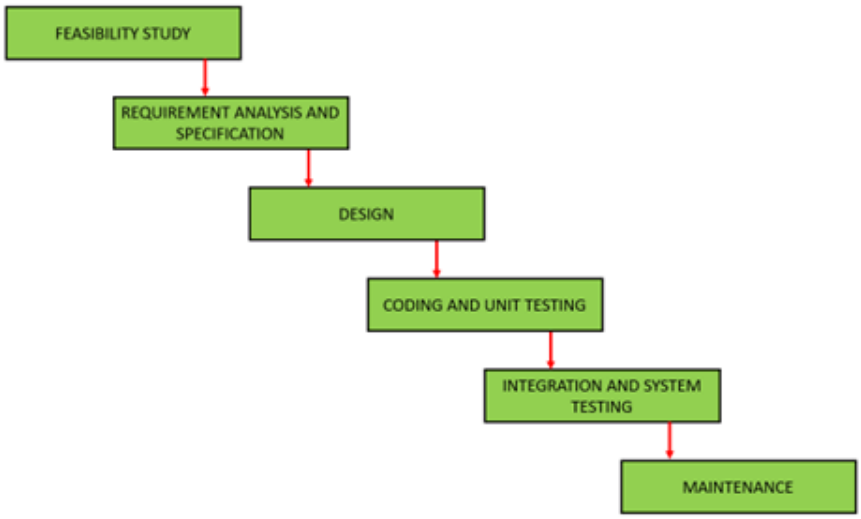


Figure 3.1: Illustration of the Waterfall methodology.

Source: GeeksForGeeks (2024).

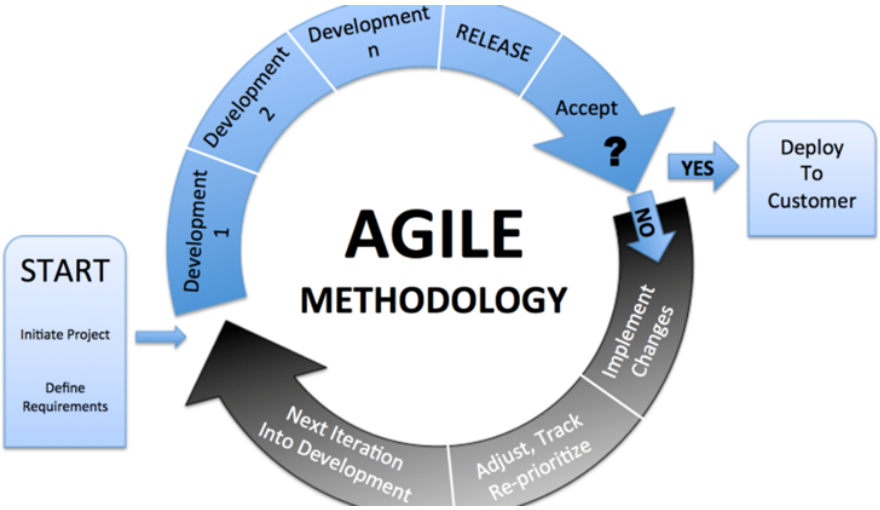


Figure 3.2: Illustration of the Agile methodology.

Source: Techchink (2023).

As this project will have a clear methodology and fixed timeline, the Waterfall methodology would be ideal to use, especially as it keeps the process straightforward. Therefore, the Waterfall methodology will be used for the project implementation.

The original timeline for this project had to be updated to meet the new assigned project deadline. The original timeline can be found in Appendix B.1, while the updated timeline can be found in Appendix B. The effectiveness of this timeline will be discussed during the evaluation stages.

3.2 Limitations and Options

Even though the dataset identified in Section 2.2.3.1 has a substantial amount of data available, exploratory data analysis (EDA) will be required to determine how much of this data is usable. Moreover, supplementary data from other datasets would likely aid in giving the data of the main dataset more context.

As discussed in Section 2.2.3.2, a LSTM neural network will be implemented for generating the main baseline model. There are two very popular deep learning Python libraries available for constructing Artificial Neural Networks (ANNs): Keras and PyTorch. Using either would be acceptable as they offer similar services and tools. However, the former offers a higher-level API compared to the latter — meaning it would be more

straightforward to use. But this comes at a cost of flexibility, that is, being more limited for constructing and adapting neural networks to a higher degree. Furthermore, Keras does not fully support machines that do not have a graphics card installed, while PyTorch offers a 'CPU-only' version. The significance of this will be explained in Section 3.3. Due to these two factors, PyTorch will be used instead of Keras.

The final system product will likely not be able to provide detailed and tailored traffic management and road safety suggestions for the target areas. This is due to the lack of granularity of information for certain areas and the would-be drastically increased system complexity (which would be beyond the scope of this project).

3.3 Design Specification

As mentioned in Objectives 4 and 5, pipelines will be developed to help automate the processes of model creation and live accident likelihood prediction. In a MLOps pipeline (Figure 3.3) — pipelines tailored to the development of ML models — there are typically five key stages:

1. Ingestion: for collecting, storing, and possibly integrating the raw data together.
2. Preprocessing: for performing any predefined preprocessing actions and transformations to the raw data.
3. Model Development: using the preprocessed data to construct a ML or DL model(s).
4. Model Monitoring: register the formulated model, so it can be monitored for model / data drift, and other changes overtime.
5. Model Deployment: deploy the model to a service or API; ready for usage.

Hence, these will be the main steps for the model creation pipeline. However, for the 'Model Deployment' stage, this will link to a separate pipeline for making the live accident likelihood predictions using real-time data. At the end of this separate pipeline will be the web app and the web API discussed in Objective 6. Apache Airflow (Apache Software Foundation, 2024), a popular and open source platform with a range of use-

ful automation capabilities, will be used for the creation of the pipelines. In addition, Airflow offers a lot of flexibility in creating pipelines for a range of purposes.

Python will be used as the preferred primary programming language for the development of these pipelines and their individual stages, particularly due to its popularity and wide-ranging support with Airflow and other tools.

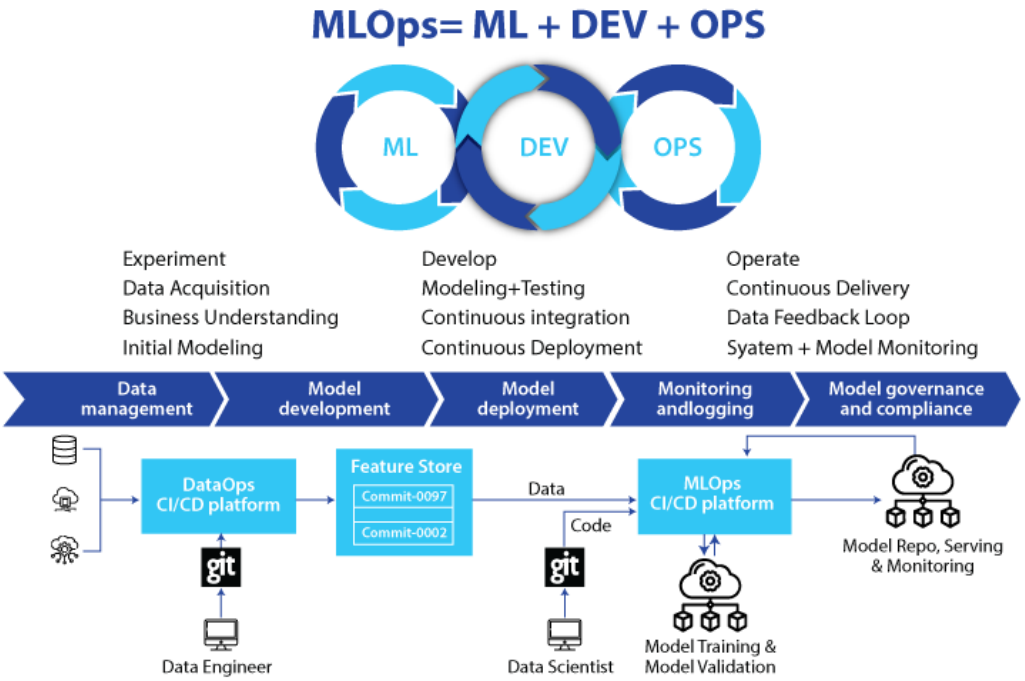


Figure 3.3: Example illustration of the stages of MLOps.

Source Bhatt (2023).

Jupyter Notebook (Jupyter, 2024) will be used to develop the implementation code through trial and testing.

To make development easier and more manageable, Conda environments (Anaconda Inc, 2024) will be used to manage installed packages. Conda environments are isolated from other environments and the main system, so if any dependency conflicts occur for instance and cannot be resolved, the environment can easily be re-created. These environments are also portable as well.

3.3.1 The predictive model

The main intention of creating the predictive model is to predict the likelihood of road traffic accidents in SCs. But the extent and feasibility of this is particularly determined by other considerations (e.g., how would it do this, what constraints exist, what is more practical and desired, etc.) and the available data. The dataset identified in Section 2.2.3.1 encourage the realisation of the constraints of any predictive model that will be formed. For instance, the predictive model will not be able to reliably predict the likelihood of RTAs for individual streets as there will likely be streets with little to no information. This would mean a detrimental accident could occur on a street where the model predicted a low likelihood of an accident happening due to the lack of information. To help resolve this, the predictions can be made less granular by making them on a borough-by-borough basis. There is a 'borough' feature present in the dataset which declares the borough where the accident occurred.

Data scientists in certain settings that leverage RTA data, such as those in emergency response services in the UK, very likely already consider the raw probability of RTAs occurring in certain regions across Greater London, and indeed the rest of the UK. For instance, if 4 out of 10 RTAs occur in a Greater London region such as Lambeth, then emergency services may be coordinated to be more active in that region compared to lower-risk areas. However, this has a limitation such that it only considers the raw probability and not necessarily the overall severity of the accidents that actually took place. Even if these other elements are considered, it would likely be a very time-consuming and bias-influenced process if done manually. This is where the predictive model will attempt to automate, accelerate and less heavily influence this challenge.

This can be perceived as a supervised time-series forecasting challenge, as the model will need to predict a target variable in the future. Based on manual analysis of the data, as indicated in Figure 2.3, the 'Accident Severity' feature would be suitable as the target variable if the predictions were to be made on a street-by-street basis. But as discussed, the predictions should be made on a borough-by-borough basis, so instead a new target variable should be created using the aggregation of the street RTA raw probabilities and their severity for each borough. Regarding the class values of 'Accident Severity', they

are 'slight', 'serious' and 'fatal'. Therefore, new class values that represent the accident likelihood in a borough should be created. The most straightforward approach would be to use a percentage value to represent the likelihood of a RTA with a certain severity. However, this conceptually does not add up as the granularity of percentage values decreases their significance. For instance, a 63% likelihood of an accident occurring can be interpreted as ambiguous and volatile. A better alternative would be to create abstract classes that progressively represent the higher likelihood of a RTA taking place. For example, categories 1, 2, 3, 4 and 5 can represent the likelihood of a RTA with a certain severity happening, where category 1 represents a very low likelihood of critical RTAs occurring in that region, and category 5 equates to a very high likelihood. This method is known as ordinal classification — where the target values have order but the distance between them is not distinguishable. Only a minimal number of categories are identified as the more possible class values there are, the more likely any models developed are less able to reference the correct category. Table 3.1 shows the preliminary possible significance of these borough accident categories.

Table 3.1: Preliminary borough RTA likelihood categories, based on an hourly basis.

Category	**Significance**
1	No or 1 RTA with a slight severity is likely to occur.
2	2 or more RTAs with a slight severity and/or 1 RTA with a serious severity are likely to occur.
3	2 RTAs with a serious severity are likely to occur.
4	1 RTA with a fatal severity and/or 3 or more RTAs with a serious severity are likely to occur.
5	1 or more RTAs with a fatal severity are likely to occur.

The categories described in Table 3.1 are influenced by individual bias, which can affect any predictions made and their importance. These category definitions should be refined after further EDA of the data.

3.3.1.1 Additional datasets

As the predictive model is intended to predict on a borough-by-borough basis, factors that affect individual accidents should not be included (as it would violate the context of the prediction). However, other borough information can be employed to help the model refine its predictions. Ideally, this information should be related to the factors in Figure 2.1, but other information outside of this can also be useful. These datasets include:

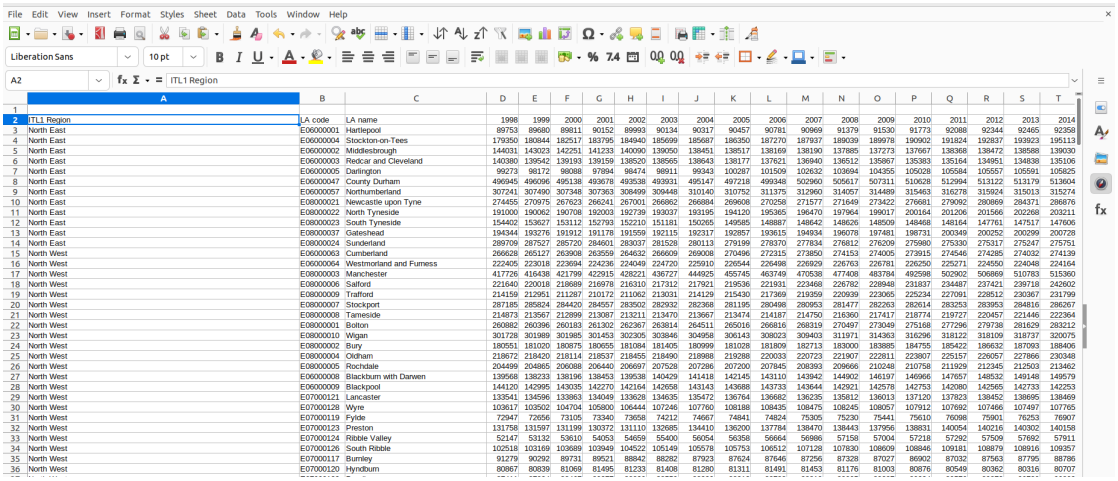
- Traffic flow: UK Department for Transport (2022) supplies yearly-averaged traffic flow information for each London borough between 1993-2022. The data is measured in million vehicle kilometers. Traffic flow likely has a high correlation with RTAs occurring, as the more vehicles there are, the higher the raw probability of an accident occuring.

- Population: Office for National Statistics (2022) has mid-year averaged population counts for each London borough between 1998-2022. This likely has an indirect importance towards RTAs. For instance, the more people there are in an area, the more likely councils have implemented safety features (such as more crossings and traffic lights).

- Gross Domestic Product (GDP): Office for National Statistics (2022) provides the GDP of each London borough between 1998-2022, based on 2022 basic prices, in pounds million. It does not take inflation into account, allowing for comparisons between prior GDP estimates. GDP can be a good indicator of a borough's finances to an extent, but coupled with the population and crime rates can provide interesting insights into RTAs as well.

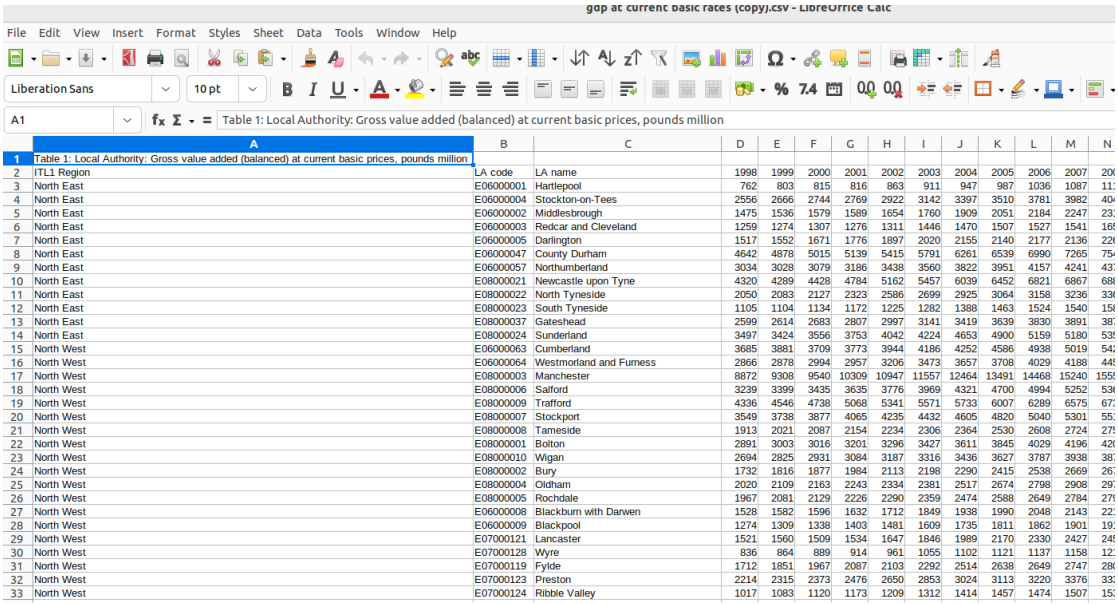
- Crime: Metropolitan Police (2024) gives a summary of crimes committed in each London borough, including more serious traffic crimes (such as theft), between April 2010-June 2022. Crime rates can give a glimpse into the attitudes of the population, and thus can impact the likelihood of RTAs happening.

Figures 3.4, 3.5, 3.6, and 3.7 show snapshots of these datasets in LibreOffice Calc.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
2	ITL1 Region	LA code	LA name	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014
3	North East	E06000001	Hartlepool	89753	89680	89811	90152	89993	90134	90317	90457	90781	90969	91379	91530	91773	92088	92344	92465	92358
4	North East	E06000004	Stockton-on-Tees	179350	180844	182517	183795	184940	185699	185687	186350	187270	187937	189039	189978	190902	191824	192837	193923	195113
5	North East	E06000002	Middlesbrough	144031	143023	142251	141233	140090	139050	138451	138517	138169	138190	137885	137273	137667	138368	138472	138588	139030
6	North East	E06000003	Redcar and Cleveland	140380	139542	139193	139159	138520	138565	138643	138177	137821	136940	136512	135867	135383	135164	134951	134838	135106
7	North East	E06000005	Darlington	99273	98172	98088	97894	98474	98911	99343	100287	101509	102632	103694	104355	105028	105584	105557	105591	105825
8	North East	E06000047	County Durham	496945	496096	495138	493678	493538	493931	495147	497218	499348	502960	505617	507311	510628	512994	513122	513179	513604
9	North East	E06000057	Northumberland	307241	307490	307348	307363	308499	309448	310140	310752	311375	312960	314057	314489	315463	316278	315924	315013	315274
10	North East	E08000021	Newcastle upon Tyne	274455	270975	267623	266241	267001	266862	266884	269608	270258	271577	271649	273422	276681	279092	280869	284371	286676
11	North East	E08000022	North Tyneside	191000	190062	190708	192003	192739	193037	193195	194120	195365	196470	197964	199017	200164	201206	201566	202268	203211
12	North East	E08000023	South Tyneside	154402	153627	153112	152793	152210	151181	150265	149585	148887	148642	148626	148509	148468	148164	147761	147517	147606
13	North East	E08000037	Gateshead	194344	193276	191912	191178	191559	192115	192317	192857	193615	194934	196078	197481	198731	200349	200252	200299	200728
14	North East	E08000024	Sunderland	289709	287527	285720	284601	283037	281528	280113	279199	278370	277834	276812	276209	275980	275330	275317	275247	275751
15	North West	E06000063	Cumberland	266628	265127	263908	263559	264632	266609	269008	270496	272315	273850	274153	274005	273915	274546	274285	274032	274139
16	North West	E06000064	Westmorland and Furness	222405	223018	223694	224236	224049	224720	225910	226544	226498	226929	226763	226781	226250	225271	224550	224048	224164
17	North West	E08000003	Manchester	417726	416438	421799	422915	428221	436727	444925	455745	463749	470538	477408	483784	492598	502902	506869	510783	515360
18	North West	E08000006	Salford	221640	220018	218689	216978	216310	217312	217921	219536	221931	223468	226782	228948	231837	234487	237421	239718	242602
19	North West	E08000009	Trafford	214159	212951	211287	210172	211062	213031	214129	215430	217369	219359	220939	223065	225234	227091	228512	230367	231799
20	North West	E08000007	Stockport	287185	285824	284420	284557	283502	282932	282368	281195	280498	280953	281477	282263	282614	283253	283953	284816	286267
21	North West	E08000008	Tameside	214873	213567	212899	213087	213211	213470	213667	213474	214187	214750	216360	217417	218774	219727	220457	221446	222364
22	North West	E08000001	Bolton	260882	260396	260183	261302	262367	263814	264511	265016	266816	268319	270497	273049	275168	277296	279738	281629	283212
23	North West	E08000010	Wigan	301728	301989	301985	301453	302305	303846	304958	306143	308023	309403	311971	314363	316296	318122	318109	318737	320075
24	North West	E08000002	Bury	180551	181020	180875	180655	181084	181405	180999	181028	181809	182713	183000	183885	184755	185422	186632	187093	188406
25	North West	E08000004	Oldham	219872	218420	218114	218537	218455	218490	218988	219288	220033	220723	221907	222811	223807	225157	226057	227866	230348
26	North West	E08000005	Rochdale	204499	204865	206088	206440	206697	207528	207286	207200	207845	208393	209666	210248	210758	211929	212345	212503	213462
27	North West	E06000008	Blackburn with Darwen	139568	138233	138196	138453	139538	140429	141418	142145	143110	143942	144902	146197	146966	147657	148532	149148	149579
28	North West	E06000009	Blackpool	144120	142995	143035	142270	142164	142658	143143	143688	143733	143644	142921	142578	142753	142080	142565	142733	142253
29	North West	E07000121	Lancaster	133841	134596	133863	134049	133628	134635	135472	136764	136682	136235	135812	136013	137120	137823	138452	138695	138469
30	North West	E07000128	Wyre	103617	103502	104704	105800	106444	107246	107760	108188	108435	108475	108245	108057	107912	107692	107466	107497	107765
31	North West	E07000119	Fylde	72947	72656	73105	73340	73658	74212	74667	74841	74824	75305	75230	75441	75610	76098	75901	76253	76907
32	North West	E07000123	Preston	131758	131597	131199	130372	131110	132685	134410	136200	137784	138470	138443	137956	138831	140054	140216	140302	140158
33	North West	E07000124	Ribble Valley	52147	53132	53610	54053	54659	55400	56054	56358	56664	56986	57158	57004	57218	57292	57509	57692	57911
34	North West	E07000126	South Ribble	102518	103169	103689	103949	104522	105149	105578	105753	106512	107128	107830	108609	108846	109181	108879	108916	109357
35	North West	E07000117	Burnley	91279	90292	89731	89521	88842	88282	87923	87624	87646	87256	87328	87027	86902	87032	87563	87795	88786
36	North West	E07000120	Hyndburn	80867	80839	81069	81495	81233	81408	81280	81311	81491	81453	81176	81003	80876	80549	80362	80316	80707

Figure 3.4: Population dataset snapshot.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Table 1: Local Authority: Gross value added (balanced) at current basic prices, pounds million													
2	ITL1 Region	LA code	LA name	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	20
3	North East	E06000001	Hartlepool	762	803	815	816	863	911	947	987	1036	1087	11
4	North East	E06000004	Stockton-on-Tees	2556	2666	2744	2769	2922	3142	3397	3510	3781	3982	40
5	North East	E06000002	Middlesbrough	1475	1536	1579	1589	1654	1760	1909	2051	2184	2247	23
6	North East	E06000003	Redcar and Cleveland	1259	1274	1307	1276	1311	1446	1470	1507	1527	1541	16
7	North East	E06000005	Darlington	1517	1552	1671	1776	1897	2020	2155	2140	2177	2136	22
8	North East	E06000047	County Durham	4642	4878	5015	5139	5415	5791	6261	6539	6990	7265	75
9	North East	E06000057	Northumberland	3034	3028	3079	3186	3438	3560	3822	3951	4157	4241	43
10	North East	E08000021	Newcastle upon Tyne	4320	4289	4428	4784	5162	5457	6039	6452	6821	6867	68
11	North East	E08000022	North Tyneside	2050	2083	2127	2323	2586	2699	2925	3064	3158	3236	33
12	North East	E08000023	South Tyneside	1105	1104	1134	1172	1225	1282	1388	1463	1524	1540	15
13	North East	E08000037	Gateshead	2599	2614	2683	2807	2997	3141	3419	3639	3830	3891	38
14	North East	E08000024	Sunderland	3497	3424	3556	3753	4042	4224	4653	4900	5159	5180	53
15	North West	E06000063	Cumberland	3685	3881	3709	3773	3944	4186	4252	4586	4938	5019	54
16	North West	E06000064	Westmorland and Furness	2866	2878	2994	2957	3206	3473	3657	3708	4029	4188	44
17	North West	E08000003	Manchester	8872	9308	9540	10309	10947	11557	12464	13491	14468	15240	155
18	North West	E08000006	Salford	3239	3399	3435	3635	3776	3969	4321	4700	4994	5252	53
19	North West	E08000009	Trafford	4336	4546	4738	5068	5341	5571	5733	6007	6289	6575	67
20	North West	E08000007	Stockport	3549	3738	3877	4065	4235	4432	4605	4820	5040	5301	55
21	North West	E08000008	Tameside	1913	2021	2097	2154	2234	2306	2364	2530	2608	2724	27
22	North West	E08000001	Bolton	2891	3003	3016	3201	3296	3427	3611	3845	4029	4196	42
23	North West	E08000010	Wigan	2694	2825	2931	3084	3187	3316	3436	3627	3787	3938	38
24	North West	E08000002	Bury	1732	1816	1877	1984	2113	2198	2290	2415	2538	2669	26
25	North West	E08000004	Oldham	2020	2109	2163	2243	2334	2381	2517	2674	2798	2908	29
26	North West	E08000005	Rochdale	1967	2081	2129	2226	2290	2359	2474	2588	2649	2784	27
27	North West	E06000008	Blackburn with Darwen	1528	1582	1596	1632	1712	1849	1938	1990	2048	2143	22
28	North West	E06000009	Blackpool	1274	1309	1338	1403	1481	1609	1735	1811	1862	1901	19
29	North West	E07000121	Lancaster	1521	1560	1509	1534	1647	1846	1989	2170	2330	2427	24
30	North West	E07000128	Wyre	836	864	889	914	961	1055	1102	1121	1137	1158	12
31	North West	E07000119	Fylde	1712	1851	1967	2087	2103	2292	2514	2638	2649	2747	28
32	North West	E07000123	Preston	2214	2315	2373	2476	2650	2853	3024	3113	3220	3376	33
33	North West	E07000124	Ribble Valley	1017	1083	1120	1173	1209	1312	1414	1457	1474	1507	15

Figure 3.5: GDP dataset snapshot.

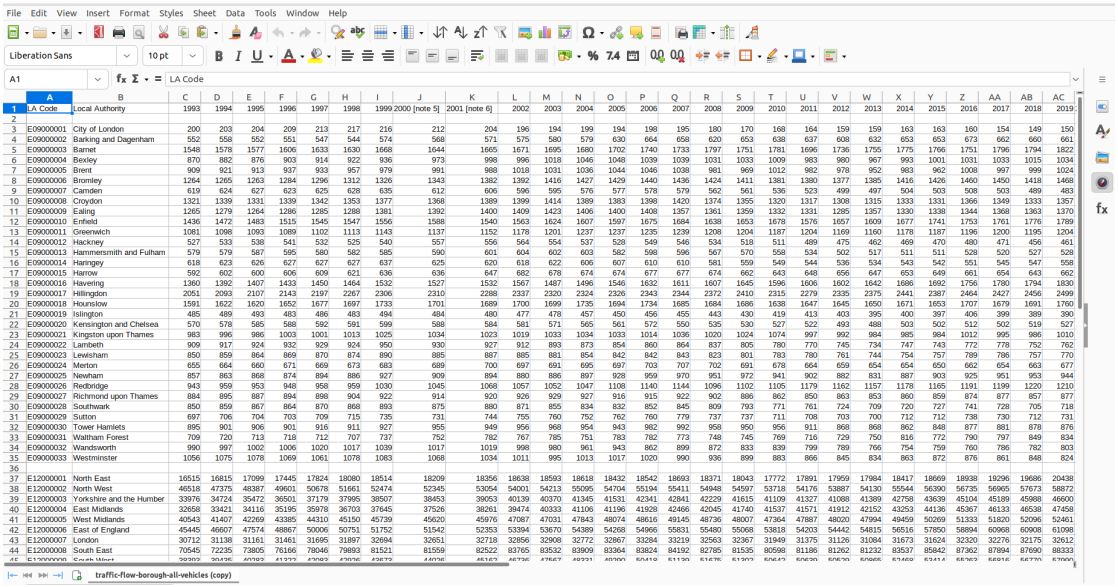


Figure 3.6: Traffic flow dataset snapshot.

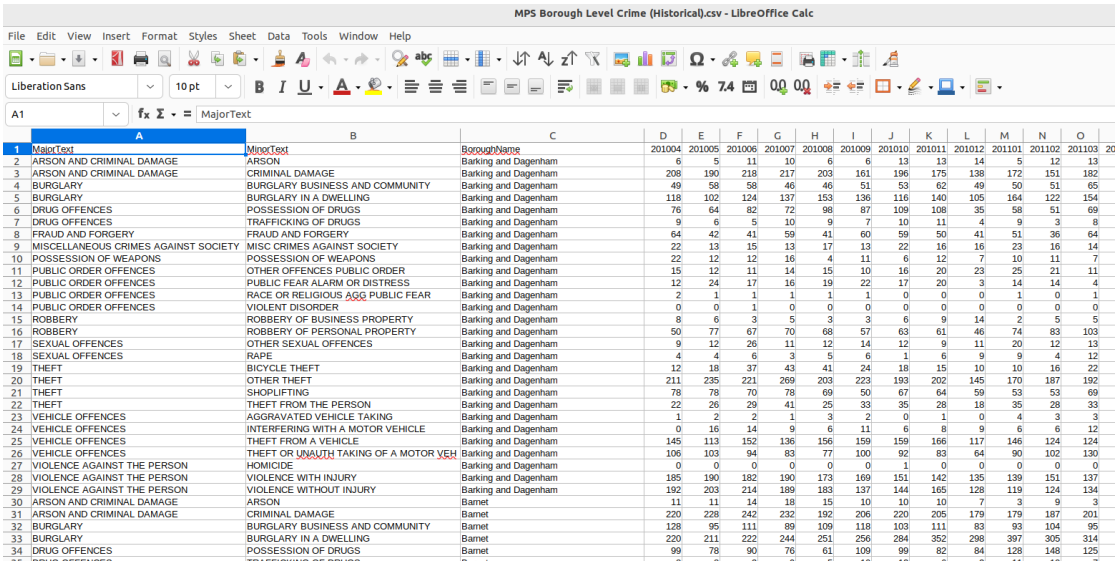


Figure 3.7: Crime dataset snapshot.

3.3.2 Model creation pipeline

3.3.2.1 Ingestion stage

There are two topics of interest to consider for the Ingestion stage: how should the data be stored and what process should be used to retrieve and possibly modify the

data. The storage database needs to be optimised for analytical purposes, as the aim is to analyse the data rather than frequently modify it. An online analytical processing (OLAP) data warehouse would meet this purpose — which are a particular category of databases designed for storing data in a read-operation-specific way. Moreover, as all the data can be presented in a tabular format (as they are all comma separated values files and have been verified for use using LibreOffice Calc), using a data warehouse suitable for structured (tabular) data would be more suitable. In addition, OLAP data warehouses have the added benefit of improved ad-hoc querying due to their more focused analytical model. Based on a online blog review by RisingWave (2023), the ClickHouse (ClickHouse Inc, 2024b) data warehouse platform seems optimal for usage. ClickHouse is an SQL column-oriented distributed data warehouse that can process large amounts of queries (concurrently) and scale with demand. It takes a hybrid approach between the traditional relational database model and more modern NoSQL databases. Its performance is highlighted by Kousha et al. (2024), as it outperforms other databases such as MySQL and Influx in almost every category (e.g., write operations, read operations, scaling capabilities, latency). This data warehouse can be deployed via Docker (Docker Inc, 2024a), a containerisation platform. Instead of creating the database on an Operating System (OS), it would be more ideal to deploy it on an isolated environment (otherwise known as a container) where it can operate independent of the OS — making development more consistent and portable. Furthermore, Docker containers are often more resource-efficient and easier to manage for development purposes. An official Docker image is available for ClickHouse (ClickHouse Inc, 2024a). To make it easier to manage and initiate containers, Docker Compose (Docker Inc, 2024b) will also be used. Docker Compose comes installed by default with more recent Docker versions.

In addition, a MariaDB server instance will be created to act as a storage for metadata information for the Airflow pipelines. This is to make the pipelines closer to a production standard, as it is more appropriate and secure to store metadata in a designated storage, rather than the default, insecure SQLite database. A MariaDB Docker image can be found at MariaDB Foundation (2024).

According to IBM Cloud Education (2021), there are two widely used approaches

for collecting and storing data from a source(s) into a database: extract, transform, load (ETL) and extract, load, transform (ELT). Typically, ETL is used when the raw data requires a number of transformations to help make it suit the database. Additionally, ETL is good for ad-hoc queries that are specific to the nature of investigating the data. But this comes with the drawback of having more schema constraints — meaning the processing of data might have to be completely re-done should the database requirements change. ELT is more recent than the traditional ETL. ELT attempts to maintain the "rawness" of the data by loading it into a database with minimal transformations (far fewer than the data stage in ETL) — which is generally more useful for business analysis and general ad-hoc querying. ELT can also be considered as extract, minimal transformation(s), load, transform; where a limited number of transformations are done to the raw data so it is compatible with the database. Given most of the raw data is in a semi-structured format and is not considered complex, it would seem more appropriate to use ELT. These specific transformations should be discovered and optimised through experimental data wrangling (done in Jupyter Notebook).

To reduce in-memory consumption of larger amounts of data, a library called Dask (Anaconda Inc and contributors, 2018a) can be used. One of the main advantages of Dask is its lazy loading, so the data is only loaded in-memory when needed. Unfortunately, there is currently no other Python packages available to enable Dask and ClickHouse to communicate directly. But there is a package called PandaHouse (Szűcs, 2020) available which can enable the direct communication between Python Pandas (a data manipulation library) (NumFocus Inc, 2024) and ClickHouse. Therefore, the data can be ingested into a Dask DataFrame, and then once it needs to be uploaded to ClickHouse, it can be converted into a Pandas DataFrame so it can be used with PandaHouse. This Pandas DataFrame will not be assigned into a variable — so it only takes up enough in-memory to be uploaded, and then be garbage collected by the interpreter.

Figure 3.8 depicts an example of Data Ingestion using a data warehouse.

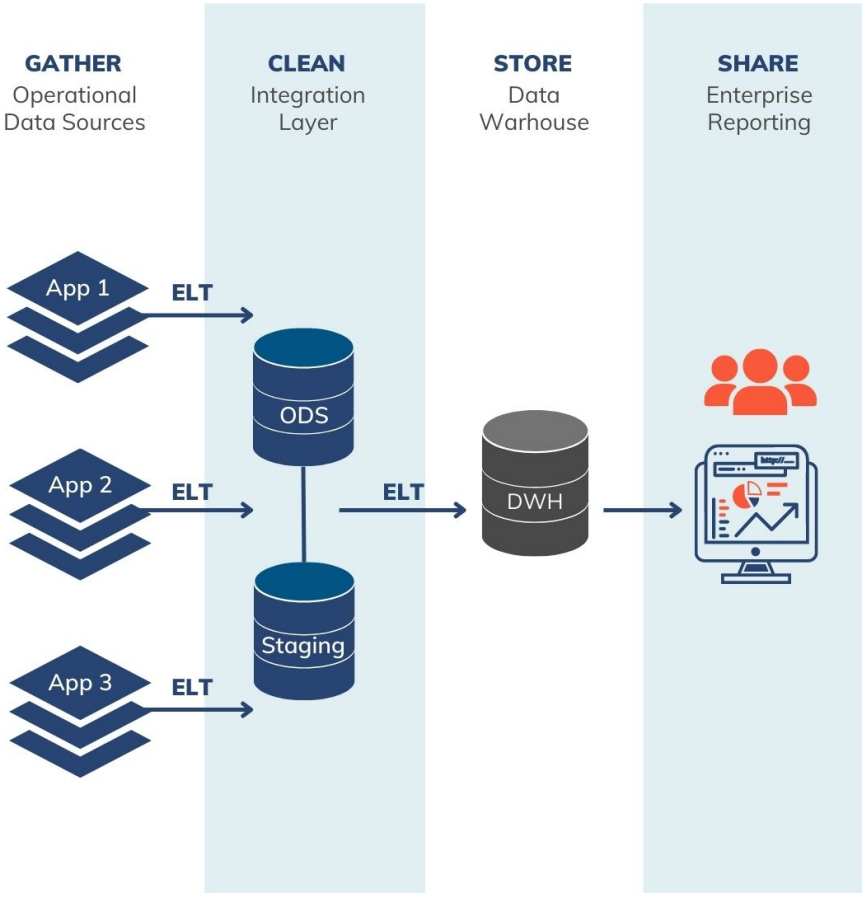


Figure 3.8: Example of Data Ingestion with a data warehouse.

Source: Costello (2018).

3.3.2.2 Preprocessing stage

Once the data has been ingested, the next step will be to analyse the fused data in more depth using EDA. The outcomes of the EDA will determine what exact preprocessing steps will be needed. For example, outlier removal, data encoding, making feature values consistent, etc. This also includes refining the accident likelihoods stated in Table 3.1. Figure 3.9 illustrates some key elements of data preprocessing. Jupyter Notebook will be used for the EDA as this would only be useful to a data scientist and not to either pipeline directly. The same Python libraries / packages mentioned in Section 3.3.2.1 can be used for data manipulation, along with SciKit Learn (Pedregosa et al., 2024) and Dask ML (Anaconda Inc and contributors, 2018b) (the Dask-compatible equi-

valent of SciKit Learn). Some example data visualisation packages include Matplotlib (MatPlotLib Team, 2024) and Seaborn (Waskom, 2024).

Moreover, as discussed in Section 3.3.1, any features that do not affect or involve boroughs as a whole should be removed, so the model can make predictions on a borough-by-borough basis. For instance, 'Highway' and 'Road Class 1', 'Speed limit', etc. This should be done before the EDA as it would be unnecessary information.

As mentioned in Section 3.3.1, as this is a time-series challenge, columns relating dates and times should be set as the index of the to-be preprocessed data. This provides temporal order to the data entries and as such enables data splitting. Once the data is organised, it is necessary to split it into training and testing sets, where the machine learning algorithms will use the training set to train a model, which will be tested against the testing set. A common ratio for this split is "80:20" — 80% of the data goes to the training set, while 20% goes to the testing set. To maintain temporal order, the data will not be shuffled before the split, so for example, the training set will have data between the years of 2010-2020 while the testing set will have data between the years of 2021-2023. To help refine the model, a validation set will be designated as a portion of the training set. The validation set allows the model to test its current "understanding" of its seen data (the training set) against unseen data (the validation set) without compromising the final testing process against the testing set, thus helping prevent temporal (target) data leakage.

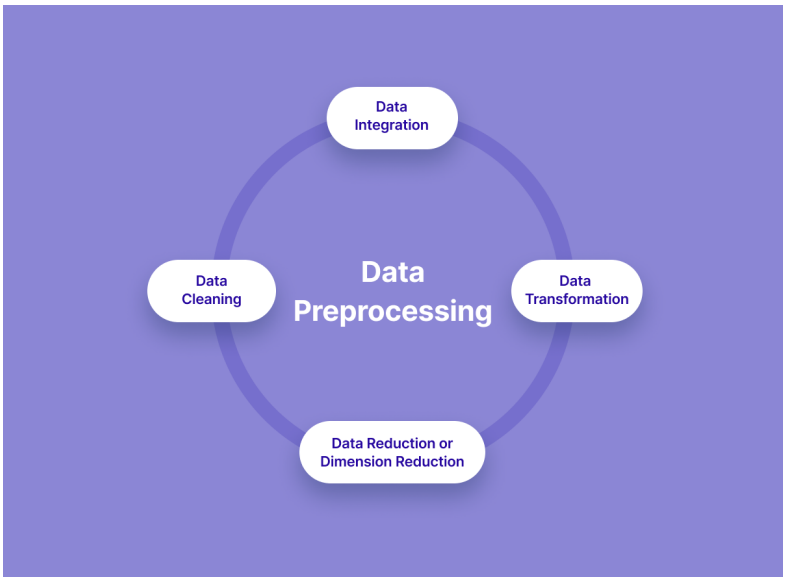


Figure 3.9: Some key elements that make up data preprocessing.

Source: Baheti (2021).

As discussed in Section 3.3.1, a function needs to be created to combine the raw probability of an accident occurring with its overall severity. At first thought, it would seem making the current 'Accident severity' values — 'slight', 'severe' and 'fatal' — worth 1, 2, and 3 respectively. These can then be added up for a specific time (e.g., for every hour) to make an overall value, then if this value exceeds a certain threshold, it will belong to one of the five categories (in Table 3.1). However, the differences in worth between should be increased, as three 'slight' accident should seemingly be less critical compared to a 'fatal' accident. Therefore, the values for each of them should be 1, 3, 9 respectively. This is bias to interpretation and should be adjusted in future by experts with more familiarity with these issues.

Once all the preprocessing steps have been completed, the preprocessed data will need to be passed to the next stage in the pipeline. In Apache Airflow, and data and MLOps pipelines in general, when creating Directed Acyclic Graphs (DAGs), the individual 'tasks' that make up the pipeline should operate independently of other 'tasks'. This is to reduce system coupling, and consequently make pipelines more flexible and easier to manage. Figure 3.10 visually shows how low coupling and high cohesion can make

systems more robust yet also more flexible. Data should not be directly passed between tasks. Instead, the data should be passed to a temporary data store where the next 'task' can access the necessary data. A NoSQL key-value store called Redis (Redis Ltd, 2024) would suit this purpose. Redis is an in-memory data store which is engineered for speed. Due to its nature of being a key-value store, it can store both structured and unstructured data.

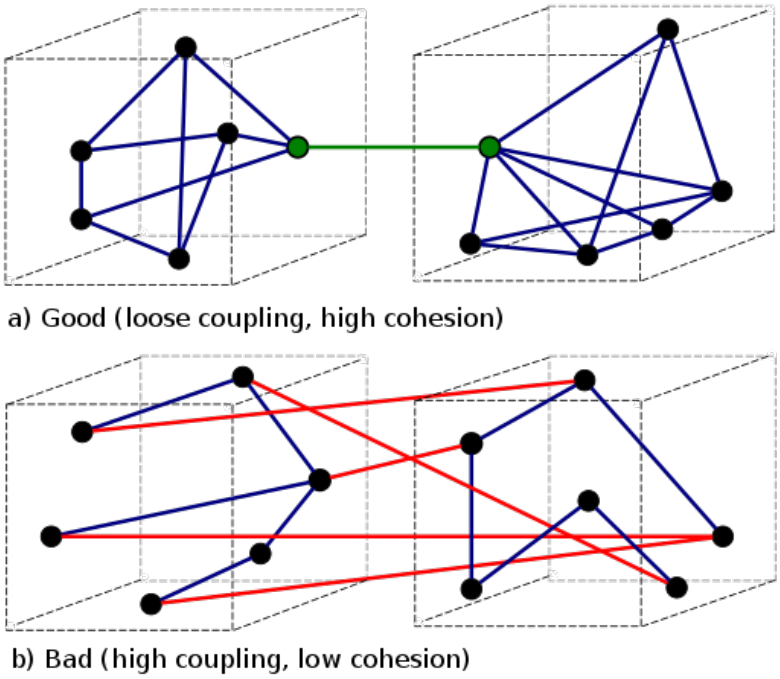


Figure 3.10: Example illustration of high and low coupling and cohesion.

Source: Sabag (2021).

3.3.2.3 Model development stage

As discussed in Section 2.2.3.2, the LSTM neural network will be made up of an input layer, hidden layers and an output layer. Within the hidden layer, there should be a LSTM layer which will be connected to a fully connected layer, which will connect to the output layer. This is illustrated in Figure 3.11, using JGraph Ltd (2023). The LSTM layers will determine what historic information should be passed to the fully connected layer, which will attempt to analyse the trends and patterns from the information it was given. Table 3.2 details the base model hyperparameters that will be used; which have

been decided through empirical experience. In the improvement stages of the implementation, these hyperparameters will be experimented with and adapted to attempt to achieve better results.

Table 3.2: LSTM neural network base hyperparameters.

Hyperparameter	**Importance**	**Default value**
Epochs	An epoch is one complete pass through the training dataset. Weights and bias are adjusted after each epoch, ready for the next epoch of training.	5
Number of classes (for classifier)	The number of classes the classifier will have available to categorise its predictions.	5
Validation split	How much of the training set should be separated and treated as testing data for better model refinement, and to help reduce overfitting.	0.2 (20% of the training set)
Optimiser	An algorithm that alters the weights and bias after each epoch in an attempt to minimise Log Loss.	Adam
Batch size	This is used to determine how many samples of the training dataset are passed to the neural network at once.	128
Optimiser learning rate	A measure of how much the optimiser should update weights and bias after each epoch. The lower the value, the better the learning rate. A lower value increases the likelihood of finding the lowest possible global minimum gradient, but can largely increase training time.	0.001

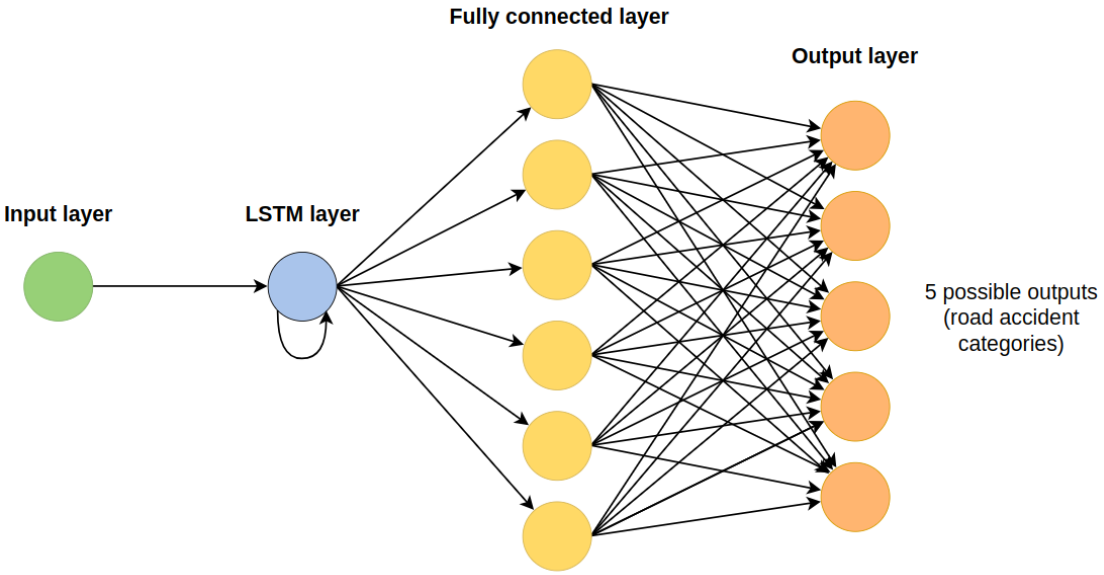


Figure 3.11: LSTM neural network implementation design illustration.

PyTorch will be used to construct and train the neural network (with the reasoning discussed in Section 3.2). Additionally, to improve flexibility and code readability, PyTorch Lightning (Lightning AI, 2024) will be used. It may be necessary to train the neural network outside the pipeline should current hardware limitations cause crashes or bugs. Should this occur, the model should be trained separately and loaded during the Model Development stage instead of created.

Regarding the implementation of ordinal classification, a specialised method that can be used is Conditional Ordinal Regression for Neural Networks (CORN) (Shi, Cao and Raschka, 2021), implemented as a Python package by Raschka Research Group (2022). CORN stands out from other previous attempts at ordinal classification primarily due to how it deals with the issue of rank inconsistency. Consistent Rank Logits (CORAL) (Cao, Mirjalili and Raschka, 2020) is also another implementation that mitigates rank inconsistency, but leads to another issue called weight-sharing. CORN addresses both of these challenges.

3.3.2.4 Model monitoring stage

Once a model has been developed and tried, it should be logged with a platform called

MLflow (Series of LF Projects, LLC, 2024). This includes its parameters and associated data. MLflow is a popular choice for tracking the progress and organising developed models, particularly for responding to issues such as data drift and model drift when it comes to dynamic data. MLflow is compatible with Python. Figure 3.12 illustrates an example workflow.

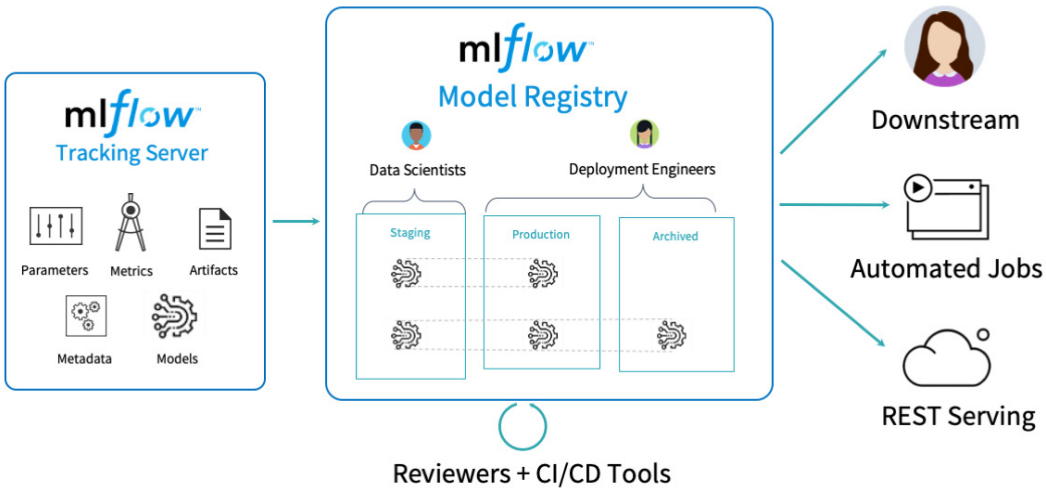


Figure 3.12: MLflow example workflow.

3.3.3 Predictions pipeline

3.3.3.1 Retrieving required live data

What live data that will need to be retrieved will depend on the feature selection process for variables relevant to predictions on a borough-by-borough basis. There are generally two main ways to retrieve information from the Internet programmatically: through the use of web scraping or using available APIs. The latter option is usually more effective as it is less subject to change — websites can be updated regularly, which can make any web scraping code potentially no longer viable; but APIs usually do not have such regular or drastic changes to their API routes. Using APIs are also more straightforward.

Based on preliminary viewing of the data (such as that shown in Figure 2.3, online services by OpenWeather (2024), SunriseSunset.io (2024) and TimeAPI (2024) offer free APIs that can provide detailed environmental information together, including weather,

rainfall, light conditions and exact time of several regions. For making requests to these APIs, a Python library called Requests can be used to make HTTP requests to certain Uniform Resource Locators (URLs) (which is how most online APIs operate). Figure 3.13 shows a straightforward example of this process.

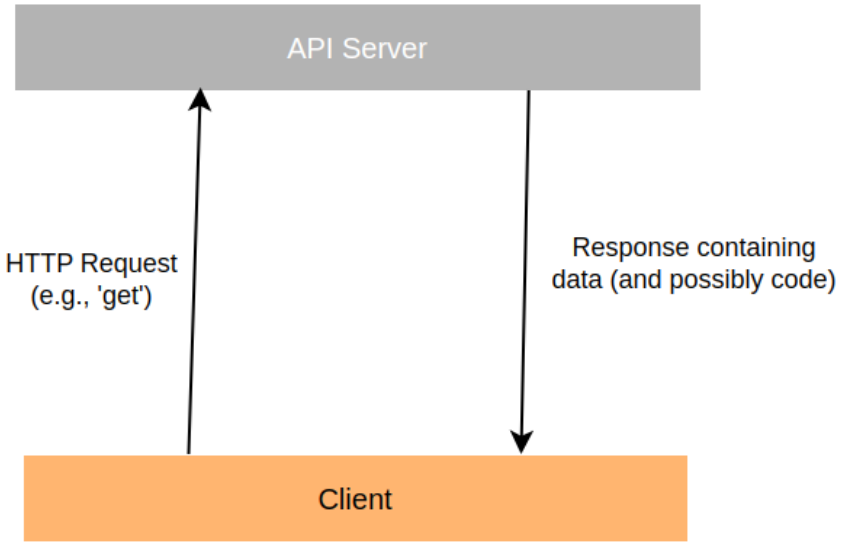


Figure 3.13: Example of an API Request with a response. Made using JGraph Ltd (2023).

3.3.3.2 Making and storing predictions

Once the live data has been retrieved, it can be fed into a model registered with MLflow to make predictions. These results should be provided to the web app (via the web API) in the form of JavaScript Object Notation (JSON). JSON is often used for light data interchanging due to its structured dictionary-like format and its natural JavaScript support.

For each set of generated predictions, they should be saved in a storage location so that the predictions can be analysed over time, allowing for further monitoring of model drift. A new ClickHouse table can be created to store these predictions, in a One Big Table format, to allow for quicker data analytics and ad-hoc querying. The features that will be stored will be similar to those finalised after the EDA process after the data has been ingested, except there will be additional columns for the prediction results and the time the predictions were made.

3.3.3.3 Use case: emergency services operations room

Besides evaluating the final model's performance, it would be useful to demonstrate how might the model be leveraged in a real-world setting — such as in an emergency services operations room (mentioned in Section 3.3.1). In these operation rooms, they often have maps available which provide them information about incidents and critical updates as part of their ongoing workload. Maps and camera footage are also shown on larger screens where applicable — so operators are made aware of important information. A web app and web API can be designed to somewhat mimic an area-wide map of Greater London, which can then be used to display the model's predictions for each borough.

Figure 3.14: Bristol Emergency Operations Centre.

Source: Bristol City Council (2024).

According to a review by Johns (2024), a popular combination of frameworks and libraries for full-stack web development is MongoDB, Express, React and NodeJS (MERN). MongoDB in this case would be replaced with ClickHouse. Express (OpenJS Foundation, 2024a) is a JavaScript framework that can be used for constructing server-side applications and APIs, while offering flexibility, useful utilities such as routing and middleware, and extremely fast response times. React (Meta OpenSource, 2024) is a JavaScript library that can be used for making web-based or mobile-based dynamic and interactive user interfaces. NodeJS (OpenJS Foundation, 2024b) is a runtime environment specialised and optimised for JavaScript. Node Package Manager (NPM) (GitHub, 2024) is

used in conjunction to NodeJS for importing useful JavaScript packages, frameworks and libraries.

In addition to the necessary software mentioned, TailWindCSS (Wathan et al., 2024), a utility-first framework for Cascading Style Sheets (CSS), will be used to help accelerate the development of the web app. TailWindCSS removes much of the complexity and organisational issues that come with implementing CSS, making development quicker and more simple.

Figure 3.15 shows an example of what the final web app may look like. This figure was made using Figma (Figma Inc, 2024).

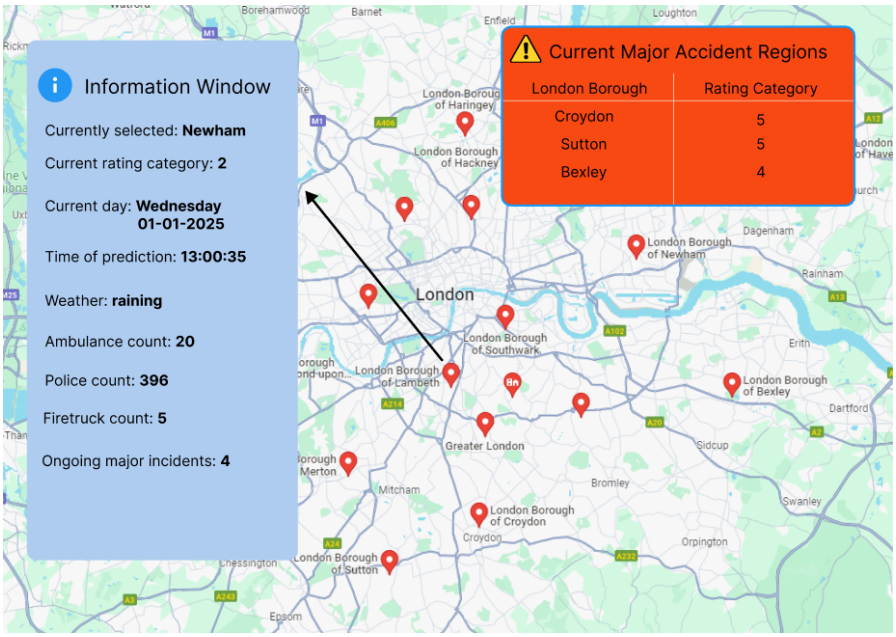


Figure 3.15: Example design of the final web app implementation.

3.3.4 Tools summary

Table 3.3 summarises the tools required and their usage for the implementation of this design specification.

Table 3.3: Summary of tools and packages required for the implementation.

Tool	**Usage**
Conda (Anaconda)	For managing installed packages and versions, while keeping them isolated from other environments.
Apache Airflow	For creating and managing the two implementation pipelines.
Apache Airflow	An OLAP data warehouse that will be used for storing the ingestion data and the final model predictions.
Dask	For data manipulation and analysis purposes during the Preprocessing and Model Development stages. It is more optimised for in-memory storage compared to Pandas due to its lazy loading capabilities.
Dask ML	A library containing functions similar to SciKit Learn, but it is designed specifically for Dask dataframes. This will be used for data manipulation purposes.
Docker	A containerisation platform that will be used for creating container instances of a ClickHouse server, MariaDB server, and a Redis store.
Docker Compose	For constructing an all-in-one file that can be run and subsequently initialise or start the required Docker containers.
Express (NodeJS)	A JavaScript framework that will be used to create the web API.
Jupyter Notebook	A platform designed for working with IPython Notebooks — which will be used for code development, and EDA for the Ingestion and Preprocessing stages.
Matplotlib and Seaborn	Python data visualisation packages that will be used for visualising data as part of the EDA.
MLflow	A platform specialised for logging and tracking developed models. This will be used to help monitor any models developed, particularly for things such as model drift and data drift.
NodeJS	A server dedicated to running JavaScript. This will be used to help host the web app and web API.

Continuation of Table 3.3	
Tool	**Usage**
PandaHouse	For importing and exporting tabular data to and from the ClickHouse server instance.
PyTorch	A Deep Learning framework that will be used to construct an LSTM neural network and model.
PyTorch Lightning	A Deep Learning framework which serves as a sort of add-on to PyTorch, which can make code development and code organisation more straightforward and standardised.
React (NodeJS)	A frontend JavaScript library that will be used to construct a dynamic web app.
Redis	An in-memory datastore that will be used to transfer data between Airflow tasks.
Requests	A Python package that will be used to retrieve API information from the Internet for the live predictions pipeline.
TailWindCSS	A CSS utility-first framework that will be used for accelerating the development of the web app, while better organising and simplifying CSS.
Node Package Manager (NPM)	A JavaScript package manager for NodeJS. This will be used to retrieve JavaScript packages, frameworks and libraries.
Pandas	A data manipulation and analysis library that will be used during the Ingestion stage to help ingest the data.
End of Table	

3.3.5 File hierarchy and overall implementation illustration

Figure 3.16 below depicts the file hierarchy of the implementation. Additionally, Figure 3.17 shows a block diagram of the pipelines and the sequence of tasks.

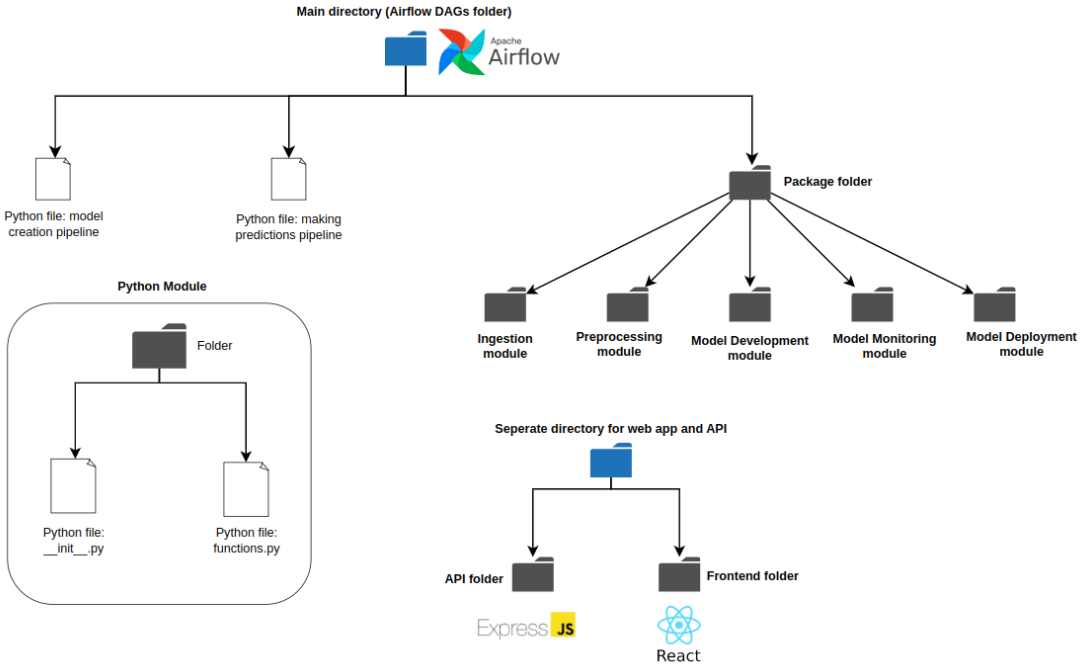


Figure 3.16: Implementation file hierarchy.

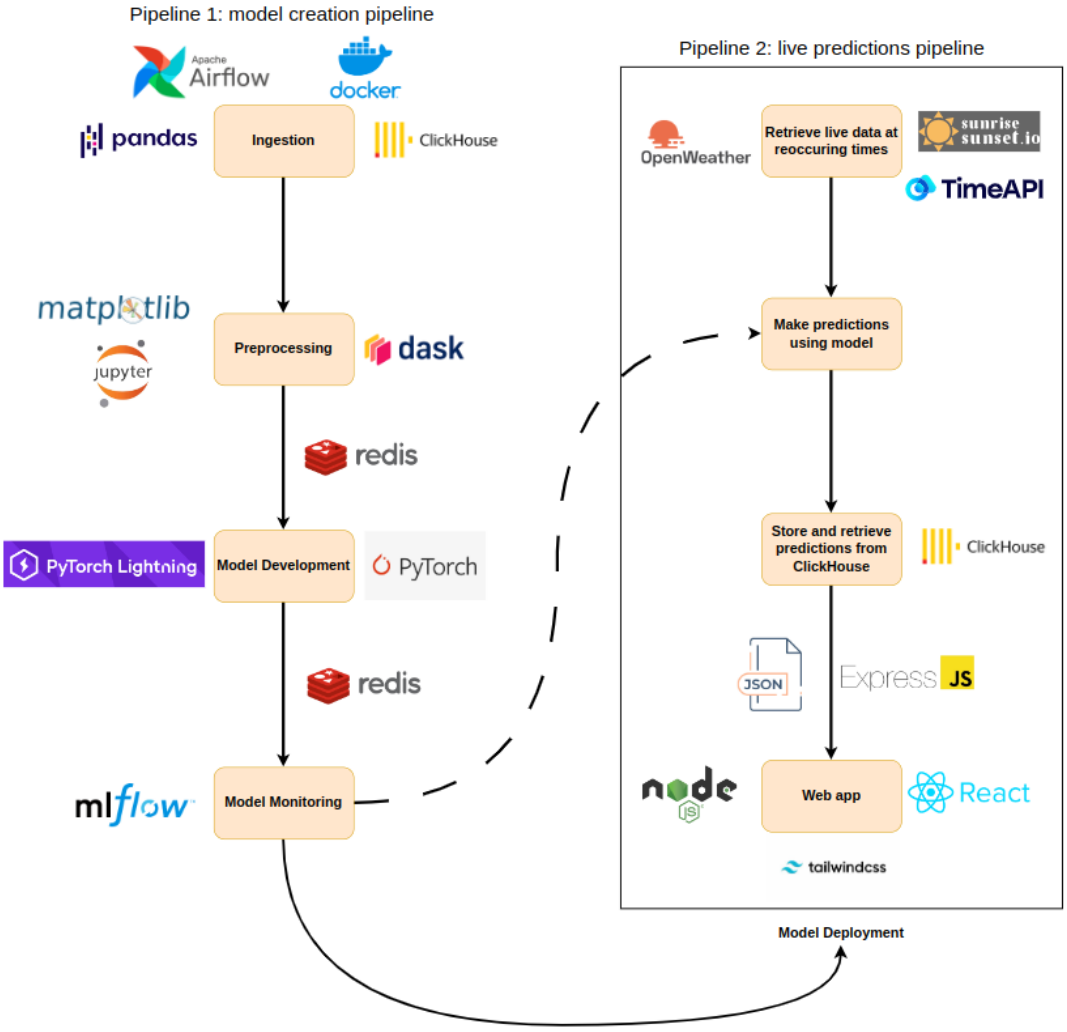


Figure 3.17: Implementation block diagram.

3.4 Testing and Evaluation

Testing the final solution to prove functionality will involve the successful running and completion of three parts: the model creation pipeline, the live predictions pipeline, and having the predictions being conveyed through the web app.

For evaluating how effective the solution will be, the focus should be on the pipelines and final model effectiveness — as these are the main components of the solution, while the web app and API are meant primarily for demonstration purposes. According to LinkedIn Community (2024), some key metrics for helping evaluate pipelines:

1. Data volume: how much data the pipeline can handle at a given time, or overtime.
2. Data latency: the time difference between getting the data from its source to its destination.
3. Data loss: the percentage of data unintentionally lost as it transverses the pipeline.
4. Time efficiency: how long the overall pipeline and its individual stages take to successfully complete.
5. Resource utilisation: how much system resources are taken up by the pipeline.
6. System uptime: the percentage of time the pipeline is operational.

From the above list, items 4 and 5 will be prioritised.

Evaluating deep learning typically involves measuring its Log Loss. Log Loss is a measure of how well the model has generalised to the data.

Regression metrics such as mean absolute error (MAE), macro-averaged MAE, mean absolute percentage error, mean squared error (MSE) and root mean squared error (RMSE), are often used for ordinal classification tasks (Baccianella, Esuli and Sebastiani, 2009, p. 284) as they can capture the significance of wrong predictions. R^2 Score is also an often-used regression metric, but Hosmer, Lemeshow and Sturdivant (2013, p. 164) advise against using it as it does not truly represent a goodness-of-fit. R^2 Score represents the comparisons between the predicted values from the fitted model to those from the base model. They argue only a metric which tests the observed and predicted values is a true measure of fit, such as the other mentioned metrics. If a confusion matrix was used to evaluate ordinal classification, it would not capture the significance of the degree a predicted value was incorrect compared with its true value (e.g., the difference between 1 and 4 is bigger than the difference between 1 and 2).

In Section 3.3.2.3, should the neural network need to be trained outside the pipeline, such as using a GPU-supported runtime on Google Colab (Google, 2024), the time it takes to train the neural network should be added to the time it takes for the Model Development stage to load and process the model.

Table 3.4: Planned implementation evaluations.

Evaluation ID	**Evaluation target**	**Evaluation metric**
1	Model creation pipeline	The time it will take overall for the individual pipelines to complete. In other words, how long it takes the pipeline to create and register a model.
2	Live predictions pipeline	The time it will take overall for the individual pipelines to complete. In summary, how long does the pipeline take to make all necessary predictions.
3	Model creation pipeline & live predictions pipeline	How much of available system resources are used by the individual pipelines.
4	Developed model	The Log Loss and accuracy of how well the model performs against the testing set.
5	Developed model	Whether the model has underfitted or overfitted to the training set.

Learning curves (as demonstrated in Figure 3.18) can be plotted to determine with a high probability whether the model has under or overfitted to the training set.

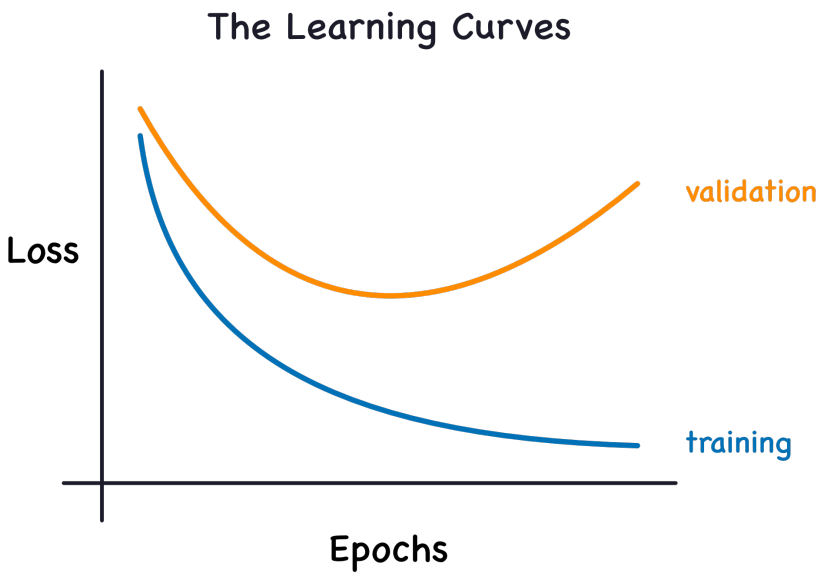


Figure 3.18: Example learning curve of a model overfitting to the training set.

Source: Ibrahim (2024).

3.5 Summary and Next Steps

In summary, two Apache Airflow pipelines will be constructed; one for model creation and one for the likelihoods of live RTAs with predictions of a certain severity. The results of live predictions and model effectiveness will be presented via a web API and web app.

The next steps, the first implementation steps, will be to start preliminary analysis of the data discussed, and subsequently program the aspects of the ingestion stage.

4 Implementation

This section will carry out and document the implementation discussed in Section 3, and where appropriate, log any changes to the strategy or slight alterations compared to following the plan.

4.1 Data Ingestion

4.1.1 Setup

This stage assumes both Docker version 27.0.3 and Conda version 23.7.4 are installed.

4.1.1.1 Creating a Conda environment

Table 4.1 details the packages that will be installed during the Conda environment creation. When installing packages from a Conda environment, there are three main channels: Anaconda, Conda Forge, and Python Package Index (PyPI). PyPI packages can be installed using Python PIP. Packages should be preferably installed via Anaconda or Conda Forge as these are designed to be compatible with Conda environments, while PyPI packages are not necessarily coded to have this compatibility. Figure 4.1 demonstrates the creation of the initial Conda environment.

Table 4.1: Initial Ingestion stage Conda environment packages.

Package	**Channel**	**Preferred Version**
Apache Airflow	Anaconda	2.4.3
ClickHouse SQLAlchemy	Anaconda	0.2.3
Dask	Anaconda	2023.11.0
Dask ML	Conda Forge	2023.3.24
Jupyter (Notebook)	Anaconda	1.0.0
MySQLClient	Anaconda	2.0.3
Python	Anaconda	3.10
Pandas	Anaconda	2.1.4
PandaHouse	PyPI	0.2.7
SQLAlchemy	Anaconda	1.4.51

```
(base) alex@alex-SATELLITE-PRO-C50-H-11J:~$ conda create --name fyp_env python=3.10 \
> sqlalchemy=1.4.51 pandas=2.1.4 clickhouse-sqlalchemy=0.2.3 airflow=2.4.3 jupyter=1.0.0 \
> mysqlclient=2.0.3 dask=2023.11.0
```

(a) Creating an environment called 'fyp_env'.

```
Downloading and Extracting Packages

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate fyp_env
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) alex@alex-SATELLITE-PRO-C50-H-11J:~$ conda activate fyp_env
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$
```

(b) Activating the environment.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ conda install -c conda-forge dask-ml=2023.3.24
```

(c) Installing Dask ML via Conda Forge.

```
alex@alex-SATELLITE-PRO-C50-H-11J: ~
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ pip install pandahouse==0.2.7
```

(d) Installing PandaHouse using PIP.

Figure 4.1: Creating the initial Conda environment.

4.1.1.2 Docker containers: ClickHouse server, MariaDB, Redis store

Table 4.2 describes common Docker Compose commands used for running compatible files, such as YAML (data serialisation langauge) files.

Figure 4.2 shows the successful start-up of the required Docker containers. The code for creating the containers, with comments, can be found in Appendix C.1.

Table 4.2: Docker Compose commands and their usage.

Command	**Usage**
docker compose –file *filename* up –wait	This will run the contents of a compose setup file by creating the relevant containers. '–wait' prevents the code progressing further unless the services are running or are healthy.
docker compose –file *filename* down	This will stop and remove all containers and networks described within the file.
docker compose –file *filename* start	This assumes the containers and network have already been created. It will resume the operations of these instead of trying to create new containers or a new network.
docker compose –file *filename* stop	This assumes the containers and network have already been created. It will stop the operations of these, but it will not remove them.

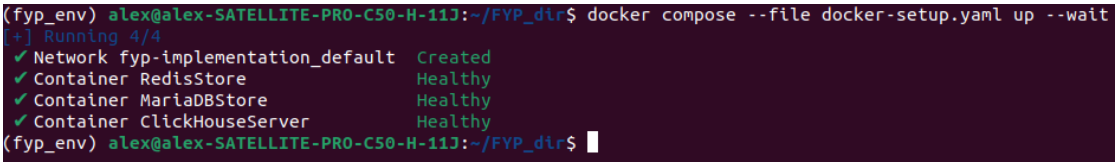


Figure 4.2: Successfully starting required Docker containers.

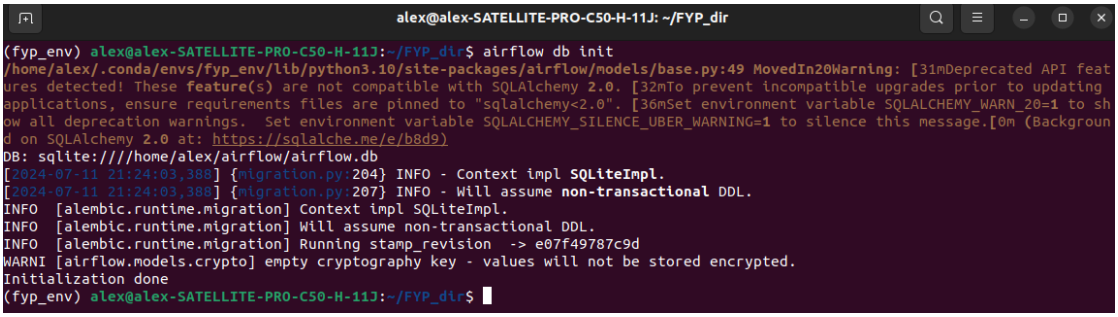
4.1.1.3 Apache Airflow setup

As discussed in Section 3.3.2.1, for Apache Airflow to operate to a production standard it requires a metadata storage like MariaDB, instead of a SQLite database. The setup of this database on the MariaDB server instance has already been done through the Docker Compose setup (Appendix C.1 and Figure 4.3).

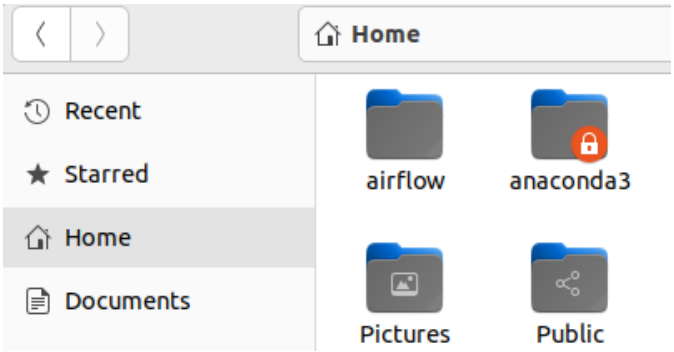
```
43      environment:
44        MARIADB_USER: AirflowUser12233
45        MARIADB_PASSWORD: AirflowPassword11223
46        MARIADB_DATABASE: airflow_metadata_db
47        MARIADB_RANDOM_ROOT_PASSWORD: 1
```

Figure 4.3: Code that creates a database in MariaDB for Airflow.

The first step is to initialise a folder for Airflow - which will contain the setup files needed to be reconfigured - which is shown in Figure 4.4. The folder is always created in the user's home directory. Subsequently, the configuration file will need to be updated with the connection string to the MariaDB database that leverages MySQL Connectors, as shown in Figure 4.5.

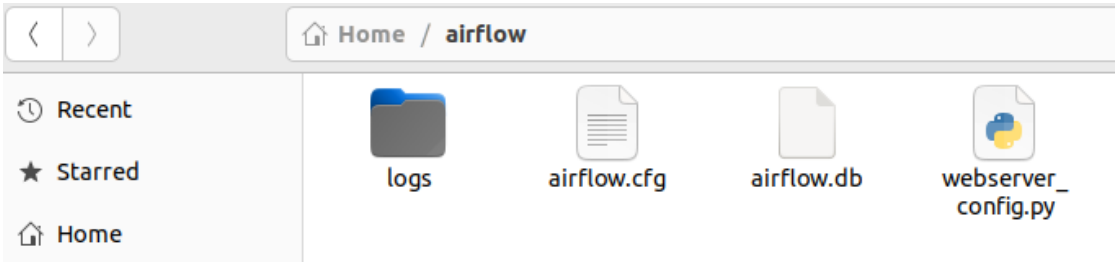


(a) Creating the Airflow folder.



(b) Confirming the successful creation of the folder.

Figure 4.4: Creating the Airflow directory.



(a) View of the Airflow directory.

```
# The executor class that airflow should use. Choices include
# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``, ``DaskExecutor``,
# ``KubernetesExecutor``, ``CeleryKubernetesExecutor`` or the
# full import path to the class when using a custom executor.
executor = LocalExecutor
```

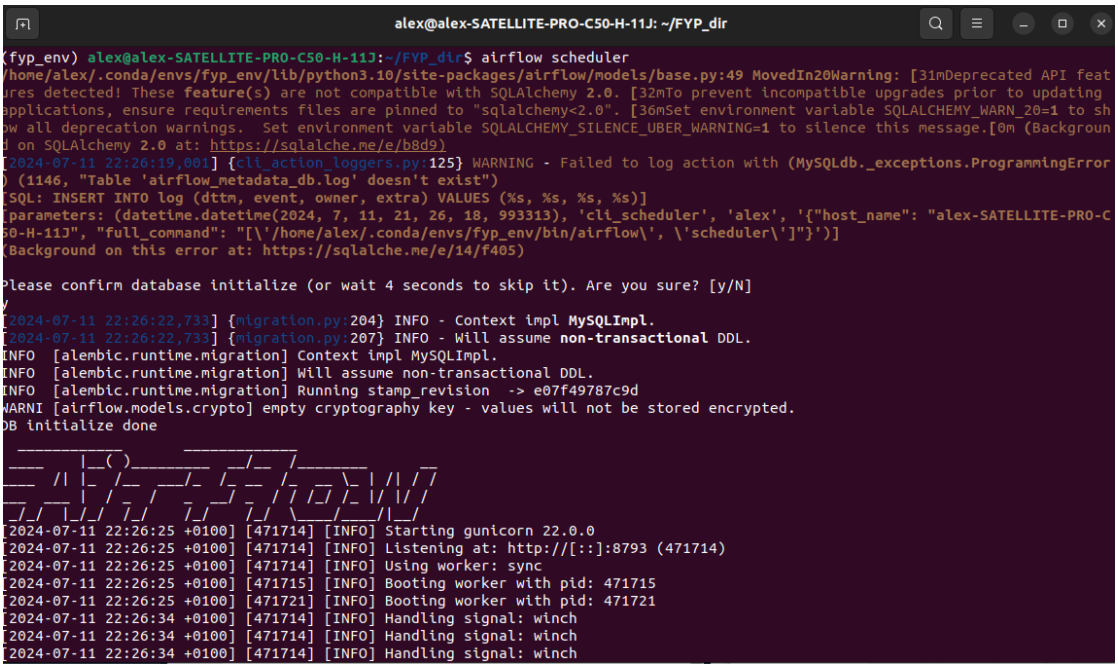
(b) Updating the 'executor' from 'SequentialExecutor' to 'LocalExecutor' in 'airflow.cfg'.

```
[database]
# The SqlAlchemy connection string to the metadata database.
# SqlAlchemy supports many different database engines.
# More information here:
# http://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html#database-uri
sql_alchemy_conn = mysql+mysqldb://AirflowUser12233:AirflowPassword11223@127.0.0.1:3306/airflow_metadata_db
```

(c) Providing the connection string to the MariaDB database via the 'sql_alchemy_conn' parameter; in 'airflow.cfg'.

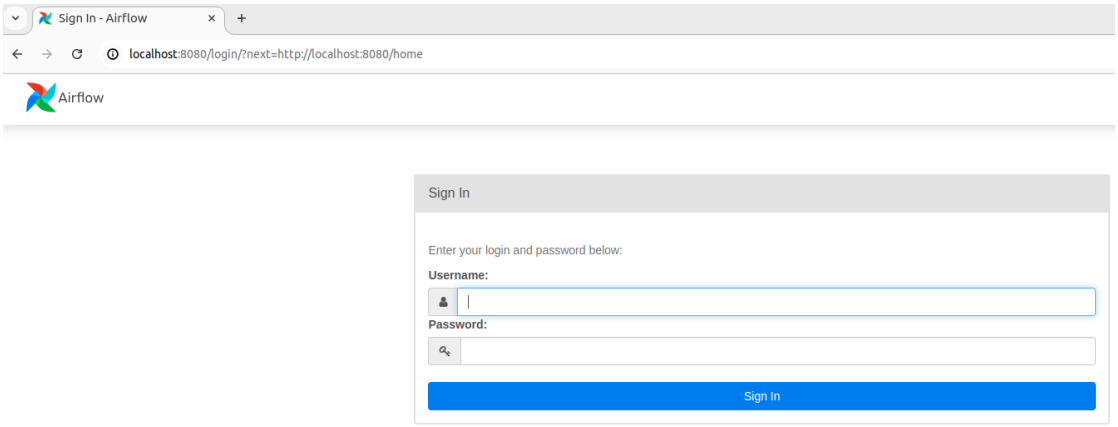
Figure 4.5: Connecting Airflow with a MariaDB database.

The next step will be to start up a Scheduler instance to manage and update Airflow tasks (e.g., jobs, DAGs, parameters, etc.). Additionally, Airflow supports a graphical user interface, which can be started using an Airflow web server instance. This is shown in Figure 4.6.



(a) Starting an Airflow Scheduler instance, and confirming the database setup.

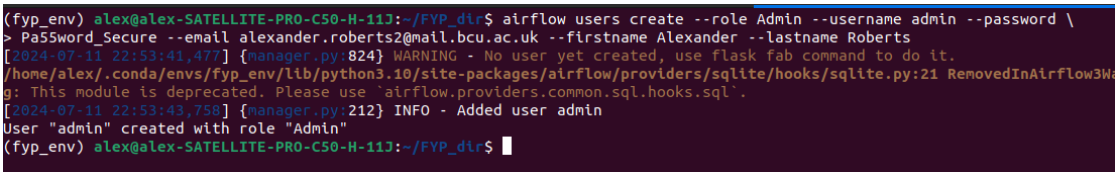
(b) Creating an Airflow web server instance.



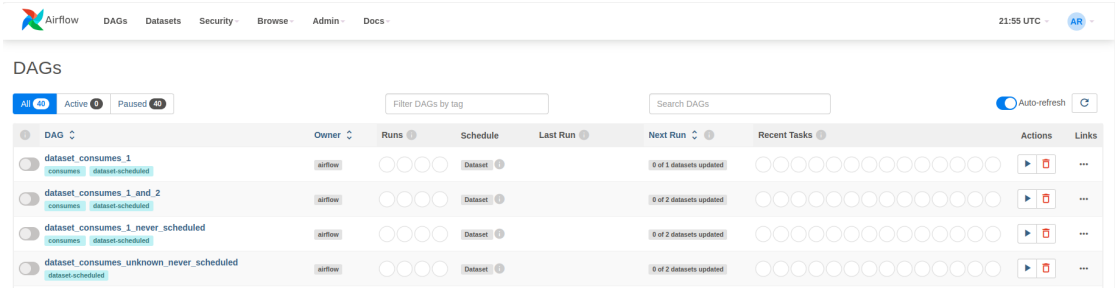
(c) Confirming Airflow is operating correctly.

Figure 4.6: Starting Apache Airflow.

The final step will be to create an Admin user to access and manage Airflow, which is shown in Figure 4.7.



(a) Creating an admin user.



(b) Logging in with the new admin user created.

Figure 4.7: Logging in and viewing the Airflow DAG dashboard.

4.1.2 Data wrangling

As mentioned in Section 3.3.2.1, data wrangling is likely required to help ingest the datasets, particularly by looking at and schema-level issues (e.g. column naming) and data-level problems (e.g. data type consistency).

Table 4.3 summarises the manual adjustments made to specified files using LibreOffice Calc. These were done manually rather than programmatically due its straightforwardness.

Table 4.3: Manual file data adjustments.

Dataset	**Adjustment(s) made**
GDP	Removed row 1 so when the data is loaded using Pandas and Dask, the column names are set correctly.
Population	Removed row 1 so when the data is loaded using Pandas and Dask, the column names are set correctly.
Traffic flow	Removed rows 2 and 36, as these are blank rows formed as a side effect of transforming the original LibreOffice Calc sheet into a CSV file. Renamed certain columns, such as '2000 [note 6]' to remove the '[note 6]' text.
RTA TfL	Removed 'ADATES_FULL' column from 2019 and 2020 datasets (as this was the combination of the date and time of each entry, but it is not needed as the date and time fields are still present). Deleted the right-most column in the 2005 dataset as this column did not actually have any data, but a whitespace. Corrected spelling mistake of the column name of 'Light Conditions' to 'Light Conditions' in the datasets for 2019 and 2020. In the datasets for 2021, 2022, and 2023, '_Collision Id' was changed to 'Accident Ref', for consistency purposes. Data was shifted for a few entries to the left adjacent column, in the 2005 dataset; indices 18876, 23813 and 25795. This was also done in the 2018 dataset, index 24322. This was needed because of the data of these rows were in the wrong columns (likely due to human error).

After making these changes, ingesting the additional datasets worked correctly without additional transformations (as shown in Appendix D.1). The TfL RTA datasets require

further transformations, which will be done programmatically as the data types need to be updated. These transformations include updating the column names of the various years of dataset files so they are consistent on the scheme-level, and updating certain columns to a more suitable data types (e.g., the collision date should be a 'datetime' type). Figure 4.8 shows the code used to perform these transformations of the Dask DataFrame.

```
# remove basic outliers
# every entry should have a Borough name with an associated accident severity for future plans
ddf = ddf[ddf['Borough'].notnull() & ddf['Accident Severity'].notnull()]
```

(a) Basic outlier removal.

```
# a pre-defined list of column names to be replaced
columns_data_to_replace = ['Speed Limit', 'Accident Severity', 'Junction Detail',
                           'Junction Control', 'Pedestrian Crossing Facilities', 'First Road Class', 'Second Road Class', 'Road Type',
                           'Light Conditions', 'Weather Details', 'Road Surface Condition', 'Special Conditions at Site', 'Carriageway Hazards',
                           'Highway Authority', 'Collision Location Details', 'Place Collision Reported']

# the below code removed numbers such as '3' from '3 30 MPH', making the result '30 MPH'
for col in columns_data_to_replace:
    ddf[col] = ddf[col].str.replace(r'^\d\s*|^-\d\s*', '', regex=True)
    # use regular expression (regex) to select prior number to actual data and remove them
    # '^' means at the beginning of the string, '\d' means if a number is present
    # '\s' means any whitespace character, '*' means however many - so however many whitespaces
    # '|' is the equivalent of OR - so if the string matches either expression, the former for positive numbers, the latter for negative numbers
```

(b) Column name replacement.

```
# some of the dates are in slightly different formats
ddf['Collision Date'] = ddf['Collision Date'].str.replace('/', '-')
ddf['Collision Date'] = ddf['Collision Date'].str.replace(" 00:00", '')

# the below is needed to ensure the dataframe types are consistent with its intended table schema
ddf['Collision Date'] = to_datetime(ddf['Collision Date'], format = "mixed", dayfirst=True)
ddf['Attendant Count'] = ddf['Attendant Count'].astype(float) # must be converted to float due to containing 'NaN' values
    # 'NaN' values are classed as floating point values
ddf['Casualty Count'] = ddf['Casualty Count'].astype(int)
ddf['Vehicle Count'] = ddf['Vehicle Count'].astype(int)
ddf['Borough Number'] = ddf['Borough Number'].astype(int)
```

(c) Updating column data types to be more representative.

Figure 4.8: Ingestion stage programmatic data wrangling.

4.1.3 Ingestion

Figure 4.9 is a sample of the control flow file for the Ingestion stage. In essence, this control flow consists of five mini task transition pipelines, where simply the data is read and stored directly into the respective ClickHouse tables (which have been custom designed for each one). The advantage of the structure and repetition of this improves predictability and reduces overall complexity.

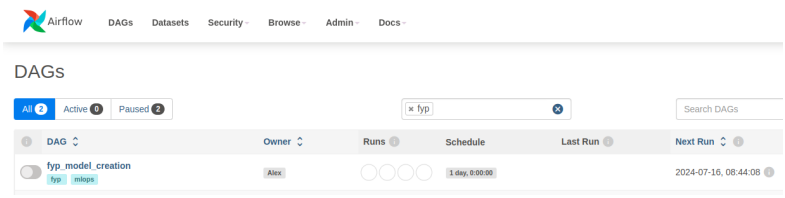
```
try:
    create_clickhouse_tables(engine_conn=engine_conn,
                             road_table_name=london_acc_table_name,
                             traffic_flow_table_name=traffic_flow_table_name,
                             population_table_name=population_table_name,
                             gdp_table_name=gdp_table_name,
                             crime_table_name=crime_table_name)

    # ingest Greater London road accident data
    # this data requires extra transformations compared to the other datasets,
        # so some custom functions are needed
    if not table_populated(engine_conn=engine_conn, table_name=london_acc_table_name):
        upload_ingested_data(
            preprocess_road_data(
                ingest_road_data(directory_path=london_acc_data_path)
                ),
            engine_conn=engine_conn,
            table_name=london_acc_table_name
        )

    # ingest traffic flow data
    if not table_populated(engine_conn=engine_conn, table_name=traffic_flow_table_name):
        upload_ingested_data(
            ingest_data(traffic_flow_data_path),
            engine_conn=engine_conn,
            table_name=traffic_flow_table_name
        )

    if not table_populated(engine_conn=engine_conn, table_name=population_table_name):
        # ingest population data
        upload_ingested_data(
            ingest_data(population_data_path),
            engine_conn=engine_conn,
            table_name=population_table_name
        )

    if not table_populated(engine_conn=engine_conn, table_name=gdp_table_name):
        # ingest GDP data
        upload_ingested_data(
            ingest_data(gdp_data_path),
            engine_conn=engine_conn,
            table_name=gdp_table_name
        )
```

Figure 4.9: Sample of the control flow function for the Ingestion stage.

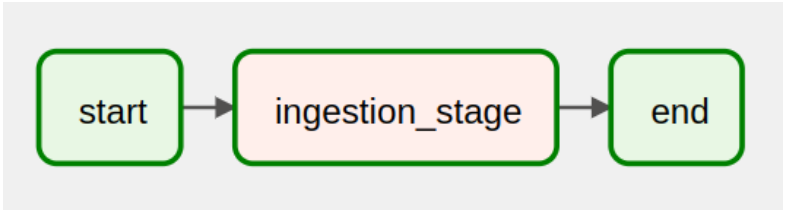
4.1.4 Pipeline after Ingestion stage

Appendix C.2.1 contains the code for the Ingestion stage 'functions.py' file, which is where the functions for the Ingestion stage are stored (as indicated in Figure 3.16). Appendix C.2.2 contains the '__init__.py' file which controls the program flow of the Ingestion stage. Finally, Appendix C.2.3 contains the code of the main pipeline with the

Ingestion stage updated. Figure 4.10 shows the Ingestion stage of the pipeline operating successfully. Appendix E.1 shows the tables in the ClickHouse server after the Ingestion stage.



(a) Model creation pipeline on the DAG dashboard.



(b) DAG graph.

Figure 4.10: Successful DAG run of the Ingestion stage.

4.2 Data Preprocessing

4.2.1 Additional environment packages

Table 4.4 details the needed additional packages that will be used for the Preprocessing and Model Development stages. Figure 4.11 shows these steps.

Table 4.4: Preprocessing and Model Development stages; additional Conda environment packages.

Package	**Channel**	**Preferred Version**
CORAL-CORN	PyPI	1.4.0
Matplotlib	Anaconda	3.8.0
PyTorch	PyTorch (PyTorch has its own dedicated channel)	2.2.2
PyTorch Lightning	PyPI	2.3.3
Redis	PyPI	5.0.4
Seaborn	Anaconda	0.12.0

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ conda install pytorch=2.2.2 cpuonly -c pytorch
```

(a) PyTorch installation.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ conda install lightning=2.3.3 --channel conda-forge
```

(b) PyTorch Lightning installation.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ conda install matplotlib=3.8.0 seaborn=0.12.0
```

(c) Installation of Matplotlib and Seaborn.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ pip install redis==5.0.4
```

(d) Redis installation.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ pip install coral-pytorch==1.4.0
```

(e) Installation of CORAL-CORN for Ordinal Regression.

Figure 4.11: Installation of required packages for Preprocessing and Model Development stages.

4.2.2 Ingested data: Exploratory Data Analysis

EDA can often become lengthy and involve numerous charts. Therefore, only the key EDA points will be discussed. Relevant graphs and charts can be found in Appendix D.2.

Train-test data leakage can occur at this stage if the data is not separated into the

training and testing sets. This is because insights gained from the EDA, and subsequent preprocessing, are influenced by the testing set, hence it is considered data leakage. A common ratio for splitting the data is "80:20", meaning 80% of the data is for training, while the remaining 20% is for testing. As the data is in the form of a time series, the data should be organised and then have the split applied. For the London recorded RTA dataset, the data can be organised by 'Collision Date'. For the other datasets, where the years are recorded as columns, some of the end columns should be removed, preferably to match the last year recorded in the London recorded RTAs dataset (as this is the main dataset). The code for retrieving the data and applying these splits can be found in Appendix C.3.4 .

Figures D.6, D.7, D.8, D.9, D.10 show the general trends of the datasets.

Based on the EDA analysis and preprocessing experimentation, several steps for preprocessing can be inferred:

- Feature selection will be required for the main RTA dataset, as many of the features do not fit the context of making predictions on a borough-by-borough basis (Figure D.11).
- Complete case analysis will be applied for removing unknown values (Figure D.20 for example), thus making columns more consistent. But this would also mean the loss of some data.
- Label encoding will be needed for encoding categorical variables into integers (for later processing by the model). One hot encoding will not be used in this case as it would greatly increase the number of dimensions of the dataset, thus possibly leading to the Curse of Dimensionality.
- The data will be need to be fused into a single dataset.
- Times of accidents (Figure D.21) will have to be converted into fixed windows. This is to make the timings more consistent, which should allow the model to distinguish the significance of the times better. It would likely be beneficial to reduce this number using time discretisation (binning). This would mean instead

of having numerous different times (e.g., 1917'), they can be binned into a certain number of categories (bins) (e.g., 19:00-20:00).

- A special function will be required to essentially take the data from the additional datasets for each year and combine them to the relevant row in the main dataset. For example, if the population count for a certain borough was 200,000 in 2015, this would have to been joined to the respective row in the main dataset.

4.2.3 Preprocessing

Due to the large number of preprocessing steps, only key actions and the overall control flow (Figure 4.12) will be discussed.

```
def preprocess_data_main(logger:logging.Logger, engine_conn:dict, london_acc_table_name:str, population_table_name:str,
                         gdp_table_name:str, traffic_flow_table_name: str, crime_table_name:str, redis_host:str,
                         redis_port:int) -> None:
    """
    Function that manages the program flow of the preprocessing stage.
    """
    try:
        upload_to_redis(
            torch_tensor_conversion(
                scale_X(
                    train_val_test_X_y_split(
                        drop_collision_date(
                            sort_values(
                                create_risk_level(
                                    label_encoding(
                                        time_discretisation(
                                            remove_duplicates_and_nan(
                                                data_fusion(
                                                    london_acc_df=date_selection_and_seperation(
                                                        create_weekday_col(
                                                            make_time_consistent(
                                                                make_light_conditions_consistent(
                                                                    remove_unknown_weather_conditions(
                                                                        standardise_borough_names(
                                                                            decapitalise_col_values(
                                                                                london_acc_feature_selection(
                                                                                    retrieve_data_from_ClickHouse(
                                                                                        engine_conn=engine_conn,
                                                                                        table_name=london_acc_table_name)
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    ),
                                                    population_df=standardise_borough_names(
                                                        rename_cols(
                                                            df=retrieve_data_from_ClickHouse(
                                                                engine_conn=engine_conn,
                                                                table_name=population_table_name),
                                                            col_rename_dict={'LA name':'Borough'}
                                                        )
```

Figure 4.12: Sample of the control flow program for the Preprocessing stage.

In summary, each dataset is collected from ClickHouse, and subsequently receive

individually-suited transformations. Afterwards, they are fused into one dataset (not all of the data from each one) and then this fused dataset receives further transformations until it is ready for the Model Development stage. This can be viewed as a programmatic pipeline. Based on the approach used, it is clear what each step is taken to transform the data (due to the function names and indentation).

However, some steps such as 'remove_duplicates_and_nan' are present after the fusion because the City of London was not included in the crime dataset — leading to missing data. This is because the City of London is not considered a London borough. Therefore, it was deemed more appropriate to use Complete Case Analysis to remove the City of London entries since there is no previous crime data for imputation.

Although not shown in Figure 4.12, a feature selection function was applied to the crime dataset to only select data that was related to vehicle offences (e.g. theft from a vehicle, speeding, drug trafficking, etc.) as these would have the biggest impact on RTA likelihoods compared to the other offences recorded. The subcategories of these can have varying importance, therefore, to avoid adding personal input bias, all of them were selected under 'vehicle offences' rather than selecting them individually.

For some of the final transformations, the data is split into the training, validation and testing sets; with a ratio of 70:10:20 (70% training, 10% validation, 20% testing). The data is organised chronologically beforehand such that the training data is the oldest entries, while the testing data will be the more recent ones. In addition, the independent features are scaled using data normalisation to a feature range of 0 to 1, just to remove the scale of the data so that the deep neural network does not interpret this as significant. Moreover, PyTorch requires its data to be converted into numerical tensors for processing. Finally, the preprocessed data is stored in Redis, ready for Model Development.

4.2.3.1 Risk level

In Figure 4.12, 'create_risk_level' and 'label_encoding' were used to construct the risk level. As discussed in Section 3.3.2.2, the numerical values for the label encoding process was 1, 3 and 9 for 'slight', 'serious' and 'fatal' respectively. Figure 4.13 shows the proportions of 'slight' to 'serious' to 'fatal' accidents. The accidents that occurred within

a certain borough on a certain date in a certain time window were aggregated together to get an overall value to represent the total "risk" of the borough at that time. For instance, if one 'slight' accident and two 'serious' accidents occurred within a time window, the "risk" value would be $1+3+3=7$. Afterwards, a function called 'categorise_risk_level' (Figure 4.14) was implemented to convert these "risk" values into one of the five ordinal categories. Initially the proportions proposed in Table 3.1 were used, by converting them into their respective numerical values (1, 3 or 9) then creating boundaries and intervals surrounding these results. The results are shown in Figure 4.15.

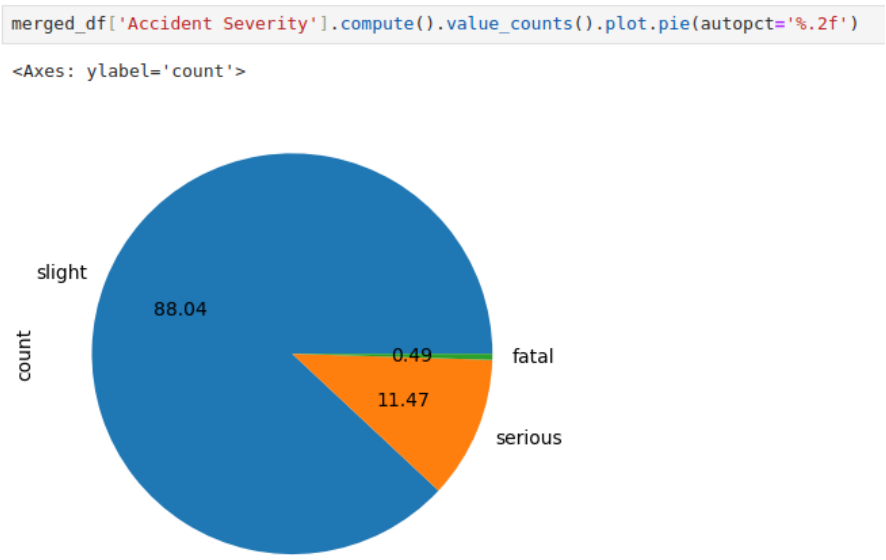


Figure 4.13: Percentage value counts of 'Accident Severity' of the ingested data.

```
def categorise_risk_level(value):
    if value value< 2:
        return 1
    elif (value >= 2) and (value < 4):
        return 2
    elif (value>=4) and (value < 7):
        return 3
    elif (value>=7) and (value < 9):
        return 4
    elif (value>=9):
        return 5
```

Figure 4.14: Categorise risk level function.

```
Risk Level
1    174567
2     42220
3      5509
5      1239
4        90
Name: count, dtype: int64
```

Figure 4.15: Initial risk category value counts.

As shown in Figure 4.15, the distribution of the target values is uneven. Unfortunately, this can lead the model being bias against the minority sets. Typically, undersampling or oversampling techniques, such as random undersampling or Synthetic Minority Over-sampling Technique (SMOTE) respectively, can be implemented to balance the target distribution. However, given that this is not a vanilla supervised learning problem, but a time-series derivative instead, these techniques might not be appropriate. If undersampling was used, a significantly large amount of data would be lost, and this could consequently distort the temporal relationship the data has. A similar case with oversampling, except instead of losing data, a considerable portion of the dataset could become synthetic. Synthetic data is often used in experimental settings, but these results can frequently become less reliable when tested against real-world data. Instead, the category boundaries will be re-evaluated to help mitigate this imbalance (as shown in Table 4.5 and Figure 4.16). This however would introduce more personal bias into the model creation.

After experimentation, 'serious' will have a value of 4 and 'fatal' will have a value of 12, 'slight' remains unchanged. Table 4.5 shows the updated boundaries for converting the risk values to risk ordinal levels.

Table 4.5: Revised borough RTA likelihood categories and their equivalent risk value boundary, based on a two-hourly basis.

Category	**Risk value boundary equivalent**
1	$value < 1$
2	$2 <= value < 4$
3	$4 <= value < 8$
4	$8 <= value < 12$
5	$value >= 12$

```
Risk Level
1    174567
3     28047
2     19313
5      1324
4       374
Name: count, dtype: int64
```

Figure 4.16: Revised risk category value counts.

4.2.4 Pipeline after Preprocessing stage.

The code for the completed preprocessing task can be found at Appendix C.3, with comments and function documentation included. Figure 4.17 shows the Airflow pipeline after the Preprocessing stage.

Figure 4.17: Successful DAG run of the Preprocessing stage.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~/FYP_dir$ pip install mlflow==2.14.3
```

(a) Installation of MLflow.

```
(fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~$ pip install tensorboard==2.17.0 \
> tensorboardX==2.6.2.2
```

(b) Installation of Tensorboard.

Figure 4.18: Installation of additional packages for the Model Development and Monitoring stage.

4.3 Model Development and Monitoring

4.3.1 Further environment package additions

Table 4.6 shows some additional Conda packages needed for this stage. Figure 4.18 shows the installation of these packages.

Tensorboard and TensorboardX are mainly used in conjuction with PyTorch Lightning Trainer to help visualise results.

Table 4.6: Model Development and Monitoring additional Conda environment packages.

Package	**Channel**	**Preferred Version**
MLflow	PyPI	2.14.3
Tensorboard	PyPI	2.17.0
TensorboardX	PyPI	2.6.2.2

4.3.2 Model creation and training

Figure 4.19 shows a sample of the control flow file for the model creation and training stage.

```python
def model_dev_and_monitoring__main(logger:logging.Logger, redis_host:str, redis_port:int, mlflow_uri:str, experiment_name:str) -> None:
    """
    Function that manages the program flow of the model development stage.
    """
    try:
        # hyperparameter constants
        hyperparams = {
            'batch_size':128,
            'criterion':corn_loss,
            'epochs':5,
            'n_features':12,
            'hidden_size':64,
            'num_layers':1,
            'learning_rate':0.001,
            'num_classes':5,
            'optimiser':Adam
        }

        # retrieve the preprocessed data
        with Redis(host=redis_host, port=redis_port) as redis_conn:
            X_train:Tensor = loads(redis_conn.get("X_train_preprocessed"))
            X_val:Tensor = loads(redis_conn.get("X_val_preprocessed"))
            X_test:Tensor = loads(redis_conn.get("X_test_preprocessed"))
            y_train:Tensor = loads(redis_conn.get("y_train_preprocessed"))
            y_val:Tensor = loads(redis_conn.get("y_val_preprocessed"))
            y_test:Tensor = loads(redis_conn.get("y_test_preprocessed"))

            redis_conn.flushdb() # flush the temporary datastore now finished with the data

        data_module  = RoadAccidentDataModule(X_train=X_train, X_test=X_test,
                                              X_val=X_val, y_train=y_train,
                                              y_test=y_test, y_val=y_val, batch_size=hyperparams['batch_size'])

        neural_network = AccidentLikelihoodNetwork(n_features=hyperparams['n_features'], hidden_size=hyperparams['hidden_size'],
                                                   output_size=hyperparams['num_classes'], batch_size=hyperparams['batch_size'],
                                                   num_layers=hyperparams['num_layers'], learning_rate=hyperparams['learning_rate'],
                                                   optimiser=hyperparams['optimiser'], criterion=hyperparams['criterion'])

        # set the connection to the MLflow server instance
        set_tracking_uri(mlflow_uri)
        # set the experiment name of the model
        set_experiment(experiment_name)
        # autolog certain evaluation metrics declared in the 'AccidentLikelihoodNetwork' module
        autolog()

        # used for managing the training and development of any model
        trainer = Trainer(max_epochs=hyperparams['epochs'], enable_progress_bar=True)

        # start a new MLflow run, so it tracks
        with start_run() as run:
            trainer.fit(neural_network, data_module)

        # log the hyperparameters used for the run
        log_params(params=hyperparams, run_id=run.info.run_id)
```

Figure 4.19: Sample of the control flow file for the Model Development and Monitoring stage.

The data is retrieved from the temporary Redis store. Then, a PyTorch Lightning 'DataModule' instance is declared. In PyTorch, data has to be wrapped in a 'Dataset' class and then further wrapped in a 'Dataloader' to be leveraged by its ML algorithms. But the PyTorch Lightning 'DataModule' makes this process more straightforward and organised (as shown in Figure 4.20). A 'LightningModule' is used to construct the neural network and associated training process all-in-one (as hinted at in Figure ...). Finally, once the training begins, certain evaluation metrics will be tracked and sent to the MLflow server (which can be started like in Figure ...), along with the model and hyperparameters.

```python
class RoadAccidentDataset(Dataset):
  def __init__(self, X:Tensor, y:Tensor):
    self.X = X.float() # convert to float to ensure the dtypes are consistent
    self.y = y.float() # still necessary even if they are already floats, e.g., float32 is different to float64

  def __len__(self): # for getting the number of instances
    return len(self.X)

  def __getitem__(self, idx): # for getting a specific set of values at a certain index
    return self.X[idx], self.y[idx]

class RoadAccidentDataModule(LightningDataModule):
  def __init__(self, X_train, X_test, X_val, y_train, y_val, y_test, batch_size):
      super().__init__()
      self.batch_size = batch_size
      self.X_train = X_train
      self.X_val = X_val
      self.X_test = X_test
      self.y_train = y_train
      self.y_val = y_val
      self.y_test = y_test

  def train_dataloader(self):
      train_dataset = RoadAccidentDataset(self.X_train, self.y_train)
      train_dataloader = DataLoader(train_dataset, batch_size=self.batch_size, shuffle=False)

      return train_dataloader

  def val_dataloader(self):
      validation_dataset = RoadAccidentDataset(self.X_val, self.y_val)
      validation_dataloader = DataLoader(validation_dataset, batch_size=self.batch_size, shuffle=False)

      return validation_dataloader

  def test_dataloader(self):
      test_dataset = RoadAccidentDataset(self.X_test, self.y_test)
      test_dataloader = DataLoader(test_dataset, batch_size=self.batch_size, shuffle=False)

      return test_dataloader
```

Figure 4.20: Leveraging the PyTorch Lightning 'DataModule'.

Figure 4.21 shows how to start the MLflow server instance.

```
fyp_env) alex@alex-SATELLITE-PRO-C50-H-11J:~/FYP_dir/mlflow$ mlflow ui
```

Figure 4.21: Starting the MLflow server instance.

4.3.2.1 Preliminary model evaluation

In order to improve the model from the baseline standard, it is necessary to perform a preliminary evaluation of its performance and then make the relevant adjustments. Both the training set and the validation enable more insight into how the model developed over time. The results are shown in Table 4.7, and the learning curves are shown in Figures 4.22, 4.23, 4.24, 4.25 (auto-generated by MLflow).

Table 4.7: Base model evaluation metrics.

Evaluation metric	**Training result (4 s.f.)**	**Validation result (4 s.f.)**
(CORN) Log Loss	0.2588	0.2860
Mean absolute error	0.3092	0.4129
Mean absolute percentage error	0.1092 (10.9%)	0.1435 (12.4%)
Macro-averaged mean absolute error	0.3092	0.4129
Mean squared error	0.6110	0.8182
Root mean squared error	0.7720	0.8999

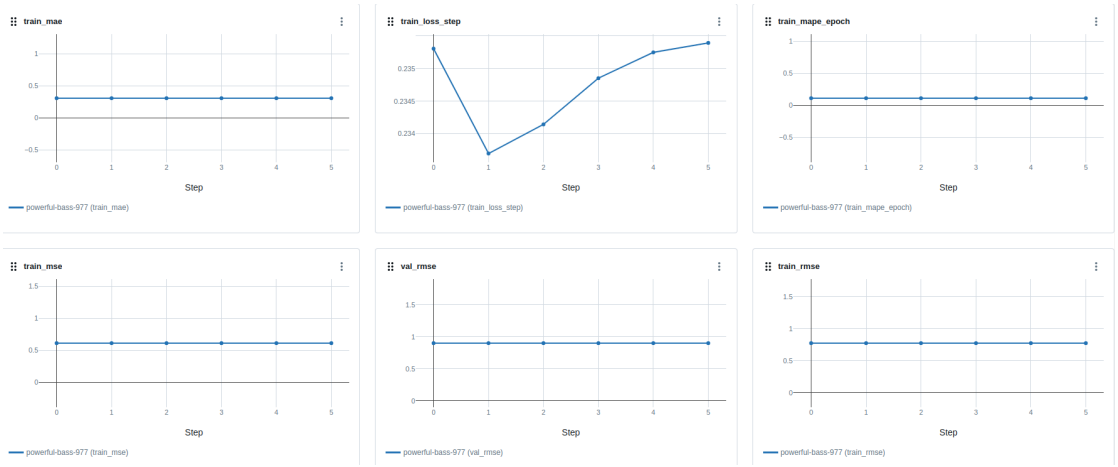


Figure 4.22: Training and validation learning curves, part 1.

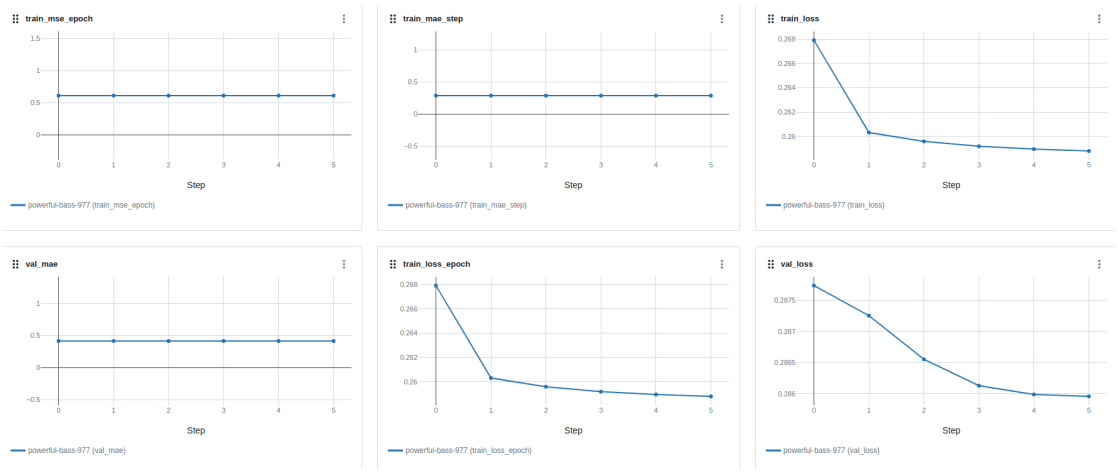


Figure 4.23: Training and validation learning curves, part 2.

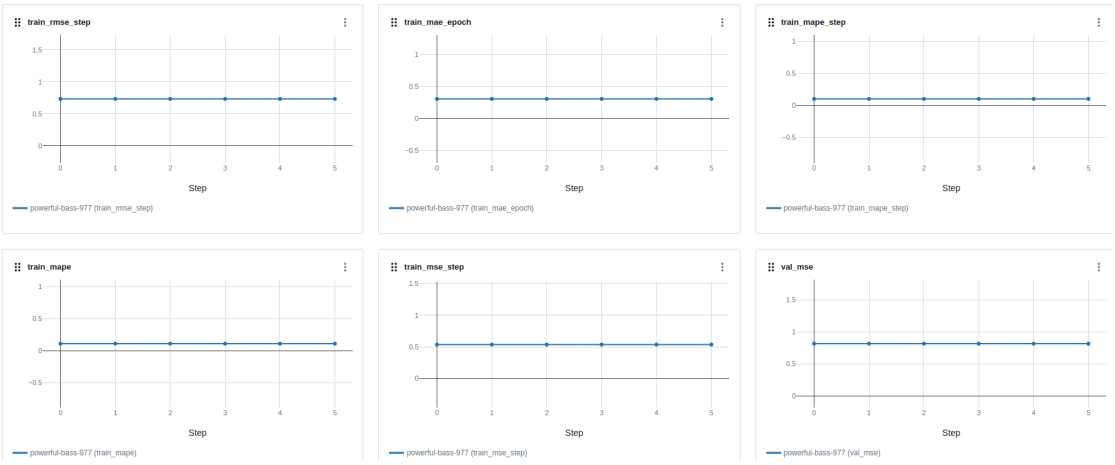


Figure 4.24: Training and validation learning curves, part 4.

Figure 4.25: Training and validation learning curves, part 4.

The results initially seemed promising, but after investigating the lack of change in most of the learning curves for the evaluation metrics not related to Log Loss, it would

seem something is not right with the model. This could be due to several influences, such as the distribution of the target variables, a lack of further information and data, a possible preprocessing oversight, or even the suitability of the dataset structure.

According to Braun (2018, p. 3) and Parsa et al. (2019, p. 1), Long Short-Term Memory (LSTM) networks are meant to work with sequences of information. The data inputted to the model in this case was not put into subset sequences, as it is an overall unnatural time sequence. This is because the dataset may have daily accident occurrences, but the specific borough and time window is what makes the sequence non-linear (as it is dependent on the date, borough and time window). However, its shape can be adjusted such that each row has its own sequence. While not conventional, it would eliminate the possible problem of sequence length invalidation. Like Z. Zhang, W. Yang and Wushour (2020), changing other hyperparameters such as the number of epochs, the number of layers in the model, batch size and learning rate can substantially improve model performance.

It appears the model has somewhat overfitted to the training data (as the training loss and error rates are less than the validation). To help resolve this, a dropout layer can be added, which adds a layer of regularisation by randomly discarding outputs of layers, forcing the model to "learn" better. This will be added between the LSTM layers.

Table 4.8 summarises the refined hyperparameters.

The code for the original neural network can be found in Appendix C.4.1 and C.4.2, while the altered neural network code can be found in Appendix C.4.3 and C.4.4.

The evaluation results for the altered neural network will be discussed in the final evaluation stage.

Table 4.8: Revised model hyperparameters.

Hyperparameter	**Value**	**Reasoning**
Epochs	10	Increasing the number of epochs enables the model to solidify and adapt what it has learnt better by consolidating more epoch results.
Optimiser learning rate	0.0001	To account for miniscule changes in the evaluation metrics, a lower learning rate can help highlight these small discrepancies better — as it can transverse more gradient minimums.
Batch size	64	Smaller batches lead to more frequent weight changes, which would be good for working in conjunction with the learning rate, as it can converge nearer to the global minimum faster. This comes with the cost of more computational resoures and time.
Dropout	0.2 (20%)	This is useful as a regularisation method to essentially force the model to have to "learn" better when some of its results are dropped.
Number of LSTM layers	2	Increasing the number of layers can enable the layers to better consolidate outputs.

4.3.3 Pipeline after Model Development and Monitoring stage

Figure 4.26 shows the successful completion of this stage in Airflow. The code for the finalised model creation pipeline can be found in Appendix C.4.5.

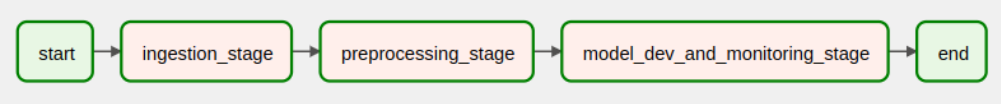


Figure 4.26: Successful DAG run after Model Development and Monitoring stage.

4.3.4 Reflective comments

DL models can become increasingly more complex and difficult to interpret, especially when experimenting with new methods. Therefore, it is crucial to have a well-grounded amount of practice with DL algorithms, on a range of problems and questions.

4.4 Model Deployment: Prediction Pipeline

4.4.1 Pipeline

Figure 4.27 shows a sample of the control flow file for the Deployment stage. The code for the Deployment stage is in Appendix C.5.

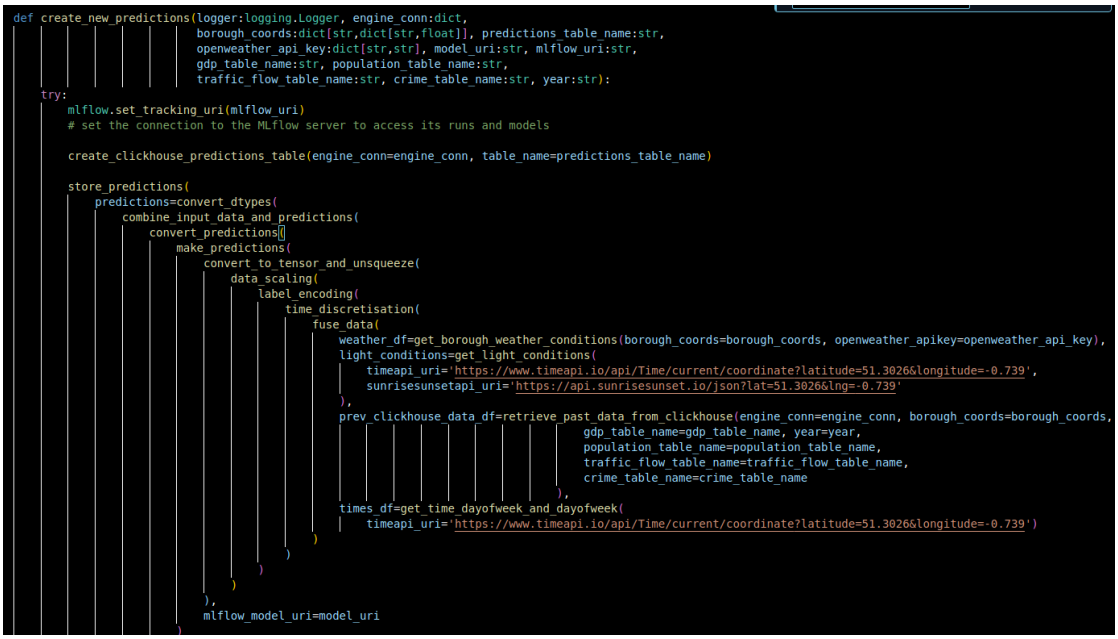


Figure 4.27: Sample of the control flow file for Model Deployment

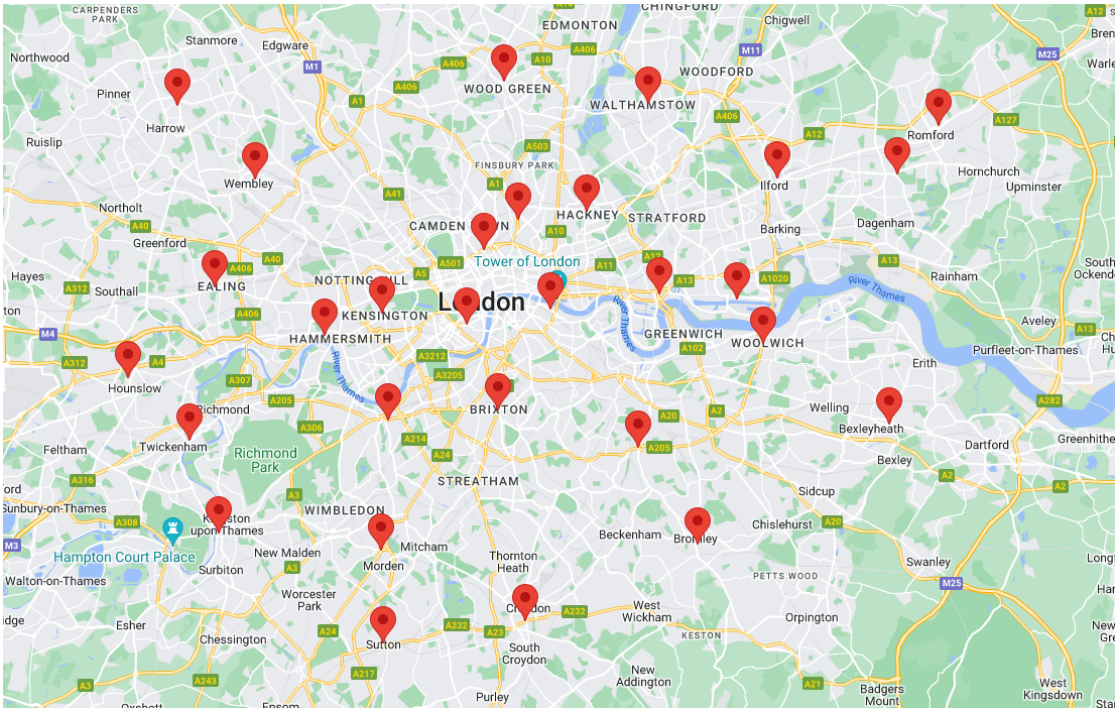
In essence, the pipeline retrieves the relevant data (matching that used to train the DL model), then fuses it together after some minimal transformations are made. Once the data has been retrieved, it goes through similar transformations as in the Preprocessing stage, ultimately being converted into a PyTorch Tensor — ready to be used for predictions. Unfortunately, the predictions received (during 'make_predictions') from the model are not in the expected format and could not be leveraged to be stored in the ClickHouse server table.

4.4.2 Web app

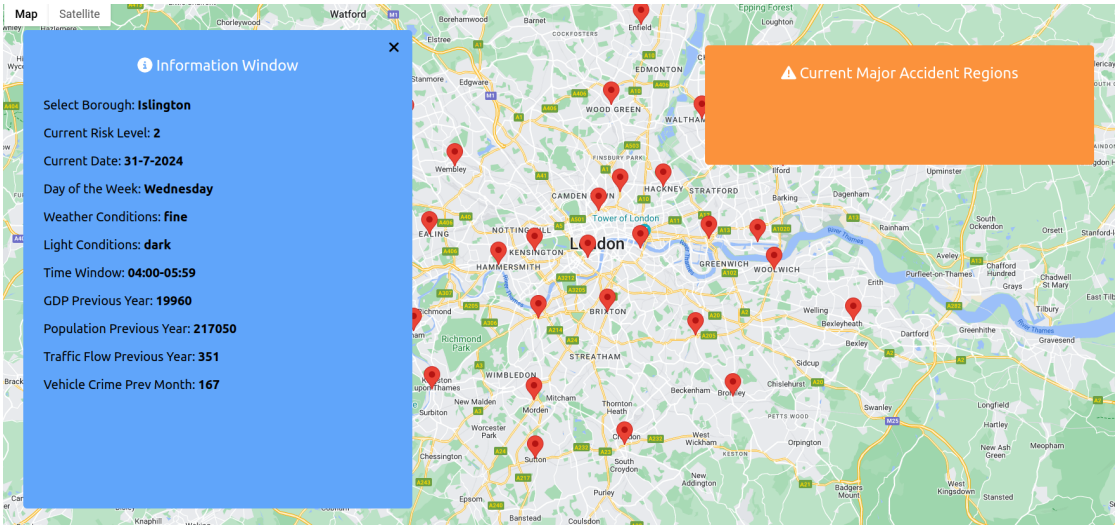
As the deployment pipeline could not operate, semi-artificial data was substituted into the created ClickHouse server table when the pipeline is run. The only synthetic part of the data is the 'Risk Level', the rest of the data was collected from the relevant channels and APIs. The result is shown in Figure 4.28.

The API was not constructed for the backend as ClickHouse offered an NPM package ('clickhouse/client-web') that could operate on the frontend just to retrieve the data from the ClickHouse server instance. In future, if additional functionality that would require

server-side processing (e.g., making a prediction through the uniform resource locator) then an API would be needed.



(a) Web app landing screen.



(b) Web app with components shown.

Figure 4.28: Web app result.

5 Evaluation

5.1 Results

5.1.1 Final model

Table 5.1 illustrates the evaluation results of the final model developed. Figures ... illustrate the learning curves of the model.

Table 5.1: Final model evaluation metrics.

Evaluation metric	**Training result (4 s.f.)**	**Validation result (4 s.f.)**	**Testing result (4 s.f.)**
(CORN) Log Loss	0.2592	0.2860	0.2786
Mean absolute error	0.3092	0.4129	0.3895
Mean absolute percentage error	0.1092 (10.9%)	0.1435 (12.4%)	0.1356 (13.56%)
Macro-averaged mean absolute error	0.3092	0.4129	Not applicable
Mean squared error	0.6110	0.8182	0.7682
Root mean squared error	0.7720	0.8999	0.8660

Figure 5.1: Final model learning curves, part 1.

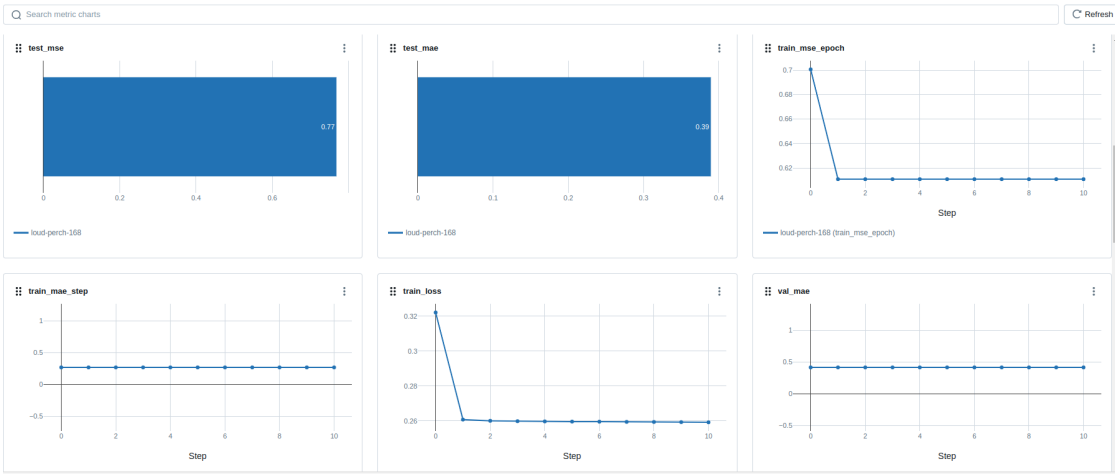


Figure 5.2: Final model learning curves, part 2.

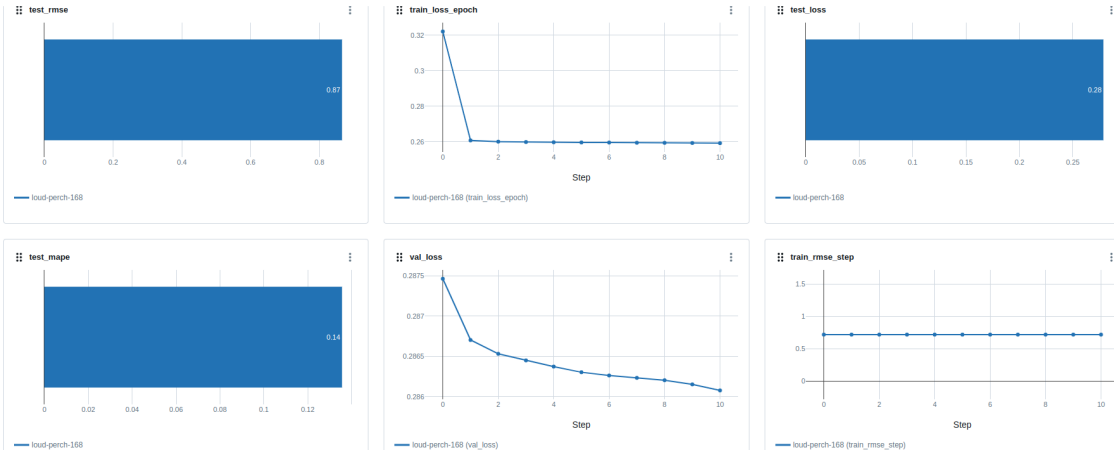


Figure 5.3: Final model learning curves, part 3.

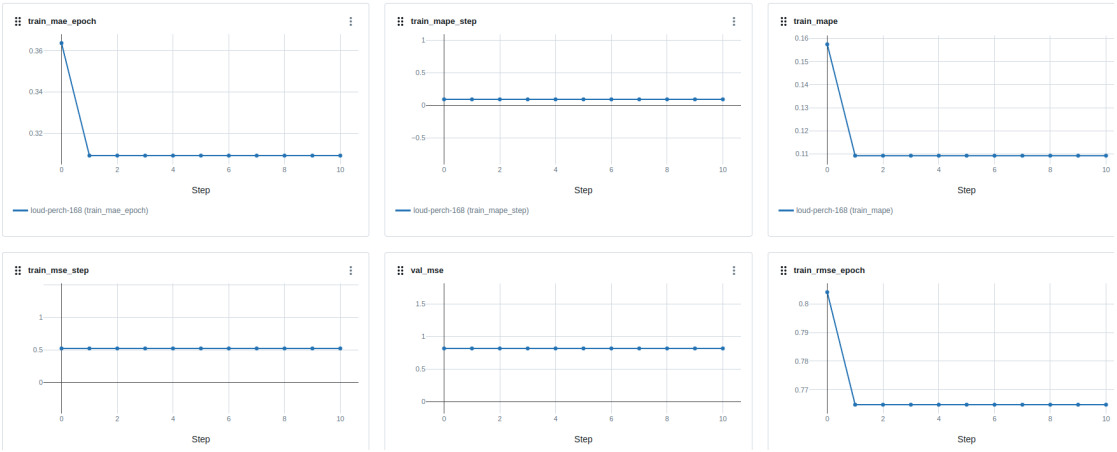


Figure 5.4: Final model learning curves, part 4.

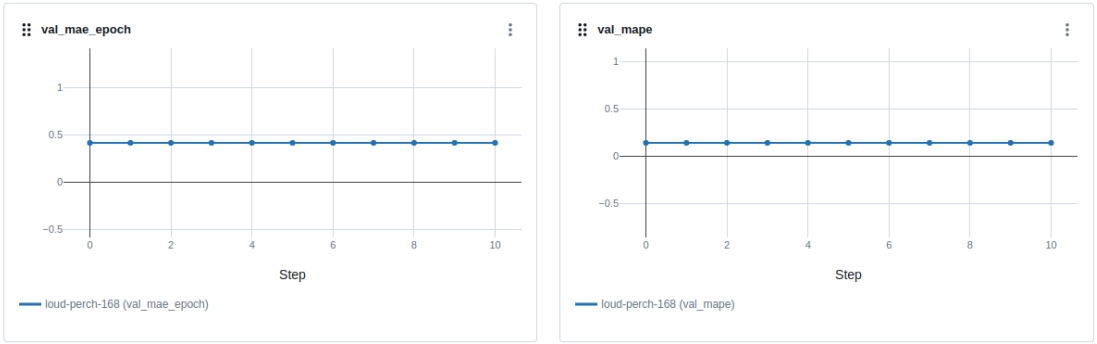


Figure 5.5: Final model learning curves, part 5.

5.1.2 Pipelines

Table 5.2 shows the time taken my the model creation pipeline to complete. The predictions pipeline took an approximate 10 seconds to complete each time previously — based on previous experimentation before the pipeline was blocked.

Table 5.2: Model creation pipeline time taken.

Model creation pipeline stage	**Time taken (seconds)**
Ingestion	8 if no tables already created, and less than 1 if the tables were already created.
Preprocessing	31
Model Development and Monitoring	445
Total	Approx. 484

5.2 Discussion

Unfortunately, the final model metrics seem to not be much better compared to the base model. This implies there is something fundamentally wrong, which is likely to be either with the structure of the data or the data itself (e.g., not enough information, too general, etc.). None of the steps taken were wrong necessarily, but perhaps some further transformations were required to properly prepare the data.

However, the metrics themselves do not indicate bad results — they actually indicate good results by themselves. But due to the trends in the learning curves, as most of them remain fairly unchanged (besides the Log Loss), it can be pointed out as irregular.

Regarding the model creation pipeline itself, it took about 8 minutes to complete. This was created however on a central processing unit-only laptop with only 8 cores available. Additionally, the number of workers enabled per 'DataLoader' could be increased (improving speed of development as they can work on the data at multiple points at a time), but this operation was restricted due to concurrency system restraints. If given a

graphics processing unit, the training time for the model would likely be reduced.

5.3 Project Progress

The timeline was not followed the best it could have been. The author's circumstances made it increasingly more difficult to maintain a regular scheduled time devoted to following the project plan. However, most project meetings and sessions were maintained and documented.

5.4 Reflective Comments

The requirements a system is designed to have, the easier it is for the complexity and dependencies of such a system to go out of control. Therefore, it is important to have backup contingencies should certain parts of the system fail.

6 Conclusions and Future Work

To conclude, with the current open source datasets available, it is not practical enough to leverage these alone to construct reliable and precise RTA predictive models. However, this could be improved using proprietary datasets in future. Additionally, this report illustrates the capabilities and versatility of MLOps, and demonstrates how they can be leveraged to help automate data ingestion and model creation.

For future work, further experimentation and exploration would be recommended for integrating the technologies available within Smart Cities, to enable a more unified approach and flow when developing them further in future. This would also open more pathways to better data collection and mining.

In addition, it is also recommended for future research to see how LSTMs could be leveraged with other deep learning algorithms to resolve spatio-temporal problems, not just pertaining to predicting road traffic accidents, but this could also be applied in other areas such as animal and insect migrations for tracking and/or management purposes.

Reference List

Ahmed, S., Hossain, M. A., Ray, S. K. et al. (2023). A study on road accident prediction and contributing factors using explainable machine learning models: analysis and performance. *Transportation Research Interdisciplinary Perspectives*, 19. Available at: https://doi.org/10.1016/j.trip.2023.100814.

Alfonsi, R., Persia, L., Antonino, T. et al. (2016). Advancements in Road Safety Management Analysis. *Transportation Research Procedia*, 14, pp. 2064–2073.

Anaconda Inc (2024). *Conda Environments*. Available at: https://docs.anaconda.com/working-with-conda/environments/ [Accessed on 26th Mar. 2024].

Anaconda Inc and contributors (2018a). *Dask*. Available at: https://docs.dask.org/en/stable/ [Accessed on 28th Mar. 2024].

Anaconda Inc and contributors (2018b). *Dask ML*. Available at: https://ml.dask.org/ [Accessed on 28th Mar. 2024].

Apache Software Foundation (2024). *Apache Airflow*. Available at: https://airflow.apache.org/ [Accessed on 20th Mar. 2024].

Association for Computing Machinary Digital Library (2023). *ACM Digital Library Board*. Available at: https://dl.acm.org/ [Accessed on 13th Dec. 2023].

Baccianella, S., Esuli, A. and Sebastiani, F. (2009). Evaluation Measures for Ordinal Regression. In: *2009 Ninth International Conference on Intelligent Systems Design and Applications*. Pisa, Italy, 30 November - 02 December 2009. New York, USA: IEEE, pp. 283–287.

Baheti, P. (2021). A Simple Guide to Data Preprocessing in Machine Learning. *Machine Learning*. [blog] 31 August. Available at: https://www.v7labs.com/blog/data-preprocessing-guide [Accessed on 30th Mar. 2024].

Bhatt, N. (2023). *Why MLOps architecture is a game changer: A comprehensive guide*. Available at: https://www.softwebsolutions.com/resources/mlops-architecture-guide.html [Accessed on 31st Mar. 2024].

Birmingham City University (2023). *BCU Library and Learning Resources*. Available at: https://www.bcu.ac.uk/library [Accessed on 13th Dec. 2023].

Boua, M., Kouabenan, D. R. and Belhaj, A. (2022). Road safety behaviors: Role of control beliefs and risk perception. *Transportation Research Part F: Traffic Psychology and Behaviour*, 91, pp. 45–57.

Braun, S. (2018). LSTM Benchmarks for Deep Learning Frameworks. *CoRR*. Available at: https://doi.org/10.48550/arXiv.1806.01818.

Bristol City Council (2024). *Bristol Emergency Operations Centre*. Available at: https://www.bristol.gov.uk/emergency-operations-centre [Accessed on 12th Apr. 2024].

Cao, W., Mirjalili, V. and Raschka, S. (2020). Rank consistent ordinal regression for neural networks with application to age estimation. *Pattern Recognition Letters*, 140, pp. 325–331. Available at: https://doi.org/10.1016/j.patrec.2020.11.008.

Celesti, A., Galletta, A., Carnevale, L. et al. (2018). An IoT Cloud System for Traffic Monitoring and Vehicular Accidents Prevention Based on Mobile Sensor Data Processing. *IEEE Sensors Journal*, 18 (12), pp. 4795–4802.

Cheng, R. (2020). LSTM Text Classification Using Pytorch. *Towards Data Science*. [blog] 30 June. Available at: https://towardsdatascience.com/lstm-text-classification-using-pytorch-2c6c657f8fc0 [Accessed on 31st Mar. 2024].

Clarivate (2023). *Web of Science*. Available at: https://www.webofscience.com/wos/woscc/basic-search [Accessed on 13th Dec. 2023].

ClickHouse Inc (2024a). *ClickHouse Server*. Available at: https://hub.docker.com/r/clickhouse/clickhouse-server/ [Accessed on 28th Mar. 2024].

ClickHouse Inc (2024b). *The real-time data warehouse for ML and GenAI*. Available at: https://clickhouse.com/ [Accessed on 24th Mar. 2024].

Costello, S. (2018). A Guide to Data Warehousing: From Strategy to Implementation. *Analytics8*. [blog] 26 July. Available at: https://www.analytics8.com/blog/what-is-a-data-warehouse/ [Accessed on 9th Apr. 2024].

Directory for Open Access Journals (2023). *Infrastructure Services for Open Access*. Available at: https://www.mongodb.com/compare/relational-vs-non-relational-databases [Accessed on 13th Dec. 2023].

Docker Inc (2024a). *Docker*. Available at: https://www.docker.com [Accessed on 28th Mar. 2024].

Docker Inc (2024b). *Docker Compose overview*. Available at: https://docs.docker.com/compose/.

Elsevier (2023a). *Science Direct*. Available at: https://www.sciencedirect.com/ [Accessed on 13th Dec. 2023].

Elsevier (2023b). *Scopus*. Available at: https://www.scopus.com/ [Accessed on 13th Dec. 2023].

Figma Inc (2024). *Figma*. Available at: https://www.figma.com/ [Accessed on 16th Apr. 2024].

Gassmann, O., Bohm, J. and Palmie, M. (2019). *Smart Cities: Introducing Digital Innovation to Cities*. Bingley, England: Emerald Publishing.

GeeksForGeeks (2024). *Waterfall Model — Software Engineering*. Available at: https://www.geeksforgeeks.org/waterfall-model/ [Accessed on 4th Feb. 2024].

Ghena, M. F. and Ghiculescu, L. D. (2023). Applicability of Waterfall and Agile Methodologies. *FAIMA Business and Management Journal*, 11 (4), pp. 55–65.

GitHub (2024). *NPM*. Available at: https://www.npmjs.com/ [Accessed on 13th Apr. 2024].

Google (2023). *Google Scholar*. Available at: https://scholar.google.co.uk/ [Accessed on 13th Dec. 2023].

Google (2024). *Google Colaboratory*. Available at: https://colab.research.google.com/ [Accessed on 17th Apr. 2024].

Greater London Authority (2024). *London Datastore*. [dataset]. Available at: https://data.london.gov.uk/ [Accessed on 6th Feb. 2024].

Hosmer, D., Lemeshow, S. and Sturdivant, R. (2013). *Applied Logistic Regression*. Hoboken, New Jersey: John Wiley and Sons.

IBM Cloud Education (2021). ELT vs. ETL: What's the Difference? *Analytics*. [blog] 14 December. Available at: https://www.ibm.com/blog/elt-vs-etl-whats-the-difference/ [Accessed on 20th Apr. 2024].

Ibrahim, M. (2024). A Deep Dive Into Learning Curves in Machine Learning. *Domain Agnostic*. [blog] 13 March. Available at: https://wandb.ai/mostafaibrahim17/ml-articles/reports/A-Deep-Dive-Into-Learning-Curves-in-Machine-Learning--Vmlldzo0NjA1ODY0 [Accessed on 20th Apr. 2024].

Institute of Electrical and Electronics Engineers Xplorer (2023). *IEEE and the Institution of Engineering and Technology*. Available at: https://ieeexplore.ieee.org/Xplore/home.jsp [Accessed on 13th Dec. 2023].

Jähi, H., Muhlrad, N., Buttler, I. et al. (2012). Investigating Road Safety Management Processes in Europe. *Procedia - Social and Behavioral Sciences*, 48, pp. 2130–2139.

JGraph Ltd (2023). *draw.io (v23.0.2)*. [computer software]. Available at: https://www.drawio.com/ [Accessed on 6th Feb. 2024].

Johns, R. (2024). A Guide to Data Warehousing: From Strategy to Implementation. *Development*. [blog] 26 January. Available at: https://hackr.io/blog/web-development-frameworks [Accessed on 13th Apr. 2024].

Jupyter (2024). *Jupyter Notebook*. Available at: https://jupyter.org/ [Accessed on 30th Mar. 2024].

Jurczenia, K. and Rak, J. (2022). A Survey of Vehicular Network Systems for Road Traffic Management. *IEEE Access*, 10, pp. 42365–42385.

Kanakala, R., Mohan, J. and Reddy, K. (2023). Modelling a Deep Network Using CNN and RNN for Accident Classification. *Measurement: Sensors*, 27. Available at: https://doi.org/10.1016/j.measen.2023.100794.

Kaneko, Y., Suzuki, M., Nagai, K. et al. (2021). Differentail effects of aging and cognitive decline on visual exploration behaviour in the elderly. *Neuroscience Research*, 171, pp. 62–66.

Kousha, P., Zhou, Q., Subramoni, H. et al. (2024). Benchmarking Modern Databases for Storing and Profiling Very Large Scale HPC Communication Data. In: *Benchmarking, Measuring, and Optimizing*. Sanya, China, 3-5 December 2023, pp. 104–119. Available at: https://doi.org/10.1007/978-981-97-0316-6_7. [Accessed on 27th Mar. 2024].

Liang, C., Ghazel, M., Cazier, O. et al. (2018). Developing accident prediction model for railway level crossings. *Safety Science*, 101, pp. 48–59.

Lightning AI (2024). *PyTorch Lightning*. Available at: https://lightning.ai/docs/pytorch/stable/ [Accessed on 4th Apr. 2024].

Lin, D.-J., Chen, M.-Y., Chiang, H.-S. et al. (2021). Intelligent Traffic Accident Prediction Model for Internet of Vehicles With Deep Learning Approach. *IEEE Transactions on Intelligent Transportation Systems*, 23 (3), pp. 2340–2349.

LinkedIn Community (2024). What are the key performance indicators and metrics for a data pipeline? *Big Data Analytics*. [blog] 11 June. Available at: https://www.linkedin.com/advice/0/what-key-performance-indicators-metrics-data [Accessed on 17th Apr. 2024].

MariaDB Foundation (2024). *MariaDB Docker image*. Available at: https://hub.docker.com/_/mariadb [Accessed on 28th Mar. 2024].

MatPlotLib Team (2024). *MatPlotLib: Visualisation with Python*. Available at: https://matplotlib.org/ [Accessed on 28th Mar. 2024].

Meta OpenSource (2024). *React*. Available at: https://react.dev/ [Accessed on 13th Apr. 2024].

Metropolitan Police (2024). *Recorded Crime: Geographic Breakdown*. [dataset]. Available at: https://data.london.gov.uk/dataset/recorded_crime_summary [Accessed on 12th July 2024].

Mohanta, B. K., Jena, D., Mohapatra, N. et al. (2022). Machine learning based accident prediction in secure IoT enable transportation system. *Journal of Intelligent and Fuzzy Systems: Applications in Engineering and Technology*, 42 (2), pp. 713–725.

Monsefi, A. K., Shiri, P., Mohammadshirazi, A. et al. (Nov. 2023). CrashFormer: A Multimodal Architecture to Predict the Risk of Crash. In: *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Advances in Urban-AI*. UrbanAI '23. Hamburg, Germany: Association for Computing Machinery, pp. 42–51.

NumFocus Inc (2024). *Python Pandas*. Available at: https://pandas.pydata.org/ [Accessed on 28th Mar. 2024].

Nzuchi, J. S., Ngoma, S. J. and Meshi, E. B. (2022). Commercial motorcyclists and road safety measures compliance. A case study of Dodoma city, central Tanzania. *Heliyon*, 8 (8). Available at: https://doi.org/10.1016/j.heliyon.2022.e10297.

Office for National Statistics (2022). *Regional gross domestic product: local authorities*. [dataset]. Available at: https://www.ons.gov.uk/economy/grossdomesticproductgdp/datasets/regionalgrossdomesticproductlocalauthorities [Accessed on 12th July 2024].

OpenJS Foundation (2024a). *Express*. Available at: https://expressjs.com/ [Accessed on 13th Apr. 2024].

OpenJS Foundation (2024b). *NodeJS*. Available at: https://nodejs.org/ [Accessed on 13th Apr. 2024].

OpenWeather (2024). *Weather API*. Available at: https://openweathermap.org/api [Accessed on 8th Apr. 2024].

Ordnance Survey (2024). *Ordnance Survey website home page*. Available at: https://www.ordnancesurvey.co.uk/ [Accessed on 8th Feb. 2024].

Ouallane, A., Bahnasse, A., Bakali, A. et al. (2022). Overview of Road Traffic Management Solutions Based on IoT and AI. *Procedia Computer Science*, 198, pp. 518–523.

Ouallane, A., Bakali, A., Bahnasse, A. et al. (2022). Fusion of engineering insights and emerging trends: Intelligent urban traffic management system. *Information Fusion*, 88, pp. 218–248.

Parsa, A. B., Chauhan, R. S., Taghipour, H. et al. (2019). Applying Deep Learning to Detect Traffic Accidents in Real Time Using Spatiotemporal Sequential Data. *Machine Learning*. Available at: https://doi.org/10.48550/arXiv.1912.06991.

Pedregosa, F., Varoquaux, G., Gramfort, A. et al. (2024). *SciKit Learn*. Available at: https://scikit-learn.org/stable/index.html [Accessed on 28th Mar. 2024].

Perri, G. and Vaiana, R. (2022). Road Safety Management of Uncontrolled Access Points: Design Criteria and Insights into Risk Factors. *Applied Sciences*, 12 (24). Available at: https://doi.org/10.3390/app122412661.

Pop, M.-D. and Proștean, O. (2018). A Comparison Between Smart City Approaches in Road Traffic Management. *Procedia — Social and Behavioural Sciences*, 238, pp. 29–36.

Raschka Research Group (2022). *CORAL & CORN PyTorch*. Available at: https://github.com/Raschka-research-group/coral-pytorch [Accessed on 4th Apr. 2024].

Redis Ltd (2024). *Redis*. Available at: https://redis.io/company/ [Accessed on 30th Mar. 2024].

Regan, M. A., Lee, J. D. and Victor, T. W. (2013). *Driver Distraction and Inattention*. London, England: Ashgate Publishing.

Ren, H., Song, Y., Liu, J. et al. (2017). A Deep Learning Approach to the Prediction of Short-term Traffic Accident Risk. Available at: https://doi.org/10.48550/arXiv.1710.09543.

Ribeiro, B., Nicolau, M. J. and Santos, A. (Mar. 2023). Machine learning for VRUs accidents prediction using V2X data. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. SAC '23. Tallinn, Estonia: Association for Computing Machinery, pp. 1789–1791.

RisingWave (2023). A Comprehensive Overview of Top 8 Real-Time OLAP Databases. *Data Engineering 101*. [blog] 6 September. Available at: https://risingwave.com/blog/a-comprehensive-overview-of-top-8-real-time-olap-databases/ [Accessed on 26th Mar. 2024].

Sabag, E. (2021). What exactly does "Low coupling, High cohesion" mean? *Wix Engineering*. [blog] 2 December. Available at: https://medium.com/wix-engineering/what-exactly-does-low-coupling-high-cohesion-mean-9259e8225372 [Accessed on 30th Mar. 2024].

Series of LF Projects, LLC (2024). *MLflow*. Available at: https://mlflow.org/ [Accessed on 3rd Apr. 2024].

Shi, X., Cao, W. and Raschka, S. (2021). Deep neural networks for rank-consistent ordinal regression based on conditional probabilities. *Pattern Analysis and Applications*, 26 (3), pp. 941–955. Available at: 10.1007/s10044-023-01181-9.

Soehodho, S. (2017). Pubic transportation development and traffic accident prevention in Indonesia. *IATSS Research*, 40 (2), pp. 76–80.

Souza, A. de, Brennand, C., Yokoyama, R. et al. (2017). Traffic management systems: A classification, review, challenges, and future perspectives. *International Journal of Distributed Sensor Networks*, 13 (4). Available at: https://doi.org/10.1177/1550147716683612.

SunriseSunset.io (2024). *Sunset and Sunrise Times API*. Available at: https://sunrisesunset.io/api/ [Accessed on 8th Apr. 2024].

Szűcs, K. (2020). *PandaHouse*. Available at: https://github.com/kszucs/pandahouse [Accessed on 28th Mar. 2024].

TBF Traffic (2023). *What is Traffic Management?* Available at: https://www.tbftraffic.com/what-is-traffic-management/ [Accessed on 8th Oct. 2023].

Techchink (2023). *What is Agile Methodology? Agile Development Methodology Steps in Detail*. Available at: https://www.techchink.com/what-is-agile-methodology/ [Accessed on 4th Feb. 2024].

The Document Foundation (2022). *LibreOffice Calc (7.3.7.2)*. [computer software]. Available at: https://www.libreoffice.org/ [Accessed on 25th Feb. 2024].

Tian, Z. and Zhang, S. (2022). Deep learning method for traffic accident prediction security. *Soft Computing*, 26, pp. 5363–5375.

TimeAPI (2024). *Time Zone API*. Available at: https://www.timeapi.io/ [Accessed on 8th Apr. 2024].

Torbaghan, M. E., Sasidharan, M., Reardon, L. et al. (2022). Understanding the potential of emerging digital technologies for improving road safety. *Accident Analysis and Prevention*, 166. Available at: https://doi.org/10.1016/j.aap.2021.106543.

Torre, F. L., Meocci, M., Domenichini, L. et al. (2019). Development of an accident prediction model for Italian freeways. *Accident Analysis and Prevention*, 124, pp. 1–11.

Transport for London (2023). *Causualties in Greater London during 2022*. [pdf] London: Transport for London. Available at: https://content.tfl.gov.uk/casualties-in-greater-london-2022.pdf [Accessed on 8th Oct. 2023].

Transport for London (2024a). *Our open data*. [dataset]. Available at: https://tfl.gov.uk/info-for/open-data-users/our-open-data#on-this-page-0 [Accessed on 6th Feb. 2024].

Transport for London (2024b). *Road safety data*. [dataset]. Available at: https://tfl.gov.uk/corporate/publications-and-reports/road-safety [Accessed on 6th Feb. 2024].

UK Department for Transport (2022). *Traffic Flows, Borough*. [dataset]. Available at: https://data.london.gov.uk/dataset/traffic-flows-borough [Accessed on 12th July 2024].

Waskom, M. (2024). *Seaborn: statistical data visualisation*. Available at: https://seaborn.pydata.org/ [Accessed on 28th Mar. 2024].

Wathan, A., Schoger, S., Hemphill, D. et al. (2024). *TailWindCSS*. Available at: https://tailwindcss.com/ [Accessed on 13th Apr. 2024].

Wegman, F. (2017). The future of road safety: A worldwide perspective. *IATSS Research*, 40 (2), pp. 66–71.

Wegman, F., Berg, H.-Y., Cameron, I. et al. (2015). Evidence-based and data-driven road safety management. *IATSS Research*, 39 (1), pp. 19–25.

Yu, L., Du, B., Hu, X. et al. (2021). Deep spatio-temporal graph convolutional network for traffic accident prediction. *Neurocomputing*, 423, pp. 135–147.

Yuan, Z., Zhou, X. and Yang, T. (Aug. 2018). Hetero-ConvLSTM: A Deep Learning Approach to Traffic Accident Prediction on Heterogenous Spatio-Temporal Data. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. London, England: Association for Computing Machinery, pp. 984–992.

Zagidullin, R. (2017). Model of Road Traffic Management in the City during Major Sporting Events. *Transportation Research Procedia*, 20, pp. 709–716.

Zhang, Z., Yang, W. and Wushour, S. (2020). Traffic Accident Prediction Based on LSTM-GBRT Model. *Journal of Computer Science and Engineering*, 2020, p. 4206919. Available at: https://doi.org/10.1155/2020/4206919.

Zhou, X., Lu, P., Zheng, Z. et al. (2020). Accident Prediction Accuracy Assessment for Highway-Rail Grade Crossings Using Random Forest Algorithm Compared with Decision Tree. *Reliability Engineering and System Safety*, 200. Available at: https://doi.org/10.1016/j.ress.2020.106931.

Bibliography

Adeshina, A. (2022). Developing a Single Page App with FastAPI. *testdriven.io*. [blog] 30 May. Available at: https://testdriven.io/blog/fastapi-react/ [Accessed on 17th Dec. 2023].

Ahmad, I., Noor, R. M., Zaba, M. R. et al. (2020). A Cooperative Heterogeneous Vehicular Clustering Mechanism for Road Traffic Management. *International Journal of Parallel Programming*, 48 (5), pp. 870–889.

Ahmed, S., Hossain, M. A., Ray, S. K. et al. (2023). A study on road accident prediction and contributing factors using explainable machine learning models: analysis and performance. *Transportation Research Interdisciplinary Perspectives*, 19. Available at: https://doi.org/10.1016/j.trip.2023.100814.

Alfonsi, R., Persia, L., Antonino, T. et al. (2016). Advancements in Road Safety Management Analysis. *Transportation Research Procedia*, 14, pp. 2064–2073.

Alosius, T. (2023). *Building a Prediction Web App using Streamlit*. Available at: https://medium.com/@tony.aloysius.77/building-a-prediction-web-app-using-streamlit-5bb881e545a [Accessed on 10th Feb. 2024].

Anaconda Inc (2024). *Conda Environments*. Available at: https://docs.anaconda.com/working-with-conda/environments/ [Accessed on 26th Mar. 2024].

Anaconda Inc and contributors (2018a). *Dask*. Available at: https://docs.dask.org/en/stable/ [Accessed on 28th Mar. 2024].

Anaconda Inc and contributors (2018b). *Dask ML*. Available at: https://ml.dask.org/ [Accessed on 28th Mar. 2024].

Apache Software Foundation (2024a). *Apache Airflow*. Available at: https://airflow.apache.org/ [Accessed on 20th Mar. 2024].

Apache Software Foundation (2024b). *Set up a Database Backend*. Available at: https://airflow.apache.org/docs/apache-airflow/2.4.3/howto/set-up-database.html [Accessed on 31st Mar. 2024].

Association for Computing Machinary Digital Library (2023). *ACM Digital Library Board*. Available at: https://dl.acm.org/ [Accessed on 13th Dec. 2023].

Augusto, J. C. (2021). *Handbook of Smart Cities*. Switzerland: Springer International Publishing.

Baber, C., Morar, N. S. and McCabe, F. (2019). Ecological Interface Design, the Proximity Compatibility Principle, and Automation Reliability in Road Traffic Management. *IEEE Transactions on Human-Machine Systems*, 49 (3), pp. 241–249.

Baccianella, S., Esuli, A. and Sebastiani, F. (2009). Evaluation Measures for Ordinal Regression. In: *2009 Ninth International Conference on Intelligent Systems Design*

and Applications. Pisa, Italy, 30 November - 02 December 2009. New York, USA: IEEE, pp. 283–287.

Baheti, P. (2021). A Simple Guide to Data Preprocessing in Machine Learning. *Machine Learning*. [blog] 31 August. Available at: https://www.v7labs.com/blog/data-preprocessing-guide [Accessed on 30th Mar. 2024].

Bauza, R. and Gozalvez, J. (2013). Traffic congestion detection in large-scale scenarios using vehicle-to-vehicle communications. *Journal of Network and Computer Applications*, 36 (5), pp. 1295–1307.

Bhatt, N. (2023). *Why MLOps architecture is a game changer: A comprehensive guide*. Available at: https://www.softwebsolutions.com/resources/mlops-architecture-guide.html [Accessed on 31st Mar. 2024].

Birmingham City University (2023). *BCU Library and Learning Resources*. Available at: https://www.bcu.ac.uk/library [Accessed on 13th Dec. 2023].

Bohnert, A. (2024). 10 Types of Neural Networks, Explained. *Artificial Intelligence*. [blog] 17 May. Available at: https://www.hackerrank.com/blog/types-of-neural-networks-explained/ [Accessed on 6th Feb. 2024].

Boskin, K. (2023). Building Web Applications with React and Python. *Full-stack Development*. [blog] 04 January. Available at: https://dev.to/kboskin/building-web-applications-with-react-and-python-2d8c [Accessed on 3rd Feb. 2024].

Boua, M., Kouabenan, D. R. and Belhaj, A. (2022). Road safety behaviors: Role of control beliefs and risk perception. *Transportation Research Part F: Traffic Psychology and Behaviour*, 91, pp. 45–57.

Boucheron, B. (2023). How To Install Node.js on Ubuntu 20.04. *DigitalOcean*. [blog] 23 January. Available at: https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04 [Accessed on 31st Mar. 2024].

Boujemaa, K. S., Berrada, I., Fardousse, K. et al. (2022). Toward Road Safety Recommender Systems: Formal Concepts and Technical Basics. *IEEE Transactions on Intelligent Transportation Systems*, 23 (6), pp. 5211–5230.

Braun, S. (2018). LSTM Benchmarks for Deep Learning Frameworks. *CoRR*. Available at: https://doi.org/10.48550/arXiv.1806.01818.

Bristol City Council (2024). *Bristol Emergency Operations Centre*. Available at: https://www.bristol.gov.uk/emergency-operations-centre [Accessed on 12th Apr. 2024].

Cao, W., Mirjalili, V. and Raschka, S. (2020). Rank consistent ordinal regression for neural networks with application to age estimation. *Pattern Recognition Letters*, 140, pp. 325–331. Available at: https://doi.org/10.1016/j.patrec.2020.11.008.

Celesti, A., Galletta, A., Carnevale, L. et al. (2018). An IoT Cloud System for Traffic Monitoring and Vehicular Accidents Prevention Based on Mobile Sensor Data Processing. *IEEE Sensors Journal*, 18 (12), pp. 4795–4802.

Centre for Environmental Data Analysis (2024). *MIDAS Open: UK hourly weather observation data, v202308*. Available at: https://data.ceda.ac.uk/badc/ukmo-midas-open/data/uk-hourly-weather-obs/dataset-version-202308/greater-london [Accessed on 6th Feb. 2024].

Cheng, R. (2020). LSTM Text Classification Using Pytorch. *Towards Data Science*. [blog] 30 June. Available at: https://towardsdatascience.com/lstm-text-classification-using-pytorch-2c6c657f8fc0 [Accessed on 31st Mar. 2024].

Clarivate (2023). *Web of Science*. Available at: https://www.webofscience.com/wos/woscc/basic-search [Accessed on 13th Dec. 2023].

ClickHouse Inc (2024a). *ClickHouse Server*. Available at: https://hub.docker.com/r/clickhouse/clickhouse-server/ [Accessed on 28th Mar. 2024].

ClickHouse Inc (2024b). *Instal ClickHouse*. Available at: https://clickhouse.com/docs/en/install [Accessed on 28th Mar. 2024].

ClickHouse Inc (2024c). *The real-time data warehouse for ML and GenAI*. Available at: https://clickhouse.com/ [Accessed on 24th Mar. 2024].

Costello, S. (2018). A Guide to Data Warehousing: From Strategy to Implementation. *Analytics8*. [blog] 26 July. Available at: https://www.analytics8.com/blog/what-is-a-data-warehouse/ [Accessed on 9th Apr. 2024].

Coursera (2024). 8 Common Types of Neural Networks. *Attention*. [blog] 23 April. Available at: https://www.coursera.org/in/articles/types-of-neural-networks?msockid=0dc41a119f886ac22b1609aa9eaf6bfc [Accessed on 6th Feb. 2024].

Department for Transport (2023). *London*. Available at: https://roadtraffic.dft.gov.uk/regions/6 [Accessed on 15th Jan. 2024].

Directory for Open Access Journals (2023). *Infrastructure Services for Open Access*. Available at: https://www.mongodb.com/compare/relational-vs-non-relational-databases [Accessed on 13th Dec. 2023].

Docker Inc (2024a). *Docker*. Available at: https://www.docker.com [Accessed on 28th Mar. 2024].

Docker Inc (2024b). *Docker Compose overview*. Available at: https://docs.docker.com/compose/.

Elsevier (2023a). *Science Direct*. Available at: https://www.sciencedirect.com/ [Accessed on 13th Dec. 2023].

Elsevier (2023b). *Scopus*. Available at: https://www.scopus.com/ [Accessed on 13th Dec. 2023].

Emanuilov, S. (2024). MongoDB vs ClickHouse for OLAP? *Medium*. [blog] 12 March. Available at: https://medium.com/@simeon.emanuilov/mongodb-vs-clickhouse-for-olap-1e430a40ade8 [Accessed on 28th Mar. 2024].

Facebook, Inc (2022). *Create React App*. Available at: https://create-react-app.dev/ [Accessed on 31st Mar. 2024].

Ferrera, N. and Colantonio, J. (2023). *How to Test a Time Machine: A Practical Guide to Test Architecture and Automation*. Birmingham, England: Packt Publishing.

Figma Inc (2024). *Figma*. Available at: https://www.figma.com/ [Accessed on 16th Apr. 2024].

Gassmann, O., Bohm, J. and Palmie, M. (2019). *Smart Cities: Introducing Digital Innovation to Cities*. Bingley, England: Emerald Publishing.

GeeksForGeeks (2024). *Waterfall Model — Software Engineering*. Available at: https://www.geeksforgeeks.org/waterfall-model/ [Accessed on 4th Feb. 2024].

Ghena, M. F. and Ghiculescu, L. D. (2023). Applicability of Waterfall and Agile Methodologies. *FAIMA Business and Management Journal*, 11 (4), pp. 55–65.

GitHub (2024). *NPM*. Available at: https://www.npmjs.com/ [Accessed on 13th Apr. 2024].

Gokulakrishnan, P. and Ganeshkumar, P. (2015). Road Accident Prevention with Instant Emergency Warning Message Dissemination in Vehicular Ad-Hoc Network. *PLoS ONE*, 10 (12). Available at: https://doi.org/10.1371/journal.pone.0143383.

Google (2023). *Google Scholar*. Available at: https://scholar.google.co.uk/ [Accessed on 13th Dec. 2023].

Google (2024a). *Code Samples*. Available at: https://developers.google.com/maps/documentation/javascript/examples/ [Accessed on 31st Mar. 2024].

Google (2024b). *Google Colaboratory*. Available at: https://colab.research.google.com/ [Accessed on 17th Apr. 2024].

Gourlay, K. (2023). Heavy traffic on Edinburgh city bypass after two cars collide near busy slip road. *Edinburgh Live*, 8 October. Available at: https://www.edinburghlive.co.uk/news/edinburgh-news/heavy-traffic-edinburgh-city-bypass-27867070 [Accessed on 10th Oct. 2023].

Government, U. K. (2023a). *Find and use roadworks data*. Available at: https://www.gov.uk/guidance/find-and-use-roadworks-data [Accessed on 6th Feb. 2024].

Government, U. K. (2023b). *Road traffic statistics*. Available at: https://www.gov.uk/government/statistical-data-sets/road-traffic-statistics-tra [Accessed on 6th Feb. 2024].

Greater London Authority (2024). *London Datastore*. [dataset]. Available at: https://data.london.gov.uk/ [Accessed on 6th Feb. 2024].

Grossi, G. and Pianezzi, D. (2017). Smart Cities: Utopia or Neoliberal Ideology? *Cities*, 69, pp. 79–85.

Gui, K., Ye, L. and Ge, J. (2019). Road surface condition detection utilizing resonance frequency and optical technologies. *Sensors and Actuators A: Physical*, 297. Available at: https://doi.org/10.1016/j.sna.2019.111540.

Halim, E. and Naratama, E. P. (Dec. 2023). Enhancing Efficiency in Road Traffic Management using Internet of Things and Vision-Based Vehicle Detection. In: *3rd International Conference on Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS)*. Bali, Indonesia: IEEE Xplorer, pp. 103–108.

Han, I. (2023). A Comprehensive Overview of Top Real-Time OLAP Databases. *Medium*. [blog] 23 November. Available at: https://medium.com/@figbear7/a-comprehensive-overview-of-top-real-time-olap-databases-ecbdc1970ba6 [Accessed on 28th Mar. 2024].

Hosmer, D., Lemeshow, S. and Sturdivant, R. (2013). *Applied Logistic Regression*. Hoboken, New Jersey: John Wiley and Sons.

IBM Cloud Education (2021). ELT vs. ETL: What's the Difference? *Analytics*. [blog] 14 December. Available at: https://www.ibm.com/blog/elt-vs-etl-whats-the-difference/ [Accessed on 20th Apr. 2024].

Ibrahim, M. (2024). A Deep Dive Into Learning Curves in Machine Learning. *Domain Agnostic*. [blog] 13 March. Available at: https://wandb.ai/mostafaibrahim17/ml-articles/reports/A-Deep-Dive-Into-Learning-Curves-in-Machine-Learning--Vmlldzo0NjA1ODY0 [Accessed on 20th Apr. 2024].

Institute of Electrical and Electronics Engineers Xplorer (2023). *IEEE and the Institution of Engineering and Technology*. Available at: https://ieeexplore.ieee.org/Xplore/home.jsp [Accessed on 13th Dec. 2023].

Jähi, H., Muhlrad, N., Buttler, I. et al. (2012). Investigating Road Safety Management Processes in Europe. *Procedia - Social and Behavioral Sciences*, 48, pp. 2130–2139.

Jamroz, K., Budzyński, M., Kustra, W. et al. (2014). Tools for Road Infrastructure Safety Management – Polish Experiences. *Transport Research Procedia*, 3, pp. 730–739.

JGraph Ltd (2023). *draw.io (v23.0.2)*. [computer software]. Available at: https://www.drawio.com/ [Accessed on 6th Feb. 2024].

Johns, R. (2024). A Guide to Data Warehousing: From Strategy to Implementation. *Development*. [blog] 26 January. Available at: https://hackr.io/blog/web-development-frameworks [Accessed on 13th Apr. 2024].

Jupyter (2024). *Jupyter Notebook*. Available at: https://jupyter.org/ [Accessed on 30th Mar. 2024].

Jurczenia, K. and Rak, J. (2022). A Survey of Vehicular Network Systems for Road Traffic Management. *IEEE Access*, 10, pp. 42365–42385.

Kanakala, R., Mohan, J. and Reddy, K. (2023). Modelling a Deep Network Using CNN and RNN for Accident Classification. *Measurement: Sensors*, 27. Available at: https://doi.org/10.1016/j.measen.2023.100794.

Kaneko, Y., Suzuki, M., Nagai, K. et al. (2021). Differentail effects of aging and cognitive decline on visual exploration behaviour in the elderly. *Neuroscience Research*, 171, pp. 62–66.

Kousha, P., Zhou, Q., Subramoni, H. et al. (2024). Benchmarking Modern Databases for Storing and Profiling Very Large Scale HPC Communication Data. In: *Benchmarking, Measuring, and Optimizing*. Sanya, China, 3-5 December 2023, pp. 104–119. Available at: https://doi.org/10.1007/978-981-97-0316-6_7. [Accessed on 27th Mar. 2024].

Lamr, M. and Skrbek, J. (Nov. 2015). Advanced Approaches to Traffic Accident Prevention. In: *26th International-Business-Information-Management-Association Conference*. Madrid, Spain: International Business Information Management Association, pp. 2236–2244.

Lancheros-Cuesta, D., Gayambuco-Ortega, L. C. and Aguilar, C. A. (June 2023). Design of an Intelligent Transportation System. In: *18th Iberian Conference on Information Systems and Technologies (CISTI)*. Averio, Portugal: IEEE Xplorer, pp. 1–6.

Lazyfloud (2023). Building a Full-Stack Web Application with FastAPI and React. *Medium*. [blog] 3 June. Available at: https://medium.com/@lazyflous/building-a-full-stack-web-application-with-fastapi-and-react-171f704d3aab [Accessed on 31st Mar. 2024].

Liang, C., Ghazel, M., Cazier, O. et al. (2018). Developing accident prediction model for railway level crossings. *Safety Science*, 101, pp. 48–59.

Lightning AI (2024). *PyTorch Lightning*. Available at: https://lightning.ai/docs/pytorch/stable/ [Accessed on 4th Apr. 2024].

Lin, D.-J., Chen, M.-Y., Chiang, H.-S. et al. (2021). Intelligent Traffic Accident Prediction Model for Internet of Vehicles With Deep Learning Approach. *IEEE Transactions on Intelligent Transportation Systems*, 23 (3), pp. 2340–2349.

LinkedIn Community (2024). What are the key performance indicators and metrics for a data pipeline? *Big Data Analytics*. [blog] 11 June. Available at: https://www.linkedin.com/advice/0/what-key-performance-indicators-metrics-data [Accessed on 17th Apr. 2024].

London Air (2024). *Data Downloads*. Available at: https://www.londonair.org.uk/london/asp/datadownload.asp [Accessed on 6th Feb. 2024].

MariaDB (2024). *MariaDB Enterprise ColumnStore*. Available at: https://mariadb.com/docs/server/products/mariadb-enterprise-columnstore/ [Accessed on 31st Mar. 2024].

MariaDB Foundation (2024). *MariaDB Docker image*. Available at: https://hub.docker.com/_/mariadb [Accessed on 28th Mar. 2024].

MatPlotLib Team (2024). *MatPlotLib: Visualisation with Python*. Available at: https://matplotlib.org/ [Accessed on 28th Mar. 2024].

Meta OpenSource (2024). *React*. Available at: https://react.dev/ [Accessed on 13th Apr. 2024].

Metropolitan Police (2024). *Recorded Crime: Geographic Breakdown*. [dataset]. Available at: https://data.london.gov.uk/dataset/recorded_crime_summary [Accessed on 12th July 2024].

Microsoft (2023). *ML For Beginners*. Available at: https://github.com/microsoft/ML-For-Beginners/tree/main [Accessed on 24th May 2024].

Mohanta, B. K., Jena, D., Mohapatra, N. et al. (2022). Machine learning based accident prediction in secure IoT enable transportation system. *Journal of Intelligent and Fuzzy Systems: Applications in Engineering and Technology*, 42 (2), pp. 713–725.

Monsefi, A. K., Shiri, P., Mohammadshirazi, A. et al. (Nov. 2023). CrashFormer: A Multimodal Architecture to Predict the Risk of Crash. In: *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Advances in Urban-AI*. UrbanAI '23. Hamburg, Germany: Association for Computing Machinery, pp. 42–51.

NodeSource Team (2024). *NodeSource Node.js Binary Distributions*. Available at: https://github.com/nodesource/distributions/blob/master/README.md [Accessed on 31st Mar. 2024].

NumFocus Inc (2024). *Python Pandas*. Available at: https://pandas.pydata.org/ [Accessed on 28th Mar. 2024].

NumPy Team (2024). *NumPy package*. Available at: https://numpy.org/ [Accessed on 28th Mar. 2024].

Nzuchi, J. S., Ngoma, S. J. and Meshi, E. B. (2022). Commercial motorcyclists and road safety measures compliance. A case study of Dodoma city, central Tanzania. *Heliyon*, 8 (8). Available at: https://doi.org/10.1016/j.heliyon.2022.e10297.

Office for National Statistics (2022). *Regional gross domestic product: local authorities*. [dataset]. Available at: https://www.ons.gov.uk/economy/grossdomesticproductgdp/datasets/regionalgrossdomesticproductlocalauthorities [Accessed on 12th July 2024].

Onojakpor, O. (2022). A Python developer's guide to React. *React*. [blog] 21 January. Available at: https://blog.logrocket.com/python-developers-guide-react/ [Accessed on 3rd Feb. 2024].

OpenJS Foundation (2024a). *Express*. Available at: https://expressjs.com/ [Accessed on 13th Apr. 2024].

OpenJS Foundation (2024b). *NodeJS*. Available at: https://nodejs.org/ [Accessed on 13th Apr. 2024].

OpenWeather (2024). *Weather API*. Available at: https://openweathermap.org/api [Accessed on 8th Apr. 2024].

Ordnance Survey (2024). *Ordnance Survey website home page*. Available at: https://www.ordnancesurvey.co.uk/ [Accessed on 8th Feb. 2024].

Ouallane, A., Bahnasse, A., Bakali, A. et al. (2022). Overview of Road Traffic Management Solutions Based on IoT and AI. *Procedia Computer Science*, 198, pp. 518–523.

Ouallane, A., Bakali, A., Bahnasse, A. et al. (2022). Fusion of engineering insights and emerging trends: Intelligent urban traffic management system. *Information Fusion*, 88, pp. 218–248.

Parsa, A. B., Chauhan, R. S., Taghipour, H. et al. (2019). Applying Deep Learning to Detect Traffic Accidents in Real Time Using Spatiotemporal Sequential Data. *Machine Learning*. Available at: https://doi.org/10.48550/arXiv.1912.06991.

Pedregosa, F., Varoquaux, G., Gramfort, A. et al. (2024). *SciKit Learn*. Available at: https://scikit-learn.org/stable/index.html [Accessed on 28th Mar. 2024].

Perri, G. and Vaiana, R. (2022). Road Safety Management of Uncontrolled Access Points: Design Criteria and Insights into Risk Factors. *Applied Sciences*, 12 (24). Available at: https://doi.org/10.3390/app122412661.

Pop, M.-D. and Proștean, O. (2018). A Comparison Between Smart City Approaches in Road Traffic Management. *Procedia — Social and Behavioural Sciences*, 238, pp. 29–36.

PyTorch Foundation (2024). *PyTorch*. Available at: https://pytorch.org/ [Accessed on 24th Mar. 2024].

Qiu, J., Wang, B. and Zhou, C. (2020). Forecasting stock prices with long-short term memory neural network based on attention mechanism. *PLoS ONE*, 15 (1). Available at: https://doi.org/10.1371/journal.pone.0227222.

Raschka Research Group (2022). *CORAL & CORN PyTorch*. Available at: https://github.com/Raschka-research-group/coral-pytorch [Accessed on 4th Apr. 2024].

Redis Ltd (2024). *Redis*. Available at: https://redis.io/company/ [Accessed on 30th Mar. 2024].

Regan, M. A., Lee, J. D. and Victor, T. W. (2013). *Driver Distraction and Inattention*. London, England: Ashgate Publishing.

Ren, H., Song, Y., Liu, J. et al. (2017). A Deep Learning Approach to the Prediction of Short-term Traffic Accident Risk. Available at: https://doi.org/10.48550/arXiv.1710.09543.

Ribeiro, B., Nicolau, M. J. and Santos, A. (Mar. 2023). Machine learning for VRUs accidents prediction using V2X data. In: *Proceedings of the 38th ACM/SIGAPP Sym-*

posium on Applied Computing. SAC '23. Tallinn, Estonia: Association for Computing Machinery, pp. 1789–1791.

RisingWave (2023a). A Comprehensive Overview of Top 8 Real-Time OLAP Databases. *Data Engineering 101*. [blog] 6 September. Available at: https://risingwave.com/blog/a-comprehensive-overview-of-top-8-real-time-olap-databases/ [Accessed on 26th Mar. 2024].

RisingWave (2023b). How to Compare Different OLAP Engines: A Comprehensive Guide. *Data Engineering 101*. [blog] 18 September. Available at: https://risingwave.com/blog/how-to-compare-different-olap-engines-a-comprehensive-guide/ [Accessed on 28th Mar. 2024].

Sabag, E. (2021). What exactly does "Low coupling, High cohesion" mean? *Wix Engineering*. [blog] 2 December. Available at: https://medium.com/wix-engineering/what-exactly-does-low-coupling-high-cohesion-mean-9259e8225372 [Accessed on 30th Mar. 2024].

Saeed, M. (2023). An Introduction to Recurrent Neural Networks and the Math That Powers Them. *Attention*. [blog] 6 January. Available at: https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/ [Accessed on 6th Feb. 2024].

Sandy, J. (2021). Choosing Between Django, Flask, and FastAPI. *testdriven.io*. [blog] 4 January. Available at: https://www.webscale.com/engineering-education/choosing-between-django-flask-and-fastapi/ [Accessed on 16th Dec. 2023].

Schulze, H. and Koßmann, I. (2010). The role of safety research in road safety management. *Safety Science*, 48 (9), pp. 1160–1166.

Series of LF Projects, LLC (2024). *MLflow*. Available at: https://mlflow.org/ [Accessed on 3rd Apr. 2024].

Shi, X., Cao, W. and Raschka, S. (2021). Deep neural networks for rank-consistent ordinal regression based on conditional probabilities. *Pattern Analysis and Applications*, 26 (3), pp. 941–955. Available at: 10.1007/s10044-023-01181-9.

Soehodho, S. (2017). Pubic transportation development and traffic accident prevention in Indonesia. *IATSS Research*, 40 (2), pp. 76–80.

Souza, A. de, Brennand, C., Yokoyama, R. et al. (2017). Traffic management systems: A classification, review, challenges, and future perspectives. *International Journal of Distributed Sensor Networks*, 13 (4). Available at: https://doi.org/10.1177/1550147716683612.

Statistics Solutions (2023). *Ordinal Regression*. Available at: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/ordinal-regression/ [Accessed on 15th Dec. 2023].

Stoffel, T. (2023). MariaDB ColumnStore Docker Quick Start Guide. *MariaDB*. [blog] 23 November. Available at: https://mariadb.com/resources/blog/mariadb-columnstore-docker-quick-start-guide/ [Accessed on 28th Mar. 2024].

SunriseSunset.io (2024). *Sunset and Sunrise Times API*. Available at: https://sunrisesunset.io/api/ [Accessed on 8th Apr. 2024].

Szűcs, K. (2020). *PandaHouse*. Available at: https://github.com/kszucs/pandahouse [Accessed on 28th Mar. 2024].

Takeuchi, S., Uchida, N. and Shibata, Y. (2019). Movement Detection Methods with Wireless Signals and Multiple Sensors on Mobile Phone for Traffic Accident Prevention Systems. *Advances in Network-based Information Systems*, 22. Available at: https://doi.org/10.1007/978-3-319-98530-5_73.

Taresh, A. A. R. and Zghair, N. A. K. (2023). Redesign of the communications network based on high availability of traffic management technologies to improve the communication. *Measurement: Sensors*, 27. Available at: https://doi.org/10.1016/j.measen.2023.100776.

TBF Traffic (2023). *What is Traffic Management?* Available at: https://www.tbftraffic.com/what-is-traffic-management/ [Accessed on 8th Oct. 2023].

Techchink (2023). *What is Agile Methodology? Agile Development Methodology Steps in Detail*. Available at: https://www.techchink.com/what-is-agile-methodology/ [Accessed on 4th Feb. 2024].

The Document Foundation (2022). *LibreOffice Calc (7.3.7.2)*. [computer software]. Available at: https://www.libreoffice.org/ [Accessed on 25th Feb. 2024].

Tian, Z. and Zhang, S. (2022). Deep learning method for traffic accident prediction security. *Soft Computing*, 26, pp. 5363–5375.

TimeAPI (2024). *Time Zone API*. Available at: https://www.timeapi.io/ [Accessed on 8th Apr. 2024].

Torbaghan, M. E., Sasidharan, M., Reardon, L. et al. (2022). Understanding the potential of emerging digital technologies for improving road safety. *Accident Analysis and Prevention*, 166. Available at: https://doi.org/10.1016/j.aap.2021.106543.

Torre, F. L., Meocci, M., Domenichini, L. et al. (2019). Development of an accident prediction model for Italian freeways. *Accident Analysis and Prevention*, 124, pp. 1–11.

Transport for London (2023). *Causualties in Greater London during 2022*. [pdf] London: Transport for London. Available at: https://content.tfl.gov.uk/casualties-in-greater-london-2022.pdf [Accessed on 8th Oct. 2023].

Transport for London (2024a). *Our open data*. [dataset]. Available at: https://tfl.gov.uk/info-for/open-data-users/our-open-data#on-this-page-0 [Accessed on 6th Feb. 2024].

Transport for London (2024b). *Road safety data*. [dataset]. Available at: https://tfl.gov.uk/corporate/publications-and-reports/road-safety [Accessed on 6th Feb. 2024].

UK Department for Transport (2022). *Traffic Flows, Borough*. [dataset]. Available at: https://data.london.gov.uk/dataset/traffic-flows-borough [Accessed on 12th July 2024].

Waskom, M. (2024). *Seaborn: statistical data visualisation*. Available at: https://seaborn.pydata.org/ [Accessed on 28th Mar. 2024].

Wathan, A., Schoger, S., Hemphill, D. et al. (2024). *TailWindCSS*. Available at: https://tailwindcss.com/ [Accessed on 13th Apr. 2024].

Wegman, F. (2017). The future of road safety: A worldwide perspective. *IATSS Research*, 40 (2), pp. 66–71.

Wegman, F., Berg, H.-Y., Cameron, I. et al. (2015). Evidence-based and data-driven road safety management. *IATSS Research*, 39 (1), pp. 19–25.

Werr, E. (2022). *London Weather Data*. Available at: https://www.kaggle.com/datasets/emmanuelfwerr/london-weather-data [Accessed on 6th Feb. 2024].

Yu, L., Du, B., Hu, X. et al. (2021a). Deep spatio-temporal graph convolutional network for traffic accident prediction. *Neurocomputing*, 423, pp. 135–147.

Yu, L., Du, B., Hu, X. et al. (2021b). Deep spatio-temporal graph convolutional network for traffic accident prediction. *Neurocomputing*, 423, pp. 135–147.

Yu, W. and Vega, F. (2020). Nonlinear system modelling using the Takagi-sugeno fuzzy model and long-short term memory cells. *Journal of Intelligent and Fuzzy Systems*, 39, pp. 4547–4556.

Yuan, Z., Zhou, X. and Yang, T. (Aug. 2018). Hetero-ConvLSTM: A Deep Learning Approach to Traffic Accident Prediction on Heterogenous Spatio-Temporal Data. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. London, England: Association for Computing Machinery, pp. 984–992.

Zagidullin, R. (2017). Model of Road Traffic Management in the City during Major Sporting Events. *Transportation Research Procedia*, 20, pp. 709–716.

Zeng, X., Jiang, Y., Ding, W. et al. (2023). A Hierarchical Spatio-Temporal Graph Convolutional Neural Network for Anomaly Detection in Videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 33 (1), pp. 200–212.

Zhang, Z., Yang, W. and Wushour, S. (2020). Traffic Accident Prediction Based on LSTM-GBRT Model. *Journal of Computer Science and Engineering*, 2020, p. 4206919. Available at: https://doi.org/10.1155/2020/4206919.

Zhou, X., Lu, P., Zheng, Z. et al. (2020). Accident Prediction Accuracy Assessment for Highway-Rail Grade Crossings Using Random Forest Algorithm Compared with

Decision Tree. *Reliability Engineering and System Safety*, 200. Available at: https://doi.org/10.1016/j.ress.2020.106931.

Appendix A Questionnaire

A.1 Questions

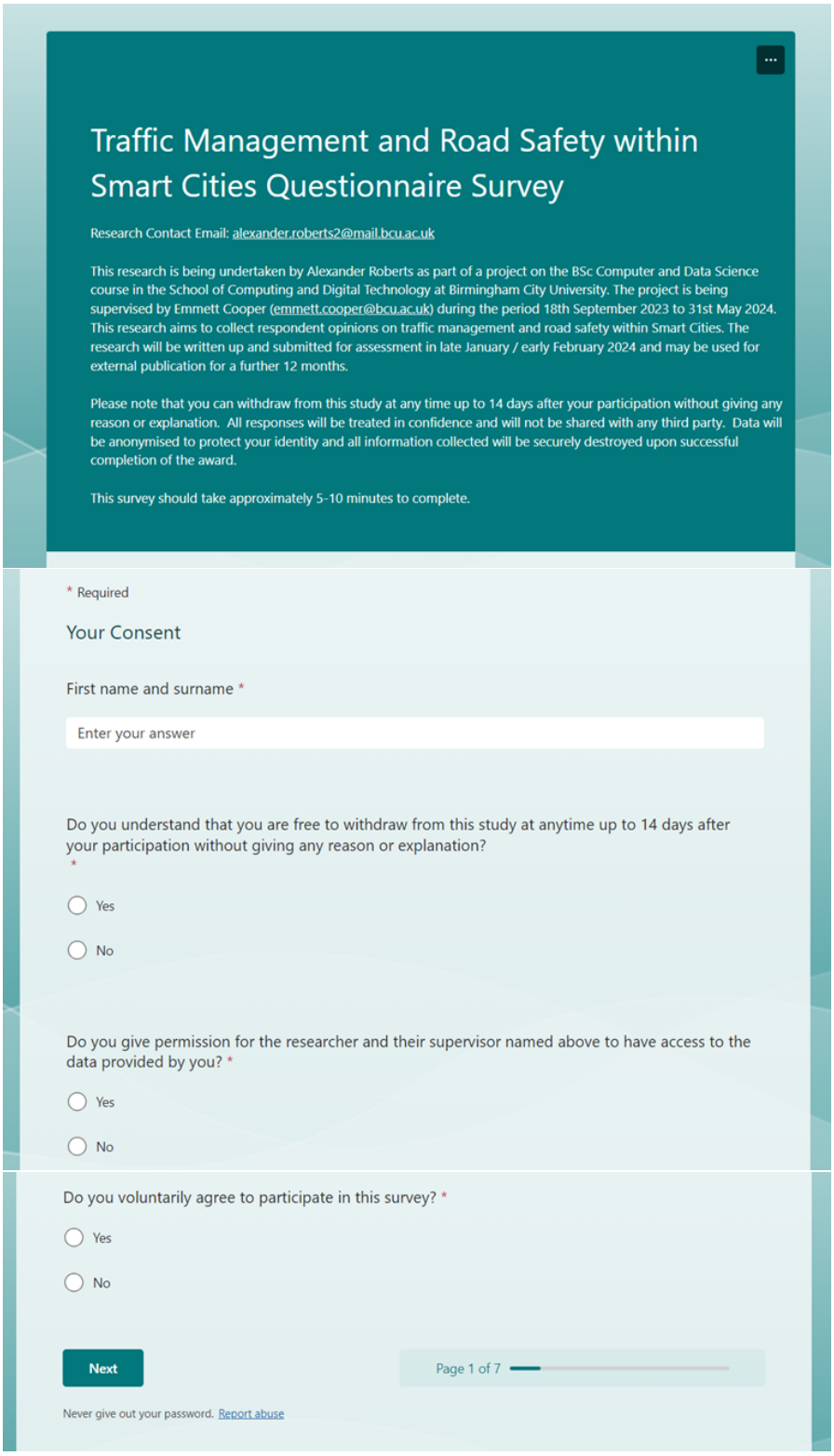


Figure A.1: Questionnaire part 1 — the respondent's consent.

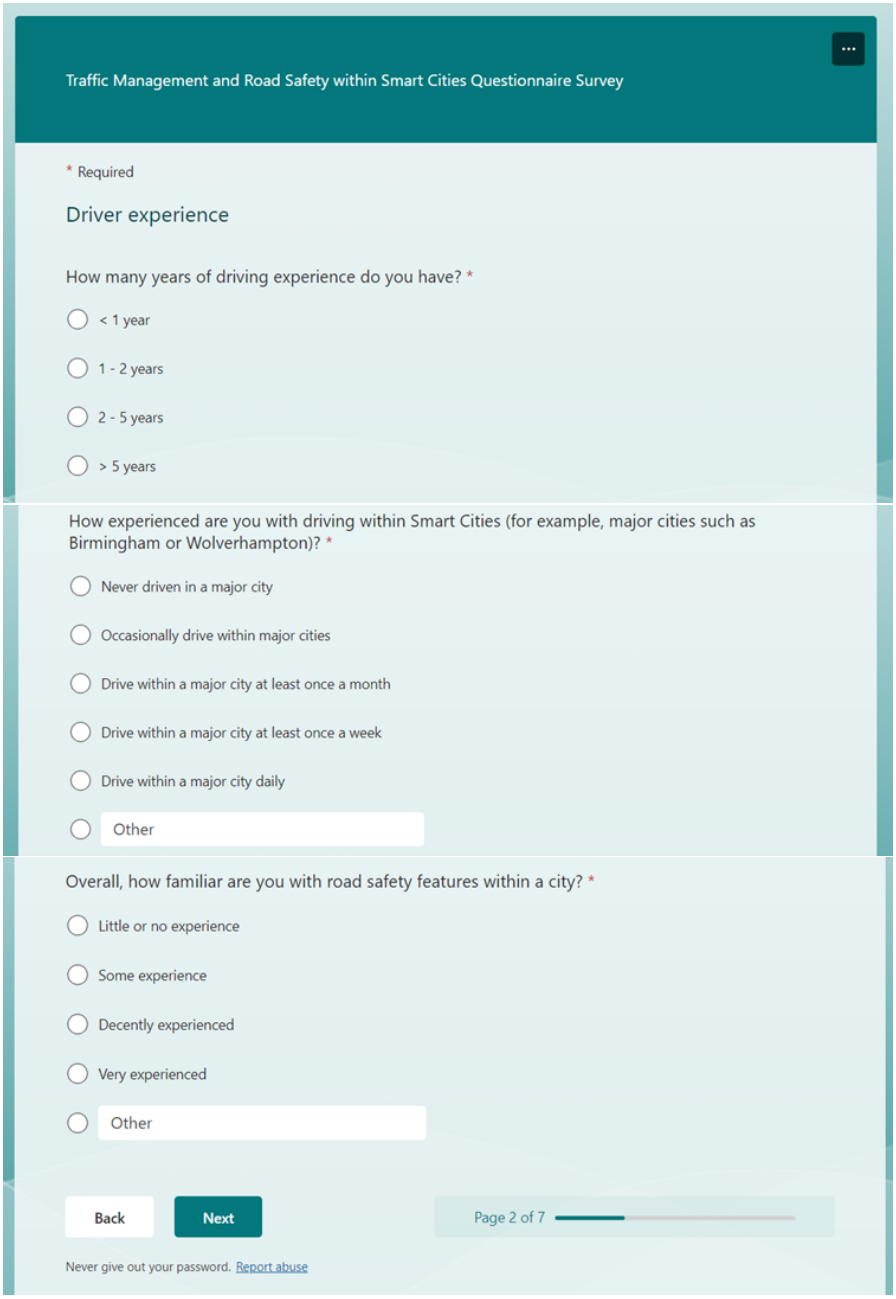


Figure A.2: Questionnaire part 2 — the respondent's experience.

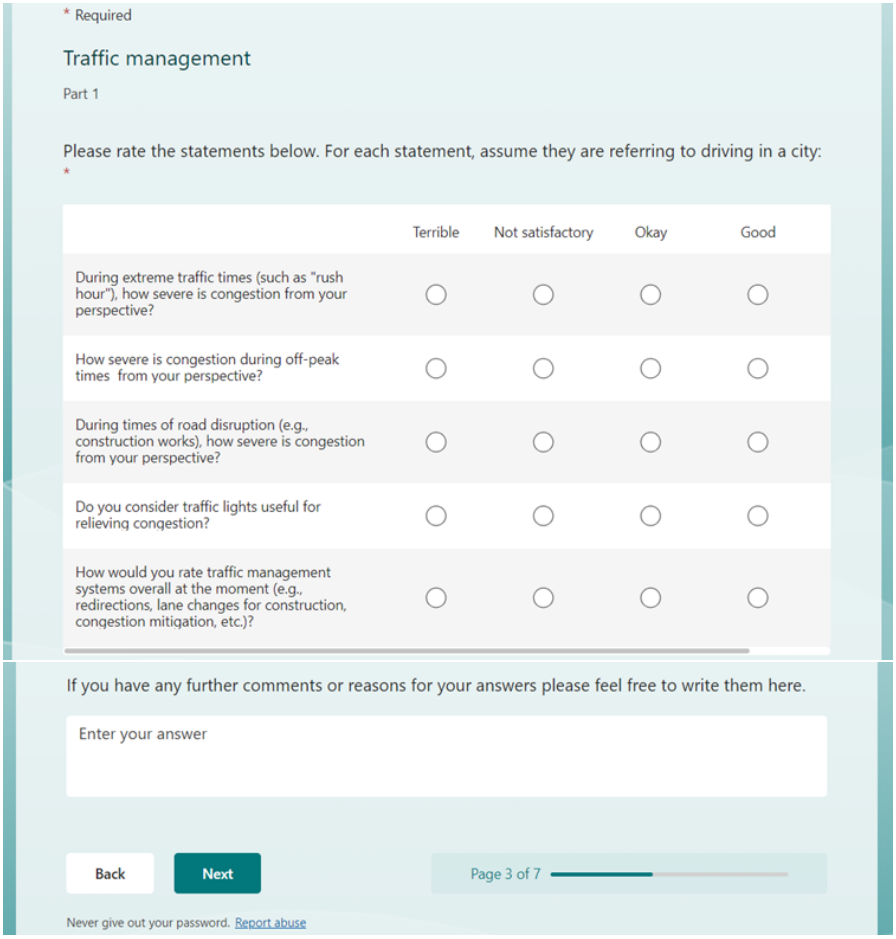


Figure A.3: Questionnaire part 3 — traffic management part 1.

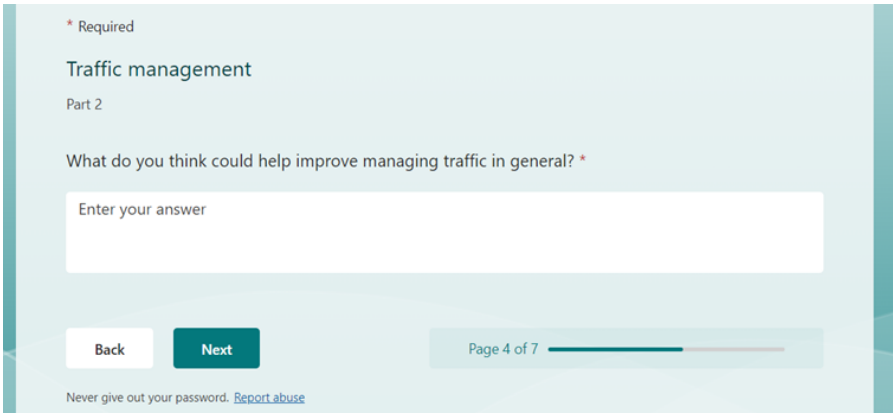


Figure A.4: Questionnaire part 4 — traffic management part 2.

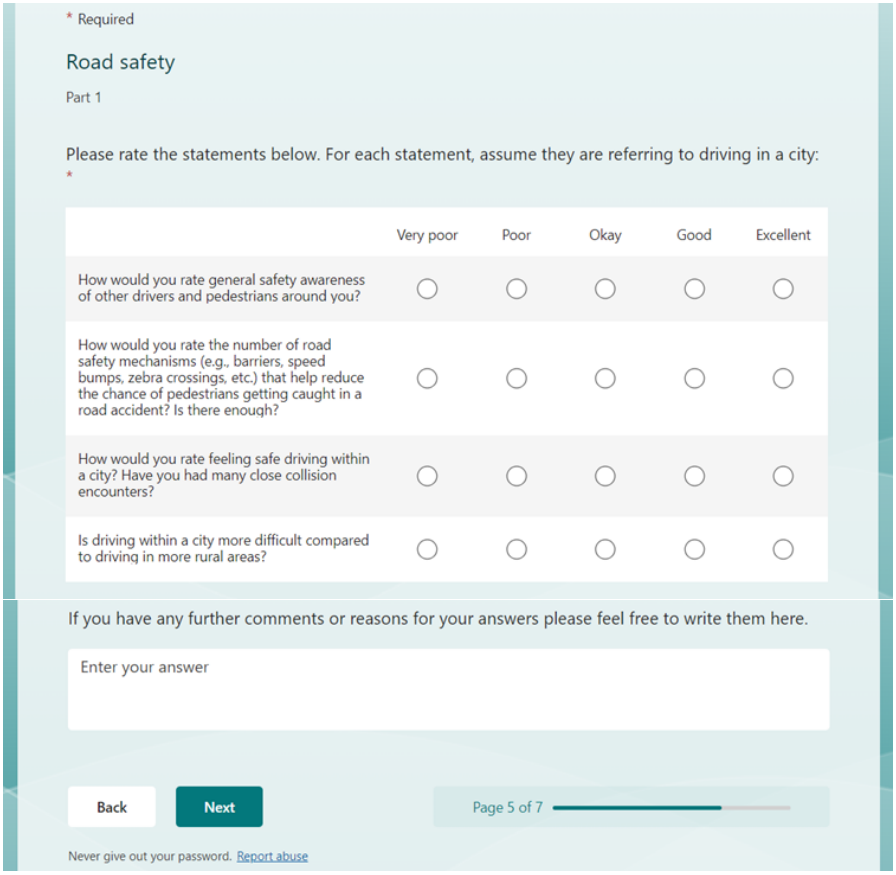


Figure A.5: Questionnaire part 5 — road safety part 1.

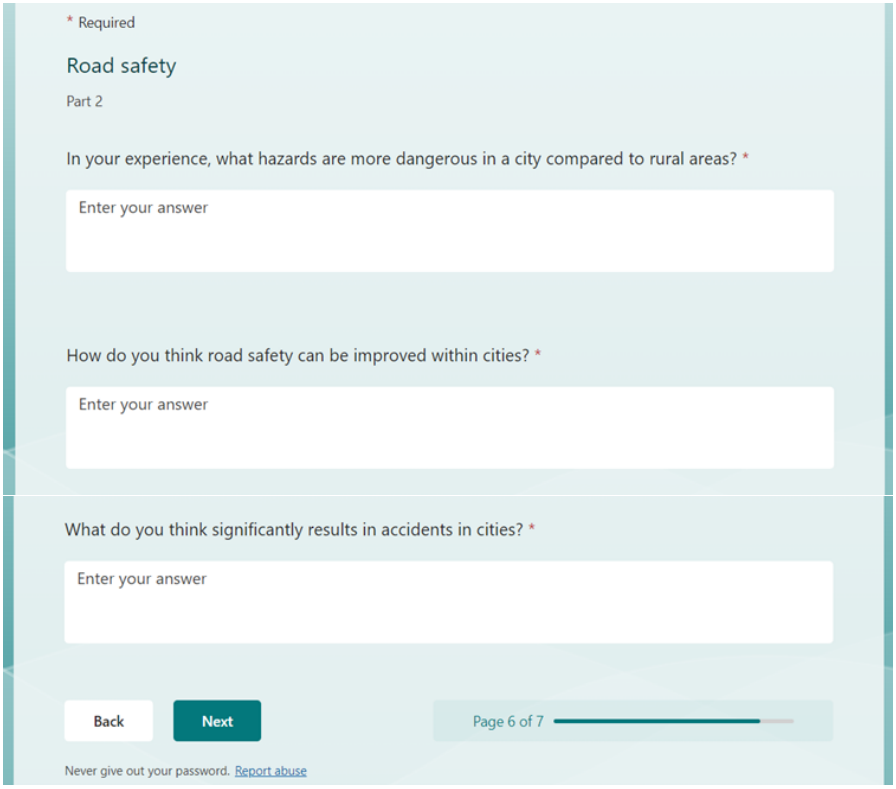


Figure A.6: Questionnaire part 6 — road safety part 2.

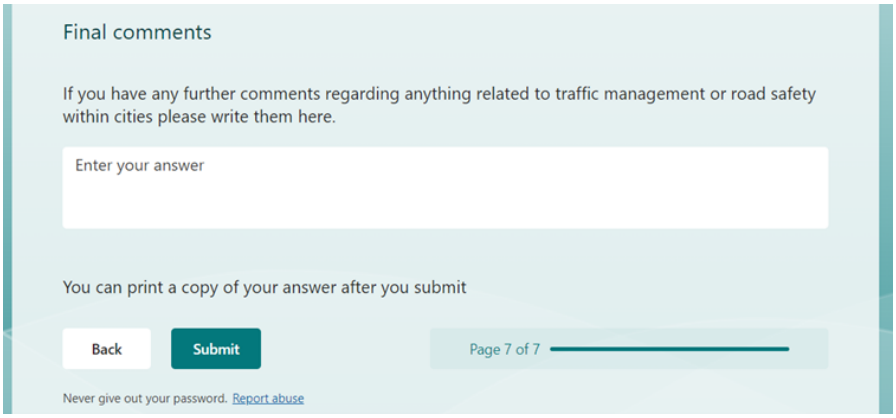


Figure A.7: Questionnaire part 7 — final comments.

A.2 Analysis

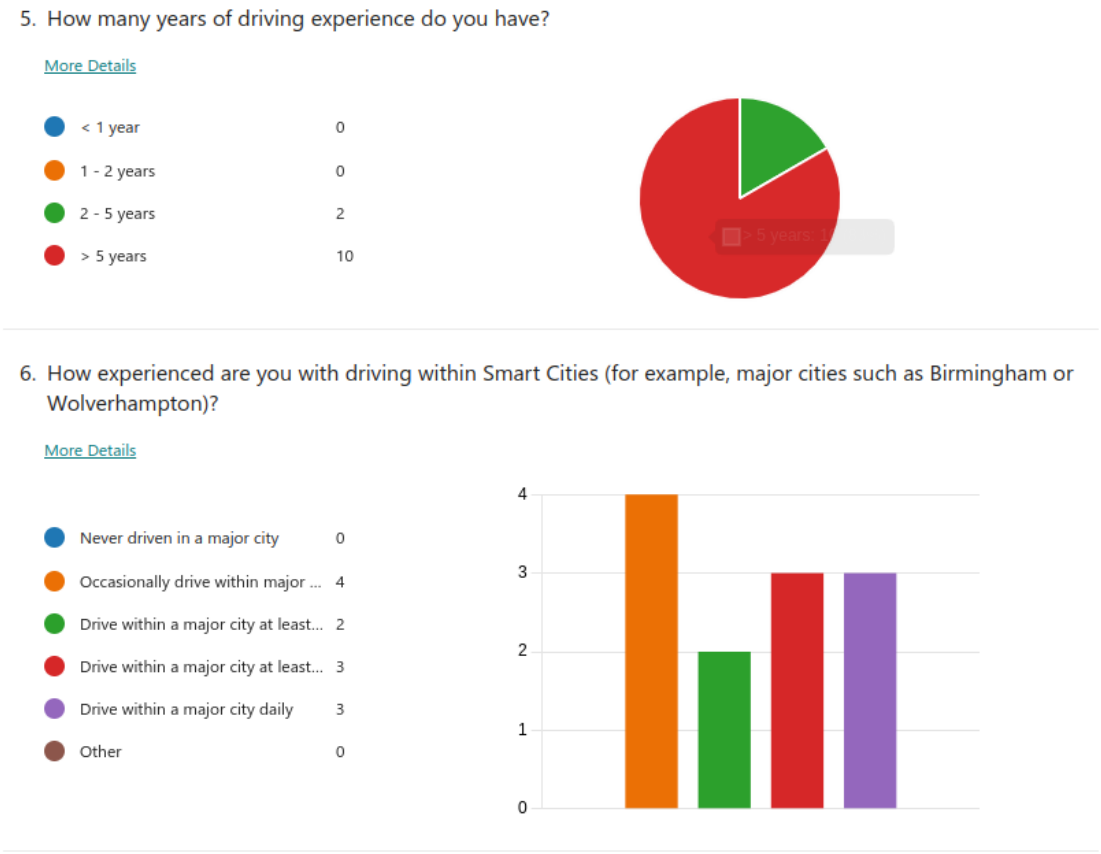


Figure A.8: Questionnaire response visualisations, part 1.

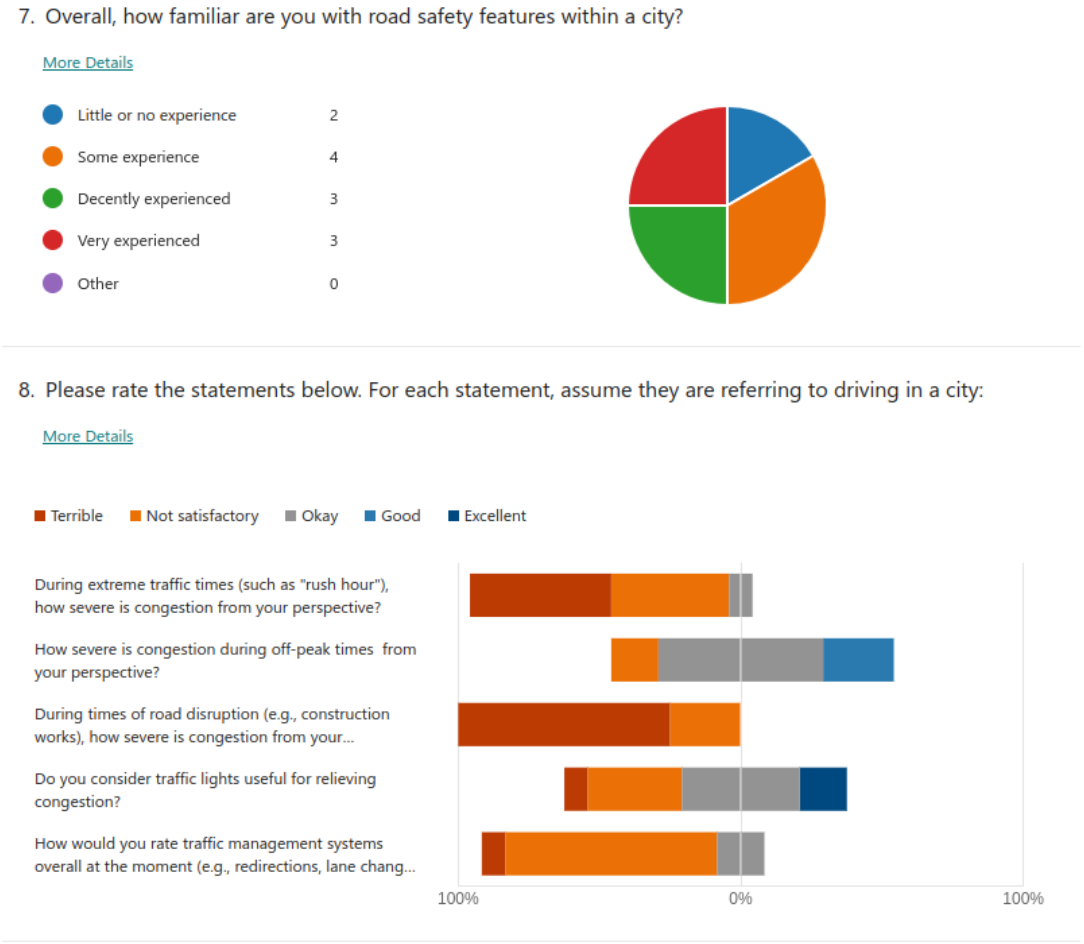


Figure A.9: Questionnaire response visualisations, part 2.

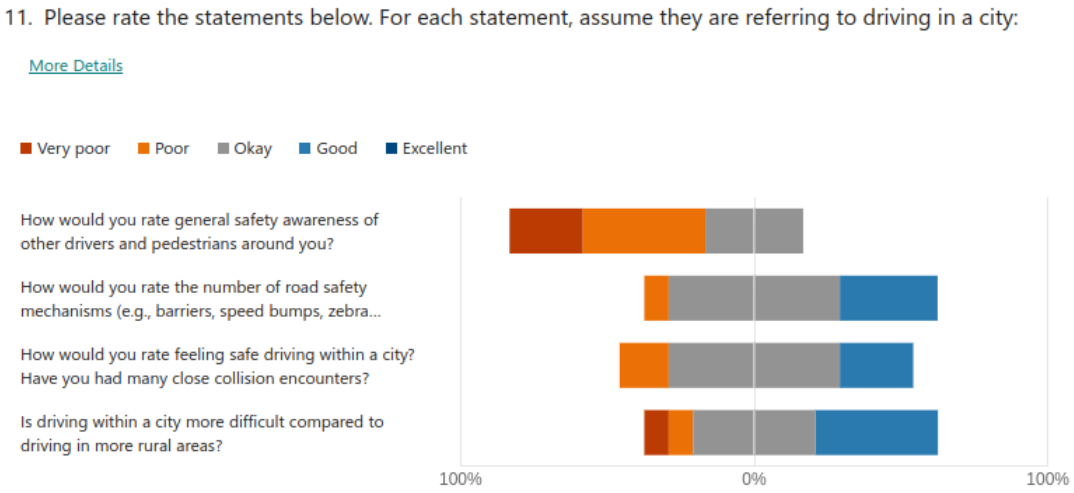


Figure A.10: Questionnaire response visualisations, part 3.

Appendix B Project Timeline

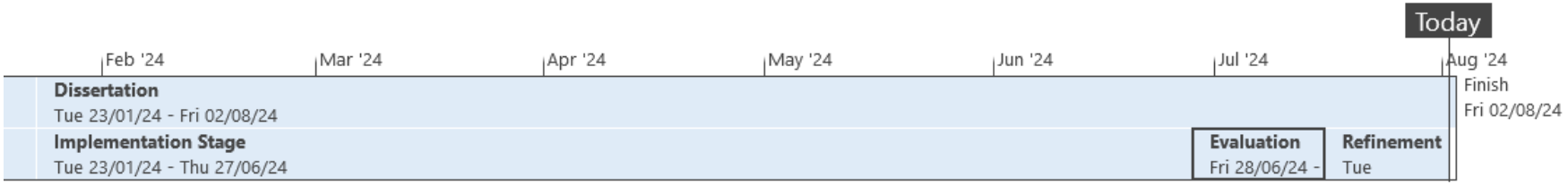


Figure B.1: Updated project timeline.

B.1 Former Project Timeline.

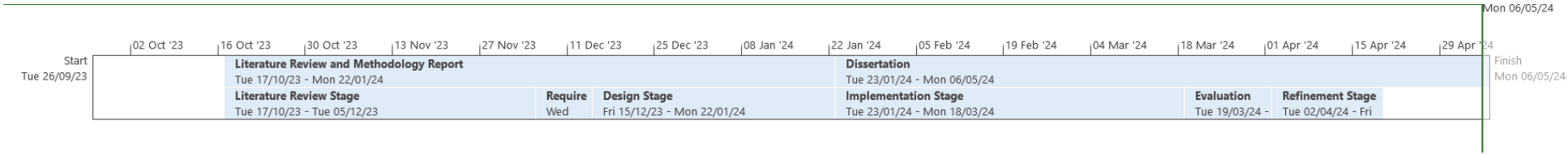


Figure B.2: Original project timeline.

Appendix C Code

C.1 Docker Compose: Containers Creation

The following code is in YAML format.

```yaml
name: fyp-implementation
services:
  ClickHouseServer:
    image: clickhouse/clickhouse-server:latest # retrieve the latest image
    ports:
      - 8123:8123 # host port : internal container port
    container_name: ClickHouseServer
    depends_on:
      MariaDBStore:
        condition: service_healthy
    environment:
      CLICKHOUSE_USER: AirflowUser122
      CLICKHOUSE_DB: airflow_storage
      CLICKHOUSE_DEFAULT_ACCESS_MANAGEMENT: 1
    healthcheck:
      test: wget --no-verbose --tries=1 --spider http://localhost:8123/ping || exit 1
      # 'wget' is a command used to retrieve web pages off of an address
        # '--no-verbose' removes any information being sent to the terminal - as it is not needed
        # '--tries' is how many attempts 'wget' should be tried, but this is not needed dues to the 'retries' of health check
        # '--spider' is a special command to not download the web page but just check to see if it exists and is responding
          # '--spider' essentially changes the output of 'wget' to be a boolean (true or false)
      interval: 10s
      retries: 3
      start_period: 30s

```

```
  RedisStore:
    image: redis:latest
    ports:
      - 6379:6379
    container_name: RedisStore
    healthcheck:
      test: redis-cli ping || exit 1
      # this 'test' will attempt to ping the Redis store through a Redis client (redis-cli)
      interval: 10s
      retries: 3
      start_period: 30s

  MariaDBStore:
    image: mariadb:latest
    ports:
      - 3306:3306
    container_name: MariaDBStore
    environment:
      MARIADB_USER: AirflowUser12233
      MARIADB_PASSWORD: AirflowPassword11223
      MARIADB_DATABASE: airflow_metadata_db
      MARIADB_RANDOM_ROOT_PASSWORD: 1
    healthcheck:
      test: ["CMD", "healthcheck.sh", "--connect", "--innodb_initialized"]
      # when a MariaDB instance is initialised, it already has a healthcheck file available
      interval: 10s
      retries: 3
      start_period: 30s

```

```
# 'test' is the command for the container to execute
# 'interval' is the time between each health check
# 'start_period' is an initial period when the results of health checks are not counted towards the maximum number of retries
# 'retries' is the number of attempts a failed health check will try before marking the service as unhealthy
```

C.2 Ingestion Stage

C.2.1 Functions file

```
# data manipulation
from dask.dataframe import DataFrame, concat, to_datetime, from_pandas
from pandas import read_csv

# server connection and table construction
from sqlalchemy import create_engine, Column, Integer, String, Float, DateTime, MetaData
from sqlalchemy.orm import Session
from clickhouse_sqlalchemy import Table
from clickhouse_sqlalchemy.engines import MergeTree
import pandahouse as ph

# for listing needed files
from os import listdir

def table_populated(engine_conn:dict, table_name:str):
    """
    Check to see if a table in the ClickHouse server is already populated.
    This is to help prevent data duplication.
    """
    connection = {
        "database":engine_conn['database'],
```

```
        "host":'http://{host}:{port}'.format(host=engine_conn['host'], port=engine_conn['port']),
        "user":engine_conn['user'],
        "password":engine_conn['password']
    }

    if(ph.execute("SELECT EXISTS(SELECT 1 FROM airflow_storage.{});".format(table_name),
                  connection=connection).decode('utf-8').find('1') != -1):
        return True
    # the above SQL attempts to retrieve a single record from the designated table in the ClickHouse server
    # the result from the executed SQL is interpreted as byte-code, which is converted
        # into a Python string using 'decode'
    # 'find' makes it so if a specific string value is not present, it will return -1
        # often known as False
    # if the table is populated, the resulting byte-code will contain '1' (true), otherwise
        # it will contain '0' (false)

    return False


def create_clickhouse_tables(engine_conn:dict,
                             road_table_name:str,
                             traffic_flow_table_name:str,
                             population_table_name:str,
                             gdp_table_name:str,
                             crime_table_name:str) -> None:
    """
    This will attempt to create the required tables in the ClickHouse server instance using the designated schemas.

    Parameters
```

```
    ----------
        | ``engine_conn``: A dictionary containing the connection details to the ClickHouse server instance.
        | ``road_table_name``: The desired table name for creating an appropriate schema for the Greater London road data.
        | ``traffic_flow_table_name``: ...
        | ``population_table_name``: ...
        | ``gdp_table_name``: ...
        | ``crime_table_name``: ...
    """
    engine_path_str = "clickhouse://{user}:{password}@{host}:{port}/{database}".format(user=engine_conn['user'],
                                                                                       password=engine_conn['password'],
                                                                                       host=engine_conn['host'],
                                                                                       port=engine_conn['port'],
                                                                                       database=engine_conn['database'])
    engine = create_engine(engine_path_str)

    road_table = Table(road_table_name, MetaData(),
                  Column(Integer, name = 'index', primary_key=True),
                  Column(String, name = 'Accident Ref'),
                  Column(String, name = 'Borough'),
                  Column(Integer, name = 'Borough Number'),
                  Column(Float, name = 'Easting'),
                  Column(Float, name = 'Northing'),
                  Column(String, name='Accident Severity'),
                  Column(Integer, name = 'Casualty Count'),
                  Column(Integer, name = 'Vehicle Count'),
                  Column(DateTime, name = 'Collision Date'),
                  Column(String, name = 'Day'),
                  Column(String, name = 'Time'),
                  Column(String, name = 'First Road Class'),
```

```
                  Column(String, name = "First Road Number"),
                  Column(String, name = "Road Type"),
                  Column(String, name = "Speed Limit"),
                  Column(String, name = "Junction Detail"),
                  Column(String, name = "Junction Control"),
                  Column(String, name = 'Second Road Class'),
                  Column(String, name = "Second Road Number"),
                  Column(String, name = "Pedestrian Crossing Facilities"),
                  Column(String, name = "Light Conditions"),
                  Column(String, name = "Weather Details"),
                  Column(String, name = "Road Surface Condition"),
                  Column(String, name = "Special Conditions at Site"),
                  Column(String, name = "Carriageway Hazards"),
                  Column(String, name = "Place Collision Reported"),
                  Column(String, name = "Collision Location Details"),
                  Column(Float, name = "Attendant Count"),
                  Column(String, name = "Highway Authority"),
                  MergeTree(order_by=('index')),
                  schema=engine_conn['database'])

    crime_table = Table(crime_table_name, MetaData(),
                    Column(Integer, name = 'index', primary_key=True),
                    Column(String, name = 'MajorText'),
                    Column(String, name = 'MinorText'),
                    Column(String, name = 'BoroughName'),
                    Column(Integer, name = '201004'),
                    Column(Integer, name = '201005'),
                    Column(Integer, name = '201006'),
                    Column(Integer, name = '201007'),
```

```
                Column(Integer, name = '201008'),
                Column(Integer, name = '201009'),
                Column(Integer, name = '201010'),
                Column(Integer, name = '201011'),
                Column(Integer, name = '201012'),
                Column(Integer, name = '201101'),
                Column(Integer, name = '201102'),
                Column(Integer, name = '201103'),
                Column(Integer, name = '201104'),
                Column(Integer, name = '201105'),
                Column(Integer, name = '201106'),
                Column(Integer, name = '201107'),
                Column(Integer, name = '201108'),
                Column(Integer, name = '201109'),
                Column(Integer, name = '201110'),
                Column(Integer, name = '201111'),
                Column(Integer, name = '201112'),
                Column(Integer, name = '201201'),
                Column(Integer, name = '201202'),
                Column(Integer, name = '201203'),
                Column(Integer, name = '201204'),
                Column(Integer, name = '201205'),
                Column(Integer, name = '201206'),
                Column(Integer, name = '201207'),
                Column(Integer, name = '201208'),
                Column(Integer, name = '201209'),
                Column(Integer, name = '201210'),
                Column(Integer, name = '201211'),
                Column(Integer, name = '201212'),
```

```
                Column(Integer, name = '201301'),
                Column(Integer, name = '201302'),
                Column(Integer, name = '201303'),
                Column(Integer, name = '201304'),
                Column(Integer, name = '201305'),
                Column(Integer, name = '201306'),
                Column(Integer, name = '201307'),
                Column(Integer, name = '201308'),
                Column(Integer, name = '201309'),
                Column(Integer, name = '201310'),
                Column(Integer, name = '201311'),
                Column(Integer, name = '201312'),
                Column(Integer, name = '201401'),
                Column(Integer, name = '201402'),
                Column(Integer, name = '201403'),
                Column(Integer, name = '201404'),
                Column(Integer, name = '201405'),
                Column(Integer, name = '201406'),
                Column(Integer, name = '201407'),
                Column(Integer, name = '201408'),
                Column(Integer, name = '201409'),
                Column(Integer, name = '201410'),
                Column(Integer, name = '201411'),
                Column(Integer, name = '201412'),
                Column(Integer, name = '201501'),
                Column(Integer, name = '201502'),
                Column(Integer, name = '201503'),
                Column(Integer, name = '201504'),
                Column(Integer, name = '201505'),
```

```
                Column(Integer, name = '201506'),
                Column(Integer, name = '201507'),
                Column(Integer, name = '201508'),
                Column(Integer, name = '201509'),
                Column(Integer, name = '201510'),
                Column(Integer, name = '201511'),
                Column(Integer, name = '201512'),
                Column(Integer, name = '201601'),
                Column(Integer, name = '201602'),
                Column(Integer, name = '201603'),
                Column(Integer, name = '201604'),
                Column(Integer, name = '201605'),
                Column(Integer, name = '201606'),
                Column(Integer, name = '201607'),
                Column(Integer, name = '201608'),
                Column(Integer, name = '201609'),
                Column(Integer, name = '201610'),
                Column(Integer, name = '201611'),
                Column(Integer, name = '201612'),
                Column(Integer, name = '201701'),
                Column(Integer, name = '201702'),
                Column(Integer, name = '201703'),
                Column(Integer, name = '201704'),
                Column(Integer, name = '201705'),
                Column(Integer, name = '201706'),
                Column(Integer, name = '201707'),
                Column(Integer, name = '201708'),
                Column(Integer, name = '201709'),
                Column(Integer, name = '201710'),
```

```
                Column(Integer, name = '201711'),
                Column(Integer, name = '201712'),
                Column(Integer, name = '201801'),
                Column(Integer, name = '201802'),
                Column(Integer, name = '201803'),
                Column(Integer, name = '201804'),
                Column(Integer, name = '201805'),
                Column(Integer, name = '201806'),
                Column(Integer, name = '201807'),
                Column(Integer, name = '201808'),
                Column(Integer, name = '201809'),
                Column(Integer, name = '201810'),
                Column(Integer, name = '201811'),
                Column(Integer, name = '201812'),
                Column(Integer, name = '201901'),
                Column(Integer, name = '201902'),
                Column(Integer, name = '201903'),
                Column(Integer, name = '201904'),
                Column(Integer, name = '201905'),
                Column(Integer, name = '201906'),
                Column(Integer, name = '201907'),
                Column(Integer, name = '201908'),
                Column(Integer, name = '201909'),
                Column(Integer, name = '201910'),
                Column(Integer, name = '201911'),
                Column(Integer, name = '201912'),
                Column(Integer, name = '202001'),
                Column(Integer, name = '202002'),
                Column(Integer, name = '202003'),
```

```
                    Column(Integer, name = '202004'),
                    Column(Integer, name = '202005'),
                    Column(Integer, name = '202006'),
                    Column(Integer, name = '202007'),
                    Column(Integer, name = '202008'),
                    Column(Integer, name = '202009'),
                    Column(Integer, name = '202010'),
                    Column(Integer, name = '202011'),
                    Column(Integer, name = '202012'),
                    Column(Integer, name = '202101'),
                    Column(Integer, name = '202102'),
                    Column(Integer, name = '202103'),
                    Column(Integer, name = '202104'),
                    Column(Integer, name = '202105'),
                    Column(Integer, name = '202106'),
                    Column(Integer, name = '202107'),
                    Column(Integer, name = '202108'),
                    Column(Integer, name = '202109'),
                    Column(Integer, name = '202110'),
                    Column(Integer, name = '202111'),
                    Column(Integer, name = '202112'),
                    Column(Integer, name = '202201'),
                    Column(Integer, name = '202202'),
                    Column(Integer, name = '202203'),
                    Column(Integer, name = '202204'),
                    Column(Integer, name = '202205'),
                    Column(Integer, name = '202206'),
                    MergeTree(order_by=('index')),
                    schema=engine_conn['database'])
```

```
    gdp_table = Table(gdp_table_name, MetaData(),
                Column(Integer, name = 'index', primary_key=True),
                Column(String, name = 'ITL1 Region'),
                Column(String, name = 'LA code'),
                Column(String, name = 'LA name'),
                Column(Integer, name = '1998'),
                Column(Integer, name = '1999'),
                Column(Integer, name = '2000'),
                Column(Integer, name = '2001'),
                Column(Integer, name = '2002'),
                Column(Integer, name = '2003'),
                Column(Integer, name = '2004'),
                Column(Integer, name = '2005'),
                Column(Integer, name = '2006'),
                Column(Integer, name = '2007'),
                Column(Integer, name = '2008'),
                Column(Integer, name = '2009'),
                Column(Integer, name = '2010'),
                Column(Integer, name = '2011'),
                Column(Integer, name = '2012'),
                Column(Integer, name = '2013'),
                Column(Integer, name = '2014'),
                Column(Integer, name = '2015'),
                Column(Integer, name = '2016'),
                Column(Integer, name = '2017'),
                Column(Integer, name = '2018'),
                Column(Integer, name = '2019'),
                Column(Integer, name = '2020'),
```

```
283                 Column(Integer, name = '2021'),
284                 Column(Integer, name = '2022'),
285                 MergeTree(order_by=('index')),
286                 schema=engine_conn['database'])
287 
288     population_table = Table(population_table_name, MetaData(),
289                 Column(Integer, name = 'index', primary_key=True),
290                 Column(String, name = 'ITL1 Region'),
291                 Column(String, name = 'LA code'),
292                 Column(String, name = 'LA name'),
293                 Column(Integer, name = '1998'),
294                 Column(Integer, name = '1999'),
295                 Column(Integer, name = '2000'),
296                 Column(Integer, name = '2001'),
297                 Column(Integer, name = '2002'),
298                 Column(Integer, name = '2003'),
299                 Column(Integer, name = '2004'),
300                 Column(Integer, name = '2005'),
301                 Column(Integer, name = '2006'),
302                 Column(Integer, name = '2007'),
303                 Column(Integer, name = '2008'),
304                 Column(Integer, name = '2009'),
305                 Column(Integer, name = '2010'),
306                 Column(Integer, name = '2011'),
307                 Column(Integer, name = '2012'),
308                 Column(Integer, name = '2013'),
309                 Column(Integer, name = '2014'),
310                 Column(Integer, name = '2015'),
311                 Column(Integer, name = '2016'),
```

```
                Column(Integer, name = '2017'),
                Column(Integer, name = '2018'),
                Column(Integer, name = '2019'),
                Column(Integer, name = '2020'),
                Column(Integer, name = '2021'),
                Column(Integer, name = '2022'),
                MergeTree(order_by=('index')),
                schema=engine_conn['database'])

    traffic_flow_table = Table(traffic_flow_table_name, MetaData(),
                            Column(Integer, name = 'index', primary_key=True),
                            Column(String, name = 'LA Code'),
                            Column(String, name = 'Local Authority'),
                            Column(Integer, name = '1993'),
                            Column(Integer, name = '1994'),
                            Column(Integer, name = '1995'),
                            Column(Integer, name = '1996'),
                            Column(Integer, name = '1997'),
                            Column(Integer, name = '1998'),
                            Column(Integer, name = '1999'),
                            Column(Integer, name = '2000'),
                            Column(Integer, name = '2001'),
                            Column(Integer, name = '2002'),
                            Column(Integer, name = '2003'),
                            Column(Integer, name = '2004'),
                            Column(Integer, name = '2005'),
                            Column(Integer, name = '2006'),
                            Column(Integer, name = '2007'),
                            Column(Integer, name = '2008'),
```

```
                              Column(Integer, name = '2009'),
                              Column(Integer, name = '2010'),
                              Column(Integer, name = '2011'),
                              Column(Integer, name = '2012'),
                              Column(Integer, name = '2013'),
                              Column(Integer, name = '2014'),
                              Column(Integer, name = '2015'),
                              Column(Integer, name = '2016'),
                              Column(Integer, name = '2017'),
                              Column(Integer, name = '2018'),
                              Column(Integer, name = '2019'),
                              Column(Integer, name = '2020'),
                              Column(Integer, name = '2021'),
                              Column(Integer, name = '2022'),
                              MergeTree(order_by=('index')),
                              schema=engine_conn['database'])

    # 'with' is being used as a context manager, so once the transaction is complete, the connection is closed automatically
    with Session(engine) as session:
        road_table.create(session.connection(), checkfirst = True)
        traffic_flow_table.create(session.connection(), checkfirst = True)
        population_table.create(session.connection(), checkfirst = True)
        gdp_table.create(session.connection(), checkfirst = True)
        crime_table.create(session.connection(), checkfirst = True)
        # 'checkfirst' will make the program check to see if the table exists already
        # if it does exist, it will not attempt to create a new table. Otherwise, it will
        session.commit() # completes the transaction

def ingest_road_data(directory_path : str) -> DataFrame:
```

```
    """
    A custom function that attempts to ingest specific Greater London road data from files in
    a given directory, and then returns the combined data. Column names are adjusted to ensure
    the final result is consistent and uniform.

    Returns
    -------
        The combined and slightly transformed data in the form of a Dask DataFrame.
    """
    # a custom dictionary containing column replacement names
    # this will be used to make the final combined dataframe columns uniform and consistent
    replace_col_names_dict = {
        'AREFNO':'Accident Ref',
        'Accident Ref.':'Accident Ref',
        'Boro':'Borough Number',
        'Road No. 1':'First Road Number',
        'Road No. 2':'Second Road Number',
        'Road No 2':'Second Road Number',
        'Road Class 1':'First Road Class',
        'Road Class 2':'Second Road Class',
        'No. of Casualties in Acc.':'Casualty Count',
        '_Casualty Count':'Casualty Count',
        'No. of Vehicles in Acc.':'Vehicle Count',
        '_Vehicle Count':'Vehicle Count',
        'Accident Date':'Collision Date',
        '_Collision Date':'Collision Date',
        'Weather':'Weather Details',
        'Light Conditions (Banded)':'Light Conditions',
        'Road Surface':'Road Surface Condition',
```

```
        'Ped. Crossing Decoded':'Pedestrian Crossing Facilities',
        'C/W Hazard':'Carriageway Hazards',
        'Special Conditions':'Special Conditions at Site',
        'Day Name':'Day',
        'Borough Name':'Borough',
        '_Collision Severity':'Accident Severity',
        'APOLICER_DECODED':'Place Collision Reported',
        'Collision Location':'Collision Location Details',
        '_Attendant Count':'Attendant Count',
        'Location':'Collision Location Details',
        'Highway':'Highway Authority'
    }

    dd_frames = []
    # array list to temporarily store the TfL files while they are being ingested
    # these file are then concatenated together into one large DataFrame

    for file in listdir(directory_path):
        ddf : DataFrame = from_pandas(read_csv(directory_path+'/'+file, encoding = 'cp1252'), npartitions=16)
        #cp1252 is the standard encoding (codec) for Western Europe
        ddf = ddf.rename(columns=replace_col_names_dict)
        # rename columns of DataFrames so data is combined properly
        dd_frames.append(ddf)

    return concat(dd_frames, interleave_partitions=True, axis = 0)
    # 'interleave_partitions=True' concatenates DataFrame ignoring its order. The order doesn't matter in this case, as it
    ↪  will be reorded later anyway
    #'axis = 0' ensures the dataframes will be concatenated row-wise
```

```
427 def ingest_data(directory_path : str) -> DataFrame:
428     """
429     A function that attempts to retreve CSV data from a specified file, then return this data in a Dask DataFrame.
430 
431     Returns
432     -------
433         A Dask DataFrame with the CSV data.
434     """
435     ddf : DataFrame = from_pandas(read_csv(directory_path), npartitions=16)
436     return ddf
437 
438 def preprocess_road_data(ddf : DataFrame) -> DataFrame:
439     """
440     Performs basic and required transformations to the road data so that it can be stored properly.
441     """
442 
443     # remove basic outliers
444     # every entry should have a Borough name with an associated accident severity for future plans
445     ddf = ddf[ddf['Borough'].notnull() & ddf['Accident Severity'].notnull()]
446 
447     # a pre-defined list of column names to be replaced
448     columns_data_to_replace = ['Speed Limit', 'Accident Severity', 'Junction Detail',
449                        'Junction Control', 'Pedestrian Crossing Facilities', 'First Road Class', 'Second Road Class',
                           ↪  'Road Type',
450                        'Light Conditions', 'Weather Details', 'Road Surface Condition', 'Special Conditions at Site',
                           ↪  'Carriageway Hazards',
451                       'Highway Authority', 'Collision Location Details', 'Place Collision Reported']
452 
453     # the below code removed numbers such as '3' from '3 30 MPH', making the result '30 MPH'
```

```
    for col in columns_data_to_replace:
        ddf[col] = ddf[col].str.replace(r'^\d\s*|^-\d\s*', '', regex=True)
        # use regular expression (regex) to select prior number to actual data and remove them
        # '^' means at the beginning of the string, '\d' means if a number is present
        # '\s' means any whitespace character, '*' means however many - so however many whitespaces
        # '|' is the equivalent of OR - so if the string matches either expression, the former for positive numbers, the
        ↪  latter for negative numbers

    # some of the dates are in slightly different formats
    ddf['Collision Date'] = ddf['Collision Date'].str.replace('/', '-')
    ddf['Collision Date'] = ddf['Collision Date'].str.replace(" 00:00", '')

    # the below is needed to ensure the dataframe types are consistent with its intended table schema
    ddf['Collision Date'] = to_datetime(ddf['Collision Date'], format = "mixed", dayfirst=True)
    ddf['Attendant Count'] = ddf['Attendant Count'].astype(float) # must be converted to float due to containing 'NaN' values
        # 'NaN' values are classed as floating point values
    ddf['Casualty Count'] = ddf['Casualty Count'].astype(int)
    ddf['Vehicle Count'] = ddf['Vehicle Count'].astype(int)
    ddf['Borough Number'] = ddf['Borough Number'].astype(int)

    return ddf

def upload_ingested_data(ddf : DataFrame, engine_conn : dict, table_name : str) -> None:
    """
    Upload a Dask DataFrame to a ClickHouse server.
    """
    # it is necessary to alter the engine connection dictionary slightly to suit the pandahouse package
    connection = {
        "database":engine_conn['database'],
```

```
        "host":'http://{host}:{port}'.format(host=engine_conn['host'], port=engine_conn['port']),
        "user":engine_conn['user'],
        "password":engine_conn['password']
    }
    ph.to_clickhouse(ddf.compute(), table_name, index = True, connection=connection)
```

C.2.2 Control flow file

```
# import logging to log any errors or info to Airflow
import logging

from .functions import *

def ingest_data_main(london_acc_data_path:str,
                     traffic_flow_data_path:str,
                     population_data_path:str,
                     gdp_data_path:str,
                     crime_data_path:str,
                     logger:logging.Logger,
                     engine_conn:dict,
                     london_acc_table_name:str,
                     traffic_flow_table_name:str,
                     population_table_name:str,
                     gdp_table_name:str,
                     crime_table_name:str) -> None:
    """
    Function that manages the program flow of the ingestion stage.
    """
    try:
```

```
        create_clickhouse_tables(engine_conn=engine_conn,
                                 road_table_name=london_acc_table_name,
                                 traffic_flow_table_name=traffic_flow_table_name,
                                 population_table_name=population_table_name,
                                 gdp_table_name=gdp_table_name,
                                 crime_table_name=crime_table_name)

        # ingest Greater London road accident data
        # this data requires extra transformations compared to the other datasets,
            # so some custom functions are needed
        if not table_populated(engine_conn=engine_conn, table_name=london_acc_table_name):
            upload_ingested_data(
                preprocess_road_data(
                    ingest_road_data(directory_path=london_acc_data_path)
                    ),
                engine_conn=engine_conn,
                table_name=london_acc_table_name
            )

        # ingest traffic flow data
        if not table_populated(engine_conn=engine_conn, table_name=traffic_flow_table_name):
            upload_ingested_data(
                ingest_data(traffic_flow_data_path),
                engine_conn=engine_conn,
                table_name=traffic_flow_table_name
            )

        if not table_populated(engine_conn=engine_conn, table_name=population_table_name):
            # ingest population data
```

```
            upload_ingested_data(
                ingest_data(population_data_path),
                engine_conn=engine_conn,
                table_name=population_table_name
            )

        if not table_populated(engine_conn=engine_conn, table_name=gdp_table_name):
            # ingest GDP data
            upload_ingested_data(
                ingest_data(gdp_data_path),
                engine_conn=engine_conn,
                table_name=gdp_table_name
            )

        if not table_populated(engine_conn=engine_conn, table_name=crime_table_name):
            # ingest crime data
            upload_ingested_data(
                ingest_data(crime_data_path),
                engine_conn=engine_conn,
                table_name=crime_table_name
            )

    except Exception as e:
        logger.critical("Error occured during ingestion stage.\n{}".format(e), exc_info=1)
    else:
        logger.info("Ingestion stage successful!")
```

C.2.3 Model creation pipeline after Ingestion stage

```
from datetime import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.empty import EmptyOperator
import logging
from fyp_package import ingestion
#setup of logger to record notable events
logging.basicConfig(level=logging.WARN) #log only levels including: WARNING, ERROR and CRITICAL
logger = logging.getLogger(__name__) #create the a longger instance with the name of '__main__'
#default arguments for the DAG; required

DEFAULT_ARGS = {
    "owner":"Alex",
    "depends_on_past":True, #every upstream task must be successful for the DAG to continue
    "email":["alexander.roberts2@mail.bcu.ac.uk"],
    "email_on_failure":False,
    "email_on_retry":False,
    "retries":0,
}
DEFAULT_ROAD_DATA_PATH = '/home/alex/FYP_dir/fyp_data/tfl_road_data'
DEFAULT_TRAFFIC_FLOW_DATA_PATH = '/home/alex/FYP_dir/fyp_data/traffic-flow-borough-all-vehicles.csv'
DEFAULT_POPULATION_DATA_PATH = '/home/alex/FYP_dir/fyp_data/ONS mid-year population estimates London boroughs.csv'
DEFAULT_GDP_DATA_PATH = '/home/alex/FYP_dir/fyp_data/gdp at current basic rates.csv'
DEFAULT_CRIME_DATA_PATH = '/home/alex/FYP_dir/fyp_data/MPS Borough Level Crime (Historical).csv'

# connection dictionary to the ClickHouse server instance
ENGINE_CONN = {
    "database":'airflow_storage',
```

```
    "host":'localhost',
    "user":'AirflowUser122',
    "password":'',
    "port":8123
}

# creates unique table names in the ClickHouser server instance
# e.g., 'london_road_accidents_12Jun24
ROAD_DATA_TABLE_NAME = "london_road_accidents_"+datetime.today().strftime('%d%b%y')
TRAFFIC_FLOW_TABLE_NAME = "traffic_flow_london_boroughs_"+datetime.today().strftime('%d%b%y')
POPULATION_TABLE_NAME = "population_london_boroughs_"+datetime.today().strftime('%d%b%y')
GDP_TABLE_NAME = "gdp_london_boroughs_"+datetime.today().strftime('%d%b%y')
CRIME_TABLE_NAME = "crime_london_boroughs_"+datetime.today().strftime('%d%b%y')

with DAG(
    "fyp_model_creation", # DAG name
    default_args=DEFAULT_ARGS,
    description="This pipeline is for the development of a machine learning model.",
    start_date=datetime(2022,10,10), # required otherwise the DAG cannot proceed
    schedule_interval=None,
    catchup=False, # prevent the pipeline from performing scheduler catchup
    tags=["fyp", "mlops"],
) as dag:
    start=EmptyOperator(task_id="start")
    ingestion_task=PythonOperator(task_id="ingestion_stage", python_callable=ingestion.ingest_data_main,
        op_kwargs={
            'london_acc_data_path':DEFAULT_ROAD_DATA_PATH,
            'traffic_flow_data_path':DEFAULT_TRAFFIC_FLOW_DATA_PATH,
            'population_data_path':DEFAULT_POPULATION_DATA_PATH,
```

```
            'gdp_data_path':DEFAULT_GDP_DATA_PATH,
            'crime_data_path':DEFAULT_CRIME_DATA_PATH,
            'logger':logger,
            'engine_conn':ENGINE_CONN,
            'london_acc_table_name':ROAD_DATA_TABLE_NAME,
            'traffic_flow_table_name':TRAFFIC_FLOW_TABLE_NAME,
            'population_table_name':POPULATION_TABLE_NAME,
            'gdp_table_name':GDP_TABLE_NAME,
            'crime_table_name':CRIME_TABLE_NAME,
        })
    end=EmptyOperator(task_id="end")
    start >> ingestion_task >> end
```

C.3 Preprocessing Stage

C.3.1 Functions file

```
# data manipulation
from pandas import cut, DateOffset, DataFrame as PandasDataFrame, Series as PandasSeries
from dask.dataframe import DataFrame, to_datetime, from_pandas
from dask_ml.preprocessing import LabelEncoder
from datetime import datetime
from sklearn.preprocessing import MinMaxScaler
from numpy import ndarray

# for static type checking, but this will not be enforced
from typing import Union

# server connection
import pandahouse as ph
from pickle import dumps
from redis import Redis

# for converting data into tensors
from torch import Tensor, tensor

def categorise_risk_level(value:int) -> int:
    if value < 2:
        return 1
    elif (value >= 2) and (value < 4):
        return 2
    elif (value>=4) and (value < 8):
        return 3
```

```
    elif (value>=8) and (value < 12):
        return 4
    elif (value>=12):
        return 5

def create_risk_level(df: DataFrame) -> DataFrame:
    """
    Create a 'Risk Level' column in the merged dataframe by transforming the 'Accident Severity' column.
    This function requires all labels of the dataframe to be numerically encoded.

    Returns
    -------
        | A Dask DataFrame with a 'Accident Severity' column transformed into a 'Risk Level' column.
    """
    # aggregation dictionary - tells the 'agg' function how to combine elements
    agg_dict = {
        'DayOfWeek': 'first',
        'Year': 'first',
        'Month': 'first',
        'Day': 'first',
        'Light Conditions': 'first',
        'Weather Details': 'first',
        'GDP Prev Year': 'first',
        'Population Prev Year': 'first',
        'Traffic Flow Prev Year': 'first',
        'Vehicle Crime Offences Prev Month': 'first',
        'Accident Severity': 'sum'
    }

```

```
    df : DataFrame = from_pandas(df.compute().groupby(by=['Borough', 'Collision Date', 'Time'],
                                                      as_index=False, dropna=True).agg(agg_dict).reset_index(drop=True),
                                                      npartitions=16)
    # 'as_index' prevents the designated columns becoming the index
    # 'dropna' is present as the nature of 'groupby' can lead to NaN values in some columns (which getting rid of does not
        # affect the original data)
    # the above is essentially used to combine rows together into one, and aggregate the individual values based on 'agg_dict'

    df = df.rename(columns={
        'Accident Severity':'Risk Level'
    })

    # now the values have been aggregated together, it is time to designate the actual risk levels
    # this can be seen in the 'categorise_risk_level' function
    df['Risk Level'] = df['Risk Level'].map_partitions(lambda df: df.apply(categorise_risk_level))

    return df

def create_weekday_col(df: DataFrame) -> DataFrame:
    """
    Take the specific date of a datetime entry (e.g., 16th September 2014) and add the corresponding
    day of the week to a new column, 'DayOfWeek'.
    """
    df['DayOfWeek'] = df['Collision Date'].dt.weekday
    return df

def crime_dataset_feature_selection(df: DataFrame) -> DataFrame:
    """
    Select features of the crime-related dataset that are linked with vehicle offences (e.g., vehicle theft, drug trafficking, etc.).
```

```
85
86      Returns
87      -------
88          | A Dask DataFrame with certain vehicle-related features selected.
89      """
90      return df[df['MajorText'] == 'VEHICLE OFFENCES']
91
92  def data_fusion(london_acc_df: DataFrame, crime_df: DataFrame, population_df: DataFrame,
93                  gdp_df: DataFrame, traffic_flow_df: DataFrame) -> DataFrame:
94      """
95      Merge the five designated datasets together into one Dask DataFrame.
96      """
97      # get only the numerical columns of the other dataframes
98      gdp_numeric_cols = [col for col in gdp_df.columns if col.isdigit()]
99      pop_numeric_cols = [col for col in population_df.columns if col.isdigit()]
100     traffic_flow_numerical_cols = [col for col in traffic_flow_df.columns if col.isdigit()]
101     crime_num_cols = [col for col in crime_df.columns if col.isdigit()]
102
103     # 'melt' is a special function used to essentially convert a wide-dataframe into a long-dataframe (reduce columns and
        ↪ increase rows)
104     # this is used due to some of the dataframes having the year and/or month set as the column name
105     gdp_long : PandasDataFrame = gdp_df.compute().melt(
106         id_vars=['Borough'],
107         value_vars=gdp_numeric_cols, var_name='Year',
108         value_name='GDP Prev Year')
109     population_long : PandasDataFrame = population_df.compute().melt(
110         id_vars=['Borough'],
111         value_vars=pop_numeric_cols, var_name='Year',
112         value_name='Population Prev Year')
```

```
    traffic_long : PandasDataFrame = traffic_flow_df.compute().melt(
        id_vars=['Borough'],
        value_vars=traffic_flow_numerical_cols, var_name='Year',
        value_name='Traffic Flow Prev Year')
    crime_long : PandasDataFrame = crime_df.compute().melt(
        id_vars=['Borough'],
        value_vars=crime_num_cols, var_name='Year_Month',
        value_name='Vehicle Crime Offences Prev Month')

    # seperate datetime entries in the crime dataframe, e.g., '201004' becomes year 2010, month 04 (April)
    crime_long['Year'] = crime_long['Year_Month'].str[:4]
    crime_long['Month'] = crime_long['Year_Month'].str[4:]

    crime_long['Year'] = crime_long['Year'].astype(int)
    crime_long['Month'] = crime_long['Month'].astype(int)

    # convert the dates in the crime dataframe into a datetime format
    # this is so each month can be pushed forward by 1 month properly
        # this is to match up entries such that current data becomes past data
    crime_long['Year_Month'] = to_datetime(crime_long['Year'].astype(str) + '-' + crime_long['Month'].astype(str),
                                           format='%Y-%m')
    crime_long['Year_Month'] = crime_long['Year_Month'] + DateOffset(months=1)

    # aggregate entries together that have the same borough and time to get the total vehicle offences for that time
    crime_long = crime_long.groupby(['Borough', 'Year',
                                     'Month'], as_index=False)['Vehicle Crime Offences Prev Month'].sum()

    # Convert the years from a string type to an integer type, ready for combining them
    gdp_long['Year'] = gdp_long['Year'].astype(int)
```

```
    population_long['Year'] = population_long['Year'].astype(int)
    traffic_long['Year'] = traffic_long['Year'].astype(int)

    # make certain data go from future data to past data for entries the year after
        # a similar thing was done for the crime dataset, except the month was pushed
        # forward by 1, as it is on a month-by-month basis
    gdp_long['Year'] = gdp_long['Year']+1
    population_long['Year'] = population_long['Year']+1
    traffic_long['Year'] = traffic_long['Year']+1

    # Merge the DataFrames
    merged_df : DataFrame = london_acc_df.merge(gdp_long, on=['Borough', 'Year'], how='left')
    merged_df = merged_df.merge(population_long, on=['Borough', 'Year'], how='left')
    merged_df = merged_df.merge(traffic_long, on=['Borough', 'Year'], how='left')

    return merged_df.merge(crime_long, on=['Borough', 'Year', 'Month'], how='left')

def date_selection_and_seperation(df: DataFrame) -> DataFrame:
    """
    For the London road accidents dataset, remove all entries before 2011 and after 2021.
    This will also break down any remaining datetimes into the respective year, month and day of the month.
    """
    df[(df['Collision Date'] > datetime(2011, 1, 1)) & (df['Collision Date'] < datetime(2022, 1, 1))]
    # date determined through EDA analysis
    df['Year'] = df['Collision Date'].dt.year
    df['Month'] = df['Collision Date'].dt.month
    df['Day'] = df['Collision Date'].dt.day
    return df

```

```
171 def decapitalise_col_values(df: DataFrame) -> DataFrame:
172     """
173     Decapitalise values of certain columns of the Greater London accident data, so values are consistent.
174     """
175 
176     df['Borough'] = df['Borough'].str.lower()
177     df['Weather Details'] = df['Weather Details'].str.lower()
178     df['Accident Severity'] = df['Accident Severity'].str.lower()
179     df['Light Conditions'] = df['Light Conditions'].str.lower()
180     return df
181 
182 def drop_collision_date(df: PandasDataFrame) -> PandasDataFrame:
183     """
184     Drop the collision date column from the merged dataframe.
185     """
186     # 'Collision Date' no longer needed
187     return df.drop(columns='Collision Date')
188 
189 def london_acc_feature_selection(df: DataFrame) -> DataFrame:
190     """
191     Select specific features of the London road accidents dataset.
192     """
193     # selected features that relate to a Borough level
194     return df[['Borough', 'Collision Date', 'Time', 'Accident Severity', 'Light Conditions', 'Weather Details']]
195 
196 def label_encoding(df: DataFrame) -> DataFrame:
197     """
198     Encode categorical features of the merged dataframe.
199     """
```

```
    le_borough = LabelEncoder()
    le_time = LabelEncoder()
    le_light_conditions = LabelEncoder()
    le_weather = LabelEncoder()

    df['Borough'] = le_borough.fit_transform(df['Borough'])
    df['Time'] = le_time.fit_transform(df['Time'])
    df['Light Conditions'] = le_light_conditions.fit_transform(df['Light Conditions'])
    df['Weather Details'] = le_weather.fit_transform(df['Weather Details'])

    df['Accident Severity'] = df['Accident Severity'].replace('slight', 1)
    df['Accident Severity'] = df['Accident Severity'].replace('serious', 4)
    df['Accident Severity'] = df['Accident Severity'].replace('fatal', 12)

    return df

def make_light_conditions_consistent(df: DataFrame) -> DataFrame:
    """
    A function to make the values of the light conditions column of the London road accidents dataset consistent.
    """
    df['Light Conditions'] = df['Light Conditions'].str.replace(r"dark.+", "dark", regex=True)
    return df

def make_time_consistent(df: DataFrame) -> DataFrame:
    """
    A function to make the values of the time of day column of the London road accidents dataset consistent.
    Additionally, this column type will be converted to datetime.
    """
    df['Time'] = df['Time'].str.replace("'", "")
```

```
    df['Time'] = df['Time'].str.replace(":", "")
    df['Time'] = df['Time'].str.replace("\\", "")
    df['Time'] = df['Time'].str.replace(r"00$", "", regex=True)
    df['Time'] = to_datetime(df['Time'], format='%H%M')
    df['Time'] = df['Time'].dt.hour
    df['Time'] = df['Time'].astype(int)
    return df

def remove_duplicates_and_nan(df: DataFrame) -> DataFrame:
    """
    Remove duplicates and NaN values of a dataset using Complete Case Analysis.
    """
    return df.drop_duplicates().dropna(how='any')

def remove_unknown_weather_conditions(df: DataFrame) -> DataFrame:
    """
    Remove entries which have 'unknown' weather conditions.
    """
    return df[df['Weather Details'] != "unknown"]

def retrieve_data_from_ClickHouse(engine_conn: dict, table_name: str) -> DataFrame:
    """
    Retrieves data from designated tables from a ClickHouse server.

    Parameters
    ----------
        | ``engine_conn``: A dictionary containing the connection information to the ClickHouse server.
        | ``table_name``: The name of the table matching that in the ClickHouse server; for the data to be retrieved from.

```

```
    Returns
    -------
        Data in the form of a Dask DataFrame.
    """
    connection = {
        "database":engine_conn['database'],
        "host":'http://{host}:{port}'.format(host=engine_conn['host'], port=engine_conn['port']),
        "user":engine_conn['user'],
        "password":engine_conn['password']
    }

    return from_pandas(ph.read_clickhouse('SELECT * FROM {}.{}'.format(
        engine_conn['database'], table_name), index = True, index_col = 'index', connection=connection), npartitions=16)

def rename_cols(df: DataFrame, col_rename_dict: dict[str, str]) -> DataFrame:
    """
    Rename columns within a Dask DataFrame.
    """
    return df.rename(columns=col_rename_dict)

def scale_X(split_data: dict[str, Union[PandasDataFrame, PandasSeries]]) -> dict[str, Union[ndarray, PandasSeries]]:
    """
    Scale the indepedent variables of the training and testing sets between the values of 0 and 1.

    Parameters
    --------
        ``split_data``: the training and testing sets (e.g., X_train, X_test, y_train, y_test).
    """
    minmaxscaler = MinMaxScaler(feature_range=(0,1))
```

```
287
288     split_data['X_train'] = minmaxscaler.fit_transform(split_data['X_train'])
289     split_data['X_val'] = minmaxscaler.transform(split_data['X_val'])
290     split_data['X_test'] = minmaxscaler.transform(split_data['X_test'])
291
292     return split_data
293
294 def standardise_borough_names(df: DataFrame) -> DataFrame:
295     """
296     Make the names of boroughs consistent by removing and editing punctuation and capitalisation.
297     """
298     df['Borough'] = df['Borough'].str.replace('-', ' ')
299     df['Borough'] = df['Borough'].str.replace('&', 'and')
300     df['Borough'] = df['Borough'].str.lower()
301     return df
302
303 def sort_values(df: DataFrame) -> PandasDataFrame:
304     """
305     Change the order of rows by organising the date and time (of day) of each entry.
306     Then, the index will be reset so as to accomodate these changes. This makes the
307     data become chronological.
308
309     Returns
310     -------
311         | A Pandas DataFrame of the sorted data. It is a Pandas DataFrame so the index order
312         is maintained, unlike in Dask due to partitioning.
313     """
314     return df.compute().sort_values(by=['Collision Date', 'Time']).reset_index(drop=True)
```

```
315     # must be converted into a Pandas DataFrame so the index is maintained when split into independent variables and target
        ↪ variable
316
317 def time_discretisation(df: DataFrame) -> DataFrame:
318     """
319     Convert a positive integer sequence of values which represent each hour of the day
320     into a select number of bins with associated labels.
321     """
322     # as 'Time' contains integer values, the bins need to be real number as well
323     # they are float values as the bins work by collecting values between each index
324         # e.g., values between 0 and 2 (not including 2) will be put in bin 1
325         # which has a label of '00:00-01:59'
326     bins = [0, 1.99, 3.99, 5.99, 7.99, 9.99, 11.99, 13.99, 15.99, 17.99, 19.99, 21.99, 23.99]
327
328     # the labels represent the time of day using a 24-hour clock
329     labels = ['00:00-01:59', '02:00-03:59', '04:00-05:59', '06:00-07:59', '08:00-09:59', '10:00-11:59', '12:00-13:59',
330         '14:00-15:59', '16:00-17:59', '18:00-19:59', '20:00-21:59', '22:00-23:59']
331
332     df['Time'] = df['Time'].map_partitions(cut, bins=bins, labels=labels, include_lowest=True, ordered=True)
333     # 'map_partitions' is a Dask function used to apply a function to each partition in the dataframe
334         # in this case, it will be the Pandas 'cut' method - which is used to bin values into discrete values
335     # 'ordered=True' will match each index in the bins and labels together
336     # 'include_lowest=True' will include the first index, so instead of values being binned like
337         # 0 < x < 2, it becomes 0 <= x < 2
338     return df
339
340 def train_val_test_X_y_split(df: PandasDataFrame) -> dict[str, Union[PandasDataFrame, PandasSeries]]:
341     """
342     Split the dataframe into one dataframe for independent variables and one dataframe
```

```
    for the target variable. Then, split these dataframes into the training and testing sets.
    """
    train_size = int(0.7 * len(df))
    test_size = int(0.2 * len(df))
    val_size = len(df)-test_size
    # 70% training, 10% validation, 20% testing
    # in chronological order, so 70% will be oldest entries, etc.

    X_train = df.iloc[0:train_size, :].drop(columns='Risk Level')
    X_val = df.iloc[train_size:val_size, :].drop(columns='Risk Level')
    X_test = df.iloc[val_size:len(df), :].drop(columns='Risk Level')
    y_train = df.iloc[0:train_size, :]['Risk Level']
    y_val = df.iloc[train_size:val_size, :]['Risk Level']
    y_test = df.iloc[val_size:len(df), :]['Risk Level']
    # separation of independent variables and target variable
    # the rows are sorted by the 'Collision Date' column, then the index is reset

    return {'X_train':X_train, 'X_val':X_val, 'X_test':X_test, 'y_train':y_train, 'y_val':y_val, 'y_test':y_test}

def torch_tensor_conversion(split_data: dict[str, Union[ndarray, PandasSeries]]) -> dict[str, Tensor]:
    """
    Convert training and testing sets into PyTorch Tensors, so they can be processed by PyTorch.

    Returns
    -------
        | A dictionary of the training and testing sets, each converted into a PyTorch tensor.
    """
    X_train = split_data['X_train']
    X_val = split_data['X_val']
```

```
    X_test = split_data['X_test']
    y_train = split_data['y_train']
    y_val = split_data['y_val']
    y_test = split_data['y_test']
    # 'X_train', 'X_val' and 'X_test' are already numpy arrays

    return {
        'X_train': tensor(X_train),
        'X_val': tensor(X_val),
        'X_test': tensor(X_test),
        'y_train': tensor(y_train.to_numpy()),
        'y_val': tensor(y_val.to_numpy()),
        'y_test': tensor(y_test.to_numpy())
    }

def upload_to_redis(data: dict[str, Tensor], redis_host: str, redis_port: int) -> None:
    """
    Upload training and testing sets into a Redis datastore, so it can be passed to the next pipeline task.
    """
    X_train = data['X_train']
    X_val = data['X_val']
    X_test = data['X_test']
    y_train = data['y_train']
    y_val = data['y_val']
    y_test = data['y_test']

    # using 'with' as a context manager, so connection is closed automatically after the code has been run
    with Redis(host=redis_host, port=redis_port) as redis_conn:
        redis_conn.set(name="X_train_preprocessed", value=dumps(X_train))
```

```
        redis_conn.set(name="X_val_preprocessed", value=dumps(X_val))
        redis_conn.set(name="X_test_preprocessed", value=dumps(X_test))
        redis_conn.set(name="y_train_preprocessed", value=dumps(y_train))
        redis_conn.set(name="y_val_preprocessed", value=dumps(y_val))
        redis_conn.set(name="y_test_preprocessed", value=dumps(y_test))

    # Pickle 'dumps' converts the data into a bytes Python object, so it can be stored in Redis more effectively
```

C.3.2 Control flow file

```
# import logging to log any errors or info to Airflow
import logging

from .functions import *

def preprocess_data_main(logger:logging.Logger, engine_conn:dict, london_acc_table_name:str, population_table_name:str,
                         gdp_table_name:str, traffic_flow_table_name: str, crime_table_name:str, redis_host:str,
                         redis_port:int) -> None:
    """
    Function that manages the program flow of the preprocessing stage.
    """
    try:
        upload_to_redis(
            torch_tensor_conversion(
                scale_X(
                    train_val_test_X_y_split(
                        drop_collision_date(
                            sort_values(
                                create_risk_level(
```

```
                                label_encoding(
                                    time_discretisation(
                                        remove_duplicates_and_nan(
                                            data_fusion(
                                                london_acc_df=date_selection_and_seperation(
                                                    create_weekday_col(
                                                        make_time_consistent(
                                                            make_light_conditions_consistent(
                                                                remove_unknown_weather_conditions(
                                                                    standardise_borough_names(
                                                                        decapitalise_col_values(
                                                                            london_acc_feature_selection(
                                                                                retrieve_data_from_ClickHouse(
                                                                                    engine_conn=engine_conn,
                                                                                    table_name=london_acc_table_name)
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                ),
                                                population_df=standardise_borough_names(
                                                    rename_cols(
                                                        df=retrieve_data_from_ClickHouse(
                                                            engine_conn=engine_conn,
                                                            table_name=population_table_name),
                                                        col_rename_dict={'LA name':'Borough'}
```

```
                    )
                ),
                gdp_df=standardise_borough_names(
                    rename_cols(
                        df=retrieve_data_from_ClickHouse(
                            engine_conn=engine_conn,
                            table_name=gdp_table_name),
                        col_rename_dict={'LA name':'Borough'}
                    )
                ),
                traffic_flow_df=standardise_borough_names(
                    rename_cols(
                        df=retrieve_data_from_ClickHouse(
                            engine_conn=engine_conn,
                            table_name=traffic_flow_table_name),
                        col_rename_dict={'Local Authority':'Borough'}
                    )
                ),
                crime_df=crime_dataset_feature_selection(
                    standardise_borough_names(
                        rename_cols(
                            df=retrieve_data_from_ClickHouse(
                                engine_conn=engine_conn, table_name=crime_table_name),
                            col_rename_dict={'BoroughName':'Borough'}
                        )
                    )
                )
            )
        )
```

```
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            ),
            redis_host=redis_host,
            redis_port=redis_port
        )
    except Exception as e:
        logger.critical("Error occured during preprocessing stage.\n{}".format(e), exc_info=1)
    else:
        logger.info("Preprocessing stage successful!")
```

C.3.3 Model creation pipeline after Preprocessing stage

```
from datetime import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.empty import EmptyOperator
import logging
from fyp_package import ingestion, preprocessing
#setup of logger to record notable events
logging.basicConfig(level=logging.WARN) #log only levels including: WARNING, ERROR and CRITICAL
logger = logging.getLogger(__name__) #create the a longger instance with the name of '__main__'
#default arguments for the DAG; required

```

```
DEFAULT_ARGS = {
    "owner":"Alex",
    "depends_on_past":True, #every upstream task must be successful for the DAG to continue
    "email":["alexander.roberts2@mail.bcu.ac.uk"],
    "email_on_failure":False,
    "email_on_retry":False,
    "retries":0,
}
DEFAULT_ROAD_DATA_PATH = '/home/alex/FYP_dir/fyp_data/tfl_road_data'
DEFAULT_TRAFFIC_FLOW_DATA_PATH = '/home/alex/FYP_dir/fyp_data/traffic-flow-borough-all-vehicles.csv'
DEFAULT_POPULATION_DATA_PATH = '/home/alex/FYP_dir/fyp_data/ONS mid-year population estimates London boroughs.csv'
DEFAULT_GDP_DATA_PATH = '/home/alex/FYP_dir/fyp_data/gdp at current basic rates.csv'
DEFAULT_CRIME_DATA_PATH = '/home/alex/FYP_dir/fyp_data/MPS Borough Level Crime (Historical).csv'

# connection dictionary to the ClickHouse server instance
ENGINE_CONN = {
    "database":'airflow_storage',
    "host":'localhost',
    "user":'AirflowUser122',
    "password":'',
    "port":8123
}
REDIS_HOST = '127.0.0.1'
REDIS_PORT = 6379

# creates unique table names in the ClickHouser server instance
# e.g., 'london_road_accidents_12Jun24
ROAD_DATA_TABLE_NAME = "london_road_accidents_"+datetime.today().strftime('%d%b%y')
TRAFFIC_FLOW_TABLE_NAME = "traffic_flow_london_boroughs_"+datetime.today().strftime('%d%b%y')
```

```
POPULATION_TABLE_NAME = "population_london_boroughs_"+datetime.today().strftime('%d%b%y')
GDP_TABLE_NAME = "gdp_london_boroughs_"+datetime.today().strftime('%d%b%y')
CRIME_TABLE_NAME = "crime_london_boroughs_"+datetime.today().strftime('%d%b%y')

with DAG(
    "fyp_model_creation", # DAG name
    default_args=DEFAULT_ARGS,
    description="This pipeline is for the development of a machine learning model.",
    start_date=datetime(2022,10,10), # required otherwise the DAG cannot proceed
    schedule_interval=None,
    catchup=False, # prevent the pipeline from performing scheduler catchup
    tags=["fyp", "mlops"],
) as dag:
    start=EmptyOperator(task_id="start")
    ingestion_task=PythonOperator(task_id="ingestion_stage", python_callable=ingestion.ingest_data_main,
        op_kwargs={
            'london_acc_data_path':DEFAULT_ROAD_DATA_PATH,
            'traffic_flow_data_path':DEFAULT_TRAFFIC_FLOW_DATA_PATH,
            'population_data_path':DEFAULT_POPULATION_DATA_PATH,
            'gdp_data_path':DEFAULT_GDP_DATA_PATH,
            'crime_data_path':DEFAULT_CRIME_DATA_PATH,
            'logger':logger,
            'engine_conn':ENGINE_CONN,
            'london_acc_table_name':ROAD_DATA_TABLE_NAME,
            'traffic_flow_table_name':TRAFFIC_FLOW_TABLE_NAME,
            'population_table_name':POPULATION_TABLE_NAME,
            'gdp_table_name':GDP_TABLE_NAME,
            'crime_table_name':CRIME_TABLE_NAME,
        })
```

```
    preprocessing_task=PythonOperator(task_id="preprocessing_stage", python_callable=preprocessing.preprocess_data_main,
        op_kwargs={
            'logger':logger,
            'engine_conn':ENGINE_CONN,
            'london_acc_table_name':ROAD_DATA_TABLE_NAME,
            'population_table_name':POPULATION_TABLE_NAME,
            'gdp_table_name':GDP_TABLE_NAME,
            'traffic_flow_table_name':TRAFFIC_FLOW_TABLE_NAME,
            'crime_table_name':CRIME_TABLE_NAME,
            'redis_host':REDIS_HOST,
            'redis_port':REDIS_PORT
        })
    end=EmptyOperator(task_id="end")
    start >> ingestion_task >> preprocessing_task >> end
```

C.3.4 Ingested data EDA

Retrieving the data from the ClickHouse server, then subsequently storing each into a Dask DataFrame.

```
london_acc_df = dd.from_pandas(
    ph.read_clickhouse('SELECT * FROM airflow_storage.london_road_accidents_24Jul24',
                       index_col='index',
                       connection=connection),
    npartitions = 16)

traffic_flow_df = dd.from_pandas(
    ph.read_clickhouse('SELECT * FROM airflow_storage.traffic_flow_london_boroughs_24Jul24',
                       index_col='index',
                       connection=connection),
    npartitions = 16)

population_df = dd.from_pandas(
    ph.read_clickhouse('SELECT * FROM airflow_storage.population_london_boroughs_24Jul24',
                       index_col='index',
                       connection=connection),
    npartitions = 16)

gdp_df = dd.from_pandas(
    ph.read_clickhouse('SELECT * FROM airflow_storage.gdp_london_boroughs_24Jul24',
                       index_col='index',
                       connection=connection),
    npartitions = 16)

crime_df = dd.from_pandas(
    ph.read_clickhouse('SELECT * FROM airflow_storage.crime_london_boroughs_24Jul24',
                       index_col='index',
                       connection=connection),
    npartitions = 16)
```

Figure C.1: Retrieving the ingested data.

London accident data

Apply the 80:20 split ratio.

```
train_size = int(0.8 * len(london_acc_df))
print("Index training set goes to: {}".format(train_size))
# 80% training
```

```
Index training set goes to: 364650
```

The below cells require the 'london_acc_df' Dask DataFrame to be come converted into a Pandas DataFrame, as the index needs to be whole and not paritioned in order to slice it correctly. This is done using 'compute', 'sort_values' and 'reset_index'.

```
london_acc_df_train = dd.from_pandas(
    london_acc_df.compute().sort_values(by='Collision Date').reset_index(drop=True).iloc[0:train_size, :],
    npartitions=16)
# the dataframe is ordered chronologically using 'Collision Date', then it has the index reset to maintain consistency
# 'drop=True' prevents the 'reset_index' adding a new column to the output dataframe which would have contained the original indexing
# '[iloc[0:train_size, :]' selects all of the columns from index 0 to the train_size index, the remainder are not used as these would make
    # up the testing set
# The resulting training set is then again converted back into a Dask DataFrame to help preserve in-memory
```

```
london_acc_df_test = dd.from_pandas(
    london_acc_df.compute().sort_values(by='Collision Date').reset_index(drop=True).iloc[train_size:len(london_acc_df), :],
    npartitions=16)
```

```
print("Original dataset shape: {} \nTraining dataset shape: {} \nTesting dataset shape: {}".format(london_acc_df.compute().shape,
                                                                                                   london_acc_df_train.compute().shape,
                                                                                                   london_acc_df_test.compute().shape))
```

```
Original dataset shape: (455813, 29)
Training dataset shape: (364650, 29)
Testing dataset shape: (91163, 29)
```

(a) Splitting the main dataset (of recorded road accidents in Greater London).

Traffic flow data

Apply the split by removing the last few columns, so it matches the main accident training dataset (which is 2019 as the last entry year).

```
traffic_flow_numerical_cols = traffic_flow_df.select_dtypes(include=['int','float'])
```

```
traffic_flow_numerical_cols.columns
```

```
Index(['1993', '1994', '1995', '1996', '1997', '1998', '1999', '2000', '2001',
       '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010',
       '2011', '2012', '2013', '2014', '2015', '2016', '2017', '2018', '2019',
       '2020', '2021', '2022'],
      dtype='object')
```

```
cols_to_drop = ['2020', '2021', '2022']
```

```
traffic_flow_df = traffic_flow_df.drop(columns=cols_to_drop)
```

(b) Splitting the additional datasets.

Figure C.2: Splitting the datasets into training and testing sets, for EDA analysis.

C.4 Model Development and Monitoring Stages

C.4.1 Original classes file

```
# model creation and training
from torch import Tensor
from torch.nn import LSTM, Linear
from torch.utils.data import Dataset, DataLoader
from coral_pytorch.dataset import corn_label_from_logits
# evaluation metrics
from torchmetrics import MeanAbsoluteError, MeanAbsolutePercentageError, MeanSquaredError

from pytorch_lightning import LightningDataModule, LightningModule

class RoadAccidentDataset(Dataset):
    def __init__(self, X:Tensor, y:Tensor):
        self.X = X.float() # convert to float to ensure the dtypes are consistent
        self.y = y.float() # still necessary even if they are already floats, e.g., float32 is different to float64

    def __len__(self): # for getting the number of instances
        return len(self.X)

    def __getitem__(self, idx): # for getting a specific set of values at a certain index
        return self.X[idx], self.y[idx]

class RoadAccidentDataModule(LightningDataModule):
    def __init__(self, X_train, X_test, X_val, y_train, y_val, y_test, batch_size):
        super().__init__()
        self.batch_size = batch_size
        self.X_train = X_train
```

```
        self.X_val = X_val
        self.X_test = X_test
        self.y_train = y_train
        self.y_val = y_val
        self.y_test = y_test

    def train_dataloader(self):
        train_dataset = RoadAccidentDataset(self.X_train, self.y_train)
        train_dataloader = DataLoader(train_dataset, batch_size=self.batch_size, shuffle=False)

        return train_dataloader

    def val_dataloader(self):
        validation_dataset = RoadAccidentDataset(self.X_val, self.y_val)
        validation_dataloader = DataLoader(validation_dataset, batch_size=self.batch_size, shuffle=False)

        return validation_dataloader

    def test_dataloader(self):
        test_dataset = RoadAccidentDataset(self.X_test, self.y_test)
        test_dataloader = DataLoader(test_dataset, batch_size=self.batch_size, shuffle=False)

        return test_dataloader

class AccidentLikelihoodNetwork(LightningModule):
    def __init__(self, n_features, hidden_size, output_size, batch_size,
                 num_layers, learning_rate,
                 optimiser, criterion):
        super().__init__()
```

```
        self.n_features = n_features
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.batch_size = batch_size
        self.num_layers = num_layers
        self.learning_rate = learning_rate
        self.optimiser = optimiser
        self.criterion = criterion

        self.train_mae = MeanAbsoluteError()
        self.train_mae_macroaveraged_epoch = MeanAbsoluteError()
        self.val_mae = MeanAbsoluteError()
        self.val_mae_macroaveraged_epoch = MeanAbsoluteError()
        self.test_mae = MeanAbsoluteError()

        self.train_mse = MeanSquaredError()
        self.val_mse = MeanSquaredError()
        self.test_mse = MeanSquaredError()

        self.train_mape = MeanAbsolutePercentageError()
        self.val_mape = MeanAbsolutePercentageError()
        self.test_mape = MeanAbsolutePercentageError()

        self.lstm = LSTM(input_size=n_features, hidden_size=hidden_size,
                         num_layers=num_layers)
        self.classifier = Linear(hidden_size, output_size-1) # -1 due to using ordinal regression
        # Linear is fully connected layer

    def forward(self, x):
```

```
        lstm_out, _ = self.lstm(x)
        output = self.classifier(lstm_out)
        return output

    def _shared_step(self, batch):
        # features is x, true_labels is y
        features, true_labels = batch
        logits=self.forward(features)

        # calculate Log Loss
        loss = self.criterion(logits, true_labels, num_classes=self.output_size)

        predicted_labels = corn_label_from_logits(logits)

        return loss, true_labels, predicted_labels

    def configure_optimizers(self):
        optimiser = self.optimiser(self.parameters(), lr=self.learning_rate)
        return optimiser

    def training_step(self, train_batch, train_idx):
        loss, true_labels, predicted_labels = self._shared_step(train_batch)
        train_mae = self.train_mae(predicted_labels, true_labels)
        self.train_mae_macroaveraged_epoch(predicted_labels, true_labels)
        train_mape = self.train_mape(predicted_labels, true_labels)
        train_mse = self.train_mse(predicted_labels, true_labels)
        train_rmse = train_mse**0.5 # equivalent to square root

        self.log('train_loss', loss, on_epoch=True)
```

```
        self.log('train_mse', train_mse, on_epoch=True)
        self.log('train_rmse', train_rmse, on_epoch=True)
        self.log('train_mae', train_mae, on_epoch=True)
        self.log('train_mape', train_mape, on_epoch=True)
        return loss

    def on_train_epoch_end(self): # for calculating the macroaveraged MAE
        self.log('train_mae_epoch', self.train_mae_macroaveraged_epoch.compute())
        self.train_mae_macroaveraged_epoch.reset()

    def validation_step(self, val_batch, val_idx):
        loss, true_labels, predicted_labels = self._shared_step(val_batch)
        val_mae = self.val_mae(predicted_labels, true_labels)
        self.val_mae_macroaveraged_epoch(predicted_labels, true_labels)
        val_mape = self.val_mape(predicted_labels, true_labels)
        val_mse = self.val_mse(predicted_labels, true_labels)
        val_rmse = val_mse**0.5

        self.log('val_loss', loss, on_epoch=True)
        self.log('val_mse', val_mse, on_epoch=True)
        self.log('val_rmse', val_rmse, on_epoch=True)
        self.log('val_mae', val_mae, on_epoch=True)
        self.log('val_mape', val_mape, on_epoch=True)
        return loss

    def on_validation_epoch_end(self): # for calculating the macroaveraged MAE
        self.log('val_mae_epoch', self.val_mae_macroaveraged_epoch.compute())
        self.val_mae_macroaveraged_epoch.reset()

```

```
    def test_step(self, test_batch, test_idx):
        loss, true_labels, predicted_labels = self._shared_step(test_batch)
        test_mae = self.test_mae(predicted_labels, true_labels)
        test_mape = self.test_mape(predicted_labels, true_labels)
        test_mse = self.test_mse(predicted_labels, true_labels)
        test_rmse = test_mse**0.5

        self.log('test_loss', loss, on_epoch=True)
        self.log('test_mse', test_mse, on_epoch=True)
        self.log('test_rmse', test_rmse, on_epoch=True)
        self.log('test_mae', test_mae, on_epoch=True)
        self.log('test_mape', test_mape, on_epoch=True)
        return loss
```

C.4.2 Original control flow file

```
# import logging to log any errors or info to Airflow
import logging

# model construction and training
from torch.optim import Adam
from torch import Tensor
from pytorch_lightning import Trainer

from coral_pytorch.losses import corn_loss

# model monitoring
from mlflow import set_tracking_uri, set_experiment, start_run, log_params
from mlflow.pytorch import autolog
```

```
14
15  # Redis server connection
16  from pickle import loads
17  from redis import Redis
18
19  from .classes import *
20
21  def model_dev_and_monitoring__main(logger:logging.Logger, redis_host:str, redis_port:int,
22                                     mlflow_uri:str, experiment_name:str) -> None:
23      """
24      Function that manages the program flow of the model development stage.
25      """
26      try:
27          # hyperparameter constants
28          hyperparams = {
29              'batch_size':128,
30              'criterion':corn_loss,
31              'max_epochs':6, # total number of epochs will be max_epochs - 1
32              'n_features':12,
33              'hidden_size':64,
34              'num_layers':1,
35              'learning_rate':0.001,
36              'num_classes':5,
37              'optimiser':Adam
38          }
39
40          # retrieve the preprocessed data
41          with Redis(host=redis_host, port=redis_port) as redis_conn:
42              X_train:Tensor = loads(redis_conn.get("X_train_preprocessed"))
```

```
            X_val:Tensor = loads(redis_conn.get("X_val_preprocessed"))
            X_test:Tensor = loads(redis_conn.get("X_test_preprocessed"))
            y_train:Tensor = loads(redis_conn.get("y_train_preprocessed"))
            y_val:Tensor = loads(redis_conn.get("y_val_preprocessed"))
            y_test:Tensor = loads(redis_conn.get("y_test_preprocessed"))

            redis_conn.flushdb() # flush the temporary datastore now finished with the data

        data_module  = RoadAccidentDataModule(X_train=X_train, X_test=X_test,
                                    X_val=X_val, y_train=y_train,
                                    y_test=y_test, y_val=y_val, batch_size=hyperparams['batch_size'])

        neural_network = AccidentLikelihoodNetwork(n_features=hyperparams['n_features'],
        ↪  hidden_size=hyperparams['hidden_size'],
                                                   output_size=hyperparams['num_classes'],
                                                   ↪  batch_size=hyperparams['batch_size'],
                                                   num_layers=hyperparams['num_layers'],
                                                   ↪  learning_rate=hyperparams['learning_rate'],
                                                   optimiser=hyperparams['optimiser'], criterion=hyperparams['criterion'])

        # set the connection to the MLflow server instance
        set_tracking_uri(mlflow_uri)
        # set the experiment name of the model
        set_experiment(experiment_name)
        # autolog certain evaluation metrics declared in the 'AccidentLikelihoodNetwork' module
        autolog()

        # used for managing the training and development of any model
        trainer = Trainer(max_epochs=hyperparams['max_epochs'], enable_progress_bar=True)
```

```
69
70          # start a new MLflow run, so it tracks
71          with start_run() as run:
72              trainer.fit(neural_network, data_module)
73
74          # log the hyperparameters used for the run
75          log_params(params=hyperparams, run_id=run.info.run_id)
76
77      except Exception as e:
78          logger.critical("Error occured during model development stage.\n{}".format(e), exc_info=1)
79      else:
80          logger.info("Model development and monitoring stage successful!")
```

C.4.3 Revised classes file

```
1  # model creation and training
2  from torch import Tensor
3  from torch.nn import LSTM, Linear
4  from torch.utils.data import Dataset, DataLoader
5  from coral_pytorch.dataset import corn_label_from_logits
6  # evaluation metrics
7  from torchmetrics import MeanAbsoluteError, MeanAbsolutePercentageError, MeanSquaredError
8
9  from pytorch_lightning import LightningDataModule, LightningModule
10
11 class RoadAccidentDataset(Dataset):
12     def __init__(self, X:Tensor, y:Tensor):
13         self.X = X.float() # convert to float to ensure the dtypes are consistent
14         self.y = y.float() # still necessary even if they are already floats, e.g., float32 is different to float64
```

```
15
16      def __len__(self): # for getting the number of instances
17          return len(self.X)
18
19      def __getitem__(self, idx): # for getting a specific set of values at a certain index
20          return self.X[idx].unsqueeze(0), self.y[idx]
21          # the above converts the features into shape (batch_size, seq_len, number of features)
22          # the true labels, y, remains the same shape
23
24  class RoadAccidentDataModule(LightningDataModule):
25      def __init__(self, X_train, X_test, X_val, y_train, y_val, y_test, batch_size):
26          super().__init__()
27          self.batch_size = batch_size
28          self.X_train = X_train
29          self.X_val = X_val
30          self.X_test = X_test
31          self.y_train = y_train
32          self.y_val = y_val
33          self.y_test = y_test
34
35      def train_dataloader(self):
36          train_dataset = RoadAccidentDataset(self.X_train, self.y_train)
37          train_dataloader = DataLoader(train_dataset, batch_size=self.batch_size, shuffle=False)
38
39          return train_dataloader
40
41      def val_dataloader(self):
42          validation_dataset = RoadAccidentDataset(self.X_val, self.y_val)
43          validation_dataloader = DataLoader(validation_dataset, batch_size=self.batch_size, shuffle=False)
```

```
44
45          return validation_dataloader
46
47      def test_dataloader(self):
48          test_dataset = RoadAccidentDataset(self.X_test, self.y_test)
49          test_dataloader = DataLoader(test_dataset, batch_size=self.batch_size, shuffle=False)
50
51          return test_dataloader
52
53  class AccidentLikelihoodNetwork(LightningModule):
54      def __init__(self, n_features, hidden_size, output_size, batch_size,
55                   num_layers, learning_rate,
56                   optimiser, criterion, dropout):
57          super().__init__()
58          self.n_features = n_features
59          self.hidden_size = hidden_size
60          self.output_size = output_size
61          self.batch_size = batch_size
62          self.num_layers = num_layers
63          self.learning_rate = learning_rate
64          self.optimiser = optimiser
65          self.dropout = dropout
66          self.criterion = criterion
67
68          self.train_mae = MeanAbsoluteError()
69          self.train_mae_macroaveraged_epoch = MeanAbsoluteError()
70          self.val_mae = MeanAbsoluteError()
71          self.val_mae_macroaveraged_epoch = MeanAbsoluteError()
72          self.test_mae = MeanAbsoluteError()
```

```
        self.train_mse = MeanSquaredError()
        self.val_mse = MeanSquaredError()
        self.test_mse = MeanSquaredError()

        self.train_mape = MeanAbsolutePercentageError()
        self.val_mape = MeanAbsolutePercentageError()
        self.test_mape = MeanAbsolutePercentageError()

        self.lstm = LSTM(input_size=n_features, hidden_size=hidden_size,
                         num_layers=num_layers, batch_first=True,
                         dropout=dropout)
        self.classifier = Linear(hidden_size, output_size-1) # -1 due to using ordinal regression
        # Linear is fully connected layer

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        output = self.classifier(lstm_out[:, -1, :])
        # lstm_out[:, -1, :] takes the last time step and uses it for the classifier
        return output

    def _shared_step(self, batch):
        # features is x, true_labels is y
        features, true_labels = batch
        logits=self.forward(features)

        # calculate Log Loss
        loss = self.criterion(logits, true_labels, num_classes=self.output_size)

```

```
        predicted_labels = corn_label_from_logits(logits)

        return loss, true_labels, predicted_labels

    def configure_optimizers(self):
        optimiser = self.optimiser(self.parameters(), lr=self.learning_rate)
        return optimiser

    def training_step(self, train_batch, train_idx):
        loss, true_labels, predicted_labels = self._shared_step(train_batch)
        train_mae = self.train_mae(predicted_labels, true_labels)
        self.train_mae_macroaveraged_epoch(predicted_labels, true_labels)
        train_mape = self.train_mape(predicted_labels, true_labels)
        train_mse = self.train_mse(predicted_labels, true_labels)
        train_rmse = train_mse**0.5 # equivalent to square root

        self.log('train_loss', loss, on_epoch=True)
        self.log('train_mse', train_mse, on_epoch=True)
        self.log('train_rmse', train_rmse, on_epoch=True)
        self.log('train_mae', train_mae, on_epoch=True)
        self.log('train_mape', train_mape, on_epoch=True)
        return loss

    def on_train_epoch_end(self): # for calculating the macroaveraged MAE
        self.log('train_mae_epoch', self.train_mae_macroaveraged_epoch.compute())
        self.train_mae_macroaveraged_epoch.reset()

    def validation_step(self, val_batch, val_idx):
        loss, true_labels, predicted_labels = self._shared_step(val_batch)
```

```
        val_mae = self.val_mae(predicted_labels, true_labels)
        self.val_mae_macroaveraged_epoch(predicted_labels, true_labels)
        val_mape = self.val_mape(predicted_labels, true_labels)
        val_mse = self.val_mse(predicted_labels, true_labels)
        val_rmse = val_mse**0.5

        self.log('val_loss', loss, on_epoch=True)
        self.log('val_mse', val_mse, on_epoch=True)
        self.log('val_rmse', val_rmse, on_epoch=True)
        self.log('val_mae', val_mae, on_epoch=True)
        self.log('val_mape', val_mape, on_epoch=True)
        return loss

    def on_validation_epoch_end(self): # for calculating the macroaveraged MAE
        self.log('val_mae_epoch', self.val_mae_macroaveraged_epoch.compute())
        self.val_mae_macroaveraged_epoch.reset()

    def test_step(self, test_batch, test_idx):
        loss, true_labels, predicted_labels = self._shared_step(test_batch)
        test_mae = self.test_mae(predicted_labels, true_labels)
        test_mape = self.test_mape(predicted_labels, true_labels)
        test_mse = self.test_mse(predicted_labels, true_labels)
        test_rmse = test_mse**0.5

        self.log('test_loss', loss, on_epoch=True)
        self.log('test_mse', test_mse, on_epoch=True)
        self.log('test_rmse', test_rmse, on_epoch=True)
        self.log('test_mae', test_mae, on_epoch=True)
        self.log('test_mape', test_mape, on_epoch=True)
```

```
160         return loss
```

C.4.4 Revised control flow file

```
# import logging to log any errors or info to Airflow
import logging

# model construction and training
from torch.optim import Adam
from torch import Tensor
from pytorch_lightning import Trainer

from coral_pytorch.losses import corn_loss

# model monitoring
from mlflow import set_tracking_uri, set_experiment, start_run, log_params
from mlflow.pytorch import autolog

# Redis server connection
from pickle import loads
from redis import Redis

from .classes import *

def model_dev_and_monitoring__main(logger:logging.Logger, redis_host:str, redis_port:int,
                                   mlflow_uri:str, experiment_name:str) -> None:
    """
    Function that manages the program flow of the model development stage.
    """
```

```
    try:
        # hyperparameter constants
        hyperparams = {
            'batch_size':64,
            'criterion':corn_loss,
            'max_epochs':11, # total number of epochs will be max_epochs - 1
            'n_features':12,
            'hidden_size':64,
            'num_layers':2,
            'learning_rate':0.0001,
            'num_classes':5,
            'optimiser':Adam,
            'dropout':0.2
        }

        # retrieve the preprocessed data
        with Redis(host=redis_host, port=redis_port) as redis_conn:
            X_train:Tensor = loads(redis_conn.get("X_train_preprocessed"))
            X_val:Tensor = loads(redis_conn.get("X_val_preprocessed"))
            X_test:Tensor = loads(redis_conn.get("X_test_preprocessed"))
            y_train:Tensor = loads(redis_conn.get("y_train_preprocessed"))
            y_val:Tensor = loads(redis_conn.get("y_val_preprocessed"))
            y_test:Tensor = loads(redis_conn.get("y_test_preprocessed"))

            redis_conn.flushdb() # flush the temporary datastore now finished with the data

        data_module  = RoadAccidentDataModule(X_train=X_train, X_test=X_test,
                                    X_val=X_val, y_train=y_train,
                                    y_test=y_test, y_val=y_val, batch_size=hyperparams['batch_size'])
```

```
55
56          neural_network = AccidentLikelihoodNetwork(n_features=hyperparams['n_features'],
            ↪  hidden_size=hyperparams['hidden_size'],
57                                                     output_size=hyperparams['num_classes'],
                                                       ↪  batch_size=hyperparams['batch_size'],
58                                                     num_layers=hyperparams['num_layers'],
                                                       ↪  learning_rate=hyperparams['learning_rate'],
59                                                     optimiser=hyperparams['optimiser'], criterion=hyperparams['criterion'],
60                                                     dropout=hyperparams['dropout'])
61
62          # set the connection to the MLflow server instance
63          set_tracking_uri(mlflow_uri)
64          # set the experiment name of the model
65          set_experiment(experiment_name)
66          # autolog certain evaluation metrics declared in the 'AccidentLikelihoodNetwork' module
67          autolog()
68
69          # used for managing the training and development of any model
70          trainer = Trainer(max_epochs=hyperparams['max_epochs'], enable_progress_bar=True)
71
72          # start a new MLflow run, so it tracks
73          with start_run() as run:
74              trainer.fit(model=neural_network, datamodule=data_module)
75              trainer.test(model=neural_network, datamodule=data_module)
76              # run final model against testing set in 'data_module'
77
78          # log the hyperparameters used for the run
79          log_params(params=hyperparams, run_id=run.info.run_id)
80
```

```
    except Exception as e:
        logger.critical("Error occured during model development stage.\n{}".format(e), exc_info=1)
    else:
        logger.info("Model development and monitoring stage successful!")
```

C.4.5 Model creation pipeline after Model Development and Monitoring stage

```
from datetime import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.empty import EmptyOperator
import logging
from fyp_package import ingestion, preprocessing, model_dev_and_monitoring
#setup of logger to record notable events
logging.basicConfig(level=logging.WARN) #log only levels including: WARNING, ERROR and CRITICAL
logger = logging.getLogger(__name__) #create the a longger instance with the name of '__main__'
#default arguments for the DAG; required

DEFAULT_ARGS = {
    "owner":"Alex",
    "depends_on_past":True, #every upstream task must be successful for the DAG to continue
    "email":["alexander.roberts2@mail.bcu.ac.uk"],
    "email_on_failure":False,
    "email_on_retry":False,
    "retries":0,
}
DEFAULT_ROAD_DATA_PATH = '/home/alex/FYP_dir/fyp_data/tfl_road_data'
DEFAULT_TRAFFIC_FLOW_DATA_PATH = '/home/alex/FYP_dir/fyp_data/traffic-flow-borough-all-vehicles.csv'
DEFAULT_POPULATION_DATA_PATH = '/home/alex/FYP_dir/fyp_data/ONS mid-year population estimates London boroughs.csv'
```

```
DEFAULT_GDP_DATA_PATH = '/home/alex/FYP_dir/fyp_data/gdp at current basic rates.csv'
DEFAULT_CRIME_DATA_PATH = '/home/alex/FYP_dir/fyp_data/MPS Borough Level Crime (Historical).csv'

# connection dictionary to the ClickHouse server instance
ENGINE_CONN = {
    "database":'airflow_storage',
    "host":'localhost',
    "user":'AirflowUser122',
    "password":'',
    "port":8123
}
REDIS_HOST = '127.0.0.1'
REDIS_PORT = 6379

MLFLOW_URI = "http://localhost:5000"
MLFLOW_EXPERIMENT_NAME = "fyp-model-development"

# creates unique table names in the ClickHouser server instance
# e.g., 'london_road_accidents_12Jun24
ROAD_DATA_TABLE_NAME = "london_road_accidents_"+datetime.today().strftime('%d%b%y')
TRAFFIC_FLOW_TABLE_NAME = "traffic_flow_london_boroughs_"+datetime.today().strftime('%d%b%y')
POPULATION_TABLE_NAME = "population_london_boroughs_"+datetime.today().strftime('%d%b%y')
GDP_TABLE_NAME = "gdp_london_boroughs_"+datetime.today().strftime('%d%b%y')
CRIME_TABLE_NAME = "crime_london_boroughs_"+datetime.today().strftime('%d%b%y')

with DAG(
    "fyp_model_creation", # DAG name
    default_args=DEFAULT_ARGS,
    description="This pipeline is for the development of a machine learning model.",
```

```
    start_date=datetime(2022,10,10), # required otherwise the DAG cannot proceed
    schedule_interval=None,
    catchup=False, # prevent the pipeline from performing scheduler catchup
    tags=["fyp", "mlops"],
) as dag:
    start=EmptyOperator(task_id="start")
    ingestion_task=PythonOperator(task_id="ingestion_stage", python_callable=ingestion.ingest_data_main,
        op_kwargs={
            'london_acc_data_path':DEFAULT_ROAD_DATA_PATH,
            'traffic_flow_data_path':DEFAULT_TRAFFIC_FLOW_DATA_PATH,
            'population_data_path':DEFAULT_POPULATION_DATA_PATH,
            'gdp_data_path':DEFAULT_GDP_DATA_PATH,
            'crime_data_path':DEFAULT_CRIME_DATA_PATH,
            'logger':logger,
            'engine_conn':ENGINE_CONN,
            'london_acc_table_name':ROAD_DATA_TABLE_NAME,
            'traffic_flow_table_name':TRAFFIC_FLOW_TABLE_NAME,
            'population_table_name':POPULATION_TABLE_NAME,
            'gdp_table_name':GDP_TABLE_NAME,
            'crime_table_name':CRIME_TABLE_NAME,
        })
    preprocessing_task=PythonOperator(task_id="preprocessing_stage", python_callable=preprocessing.preprocess_data_main,
        op_kwargs={
            'logger':logger,
            'engine_conn':ENGINE_CONN,
            'london_acc_table_name':ROAD_DATA_TABLE_NAME,
            'population_table_name':POPULATION_TABLE_NAME,
            'gdp_table_name':GDP_TABLE_NAME,
            'traffic_flow_table_name':TRAFFIC_FLOW_TABLE_NAME,
```

```
            'crime_table_name':CRIME_TABLE_NAME,
            'redis_host':REDIS_HOST,
            'redis_port':REDIS_PORT
        })
    model_dev_and_monitoring_task=PythonOperator(task_id="model_dev_and_monitoring_stage",
                                                 python_callable=model_dev_and_monitoring.model_dev_and_monitoring__main,
        op_kwargs={
            'logger':logger,
            'redis_host':REDIS_HOST,
            'redis_port':REDIS_PORT,
            'mlflow_uri':MLFLOW_URI,
            'experiment_name':MLFLOW_EXPERIMENT_NAME
        })
    end=EmptyOperator(task_id="end")
    start >> ingestion_task >> preprocessing_task >> model_dev_and_monitoring_task >> end
```

C.5 Deployment Stage

C.5.1 Functions file

```
import requests
from mlflow.pytorch import load_model
from pandas import DataFrame, concat, cut
from json import loads
from datetime import datetime
import numpy as np
from numpy import ndarray

from datetime import datetime

```

```
from copy import deepcopy

from torch import Tensor

from sklearn.preprocessing import LabelEncoder, MinMaxScaler

from sqlalchemy import create_engine, Column, Integer, String, MetaData
from sqlalchemy.orm import Session
from clickhouse_sqlalchemy import Table
from clickhouse_sqlalchemy.engines import MergeTree
import pandahouse as ph

def convert_predictions(data: tuple[DataFrame, Tensor, datetime]) -> tuple[DataFrame, list[int], datetime]:
    """
    Convert predictions made by the LSTM model into relevant risk levels. If the tensor is [w, x, y, z], the risk level
    is equal to the number of these values past the threshold of 1. E.g., so if w > 1, x > 1, y > 1 and z > 1, then
    the risk level would be 5 due to the nature of ordinal classification.
    """
    predictions:Tensor = data[1]
    list_predictions = predictions.tolist()
    for prediction_set_idx in range(len(list_predictions)):
        if ((list_predictions[prediction_set_idx][0] > 1) and (list_predictions[prediction_set_idx][1] > 1)
            and (list_predictions[prediction_set_idx][2] > 1) and (list_predictions[prediction_set_idx][3] > 1)):
            list_predictions[prediction_set_idx] = 5
        elif ((list_predictions[prediction_set_idx][0] > 1) and (list_predictions[prediction_set_idx][1] > 1)
            and (list_predictions[prediction_set_idx][2] > 1)):
            list_predictions[prediction_set_idx] = 4
        elif ((list_predictions[prediction_set_idx][0] > 1) and (list_predictions[prediction_set_idx][1] > 1)):
            list_predictions[prediction_set_idx] = 3
```

```
        elif (list_predictions[prediction_set_idx][0] > 1):
            list_predictions[prediction_set_idx] = 2
        elif (list_predictions[prediction_set_idx][0] < 1):
            list_predictions[prediction_set_idx] = 1

    return (data[0], list_predictions, data[2])


def convert_dtypes(df: DataFrame) -> DataFrame:
    """
    Convert all float types in a DataFrame into integers. Also, convert a 'Time' column
    into string type.
    """
    float_cols = df.select_dtypes(float)
    df[float_cols.columns] = df[float_cols.columns].astype(int)
    df['Time'] = df['Time'].astype(str)
    return df

def create_clickhouse_predictions_table(engine_conn:dict, table_name:str) -> None:
    """
    A function that will create a table in a CiickHouse server for storing predictions from generated models.
    """
    engine_path_str = "clickhouse://{user}:{password}@{host}:{port}/{database}".format(user=engine_conn['user'],
                                                                                       password=engine_conn['password'],
                                                                                       host=engine_conn['host'],
                                                                                       port=engine_conn['port'],
                                                                                       database=engine_conn['database'])

    engine = create_engine(engine_path_str)
```

```
69
70      # declaring the table type
71      predictions_table = Table(table_name, MetaData(),
72          Column(Integer, name='ID', primary_key=True),
73          Column(String, name='Borough'),
74          Column(Integer, name='GDP Prev Year'),
75          Column(Integer, name='Population Prev Year'),
76          Column(Integer, name='Traffic Flow Prev Year'),
77          Column(Integer, name='Vehicle Crime Prev Month'),
78          Column(String, name='Weather Conditions'),
79          Column(String, name='Light Conditions'),
80          Column(String, name='Time'),
81          Column(Integer, name='Year'),
82          Column(Integer, name='Month'),
83          Column(Integer, name='Day'),
84          Column(String, name='DayOfWeek'),
85          Column(Integer, name='Predictions'),
86          MergeTree(order_by=('ID')),
87          schema=engine_conn['database'])
88
89      with Session(engine) as session:
90          predictions_table.create(session.connection(), checkfirst=True)
91          # 'checkfirst' will prevent a new table being created if the table already exists
92          session.commit()
93
94  def store_predictions(engine_conn:dict, table_name:str, predictions:DataFrame) -> None:
95      """
96      Take model predictions and store them in a ClickHouse table.
97
```

```
    Parameters
    ----------
        | `engine_conn`: contains the connection info to the ClickHouse server database.
        | `table_name`: the name of the table to write the predictions to.
        | `predictions`: a Pandas DataFrame containing the model input data and the prediction results.
    """
    connection = {
        "database":engine_conn['database'],
        "host":'http://{host}:{port}'.format(host=engine_conn['host'], port=engine_conn['port']),
        "user":engine_conn['user'],
        "password":engine_conn['password']
    }

    ph.to_clickhouse(predictions, table_name, index = False, connection=connection)
    # index of the dataframe is not needed as the prediction time will act as the index

def get_light_conditions(timeapi_uri:str, sunrisesunsetapi_uri:str) -> str:
    """
    Retrieve the current light conditions using the specified APIs.
    """
    # retrieve current time
    current_time_london = loads(
        requests.get(timeapi_uri).content.decode('utf-8')
    )
    current_light_conditions = loads(
        requests.get(sunrisesunsetapi_uri).content.decode('utf-8')
    )

    sunset = datetime.strptime(current_light_conditions['results']['sunset'], "%I:%M:%S %p").time()
```

```
    sunrise = datetime.strptime(current_light_conditions['results']['sunrise'], "%I:%M:%S %p").time()
    current_datetime = datetime.strptime(current_time_london['dateTime'].split('.')[0], "%Y-%m-%dT%H:%M:%S" ).time()

    if((current_datetime >=  sunrise) and (current_datetime < sunset)):
        return'light'

    return'dark'

def get_borough_weather_conditions(borough_coords:dict[str, dict[str,float]], openweather_apikey:str) -> DataFrame:
    """
    Get the weather conditions of the specified borough.
    """
    df = DataFrame()
    df['Borough'] = np.nan
    df['Weather Conditions'] = np.nan

    for key, value in borough_coords.items():
        current_borough_weather = loads(
            requests.get('https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={long}&appid={api_key}'.format(
                lat=value['lat'],
                long=value['long'],
                api_key=openweather_apikey
            )).content.decode('utf-8'))['weather'][0]['main']

        match current_borough_weather.lower():
            case 'clear':
                current_borough_weather = 'fine'
            case 'rain':
                current_borough_weather = 'raining'
```

```
            case 'snow':
                current_borough_weather = 'snowing'
            case 'cloud':
                current_borough_weather = 'fine'
            case 'drizzle':
                current_borough_weather = 'other'
            case _:
                current_borough_weather = 'other'

        df = concat([df, DataFrame({
            'Borough': key,
            'Weather Conditions': current_borough_weather
        }, index=[0])], axis=0)

    return df

def retrieve_past_data_from_clickhouse(engine_conn:dict, borough_coords:dict[str, dict[str, float]], gdp_table_name:str,
                                       traffic_flow_table_name:str,
                                       population_table_name:str, crime_table_name:str,
                                       year:str="2021") -> DataFrame:
    """
    Retrieve specific past data from a ClickHouse server instance.
    """
    connection = {
        "database":engine_conn['database'],
        "host":'http://{host}:{port}'.format(host=engine_conn['host'], port=engine_conn['port']),
        "user":engine_conn['user'],
        "password":engine_conn['password']
    }
```

```
185
186     gdp:DataFrame = ph.read_clickhouse(
187         'SELECT `{}`, `LA name` FROM {}.{}'.format(year, engine_conn['database'], gdp_table_name),
188         connection=connection)
189     traffic_flow:DataFrame = ph.read_clickhouse(
190         'SELECT `{}`, `Local Authority` FROM {}.{}'.format(year, engine_conn['database'], traffic_flow_table_name),
191         connection=connection)
192     population:DataFrame = ph.read_clickhouse(
193         'SELECT `{}`, `LA name` FROM {}.{}'.format(year, engine_conn['database'], population_table_name),
194         connection=connection)
195
196     crime_col_to_get = year + str(datetime.now().month).zfill(2)
197
198     vehicle_crime:DataFrame = ph.read_clickhouse(
199         'SELECT `{}`, `BoroughName`, `MajorText`, `MinorText` FROM {}.{}'.format(crime_col_to_get, engine_conn['database'],
200                                                                                  crime_table_name),
201         connection=connection)
202
203     vehicle_crime = vehicle_crime[vehicle_crime['MinorText'].isin(['THEFT FROM A VEHICLE', 'TRAFFICKING OF DRUGS'])]
204
205     gdp['LA name'] = gdp['LA name'].str.replace('-', ' ')
206     gdp['LA name'] = gdp['LA name'].str.replace('&', 'and')
207     gdp['LA name'] = gdp['LA name'].str.lower()
208
209     population['LA name'] = population['LA name'].str.replace('-', ' ')
210     population['LA name'] = population['LA name'].str.replace('&', 'and')
211     population['LA name'] = population['LA name'].str.lower()
212
213     traffic_flow['Local Authority'] = traffic_flow['Local Authority'].str.replace('-', ' ')
```

```
    traffic_flow['Local Authority'] = traffic_flow['Local Authority'].str.replace('&', 'and')
    traffic_flow['Local Authority'] = traffic_flow['Local Authority'].str.lower()

    vehicle_crime['BoroughName'] = vehicle_crime['BoroughName'].str.replace('-', ' ')
    vehicle_crime['BoroughName'] = vehicle_crime['BoroughName'].str.replace('&', 'and')
    vehicle_crime['BoroughName'] = vehicle_crime['BoroughName'].str.lower()

    df = DataFrame()
    df['Borough'] = np.nan
    df['GDP Prev Year'] = np.nan
    df['Population Prev Year'] = np.nan
    df['Traffic Flow Prev Year'] = np.nan
    df['Vehicle Crime Prev Month'] = np.nan
    #df['Vehicle Crime Prev Month'] = np.nan

    for key, _ in borough_coords.items():
        gdp_prev_year = gdp[gdp['LA name'] == key][year].values[0]
        population_prev_year = population[population['LA name'] == key][year].values[0]
        traffic_flow_prev_year = traffic_flow[traffic_flow['Local Authority'] == key][year].values[0]
        crime_prev_month = vehicle_crime[vehicle_crime['BoroughName'] ==
        ↪  key].groupby('BoroughName').agg('sum')[crime_col_to_get].values[0]

        df = concat([df, DataFrame({
            'Borough': key,
            'GDP Prev Year':gdp_prev_year,
            'Population Prev Year':population_prev_year,
            'Traffic Flow Prev Year':traffic_flow_prev_year,
            'Vehicle Crime Prev Month':crime_prev_month
        }, index=[0])], axis=0)
```

```
242
243     return df
244     #vehicle_crime:Series = ph.read_clickhouse('SELECT {} FROM {}.{}'.format(year, engine_conn['database'], gdp_table_name),
        ↪  index = True, index_col = 'index', connection=connection)
245
246 def make_predictions(data: tuple[DataFrame, Tensor], mlflow_model_uri:str) -> tuple[DataFrame, Tensor, datetime]:
247     """
248     Make predictions using unseen data on a modal stored in MLflow.
249
250     Returns
251     -------
252         | A tuple containing the original data, the predictions, and the time of prediction.
253     """
254     loaded_model = load_model(mlflow_model_uri)
255     return (data[0], loaded_model(data[1]), datetime.now())
256
257 def combine_input_data_and_predictions(data:tuple[DataFrame, list[int], datetime]) -> DataFrame:
258     input_data:DataFrame = data[0]
259     predictions:ndarray = data[1]
260     prediction_time:datetime = data[2]
261
262     prediction_time = prediction_time.replace(microsecond=0)
263
264     # replicate the prediction time to be the same as the length of the number of predictions
265     # this is so each row can get the prediction time
266     # prediction_time_df = concat([DataFrame({
267     #         'Prediction DateTime':prediction_time
268     #     }, index=[0])]*len(input_data), ignore_index=True)
269
```

```
    # add the predictions to a new column in the dataframe
    input_data['Predictions'] = predictions

    #merged_data = concat([input_data.reset_index(drop=True), prediction_time_df.reset_index(drop=True)], axis=1,
    ↪  join='inner')
    return input_data

def label_encoding(original_data:DataFrame) -> tuple[DataFrame,DataFrame]:
    input_df = deepcopy(original_data) # make a deepcopy so does not reference original data
    le_borough = LabelEncoder()
    le_time = LabelEncoder()
    le_light_conditions = LabelEncoder()
    le_weather = LabelEncoder()
    le_dayofweek = LabelEncoder()

    input_df['Borough'] = le_borough.fit_transform(input_df['Borough'])
    input_df['Time'] = le_time.fit_transform(input_df['Time'])
    input_df['Light Conditions'] = le_light_conditions.fit_transform(input_df['Light Conditions'])
    input_df['Weather Conditions'] = le_weather.fit_transform(input_df['Weather Conditions'])
    input_df['DayOfWeek'] = le_dayofweek.fit_transform(input_df['DayOfWeek'])

    return (original_data, input_df)

def data_scaling(data: tuple[DataFrame,DataFrame]) -> tuple[DataFrame,DataFrame]:
    min_max_scaler = MinMaxScaler(feature_range=(0,1))
    return (data[0], min_max_scaler.fit_transform(data[1]))

def time_discretisation(df: DataFrame) -> DataFrame:
    # as 'Time' contains integer values, the bins need to be real number as well
```

```
    # they are float values as the bins work by collecting values between each index
        # e.g., values between 0 and 2 (not including 2) will be put in bin 1
        # which has a label of '00:00-01:59'
    bins = [0, 1.99, 3.99, 5.99, 7.99, 9.99, 11.99, 13.99, 15.99, 17.99, 19.99, 21.99, 23.99]

    # the labels represent the time of day using a 24-hour clock
    labels = ['00:00-01:59', '02:00-03:59', '04:00-05:59', '06:00-07:59', '08:00-09:59', '10:00-11:59', '12:00-13:59',
        '14:00-15:59', '16:00-17:59', '18:00-19:59', '20:00-21:59', '22:00-23:59']

    df['Time'] = cut(x=df['Time'], bins=bins, labels=labels, include_lowest=True, ordered=True)

    return df

def get_time_dayofweek_and_dayofweek(timeapi_uri:str) -> DataFrame:
    current_time_london = loads(
        requests.get(timeapi_uri).content.decode('utf-8')
    )

    return DataFrame({
        'Time': current_time_london['hour'],
        'Year': current_time_london['year'],
        'Month': current_time_london['month'],
        'Day': current_time_london['day'],
        'DayOfWeek': current_time_london['dayOfWeek']
    }, index=[0])

def fuse_data(weather_df:DataFrame, prev_clickhouse_data_df:DataFrame, light_conditions:str, times_df:DataFrame) -> DataFrame:
    merged_df = prev_clickhouse_data_df.merge(weather_df, on='Borough')

```

```
    # replicate the single light conditions across the data
    light_conditions_df = concat([DataFrame({
            'Light Conditions':light_conditions
        }, index=[0])]*len(merged_df), ignore_index=True)
    times_df = concat([times_df]*len(merged_df), axis=0)

    merged_df = concat([merged_df.reset_index(drop=True), light_conditions_df.reset_index(drop=True)], axis=1, join='inner')

    merged_df = concat([merged_df.reset_index(drop=True), times_df.reset_index(drop=True)], axis=1, join='inner')

    return merged_df

def convert_to_tensor_and_unsqueeze(data: tuple[DataFrame,DataFrame]) -> tuple[DataFrame,Tensor]:
    """
    Convert a Pandas DataFrame into a PyTorch Tensor.

    Parameters
    ----------
        | `data`: a tuple containing two DataFrames, one for the original data at index 0,
        while the other at index 1 is the transformed and to be transformed data into a Tensor.

    Returns
    -------
        | A tuple with the original data, as a DataFrame, at index 0, and a Tensor at index 1.
    """
    return (data[0], Tensor(data[1]))
```

C.5.2 Control flow file

```
# import logging to log any errors or info to Airflow
import logging

import mlflow

from .functions import *

def create_new_predictions(logger:logging.Logger, engine_conn:dict,
                           borough_coords:dict[str,dict[str,float]], predictions_table_name:str,
                           openweather_api_key:dict[str,str], model_uri:str, mlflow_uri:str,
                           gdp_table_name:str, population_table_name:str,
                           traffic_flow_table_name:str, crime_table_name:str, year:str):
    try:
        mlflow.set_tracking_uri(mlflow_uri)
        # set the connection to the MLflow server to access its runs and models

        create_clickhouse_predictions_table(engine_conn=engine_conn, table_name=predictions_table_name)

        store_predictions(
            predictions=convert_dtypes(
            combine_input_data_and_predictions(
            convert_predictions(
            make_predictions(
            convert_to_tensor_and_unsqueeze(
            data_scaling(
                label_encoding(
                    time_discretisation(
                        fuse_data(
```

```
                        weather_df=get_borough_weather_conditions(borough_coords=borough_coords,
                        ↪  openweather_apikey=openweather_api_key),
                        light_conditions=get_light_conditions(

                            ↪  timeapi_uri='https://www.timeapi.io/api/Time/current/coordinate?latitude=51.3026&longitude=-0.739',
                            sunrisesunsetapi_uri='https://api.sunrisesunset.io/json?lat=51.3026&lng=-0.739'
                        ),
                        prev_clickhouse_data_df=retrieve_past_data_from_clickhouse(engine_conn=engine_conn,
                        ↪  borough_coords=borough_coords,
                                                    gdp_table_name=gdp_table_name, year=year,
                                                    population_table_name=population_table_name,
                                                    traffic_flow_table_name=traffic_flow_table_name,
                                                    crime_table_name=crime_table_name
                                                ),
                        times_df=get_time_dayofweek_and_dayofweek(

                            ↪  timeapi_uri='https://www.timeapi.io/api/Time/current/coordinate?latitude=51.3026&longitude=-0.739')
                    )
                )
            )
        )
        ),
        mlflow_model_uri=model_uri
        )
        )
        )
        ),
        table_name=predictions_table_name,
        engine_conn=engine_conn
```

```
        )

    except Exception as e:
        logger.critical("Error occured during deployment stage.\n{}".format(e), exc_info=1)
    else:
        logger.info("Deployment stage successful!")
```

C.5.3 Predictions pipeline

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.empty import EmptyOperator
from fyp_package import deployment
import logging
#setup of logger to record notable events
logging.basicConfig(level=logging.WARN) #log only levels including: WARNING, ERROR and CRITICAL
logger = logging.getLogger(__name__) #create the a longger instance with the name of '__main__'
#default arguments for the DAG; required

DEFAULT_ARGS = {
    "owner":"Alex",
    "depends_on_past":True, #every upstream task must be successful for the DAG to continue
    "email":["alexander.roberts2@mail.bcu.ac.uk"],
    "email_on_failure":False,
    "email_on_retry":False,
    "retries":3,
}

```

```
ENGINE_CONN = {
    "database":'airflow_storage',
    "host":'localhost',
    "user":'AirflowUser122',
    "password":'',
    "port":8123
}

LONDON_BOROUGH_LAT_LONG = {
    "barking and dagenham": {
        "lat": 51.5607,
        "long": 0.1557
    },
    "barnet": {
        "lat": 52.6252,
        "long": -0.1517
    },
    "bexley": {
        "lat": 51.4549,
        "long": 0.1505
    },
    "brent": {
        "lat": 51.5588,
        "long": -0.2817
    },
    "bromley": {
        "lat": 51.4039,
        "long": 0.0198
    },
```

```
    "camden": {
        "lat": 51.5290,
        "long": -0.1255
    },
    "croydon": {
        "lat": 51.3714,
        "long": -0.0977
    },
    "ealing": {
        "lat": 51.5130,
        "long": -0.3089
    },
    "enfield": {
        "lat": 51.6538,
        "long": -0.0799
    },
    "greenwich": {
        "lat": 51.4892,
        "long": 0.0648
    },
    "hackney": {
        "lat": 51.5450,
        "long": -0.0553
    },
    "hammersmith and fulham": {
        "lat": 51.4927,
        "long": -0.2339
    },
    "haringey": {
```

```
        "lat": 51.6000,
        "long": -0.1119
    },
    "harrow": {
        "lat": 51.5898,
        "long": -0.3346
    },
    "havering": {
        "lat": 51.5812,
        "long": 0.1837
    },
    "hillingdon": {
        "lat": 51.5441,
        "long": -0.4760
    },
    "hounslow": {
        "lat": 51.4746,
        "long": -0.3680
    },
    "islington": {
        "lat": 51.5416,
        "long": -0.1022
    },
    "kensington and chelsea": {
        "lat": 51.5020,
        "long": -0.1947
    },
    "kingston upon thames": {
        "lat": 51.4085,
```

```
        "long": -0.3064
    },
    "lambeth": {
        "lat": 51.4607,
        "long": -0.1163
    },
    "lewisham": {
        "lat": 51.4452,
        "long": -0.0209
    },
    "merton": {
        "lat": 51.4014,
        "long": -0.1958
    },
    "newham": {
        "lat": 51.5077,
        "long": 0.0469
    },
    "redbridge": {
        "lat": 51.5590,
        "long": 0.0741
    },
    "richmond upon thames": {
        "lat": 51.4479,
        "long": -0.3260
    },
    "southwark": {
        "lat": 51.5035,
        "long": -0.0804
```

```
    },
    "sutton": {
        "lat": 51.3618,
        "long": -0.1945
    },
    "tower hamlets": {
        "lat": 51.5099,
        "long": -0.0059
    },
    "waltham forest": {
        "lat": 51.5908,
        "long": -0.0134
    },
    "wandsworth": {
        "lat": 51.4567,
        "long": -0.1910
    },
    "westminster": {
        "lat": 51.4973,
        "long": -0.1372
    }
}

MLFLOW_URI = "http://localhost:5000"
MLFLOW_EXPERIMENT_NAME = "fyp-model-development"

DEFAULT_PREDICTIONS_TABLE_NAME = "london_borough_road_accident_predictions"

OPENWEATHER_API_KEY = 'e5d35b72f07a5050110e80715a67fbb6'
```

```
166
167 # these would possibly be set manually by data scientists
168 # but is designed to work after model creation pipeline but demonstration purposes
169 ROAD_DATA_TABLE_NAME = "london_road_accidents_"+datetime.today().strftime('%d%b%y')
170 TRAFFIC_FLOW_TABLE_NAME = "traffic_flow_london_boroughs_"+datetime.today().strftime('%d%b%y')
171 POPULATION_TABLE_NAME = "population_london_boroughs_"+datetime.today().strftime('%d%b%y')
172 GDP_TABLE_NAME = "gdp_london_boroughs_"+datetime.today().strftime('%d%b%y')
173 CRIME_TABLE_NAME = "crime_london_boroughs_"+datetime.today().strftime('%d%b%y')
174 PREDICTIONS_TABLE_NAME = "borough_risk_level_predictions"
175
176 CHOSEN_YEAR_FOR_PREDICTIONS = '2021'
177
178 MODEL_URI = 'runs:/39a2441b181240bd83ef9491b9407914/model'
179
180 with DAG(
181     "fyp_predictions_pipeline",
182     default_args=DEFAULT_ARGS,
183     description="This pipeline is for creating new predictions for road traffic accidents.",
184     start_date=datetime(2020,10,10), #required otherwise the DAG cannot proceed
185     schedule_interval=timedelta(hours=2), # run the DAG every 2 hours
186     catchup=False,
187     tags=["fyp", "mlops", "predictions"],
188 ) as dag:
189     start=EmptyOperator(task_id="start")
190     deployment_task=PythonOperator(task_id="deployment_stage", python_callable=deployment.create_new_predictions,
191         op_kwargs={
192             'logger':logger,
193             'engine_conn':ENGINE_CONN,
194             'year':CHOSEN_YEAR_FOR_PREDICTIONS,
```

```
            'predictions_table_name':PREDICTIONS_TABLE_NAME,
            'borough_coords':LONDON_BOROUGH_LAT_LONG,
            'gdp_table_name':GDP_TABLE_NAME,
            'population_table_name':POPULATION_TABLE_NAME,
            'traffic_flow_table_name':TRAFFIC_FLOW_TABLE_NAME,
            'crime_table_name':CRIME_TABLE_NAME,
            'openweather_api_key':OPENWEATHER_API_KEY,
            'model_uri':MODEL_URI,
            'mlflow_uri':MLFLOW_URI
        })
    end=EmptyOperator(task_id="end")
    start >> deployment_task >>end
```

C.5.4 Web app

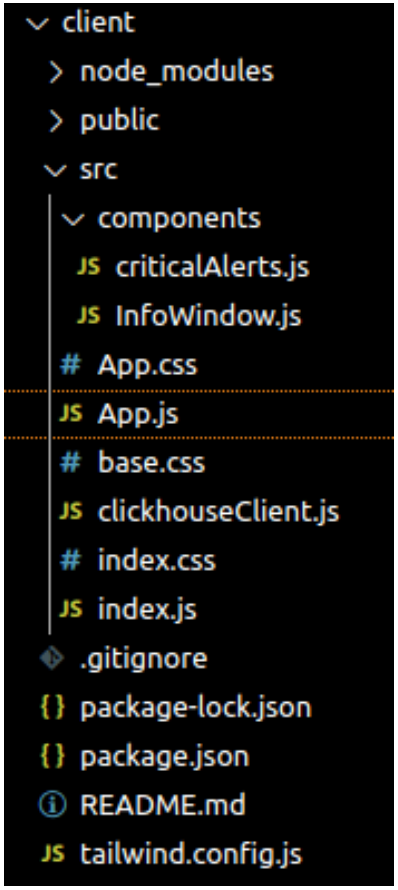


Figure C.3: File structure for the web app.

C.5.4.1 NPM dependency file

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@clickhouse/client-web": "^1.4.0",
    "@googlemaps/js-api-loader": "^1.16.6",
    "@vis.gl/react-google-maps": "^1.1.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
```

```
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "tailwindcss": "^3.4.4"
  }
}
```

C.5.4.2 'clickhouseClient.js'

```
import { createClient } from '@clickhouse/client-web';

const client = createClient({
    url: 'http://localhost:8123',
    username: 'AirflowUser122',
    password: '',
    database: 'airflow_storage',
    application: 'pingpong',
});

export default client;
```

C.5.4.3 Main app

```
import './App.css';
import {APIProvider, Marker, Map} from '@vis.gl/react-google-maps';
import InfoWindow from './components/InfoWindow';
import AlertWindow from './components/criticalAlerts';
import client from './clickhouseClient';
import { useEffect, useState } from 'react';

function App() {
  const londonPositions = {
    "barking and dagenham": {
        "lat": 51.5607,
        "lng": 0.1557
    },
    "barnet": {
        "lat": 52.6252,
        "lng": -0.1517
    },
    "bexley": {
        "lat": 51.4549,
        "lng": 0.1505
    },
    "brent": {
        "lat": 51.5588,
        "lng": -0.2817
    },
    "bromley": {
        "lat": 51.4039,
```

```
        "lng": 0.0198
    },
    "camden": {
        "lat": 51.5290,
        "lng": -0.1255
    },
    "croydon": {
        "lat": 51.3714,
        "lng": -0.0977
    },
    "ealing": {
        "lat": 51.5130,
        "lng": -0.3089
    },
    "enfield": {
        "lat": 51.6538,
        "lng": -0.0799
    },
    "greenwich": {
        "lat": 51.4892,
        "lng": 0.0648
    },
    "hackney": {
        "lat": 51.5450,
        "lng": -0.0553
    },
    "hammersmith and fulham": {
        "lat": 51.4927,
        "lng": -0.2339
```

```
    },
    "haringey": {
        "lat": 51.6000,
        "lng": -0.1119
    },
    "harrow": {
        "lat": 51.5898,
        "lng": -0.3346
    },
    "havering": {
        "lat": 51.5812,
        "lng": 0.1837
    },
    "hillingdon": {
        "lat": 51.5441,
        "lng": -0.4760
    },
    "hounslow": {
        "lat": 51.4746,
        "lng": -0.3680
    },
    "islington": {
        "lat": 51.5416,
        "lng": -0.1022
    },
    "kensington and chelsea": {
        "lat": 51.5020,
        "lng": -0.1947
    },
```

```
    "kingston upon thames": {
        "lat": 51.4085,
        "lng": -0.3064
    },
    "lambeth": {
        "lat": 51.4607,
        "lng": -0.1163
    },
    "lewisham": {
        "lat": 51.4452,
        "lng": -0.0209
    },
    "merton": {
        "lat": 51.4014,
        "lng": -0.1958
    },
    "newham": {
        "lat": 51.5077,
        "lng": 0.0469
    },
    "redbridge": {
        "lat": 51.5590,
        "lng": 0.0741
    },
    "richmond upon thames": {
        "lat": 51.4479,
        "lng": -0.3260
    },
    "southwark": {
```

```
        "lat": 51.5035,
        "lng": -0.0804
    },
    "sutton": {
        "lat": 51.3618,
        "lng": -0.1945
    },
    "tower hamlets": {
        "lat": 51.5099,
        "lng": -0.0059
    },
    "waltham forest": {
        "lat": 51.5908,
        "lng": -0.0134
    },
    "wandsworth": {
        "lat": 51.4567,
        "lng": -0.1910
    },
    "westminster": {
        "lat": 51.4973,
        "lng": -0.1372
    }
}

  const [currentSelection, setCurrentSelection] = useState(0)
  const [rows, setRows] = useState([])

  useEffect(() => {
```

```
    const fetchData = async() => {
      try {
      const result = await client.query({
        query: `SELECT * FROM airflow_storage.borough_risk_level_predictions`,
        format: 'JSONEachRow'
      });
      const jsonResult = await result.json();
      setRows(jsonResult);
      } catch (error) {
        console.error('Error fetching data: ', error)
      }
    }

    fetchData()
  }, [])

  return (
    <div className="App">
      {currentSelection && <InfoWindow data={currentSelection} setCurrentSelection={setCurrentSelection} />}
      {currentSelection && <AlertWindow />}
      <APIProvider apiKey={'AIzaSyBpXTBrHLxDMe1QblQMN2YiaD-0XNep_4o'}>
        <Map
          style={{width: '100vw', height: '100vh'}}
          center={{lat: 51.473356, lng: -0.144419}}
          disableDoubleClickZoom={true}
          zoom={11}
          zoomControl={false}
          gestureHandling={'greedy'}
        />
```

```
        {Object.entries(londonPositions).map((place) => {
          return <Marker key={place} position={place[1]} onClick={() => {
            setCurrentSelection(rows.find(row => row['Borough'] === place[0])) }} />
        })}
      </APIProvider>
    </div>
  );
}

export default App;
```

C.5.4.4 'InfoWindow.js'

```
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'
import { faInfoCircle, faClose } from '@fortawesome/free-solid-svg-icons'

const infoIcon = <FontAwesomeIcon icon={faInfoCircle} />
const closeBtnIcon = <FontAwesomeIcon icon={faClose} />

const capitaliseWords = (str) => {
    const exceptions = ['and']
    return str.split(' ').map(word => {
        if (exceptions.includes(word.toLowerCase())) {
            return word.toLowerCase();
        }
        return word.charAt(0).toUpperCase() + word.slice(1).toLowerCase();
    }).join(' ')
}
```

```
16
17  const InfoWindow = ({data, setCurrentSelection}) => {
18      return(
19          <section id="infoWindow"
            ↪  className="absolute z-50 w-1/3 h-4/5 transition-all duration-200 bg-blue-400 top-14 left-10 rounded-md">
20              <button className='absolute top-3 right-5 text-2xl' onClick={() => setCurrentSelection(0)}>
21                  {closeBtnIcon}
22              </button>
23              <div className='mt-10'>
24                  <h1 className='text-white text-2xl mt-7'>{infoIcon} Information Window</h1>
25                  <div className='mt-4 ml-8 float-left text-xl'>
26                      <ul className='list-none [&>*]:mt-4 text-left'>
27                          <li>Select Borough: <b>{capitaliseWords(data.Borough)}</b></li>
28                          <li>Current Risk Level: <b>{data.Predictions}</b></li>
29                          <li>Current Date: <b>{data.Day}-{data.Month}-{data.Year}</b></li>
30                          <li>Day of the Week: <b>{data.DayOfWeek}</b></li>
31                          <li>Weather Conditions: <b>{data['Weather Conditions']}</b></li>
32                          <li>Light Conditions: <b>{data['Light Conditions']}</b></li>
33                          <li>Time Window: <b>{data['Time']}</b></li>
34                          <li>GDP Previous Year: <b>{data['GDP Prev Year']}</b></li>
35                          <li>Population Previous Year: <b>{data['Population Prev Year']}</b></li>
36                          <li>Traffic Flow Previous Year: <b>{data['Traffic Flow Prev Year']}</b></li>
37                          <li>Vehicle Crime Prev Month: <b>{data['Vehicle Crime Prev Month']}</b></li>
38                      </ul>
39                  </div>
40
41              </div>
42          </section>
43      )
```

```
}

export default InfoWindow;
```

C.5.4.5 'criticalAlerts.js'

```
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'
import { faTriangleExclamation } from '@fortawesome/free-solid-svg-icons'

const alertIcon = <FontAwesomeIcon icon={faTriangleExclamation} />

const AlertWindow = () => {
    return(
        <div id="infoWindow"
        ↪  className="absolute z-50 w-1/3 h-1/5 transition-all duration-200 bg-orange-400 top-20 right-28 rounded-md">
            <div>
                <h1 className='text-white text-2xl mt-7'>{alertIcon} Current Major Accident Regions</h1>
            </div>
        </div>
    )
}

export default AlertWindow;
```

Appendix D Additional Exploratory Data Analysis

D.1 Ingestion Stage: Correct Dask DataFrames of Additional Datasets

```
crime_df = dd.from_pandas(pd.read_csv('/home/alex/FYP_dir/fyp_data/MPS Borough Level Crime (Historical).csv'), npartitions=16)
population_df = dd.from_pandas(pd.read_csv('/home/alex/FYP_dir/fyp_data/ONS mid-year population estimates London boroughs.csv'), npartitions
gdp_df = dd.from_pandas(pd.read_csv('/home/alex/FYP_dir/fyp_data/gdp at current basic rates.csv'), npartitions=16)
traffic_flow_df = dd.from_pandas(pd.read_csv('/home/alex/FYP_dir/fyp_data/traffic-flow-borough-all-vehicles.csv'), npartitions=16)
```

Figure D.1: Reading from the files using Pandas and Dask.

```
population_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 361 entries, 0 to 360
Data columns (total 28 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   ITL1 Region  361 non-null    object
 1   LA code      361 non-null    object
 2   LA name      361 non-null    object
 3   1998         361 non-null    int64
 4   1999         361 non-null    int64
 5   2000         361 non-null    int64
 6   2001         361 non-null    int64
 7   2002         361 non-null    int64
 8   2003         361 non-null    int64
 9   2004         361 non-null    int64
 10  2005         361 non-null    int64
 11  2006         361 non-null    int64
 12  2007         361 non-null    int64
 13  2008         361 non-null    int64
 14  2009         361 non-null    int64
 15  2010         361 non-null    int64
 16  2011         361 non-null    int64
 17  2012         361 non-null    int64
 18  2013         361 non-null    int64
 19  2014         361 non-null    int64
 20  2015         361 non-null    int64
 21  2016         361 non-null    int64
 22  2017         361 non-null    int64
 23  2018         361 non-null    int64
 24  2019         361 non-null    int64
 25  2020         361 non-null    int64
 26  2021         361 non-null    int64
 27  2022         361 non-null    int64
dtypes: int64(25), object(3)
memory usage: 79.1+ KB
```

Figure D.2: Population DataFrame info.

```
gdp_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 361 entries, 0 to 360
Data columns (total 28 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   ITL1 Region  361 non-null    object
 1   LA code      361 non-null    object
 2   LA name      361 non-null    object
 3   1998         361 non-null    int64 
 4   1999         361 non-null    int64 
 5   2000         361 non-null    int64 
 6   2001         361 non-null    int64 
 7   2002         361 non-null    int64 
 8   2003         361 non-null    int64 
 9   2004         361 non-null    int64 
 10  2005         361 non-null    int64 
 11  2006         361 non-null    int64 
 12  2007         361 non-null    int64 
 13  2008         361 non-null    int64 
 14  2009         361 non-null    int64 
 15  2010         361 non-null    int64 
 16  2011         361 non-null    int64 
 17  2012         361 non-null    int64 
 18  2013         361 non-null    int64 
 19  2014         361 non-null    int64 
 20  2015         361 non-null    int64 
 21  2016         361 non-null    int64 
 22  2017         361 non-null    int64 
 23  2018         361 non-null    int64 
 24  2019         361 non-null    int64 
 25  2020         361 non-null    int64 
 26  2021         361 non-null    int64 
 27  2022         361 non-null    int64 
dtypes: int64(25), object(3)
memory usage: 79.1+ KB
```

Figure D.3: GDP DataFrame info.

```
crime_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 931 entries, 0 to 930
Columns: 150 entries, MajorText to 202206
dtypes: int64(147), object(3)
memory usage: 1.1+ MB
```

Figure D.4: Crime DataFrame info.

```
traffic_flow_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 46 entries, 0 to 45
Data columns (total 32 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   LA Code          46 non-null     object
 1   Local Authority  46 non-null     object
 2   1993             46 non-null     int64 
 3   1994             46 non-null     int64 
 4   1995             46 non-null     int64 
 5   1996             46 non-null     int64 
 6   1997             46 non-null     int64 
 7   1998             46 non-null     int64 
 8   1999             46 non-null     int64 
 9   2000             46 non-null     int64 
 10  2001             46 non-null     int64 
 11  2002             46 non-null     int64 
 12  2003             46 non-null     int64 
 13  2004             46 non-null     int64 
 14  2005             46 non-null     int64 
 15  2006             46 non-null     int64 
 16  2007             46 non-null     int64 
 17  2008             46 non-null     int64 
 18  2009             46 non-null     int64 
 19  2010             46 non-null     int64 
 20  2011             46 non-null     int64 
 21  2012             46 non-null     int64 
 22  2013             46 non-null     int64 
 23  2014             46 non-null     int64 
 24  2015             46 non-null     int64 
 25  2016             46 non-null     int64 
 26  2017             46 non-null     int64 
 27  2018             46 non-null     int64 
 28  2019             46 non-null     int64 
 29  2020             46 non-null     int64 
 30  2021             46 non-null     int64 
 31  2022             46 non-null     int64 
dtypes: int64(30), object(2)
memory usage: 11.6+ KB
```

Figure D.5: Traffic flow DataFrame info.

D.2 EDA Prior to Preprocessing

```
london_acc_df_train.groupby(london_acc_df_train['Collision Date'].dt.year).size().compute().plot()
plt.rcParams["figure.figsize"] = (10,10)
plt.xlabel("Year")
plt.ylabel("Number of Road Accidents across Greater London")
```

```
Text(0, 0.5, 'Number of Road Accidents across Greater London')
```

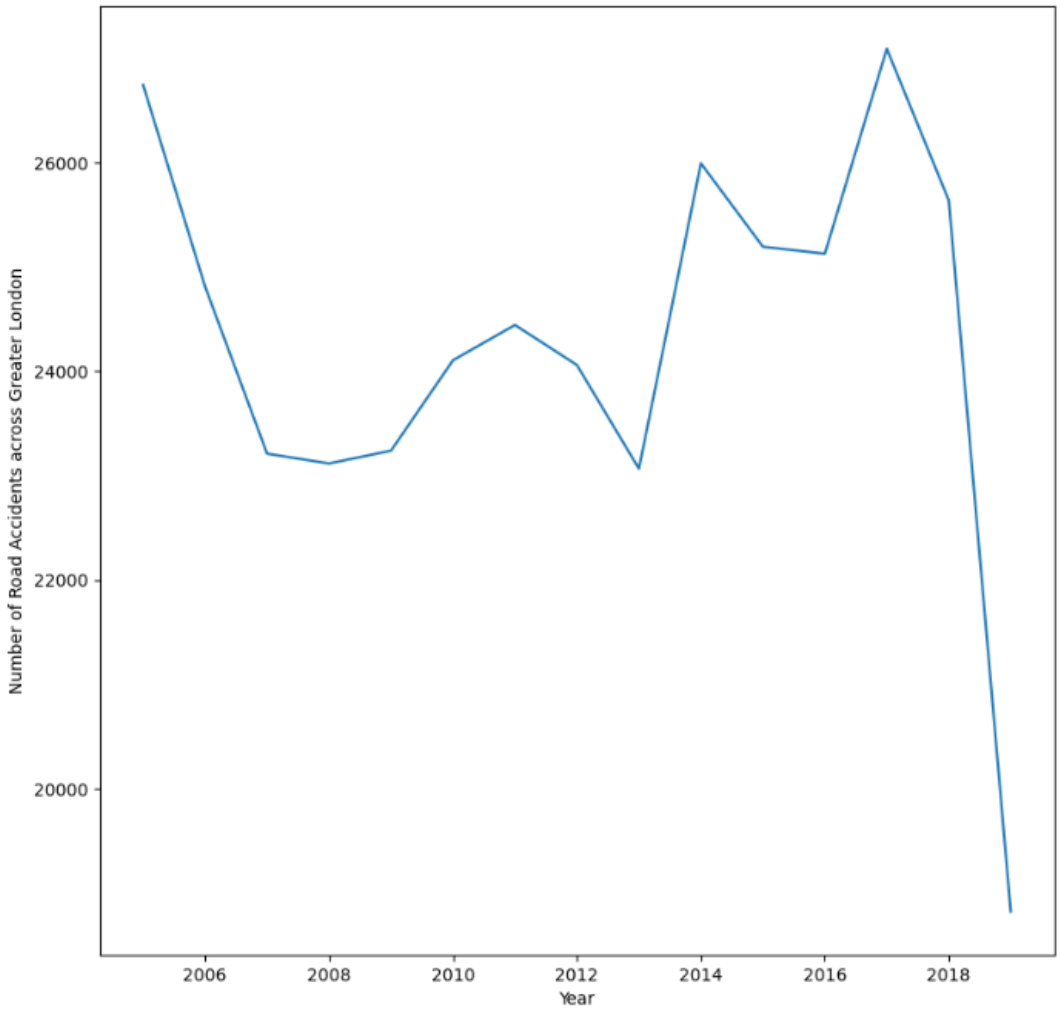


Figure D.6: Total number of recorded road accidents across Greater London over time.

```
population_greater_london = population_df[population_df['ITL1 Region'] == 'London']
# select data related to Greater London
plt.rcParams["figure.figsize"] = (15,10)
plt.plot(population_greater_london.select_dtypes('int').compute().mean())
plt.xlabel('Year')
plt.ylabel('Mean Population Count per London Borough')
plt.show()
```

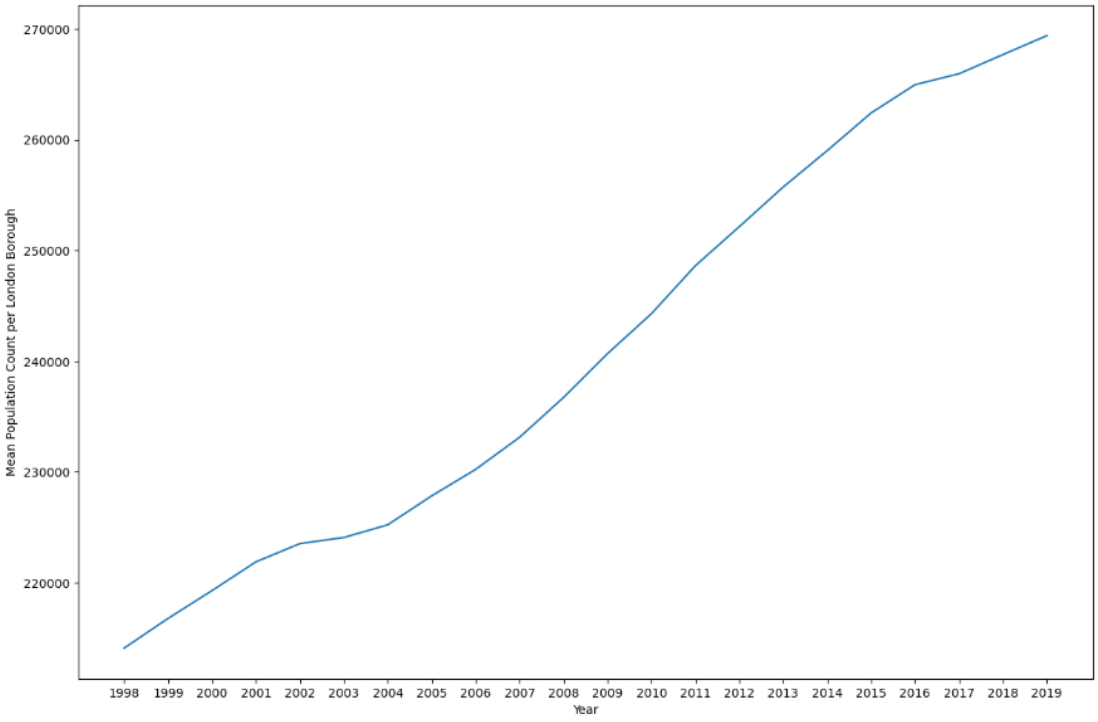


Figure D.7: Mean population count across each London borough per year.

```
gdp_greater_london = gdp_df[gdp_df['ITL1 Region'] == 'London']
plt.plot(gdp_df.select_dtypes('int').compute().mean())
plt.xlabel('Year')
plt.ylabel('Mean GDP per London Borough')
plt.rcParams["figure.figsize"] = (15,8)
plt.show()
```

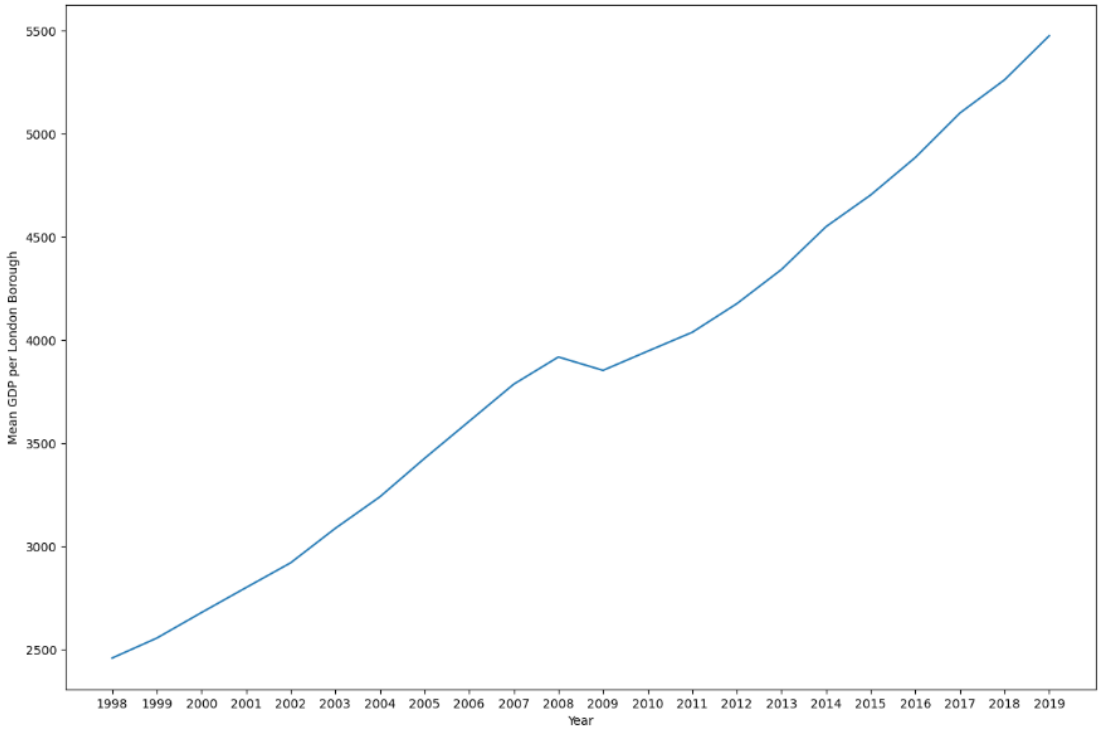


Figure D.8: Mean GDP across each London borough per year.

```
traffic_flow_greater_london = traffic_flow_df[traffic_flow_df['Local Authority'] == 'London']
# select only the figures relating to Greater London
plt.rcParams["figure.figsize"] = (15,10)
plt.plot(traffic_flow_greater_london.select_dtypes('int').compute().mean())
plt.xlabel('Year')
plt.ylabel('Mean Million Vehicle Kilometres Travelled per London Borough')
plt.show()
```

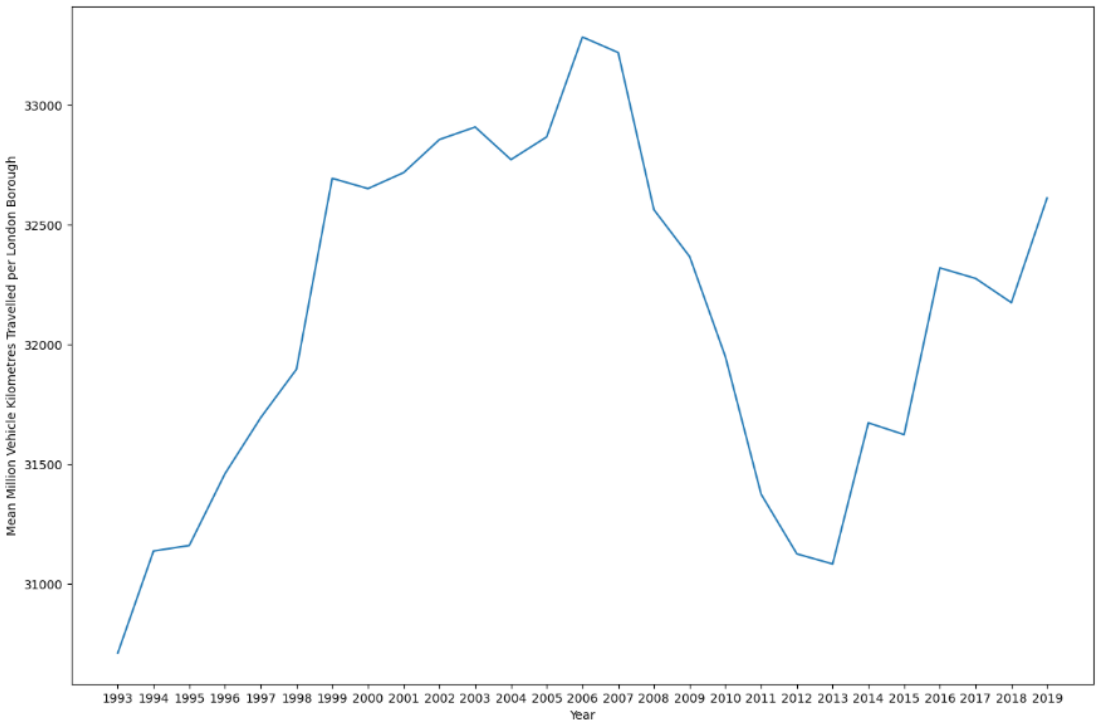


Figure D.9: Mean million vehicle kilometres travelled across each London borough per year.

```
plt.rcParams["figure.figsize"] = (10,10)
plt.plot(feature_sel_crime_df.select_dtypes('int').compute().mean())
plt.xlabel('Year')
plt.ylabel('Mean Number of Vehicle-related Offences per London Borough')
plt.show()
```

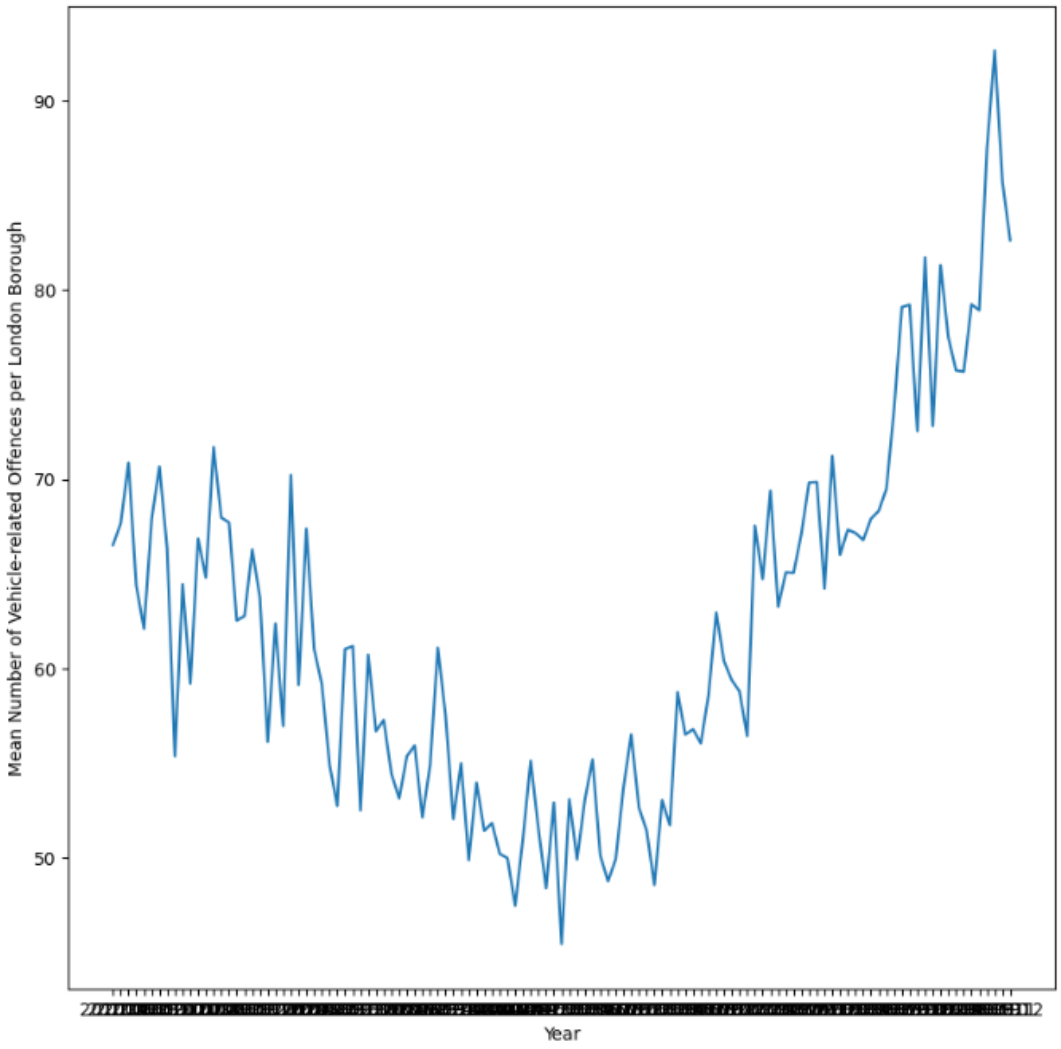


Figure D.10: Mean number of vehicle-related crimes committed across each London borough per month.

Getting some metadata information

```
london_acc_df_train.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 364650 entries, 0 to 364649
Data columns (total 28 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   Accident Ref                    364650 non-null  object
 1   Borough                         364650 non-null  object
 2   Borough Number                  364650 non-null  int32
 3   Easting                         364650 non-null  float32
 4   Northing                        364650 non-null  float32
 5   Casualty Count                  364650 non-null  int32
 6   Vehicle Count                   364650 non-null  int32
 7   Collision Date                  364650 non-null  datetime64[ns]
 8   Day                             364650 non-null  object
 9   Time                            364650 non-null  object
 10  First Road Class                358723 non-null  object
 11  First Road Number               364650 non-null  object
 12  Road Type                       364650 non-null  object
 13  Speed Limit                     345827 non-null  object
 14  Junction Detail                 364650 non-null  object
 15  Junction Control                360255 non-null  object
 16  Second Road Class               350095 non-null  object
 17  Second Road Number              364650 non-null  object
 18  Pedestrian Crossing Facilities  364650 non-null  object
 19  Light Conditions                364650 non-null  object
 20  Weather Details                 364650 non-null  object
 21  Road Surface Condition          364650 non-null  object
 22  Special Conditions at Site      27373 non-null   object
 23  Carriageway Hazards             26739 non-null   object
 24  Place Collision Reported        18823 non-null   object
 25  Collision Location Details      339840 non-null  object
 26  Attendant Count                 364650 non-null  float32
 27  Highway Authority               364650 non-null  object
dtypes: datetime64[ns](1), float32(3), int32(3), object(21)
memory usage: 69.6+ MB
```

Figure D.11: Main dataset, recorded road accidents DataFrame metadata.

```
gdp_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 361 entries, 0 to 360
Data columns (total 25 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   ITL1 Region  361 non-null    object
 1   LA code      361 non-null    object
 2   LA name      361 non-null    object
 3   1998         361 non-null    int32 
 4   1999         361 non-null    int32 
 5   2000         361 non-null    int32 
 6   2001         361 non-null    int32 
 7   2002         361 non-null    int32 
 8   2003         361 non-null    int32 
 9   2004         361 non-null    int32 
 10  2005         361 non-null    int32 
 11  2006         361 non-null    int32 
 12  2007         361 non-null    int32 
 13  2008         361 non-null    int32 
 14  2009         361 non-null    int32 
 15  2010         361 non-null    int32 
 16  2011         361 non-null    int32 
 17  2012         361 non-null    int32 
 18  2013         361 non-null    int32 
 19  2014         361 non-null    int32 
 20  2015         361 non-null    int32 
 21  2016         361 non-null    int32 
 22  2017         361 non-null    int32 
 23  2018         361 non-null    int32 
 24  2019         361 non-null    int32 
dtypes: int32(22), object(3)
memory usage: 40.9+ KB
```

Figure D.12: GDP DataFrame metadata.

```
population_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 361 entries, 0 to 360
Data columns (total 25 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   ITL1 Region  361 non-null    object
 1   LA code      361 non-null    object
 2   LA name      361 non-null    object
 3   1998         361 non-null    int32 
 4   1999         361 non-null    int32 
 5   2000         361 non-null    int32 
 6   2001         361 non-null    int32 
 7   2002         361 non-null    int32 
 8   2003         361 non-null    int32 
 9   2004         361 non-null    int32 
 10  2005         361 non-null    int32 
 11  2006         361 non-null    int32 
 12  2007         361 non-null    int32 
 13  2008         361 non-null    int32 
 14  2009         361 non-null    int32 
 15  2010         361 non-null    int32 
 16  2011         361 non-null    int32 
 17  2012         361 non-null    int32 
 18  2013         361 non-null    int32 
 19  2014         361 non-null    int32 
 20  2015         361 non-null    int32 
 21  2016         361 non-null    int32 
 22  2017         361 non-null    int32 
 23  2018         361 non-null    int32 
 24  2019         361 non-null    int32 
dtypes: int32(22), object(3)
memory usage: 40.9+ KB
```

Figure D.13: Population DataFrame metadata.

```
traffic_flow_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 46 entries, 0 to 45
Data columns (total 29 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   LA Code          46 non-null     object
 1   Local Authority  46 non-null     object
 2   1993             46 non-null     int32 
 3   1994             46 non-null     int32 
 4   1995             46 non-null     int32 
 5   1996             46 non-null     int32 
 6   1997             46 non-null     int32 
 7   1998             46 non-null     int32 
 8   1999             46 non-null     int32 
 9   2000             46 non-null     int32 
 10  2001             46 non-null     int32 
 11  2002             46 non-null     int32 
 12  2003             46 non-null     int32 
 13  2004             46 non-null     int32 
 14  2005             46 non-null     int32 
 15  2006             46 non-null     int32 
 16  2007             46 non-null     int32 
 17  2008             46 non-null     int32 
 18  2009             46 non-null     int32 
 19  2010             46 non-null     int32 
 20  2011             46 non-null     int32 
 21  2012             46 non-null     int32 
 22  2013             46 non-null     int32 
 23  2014             46 non-null     int32 
 24  2015             46 non-null     int32 
 25  2016             46 non-null     int32 
 26  2017             46 non-null     int32 
 27  2018             46 non-null     int32 
 28  2019             46 non-null     int32 
dtypes: int32(27), object(2)
memory usage: 5.8+ KB
```

Figure D.14: Traffic flow DataFrame metadata.

```
crime_df.compute().info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 931 entries, 0 to 930
Columns: 120 entries, MajorText to 201912
dtypes: int32(117), object(3)
memory usage: 451.0+ KB
```

Figure D.15: Crime DataFrame metadata.

Plotting total number of accidents per region.

```
sns.set(rc={'figure.figsize':(10,10)})
plot = sns.countplot(london_acc_df_train.compute(), x='Borough', order=london_acc_df_train['Borough'].compute().value_counts().index)
plot.tick_params(axis='x', labelrotation=90)
```

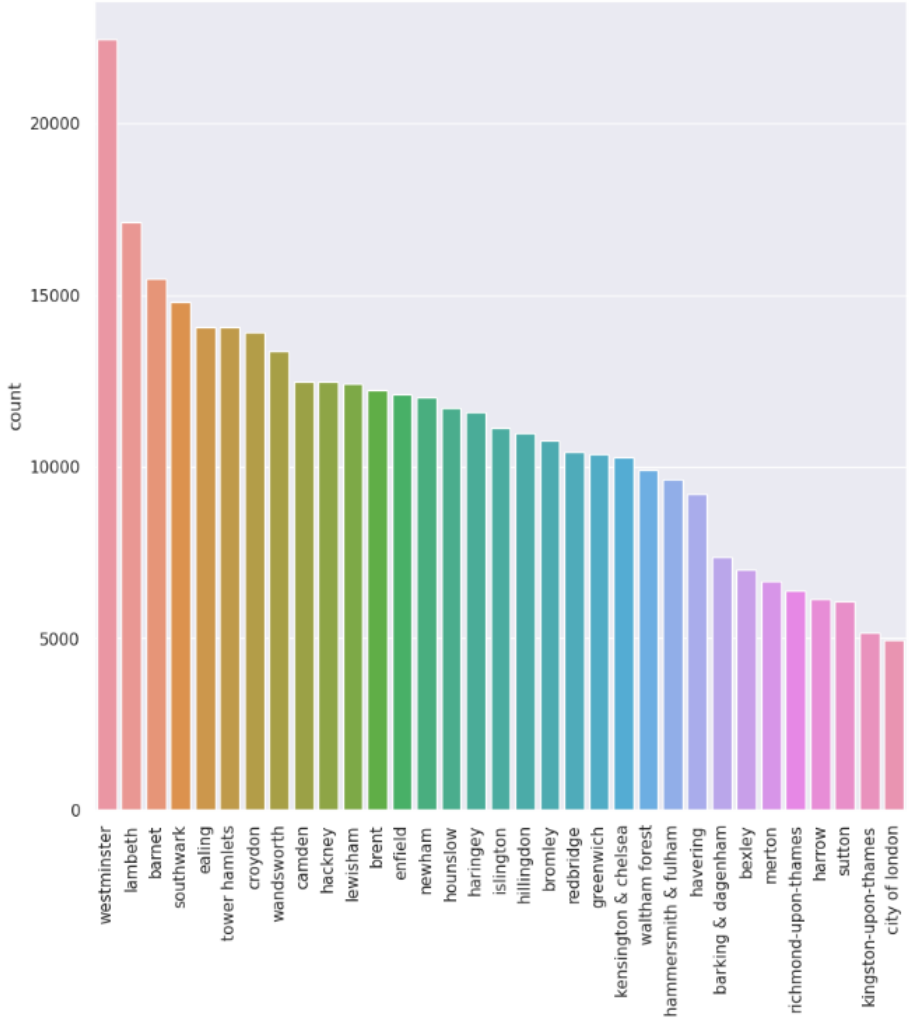


Figure D.16: The total number of road accidents recorded per London borough.

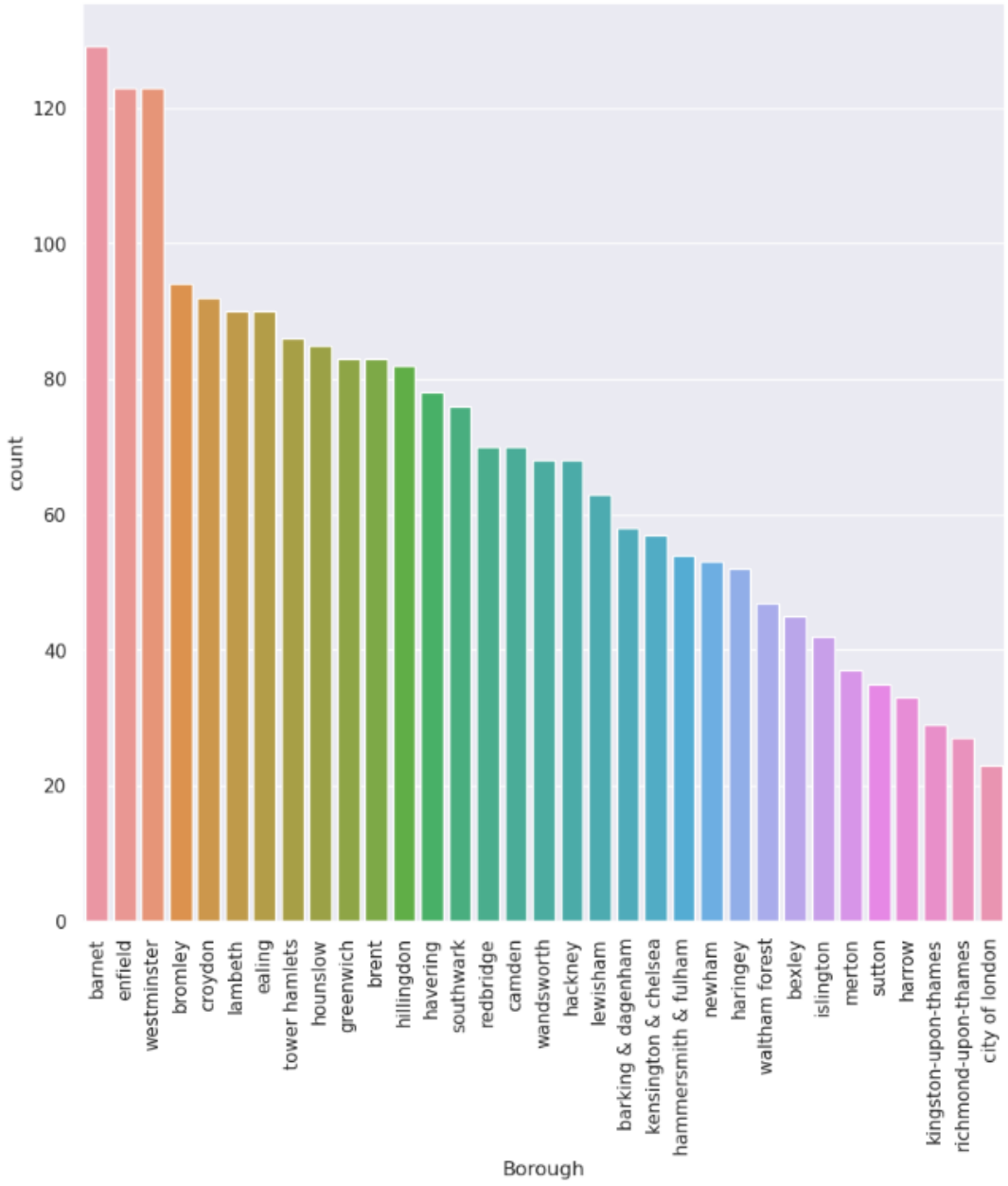


Figure D.17: The total number of 'fatal' road accidents recorded per London borough.

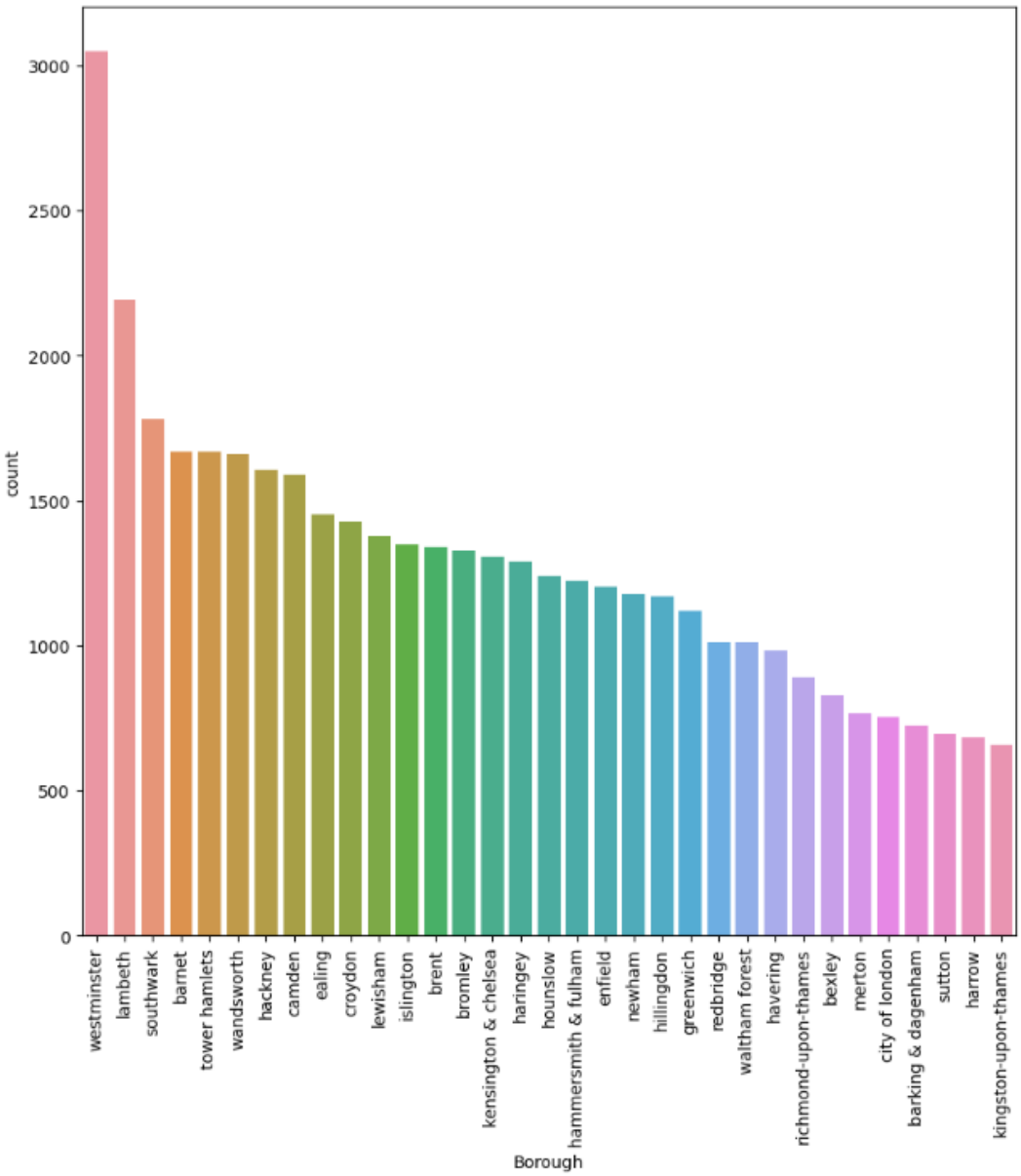


Figure D.18: The total number of 'serious' road accidents recorded per London borough.

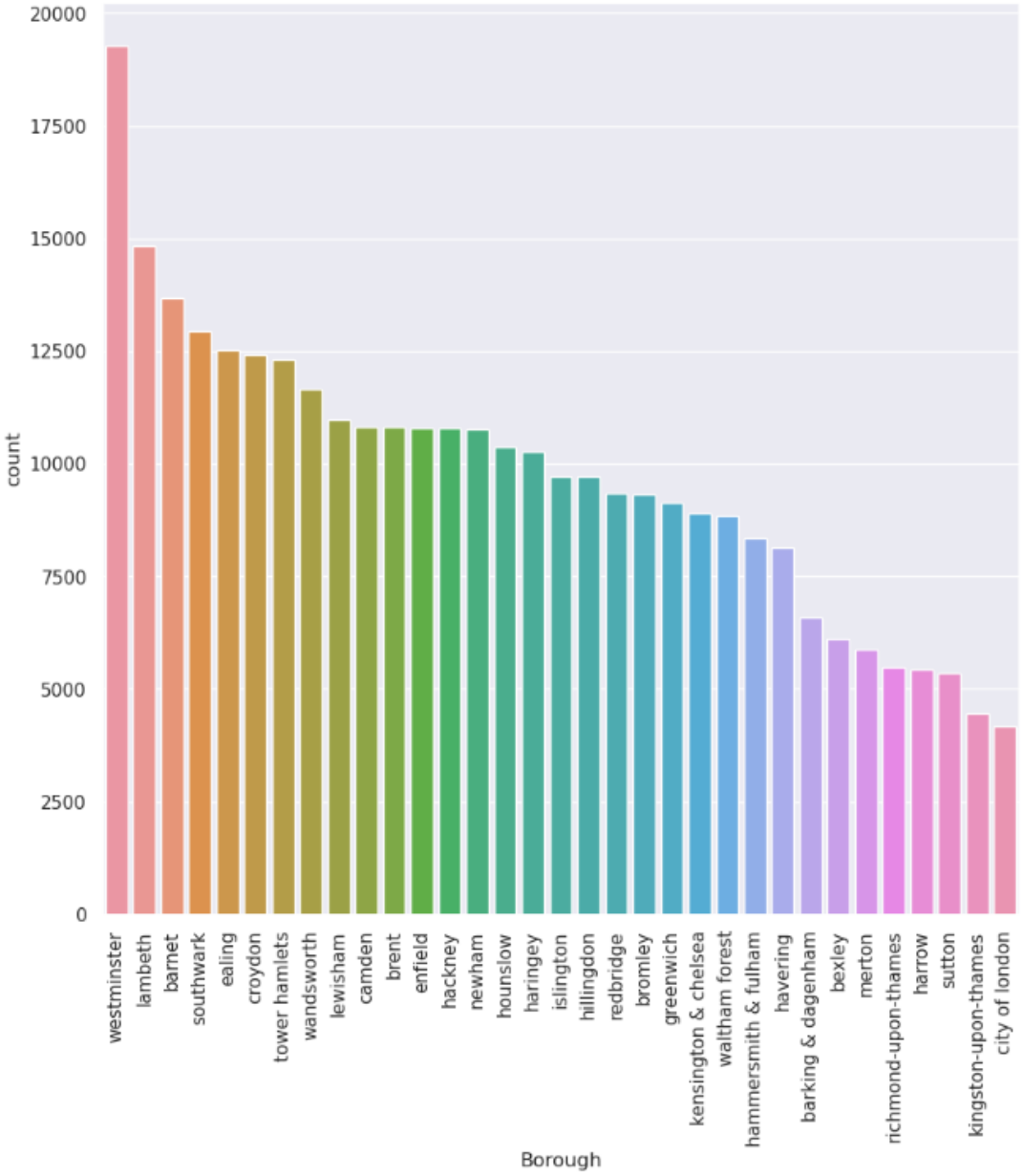


Figure D.19: The total number of 'slight' road accidents recorded per London borough.

```
Weather Details
Fine                    293337
Raining                  34680
FINE                     15105
Unknown                   6807
Other                     5140
Fine/High Winds           2050
Raining/High Winds        1803
RAINING                   1750
Snowing                   1283
OTHER                      818
UNKNOWN                    736
Fog/Mist                   606
FINE/HIGH WINDS            203
RAINING/HIGH WINDS         124
Snowing/High Winds         121
SNOWING                     69
FOG/MIST                    15
SNOWING/HIGH WINDS           3
Name: count, dtype: int64
```

Figure D.20: The value counts of the 'Weather Details' column in the main dataset.

```
london_acc_df_train['Time'].compute().value_counts()
```

```
Time
'1800       2944
'1830       2625
'1730       2606
'1700       2601
'1600       2487
            ...
23:32:00       1
02:08          1
06:51:00       1
02:31:00       1
15:51:00       1
Name: count, Length: 4209, dtype: int64
```

Figure D.21: The value counts of the 'Time' column in the main dataset.

```
(base) alex@alex-SATELLITE-PRO-C50-H-11J:~$ docker exec -it ClickHouseServer \
> clickhouse-client
ClickHouse client version 24.3.1.2672 (official build).
Connecting to localhost:9000 as user AirflowUser122.
Connected to ClickHouse server version 24.3.1.

Warnings:
 * Delay accounting is not enabled, OSIOWaitMicroseconds will not be gathered. Y
ou can enable it using `echo 1 > /proc/sys/kernel/task_delayacct` or by using sy
sctl.

6b555f57e07e :)
```

Figure E.1: Accessing the ClickHouse server instance.

```
6b555f57e07e :) use airflow_storage

USE airflow_storage

Query id: 9f560c4f-8ffb-442c-99af-879597a33c7d

Ok.

0 rows in set. Elapsed: 0.002 sec.

6b555f57e07e :) show tables
```

Figure E.2: Commands to view the tables of the designated database.

Appendix E Extra Implementation Figures

E.1 Ingestion stage

Figure E.1 uses the command, 'docker exec -it ClickHouseServer clickhouse-client' to access the ClickHouse server instance. 'exec' means to execute a command on the container, '-it' makes initialises an interactive session with the container, and 'clickhouse-client' is a special command to access the server instance and manage it using SQL.

Figure E.3: Viewing the tables created after the Ingestion stage.