ESCUELA POLITÉCNICA

# UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

MÁSTER EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

**SmartTLC**

**A Smart Traffic Light Control System for Urban Environments**


# UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

MÁSTER EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

**SmartTLC**
**A Smart Traffic Light Control System for Urban Environments**

**Autor: José Ramón Lozano Pinilla**

**Tutor: Cristina Vicente-Chicote**

**Co-Tutor: Juan Francisco Inglés-Romero**

# Acknowledgments

I would like to express my admiration and gratitude to my thesis supervisor, Cristina, for his constant help, without asking for anything in return, and for having supported me during these last years unconditionally.

I would like to thank to all the Universidad de Extremadura teachers and researchers, specially to those at Escuela Politécnina in Cáceres, for teaching me the true meaning of being an engineer, and how to deal with a problem.

I would also like to extend my thanks to the SPILab research group and all the people who compose it, as they made me feel like home every time, and helped me a lot these years back.

I would like to extend my thanks to my friends that were always there on the worst moments, and that always tried to help me out when I was blocked.

Finally, I wish to thank my parents and my sister for their unconditional support and encouragement not only throughout this Master's Thesis but also on all my past academic life and my future Ph.D.

# Resumen

El crecimiento en el número de vehiculos de combustión repercute negativamente tanto en el medio ambiente y la economía como en la vida diaria de los ciudadanos. De hecho, el sector del transporte es el que más contribuye a las emisiones de gases de efecto invernadero (GEI) y los problemas derivados de la congestión del tráfico de los que más preocupan en las ciudades de todo el mundo. Hoy en día, cada vez más ciudades apuestan por el despliegue de arquitecturas TIC para monitorizar, en tiempo real, el tráfico y su impacto medioambiental (contaminación del aire, ruido, etc.)

En esta línea, como parte de este Proyecto Fin de Máster, se desarrolla el framework ***SmartTLC***, orientado a permitir la simulación y comparación de diferentes algoritmos adaptativos de control semafórico basados en el uso de información contextual (datos históricos, en tiempo real o ambos). Además, este framework permite a los diseñadores seleccionar la mejor estrategia de control semafórico posible para diferentes situaciones, indicando cuál de ellas obtiene un mejor resultado en términos de reducción de la congestión del tráfico. Los resultados experimentales obtenidos hasta el momento demuestran que el uso de un enfoque adaptativo y sensible al contexto mejora significativamente la fluidez del tráfico, reduciendo el tiempo de espera de los vehiculos, en particular, en aquellas carreteras en las que existe una mayor densidad de tráfico.

**Palabras clave**. Control semafórico adaptativo, sensibilidad al contexto, congestión de tráfico, Ciudades Inteligentes, Sistemas de Transporte Inteligentes.

# Abstract

The increasing number of fuel-based vehicles has several negative impacts on the environment, the economy and citizen's daily life. In fact, the transportation sector is, by far, the largest contributor to Green-House Gas (GHG) emissions, being traffic congestion one of the biggest concerns in all-size cities worldwide. Actually, more and more cities are deploying ICT-based infrastructures to monitor the traffic and its environmental impact (air pollution, noise, etc.).

In this line, this Master Thesis develops the ***SmartTLC*** software framework, aimed at enabling the simulation and comparison of different traffic light adaptive control algorithms based on contextual data (either historical, real-time or both). This framework allows designers to select the best traffic light control strategy for different situations, showing which one achieves better results in terms of reducing traffic congestion. The experimental results obtained so far demonstrate that the adoption of a context-aware adaptive approach significantly improves traffic fluidity, reducing vehicle waiting time, in particular, in roads with a higher traffic demand.

**Keywords**. Adaptive Traffic Light Control, Context-Awareness, Traffic Congestion, Smart Cities, Intelligent Transport System (ITS).

# Contents


**1 Introduction** **1**

1.1 Problem statement . . . . . 2

1.2 Intended Goals . . . . . 4

1.3 Methodology . . . . . 5

1.3.1 Work breakdown into tasks . . . . . 6

1.3.2 Task planning and scheduling . . . . . 8

1.4 Document structure . . . . . 8

**2 Background and State of the Art** **11**

2.1 Basic concepts related to road traffic signaling . . . . . 11

2.1.1 What is traffic? . . . . . 11

2.1.2 What is traffic management? . . . . . 12

2.1.3 What is a traffic light? . . . . . 12

2.1.4 What is an Intelligent Transportation System? . . . . . 15

2.2 Related works and tools . . . . . 15

2.2.1 Literature review . . . . . 16

2.2.2 Tool review . . . . . 17

**3 Analysis and Design** **23**

3.1 Scenario . . . . . 23

3.1.1 Road network topology . . . . . 24

3.1.2 Traffic types . . . . . 25

3.1.3 Traffic light algorithms . . . . . 27

3.2 Requirements analysis . . . . . 29
3.3 Design . . . . . 30
3.3.1 Base architecture . . . . . 31
3.3.2 Architecture evolution . . . . . 32
3.3.3 Additional considerations . . . . . 33
3.3.4 SmartTLC architecture . . . . . 34
3.3.5 SmartTLC components . . . . . 35
3.3.6 Sequence diagram . . . . . 38

**4 Implementation and Development** **43**
4.1 Traffic Light Study . . . . . 44
4.2 Traffic Light Controller . . . . . 45
4.3 Traffic Light Predictor . . . . . 47
4.3.1 Dataset generation . . . . . 48
4.3.2 Model training process . . . . . 53
4.3.3 Model predictor selection . . . . . 55
4.4 Traffic Analyzer . . . . . 56
4.5 Traffic Light Adapter . . . . . 57
4.6 Component deployment . . . . . 59
4.6.1 Deployment strategy . . . . . 60
4.6.2 Container list . . . . . 60

**5 Result Presentation and Discussion** **63**
5.1 Simulation examples . . . . . 63
5.2 Results and discussion . . . . . 65
5.2.1 TLS component studies . . . . . 65
5.2.2 Dataset generation . . . . . 70
5.2.3 Predictor models training . . . . . 75
5.2.4 Examples execution . . . . . 80

**6 Conclusions and Future Works** **117**

6.1 Technical conclusions . . . . . 117

6.2 Problems found . . . . . 118

6.3 Future Works . . . . . 121

6.4 Personal conclusions . . . . . 122

**Appendices** **127**

**A Installation guide** **127**

A.1 What is Docker? . . . . . 127

A.2 What is Anaconda? . . . . . 128

A.3 Installation . . . . . 129

A.3.1 Windows 10 . . . . . 129

A.3.2 Ubuntu 18.04 . . . . . 129

**B User manual** **133**

B.1 Generation features . . . . . 133

B.2 Deployment guide . . . . . 136

B.2.1 Components execution . . . . . 137

B.2.2 InfluxDB queries . . . . . 142

B.2.3 Grafana configuration . . . . . 144

**C Machine Learning concepts used in SmartTLC** **147**

C.1 Machine Learning Algorithms . . . . . 147

C.1.1 Linear Regression . . . . . 148

C.1.2 Logistic Regression . . . . . 148

C.1.3 Naive Bayes Classifier . . . . . 149

C.1.4 Support Vector Machine . . . . . 151

C.1.5 K-Nearest Neighbors . . . . . 154

C.1.6 Decision Trees . . . . . 156

C.1.7 Ensemble methods . . . . . 157

C.2 Classification metrics for model performance . . . . . 160
C.2.1 Confusion Matrix . . . . . 160
C.2.2 Receiver Operator Curve . . . . . 163
C.3 Training process . . . . . 163
C.3.1 Basic Train-Test split . . . . . 164
C.3.2 Cross-validation (K-Fold split) . . . . . 164

**Bibliography** **166**

# List of Tables


2.1 Simulator comparison. . . . . 21

3.1 Traffic generic flows types with number of vehicles per hour and its range. . . . . 26
3.2 Traffic flow types based on directions. . . . . 26
3.3 Traffic Light algorithm variations based on time interval. . . . . 28
3.4 Proportion by traffic flow type. . . . . 28
3.5 Traffic Light algorithm variations based on vehicles proportion. . . . . 29

4.1 Traffic analyzer vehicle bounds per traffic type. . . . . 57

5.1 Best traffic light algorithm (5 seconds interval) per traffic type. . . . . 66
5.2 Best traffic light algorithm (proportion) per traffic type. . . . . 68
5.3 Swapped days on the generated dataset. . . . . 72
5.4 Average of time elapsed and F1 score in TLP models. . . . . 76
5.5 Best TLP date-based models based on F1 score. . . . . 77
5.6 Best TLP context-based models based on F1 score . . . . . 78

C.1 Comparison between the different Naive Bayes algorithms. . . . . 151

# List of Figures


1.1 Gantt diagram . . . . . 8

2.1 Traffic light phases. . . . . 13
2.2 Traffic light coordination example. . . . . 14
2.3 Eclipse SUMO . . . . . 18
2.4 AnyLogic - Road Traffic Simulator Software . . . . . 20
2.5 AimsunNext traffic simulator . . . . . 20

3.1 Junction possible directions. . . . . 25
3.2 Bachelor Thesis architecture. . . . . 31
3.3 Smart TLC architecture. . . . . 34
3.4 Smart TLC full sequence diagram. . . . . 39

4.1 Smart TLC main components. . . . . 43
4.2 Time patterns. . . . . 49

5.1 Plot with the traffic light study with an interval of 5 seconds. . . . . 67
5.2 Plot with the traffic light study with traffic flow proportion. . . . . 69
5.3 Different generated calendar patterns. . . . . 74
5.4 Monday simulation - Time pattern. . . . . 82
5.5 One day simulation - No adaptation. . . . . 83
5.6 One day simulation - Only predictions based on date adaptation. . . . 84
5.7 One day simulation - Only real-time analyzer adaptation. . . . . 86
5.8 One day simulation - Both analyzer and context predictor adaptation. . 89

5.9 One day simulation - Total waiting time summary. . . . . 91
5.10 Date interval simulation - Time pattern. . . . . 92
5.11 Date interval simulation - No adaptation. . . . . 93
5.12 Date interval simulation - Only predictions based on date adaptation. . 95
5.13 Date interval simulation - Only real-time analyzer adaptation. . . . . 98
5.14 Date interval simulation - Both analyzer and context predictor adaptation. 101
5.15 Date interval simulation - Total waiting time summary . . . . . 103
5.16 Random day simulation - Time pattern. . . . . 104
5.17 Random pattern simulation - No adaptation. . . . . 105
5.18 Random pattern simulation - Only predictions based on date adaptation. 107
5.19 Random pattern simulation - Only real-time analyzer adaptation. . . . 109
5.20 Random pattern simulation - Both analyzer and context predictor adaptation. . . . . 111
5.21 Random pattern simulation - Total waiting time summary. . . . . 113

B.1 InfluxDB web page. . . . . 143
B.2 Grafana add data source. . . . . 145

C.1 Linear vs Logistic Regression. . . . . 149
C.2 Support Vector Machine. . . . . 152
C.3 Support Vector Machine kernel function. . . . . 154
C.4 KNN algorithm main steps. . . . . 155
C.5 Decision Tree example. . . . . 156
C.6 Bagging example. . . . . 158
C.7 Example of random forest simplified. . . . . 159
C.8 Example of a normalized confusion matrix with three classes. . . . . 160
C.9 Different ROC curves . . . . . 163
C.10 K-Fold process. . . . . 165

# Chapter 1

# Introduction

The increasingly growing number of vehicles, nowadays mostly fuel-based, has important negative impacts not only in the environment and in the personal and global economy, but also in our daily life: long waiting times and product delivery delays due to traffic congestion, noisy and more prone to accidents urban areas, etc. [1].

In 2019, the transportation sector was the largest contributor to *Green-House Gas* (GHG) emissions, accounting up to 23% in Europe (29% in the USA) of the total GHG emissions. Up to 82% of these emissions were produced by light-duty vehicles (58%) and medium and heavy trucks (24%) [2]. It is worth noting that these vehicles are not homogeneously distributed across all continents, countries and regions. For instance, Asia (with 518M vehicles), Europe (419M) and North America (350M) account almost 90% of the vehicles worldwide. However, the breakdown *per capita* is quite different: 0.71 (710 vehicles per thousand people) in North America, 0.52 in Europe and 0.14 in Asia. The USA is, by far, the country with more cars *per capita* (0.89).

According to the *International Energy Outlook 2021* [3], electric vehicles currently make up only 30% of the 1.446 billion cars estimated on Earth in 2022 [4], which is definitely not enough to meet climate change goals. Furthermore, the report seriously warns that emissions from the transportation sector are expected to increase through 2050 unless world leaders establish legal and regulatory changes. In this context, the implementation of new mobility management policies becomes an urgent must.

The INRIX company, which provides location-based data and analytics, offers yearly a *Global Traffic Scorecard* [5] including traffic information about more than 1000 cities belonging to 50 countries in all continents. According to this report, London, Paris, Brussels, Moscow and New York are the top 5 most congested cities worldwide with 148, 140, 134, 108 and 102 hours lost in congestion per driver in 2021, respectively. The INRIX report also details how the COVID-19 pandemic has affected traffic congestion since 2019. The lockdowns and quarantines imposed worldwide in order to prevent the expansion of the SARS-Cov-2 virus dramatically reduced all kinds of mobility. As a consequence, traffic congestion nominally disappeared during the first half of 2020 and, in spite of its slow growth since the relaxation of the mobility restrictions, it is not expected to return to pre-COVID levels, at least in 2022. In spite of that, dealing with traffic congestion, particularly in urban environments, has become the main issue being addressed by cities of all sizes and is one of the central axes of the *Smart City* concept [6].

Traffic congestion refers to an excess of vehicles on part of a road at a given time, resulting in the reduction in their average speed and, in some cases, frequent stops and goes. Traffic congestion is typically due to: (1) punctual events related to traffic accidents, road work, or bad weather conditions (e.g., heavy rain, hail or snow); (2) changes in the traffic patterns, e.g., an unexpected increase of the road demand due to special events involving many people commuting by car; or (3) a flawed design of the road capacity or of the traffic light signaling. The bottlenecks caused by traffic congestion have many negative effects, affecting the environment (increased GHG emissions) and the citizens, both those commuting (longer travel times and increased costs due to higher fuel consumption) as well as those living in congested areas (higher noise and air pollution levels).

## 1.1 Problem statement

This Master Thesis proposes a smart traffic light control system aimed at helping designers analyze how different strategies behave under different situations to select

the one that provides better results in terms of alleviating traffic congestion.

Addressing traffic congestion issues is a highly complex challenge that involves (1) context-awareness, enabling real-time mobility monitoring (road demand, average waiting time, GHG emissions, etc.); (2) the identification of (eventually changing) traffic patterns; and (3) the adequacy of the road infrastructures and of the traffic lights that control the vehicular (and pedestrian) flows in urban environments.

Nowadays, more and more cities count on sensors enabling the monitoring of road occupancy, air quality, location of public transportation vehicles, etc. Besides, some cities are also deploying wide *Internet-of-Things* (IoT) sensor networks aimed at retrieving the richer data possible. Some initiatives even consider retrieving contextual information from vehicles or from the citizens' mobile devices [7].

Appropriately processing the context data provided by the previously mentioned sensors can be useful (1) to predict future road demands and adequately plan and design the required infrastructures (e.g., new roads of the appropriate size and type); (2) to identify relevant traffic patterns (e.g., daily peak hours) in order to schedule the best (predefined) traffic light control policies; and (3) to react to unforeseen situations (e.g., an unexpected increase of road demand), enabling the dynamic adaptation of the traffic light control strategy.

In this context, this Master Thesis introduces ***SmartTLC: a Smart Traffic Light Control system for urban environments***. **SmartTLC** provides a framework to test and compare (in terms of time lost in congestion) four different traffic light control strategies, namely: (1) a non-adaptive approach based on traffic lights with static, predefined phases; (2) an adaptive approach that modifies the traffic light control strategy according to the traffic patterns predicted from historical data; (3) an adaptive approach that uses real-time contextual information to dynamically regulate the traffic light phases; and (4) an adaptive approach that combines both historical and real-time contextual information to regulate the traffic lights.

## 1.2 Intended Goals

This Master Thesis is targeted at ***monitoring the contextual information related to traffic light intersections*** (junctions), in order to be aware of the current traffic flows. The hypothesis is that this information, together with historical traffic data, can be used to ***adapt automatically the traffic light control algorithms*** in order to minimize the amount of time vehicles should wait at the junctions. Reducing this waiting time would also indirectly improve other metrics such as pollution, GHG emissions, noise, etc. In order to find the best traffic light adaptation policies, ***different approaches will be implemented and their results compared***.

***The main goal of this Master Thesis is to develop a software framework that allows to compare different traffic light adaptation strategies to help designers decide which one performs better (in terms of reducing congestion and vehicle waiting time) in different situations.***

This general objective can be divided, in turn, into the following sub-goals:

- Identify relevant traffic flow patterns in a junction considering the number of vehicles crossing it on each direction.
- Analyze different traffic light control algorithms and select the one that performs better for each traffic flow pattern.
- Predict the traffic flow pattern on a given date and hour based only on historical context data.
- Detect the current traffic flow pattern based only on real-time context data.
- Analyze and compare the results obtained by different traffic light adaptation approaches that make use of traffic pattern predictions based on historical data, real-time data, or a combination of both.

It is worth noting that the experiments performed throughout this work are based on a simple traffic network topology, including only one junction with four main roads (North, East, South and West) which, in turn, are grouped into two main directions (North-South and East-West). This scenario and other possible ones are later detailed in Section 3.1.

## 1.3 Methodology

Based on the research nature of this Project, an agile approach has been followed. The main reasons for selecting an agile methodology were the following: (1) it is oriented to goal completion, while it is flexible enough to allow the inclusion of new goals and the modification of existing ones; (2) its progress is checked through regular meetings, providing continuous feedback to both the student and the supervisor; (3) in the event that one of the features of the project fails or worsens the solution, it can be quickly identified and corrected; and (4) the project documentation is incrementally developed as the project progresses.

The work presented here has been defined as part of a long-term scientific research line, and it will be the basis for a Doctoral Thesis to be carried out after completing the Master. Moreover, the research focus of this work has allowed us to delve into certain aspects that, otherwise, would not be so important, such as making a deep and critical analysis and evaluation of different design and implementation alternatives.

Weekly meetings were scheduled, always following the same structure: (1) the meeting began showing the advances achieved during the last week, gathered in the documentation; (2) the main difficulties found (if any) were commented, and the possible causes and potential solutions were discussed; (3) weekly goal achievement was checked; (4) previous goals were adjusted (when needed) and new features and goals for the next week were defined; and, finally, (5) the decisions made during the meeting were documented for sake of traceability.

To support this methodology, tools like *Click Up* (a project management tool used as a canvas, where different tasks and goals can be defined and scheduled) and *GitHub*

(as a version control system and repository for storing the framework implementation, the examples, and the experimental results) were used.

### 1.3.1 Work breakdown into tasks

The work proposed in this Project has been divided into the following tasks:

- **Requirements analysis**: this task, together with the one dealing with the design, are considered central as they lay the foundations of the proposed framework. This task defines the main system requirements, along with its intended goals. In order to get a good understanding of the problem domain, plenty of references about vehicular traffic, traffic congestion and current traffic light systems were reviewed. This also helped us identify the problems and limitations of current traffic congestion control mechanisms. The time invested on this task was 15h.

- **Analysis of related works and tools**: in order to contextualize the development of the framework, different approaches and tools aimed at addressing traffic congestion problems were analyzed. This study helped us focus the Project on those aspects not solved (or even addressed) by other approaches and tools. The time invested on this task was 40h.

- **System design**: this task aims at defining the framework architecture in terms of its components, their behavior and the interactions existing among them. The time invested on this task was 10h.

- **Development of the *Traffic Light Study* component**: this component plays a key role as it provides traffic light engineers with the required information (average waiting time per vehicle) to select the best control algorithm for a given traffic pattern. The time invested on this task was 70h.

- **Development of the *Traffic Light Controller* and the *Traffic Light Adapter* components**: the former, gathers and makes available to all others relevant traffic context data (so far provided by the traffic simulator) while the later, adapts the

traffic light control algorithm according to the predicted/identified traffic pattern. The time invested on this task was 100h.

- **Development of the *Traffic Light Predictor* component**: this task develops and trains different machine learning algorithms so that they can predict the traffic pattern on any give date and hour based on historical context data. The dataset containing the (so far simulated) historical data was also generated as part of this task. The time invested on this task was 90h.

- **Development of the *Traffic Analyzer* component**: this task develops a component that analyzes the current traffic flows (provided by the *Traffic Light Controller*) and, from it, determines the current traffic pattern. The time invested on this task was 25h.

- **Development of the *Data storage* and *Data Visualization* components**: this tasks develops two auxiliary components: the former stores and manages all the data published by all the other components (time invested: 6h), while the later allows to graphically display those data (time invested: 5h).

- **Additional component development**: this task relates with the configuration of the publication-subscription middleware enabling the communication of all the previous components, and the development of two additional auxiliary modules enabling, respectively, the recording of the experiments (*Recorder*) and their reproduction (*Payer*). The time invested on this task was 10h.

- **System deployment and testing**: once all the previous components were implemented, this task was in charge of deploying and testing them together. The time elapsed on this task is 25 hours.

- **Experiment execution**: as part of this task, several experiments were executed using the developed framework. The results obtained in these experiments were analyzed and compared, and conclusions about the different traffic light control adaptation mechanisms were extracted. The time invested on this task was 20h.

- **Documentation**: this task runs concurrently with all the previous ones throughout the Project. It includes both the external and the internal (*in-code*) documentation of the framework developed as part of this Project. The time invested on this task was 90h.

### 1.3.2 Task planning and scheduling

Figure 1.1 shows the Gantt diagram of the Project, which started in September 2021 and was completed by December 2021. The development and documentation of the Project took about 500 hours to be completed. The workload associated to each task is detailed in the previous section.

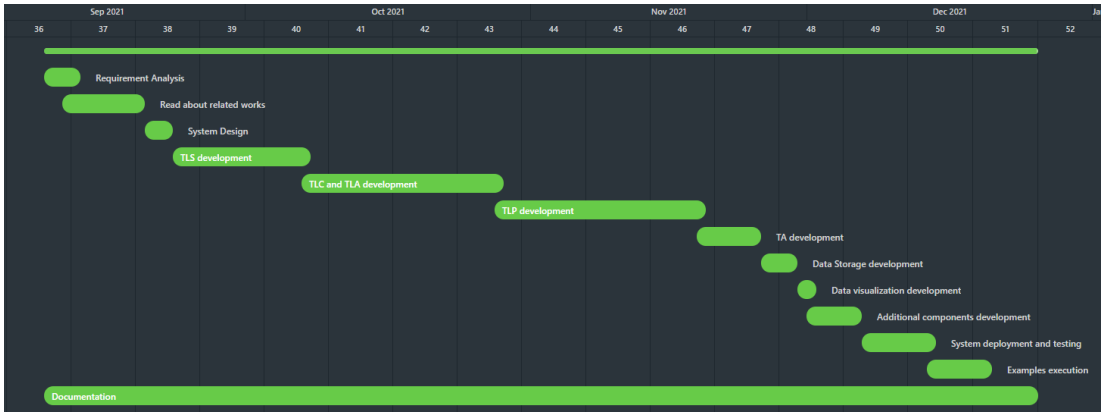



Figure 1.1: Gantt diagram

## 1.4 Document structure

The rest of this document is organized as follows:

- **Chapter 2** introduces traffic-related concepts, along with a review of related works and existing traffic simulation frameworks;

- **Chapter 3** outlines possible traffic scenarios, and describes the system requirements analysis and the evolution of the software architecture, from its original to its final design;

- **Chapter 4** details the main components of SmartTLC, reporting on the relevant implementation decisions made for each of them;
- **Chapter 5** presents and discusses the experimental results obtained during the Project; and, finally,
- **Chapter 6** draws some conclusions and outlines future works.

# Chapter 2

# Background and State of the Art

This chapter introduces the main concepts related to traffic light signaling and control, and reviews state-of-the-art works and tools related to this area.

## 2.1 Basic concepts related to road traffic signaling

There are several relevant concepts related to road traffic signaling [8] that need to be clearly defined to establish a solid basis on which building the the proposed framework.

### 2.1.1 What is traffic?

There are different types of traffic depending on the users, the environment and the types of vehicles involved.

The Oxford Dictionary defines traffic as "the vehicles that are on a road at a particular time" [9]. Obviously, this definition focuses on vehicular road traffic. In this context, both traffic laws (which establish the legal framework for vehicle circulation) and informal rules (which typically govern the interactions between drivers and between drivers and pedestrians) apply. Lane, crossroad, intersection, traffic light, traffic signal, etc., are concepts also related to vehicular road traffic.

### 2.1.2 What is traffic management?

Traffic management [10] is a logistics branch related to the planning, control and purchase of transport services required to move vehicles, independently of their type: road vehicles, watercraft, aircraft, etc. This management should not be confused with road traffic control, as it involves directing both vehicular and pedestrian traffic around a construction area, accident or anything that could cause danger to them. In fact, these worksites usually involve closing a road or a part of it. This redirection aims at ensuring the safety of emergency response teams, construction workers and even the general public. This control uses different means of traffic monitoring to manage traffic flows and concerns about traffic congestion.

### 2.1.3 What is a traffic light?

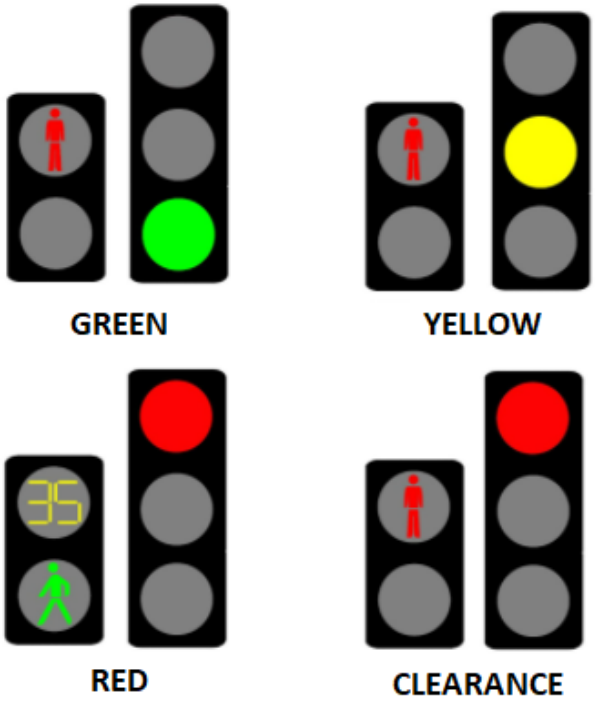
Also known as a traffic signal, it is a signalling device located mainly on-road intersections (where two or more roads meet each other) and pedestrian crossings to control traffic flows. Furthermore, these devices are used for traffic control and coordination. There are several types of traffic lights (road, pedestrian, tram, etc.) but in this case, the described is the road type, because it is the one used on the project.

According to Oxford Learner's Dictionary, a traffic light is "a signal that controls the traffic on a road, by means of red, yellow, and green lights that show when you must stop and when you can go" [11].

These devices are composed of three different lights: red, amber and green; with which it is possible to define different phases on a given traffic light, indicating one set of possible movements that are allowed to happen at the same time; for example, only allowing right turns and forward but no left turns. In fact, traffic lights phases are not standardized at all, as some countries have additional rules for signalling, but all of them have four main phases that mean the same. These phases are shown in Figure 2.1 and are:

- **Green phase**: allows traffic to proceed in the given direction.

- **Amber phase**: it is also known as "Yellow phase" and it warns that the phase is going to change to the red one.
- **Red phase**: prohibits any traffic from proceeding in a given direction.
- **Clearance**: it is a phase where all the lights are set to the red state. It is used to end a remaining phase, e.g. once the green phase on pedestrian crossing ends.



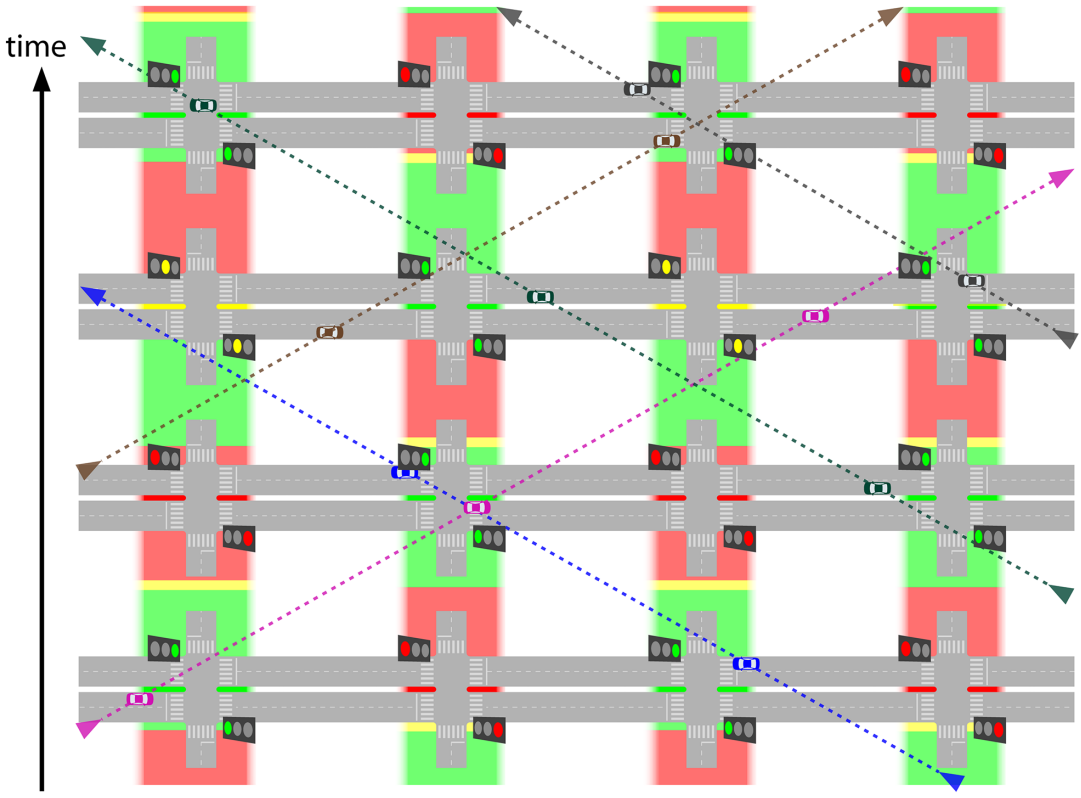

*Source:* Documentation owner
Figure 2.1: Traffic light phases.

Some of the additional phase variations could be: (1) **red and amber phases together** which indicates that the phase is going to change to green, used in the United Kingdom and Germany, among others; and (2) **amber alone** that like the previous one, it indicates that the phase is going to change to green, but in this case only if the previous phase was red, only used on Russia.

It is worth mentioning that as there are a wide variety of traffic light control systems, that can be divided into four main types:

- **Fixed time control**: this kind of control is an electro-mechanical simple and old fashion controller that is used nowadays, that can only control one way at a given time.

- **Coordinated control**: it is important to differentiate between coordinated signals and synchronized signals, as this last one means that all signals change at the same time. The coordinated signals are controlled from a master controller, setting the green phases in "cascade". Even this coordination can be accomplished in real-time, in order to deal with changing traffic patterns, by using different detectors. This behavior can be seen in Figure 2.2.




*Source:* Institute for Transportation and Development Policy (`http://shorturl.at/zDS57`)
Figure 2.2: Traffic light coordination example.

- **Adaptive control**: in this strategy, the traffic signal timing adapts based on the actual traffic demand. This control is achieved by using an adaptive traffic

control system based on both software and hardware.

- **Other types**: there are other types of control that can not be grouped along with the previous ones. These are: failure, part-time operation, rail-road preemption, bus and transport priority, emergency vehicles, actuated control (by pressing a detector), etc.

Lastly, it is important to describe different traffic light related concepts such as: (1) effective green time, that is the time when vehicles are considered effectively moving; (2) cycle time, the time between the green light is repeated, passing once to red, on those situations that there are not used any detectors as these modify the cycle time; (3) the red time is the cycle time minus the effective green time; and (4) the lost time is the period between the end of the green phase and the start of the next green phase.

### 2.1.4 What is an Intelligent Transportation System?

Intelligent Transport Systems (ITS) [12] refers to the use of communication and information technologies in the transport industry. On the one hand, the main purposes of ITS are: (1) improving people and vehicles mobility, (2) increasing safety by reducing traffic congestion effectively, (3) aiming at transport goals as demand management, public transport or emergency vehicles. On the other hand, ITS main applications are: (1) improving traffic flow by reducing traffic congestion, (2) improving air quality by reducing pollution levels locally and (3) improving safety by providing warnings in advance. In fact, the proposed framework fits perfectly as an ITS.

## 2.2 Related works and tools

Reviewing the literature, it is possible to find several related works aimed at solving problems similar to those selected to be covered in this Project. In fact, these works are directly related to the use of contextual information in traffic scenarios in order to reduce traffic congestion or vehicles' waiting time. Besides, it is also possible to find

several tools to simulate different traffic scenarios. A comparison among them will help us decide which one fits better the Project requirements. Next, the following two sections, provide an overview of both related works and tools.

### 2.2.1 Literature review

The work presented in [13] points out that the unreliability of route services such as Google Maps (due to its inability to deal with sudden updates related to accidents or other traffic problems that may appear on a route), may negatively affect the traffic conditions. In order to solve this problem, the authors propose an efficient context-aware vehicle incidents route service management for an ITS, which monitors contextual information (e.g., incidents, weather conditions, etc.) and, based on it, proposes alternative routes. This work aims at reducing the waiting time of the vehicles by providing alternative routes based on the contextual information, which in fact is achieved. However, the system itself does not adapt its behavior based on this information which, in fact, is one of the main purposes of this Project.

The IoT-based framework presented in [14] aims at managing road traffic congestion by using the available infrastructures and resources from an Intelligent Traffic Management System (ITMS) [15]. In order to achieve this, the proposed framework uses traffic contextual information (e.g., number of vehicles or their speed) to infer knowledge and make decisions on congestion control (e.g. controlling traffic lights). Furthermore, the framework monitors the traffic status in order to offer it to transport authorities. This work is similar to the one proposed in this Project. However, it does not consider different historical context data or different adaptation techniques which results could be compared. In fact, its real contribution seems to be the definition of the contextual monitoring traffic framework and several services that could be of value to end-users, but there is no application described in real-world scenarios.

The work presented in [16] also proposes a framework which combines the Virtual Traffic Light (VTL) [17] technology with Fog Computing [18] to replace the actual traffic light systems. Besides, they use Vehicular Ad-hoc Networks (VANETs) [19]

communication to handle the traffic congestion and optimize the traffic light signaling. The main issue with this framework is that it is based on the Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) communications, which are not yet as extended as required in current vehicles to make this approach feasible in real-world scenarios.

The IoT-based Intelligent Traffic Congestion Control System proposed in [20] aims at reducing the traffic congestion on a given road topology, based on the contextual traffic density detected by different sensors. With this density information, they define several green phase durations in order to create different traffic light algorithms. This approach is similar to the one followed in this Project with the real-time context data analyzer. However, they do not consider the potentials of using also historical context data. Besides, the paper does not include any experimental result and, thus, it is not possible to measure whether their proposal can really reduce (and to what extent) the vehicles' waiting time or the traffic congestion.

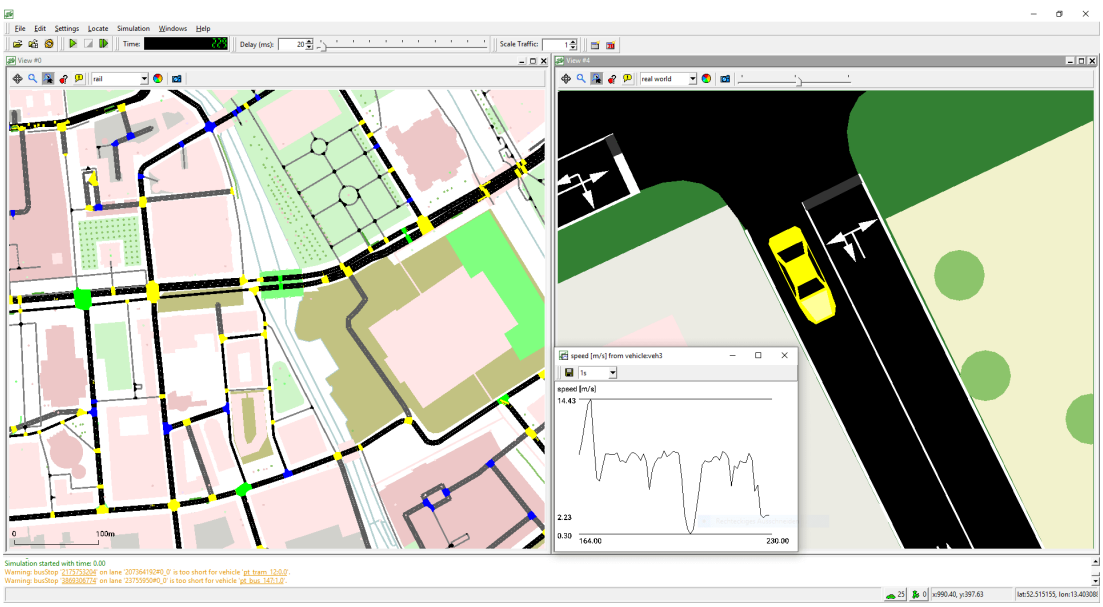
The mixed autonomy traffic framework (FLOW) [21] [22] allows the application of Deep Reinforcement Learning (DRL) approaches to control autonomous vehicles and intelligent infrastructure such as smart traffic lights. Its main purpose is to offer a framework enabling the definition and testing of DRL techniques in traffic problems, such as bottleneck decongestion [23] or the use of autonomous vehicles to check their impact on the traffic environment (e.g. congestion reduction, fewer accidents, etc.). The main difference between FLOW and the framework proposed in this Project is that FLOW only allows using DRL techniques for traffic control, while our framework is open to any kind of adaptation technique (including DRL).

### 2.2.2 Tool review

This section briefly reviews and compares some of the nowadays most widely used traffic simulators, enabling traffic-related context monitoring and interaction with the traffic light control systems.

#### 2.2.2.1 Eclipse SUMO

Eclipse SUMO (Simulation of Urban MObility) [24] is a free open source traffic simulator developed by the Eclipse Team. SUMO allows the design of a wide variety of traffic scenarios in which it is possible to define: (1) different road networks, intersections, traffic roundabouts, etc.; (2) a multitude of sensors and actuators (cameras, passage sensors, environmental sensors, traffic lights of different types, crossing barriers, etc.); and (3) a broad variety of actors, including pedestrians and vehicles (cars, bicycles, motorcycles, buses, emergency vehicles, etc.) as can be shown in Figure 2.3.



*Source:* Eclipse ORG (`http://shorturl.at/mrtA9`)
Figure 2.3: Eclipse SUMO

The main Eclipse SUMO features are: (1) simulation of self-driving vehicles, (2) simulation of interconnected vehicles, (3) traffic management, (4) simulation at the microscopic level, (5) multimodal traffic simulation, (6) online interaction, (7) import of existing topologies, (8) on-demand traffic generation, (9) traffic light management, (10) efficiency and (11) portability.

It is worth mentioning the Eclipse SUMO is highly configurable by using XML

files. Some of these configuration files can be: network topology; vehicles trips, routes or flows; traffic light algorithms; additional sensors, etc. Besides, this simulator offers an API known as TraCI (**Tra**ffic **C**ontrol **I**nterface) [25], which provides access to SUMO road traffic simulation; allows to retrieve values of simulated objects and to manipulate their behavior on execution time.

#### 2.2.2.2 AnyLogic

AnyLogic [26] is a multimethod simulation modelling tool that is developed by the AnyLogic Company. It is a subscription-based payment simulator that offers a free trial of 30 days.

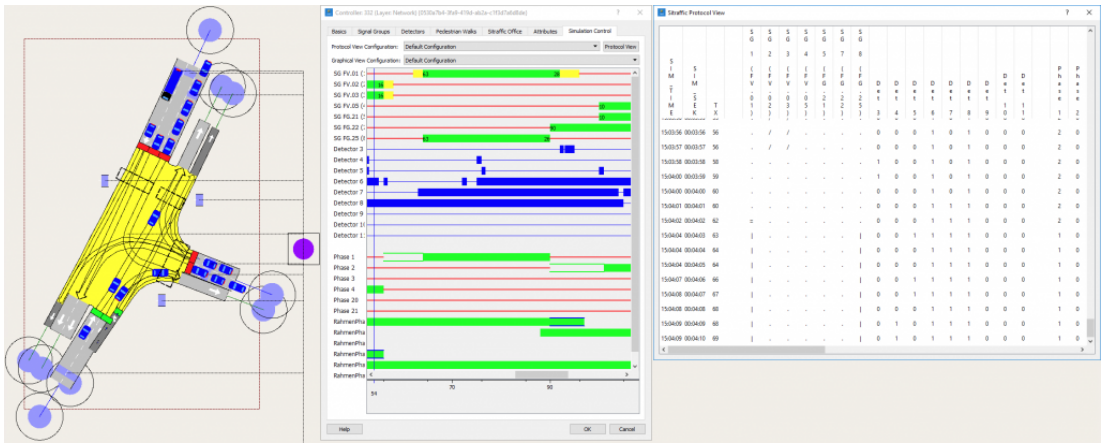
One of the many simulators Anylogic provides is a Road Traffic Simulator Software, shown in Figure 2.4 which is used for: (1) traffic planning, (2) throughput analysis as general statistics for congestion and traffic jams, (3) traffic light timing and sequencing in order to develop a system-wide optimization and (4) the integration of public object and buildings into road networks.

#### 2.2.2.3 Aimsun Next

Aimsun Next [27] is a traffic simulation software developed by the Aimsun Company. It is also considered a multi-resolution and multi-model modeling tool that simulates mobility on any scale. Like the previous tool, it is a subscription-based payment simulator, offering 30 days of a free trial.

The main functionality of this simulator is the simulation and prediction of interactions between different kinds of road users as vehicles or pedestrians, in order to evaluate different traffic operations and their environmental impact. It allows the following traffic models: microscopic (individual, low level), mesoscopic (object groups, medium level), macroscopic (full network, high level) and a hybrid approach. This simulator is shown in Figure 2.5.

*Source:* AnyLogic (`http://shorturl.at/orzXZ`)
Figure 2.4: AnyLogic - Road Traffic Simulator Software



*Source:* Aimsun (`http://shorturl.at/csBR5`)
Figure 2.5: AimsunNext traffic simulator

#### 2.2.2.4 Comparison

To conclude with this section, Table 2.1 summarizes and compares the main features of the three traffic simulators previously reviewed. As shown in this table, the simulator

tool that better fits the aim of this Project is Eclipse SUMO (together with its TraCI interface), as it offers all the required features while the others do not.

| | **Eclipse SUMO** | **AnyLogic** | **Aimsun** |
|---|---|---|---|
| Open source | ✓ | ✗ | ✓ |
| Traffic oriented | ✓ | ✗ | ✓ |
| Environmental information | ✓ | ✗ | ✓ |
| Multi-modal | ✓ | ✓ | ✓ |
| Customizable | ✓ | ✓ | ✓ |
| Free | ✓ | ✗(*) | ✗(*) |

Table 2.1: Simulator comparison.
(*) 30 free-trial.

# Chapter 3

# Analysis and Design

This chapter starts with a description of the main scenario that will be used to test the system behavior; then, it describes the system requirements analysis; and, finally, it details the system design, including the software architecture used as an starting point and the improvements made to adjust it to fit the Project requirements.

## 3.1 Scenario

The scenario itself can be defined based on three different parts, each one of them related to a given kind of traffic simulation information. These components are: (1) road network topology, (2) traffic time patterns, (3) traffic lights algorithms.

It is important to mention that the scenario that will be described below, is not the only one thought to be used in the framework, but as their complexity and the huge amount of time needed to implement it, there have been decided to use only this base scenario. All the possible scenarios defined are described below, based on topology and the algorithm that will use the different traffic lights.

The possible scenarios, based on the **topology**, are:

- Single junction with traffic flows from two different directions: from north to south, and vice versa; and from west to east, and vice versa; not allowing turns in the junction;

- Two adjacent junctions with the same traffic flow as the previous one, both allowing and not right and left turns in the junction; and
- A grid with N columns and M rows of interconnected junctions, both allowing and not right and left turns;

The possible scenarios, based on the different **traffic light algorithms**, are:

- Static time phases algorithm;
- Adaptive time phases algorithm;
- Actuated algorithm; and
- Button-type algorithm, very similar to the actuated one.

All the possible scenarios are defined based on the union of a given topology and a traffic light algorithm scenario, having a total of 20 different scenarios. These scenarios have been defined to be developed in the Ph.D. thesis as in this Master's thesis they could not be achieved due to time constraints. In fact, for this thesis, the only scenarios developed are those related to the static and adaptive time phase algorithms with a single junction in which it is not allowed to make turns.

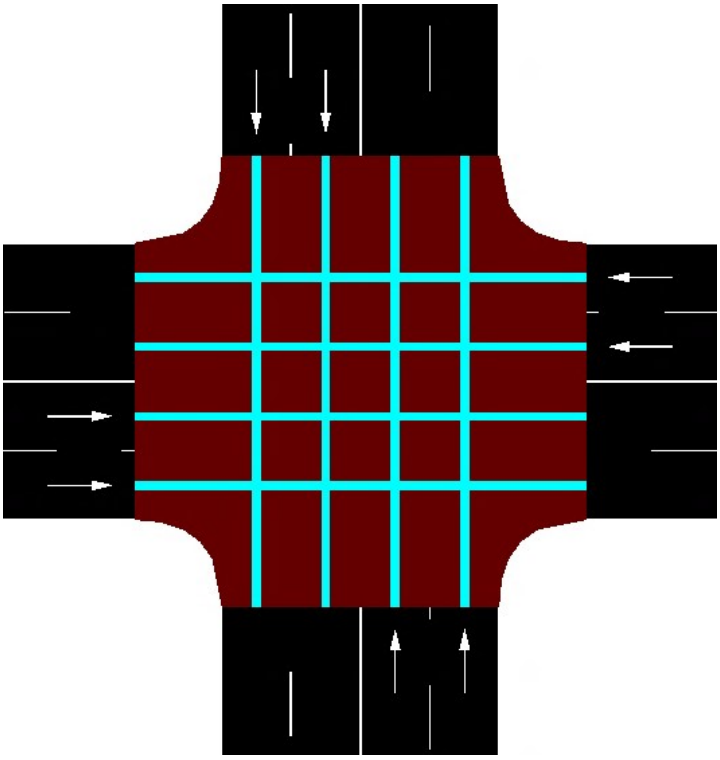
### 3.1.1 Road network topology

One of the main intended goals of the system is to adapt a single junction traffic light algorithm, in order to decrease the amount of waiting time based on the vehicle's flow on that junction.

Based on this statement, the defined network topology is composed by four main directions, each one of them with two lanes, related to north, east, south and west respectively, and a single junction connecting each one of the defined lanes, acting as a single center of the full topology.

It is important to mention that the mentioned junction has four different traffic lights, each one of them related to a direction, and the only one possible direction is

going forward, forbidding the turn directions as they are not relevant for this scenario because there is only one junction. The full topology of the road network is shown in Figure 3.1. Note that the blue color only indicates that the direction is valid.



*Source:* Documentation owner
Figure 3.1: Junction possible directions.

Besides, the traffic used in this scenario is right-handed because it is the most used traffic direction worldwide, around 72% [28]. The other option is to use the left-handed traffic which is also supported by Eclipse SUMO, allowing the two possible kinds of traffic used in real life.

### 3.1.2 Traffic types

In order to represent different kinds of flows related to the traffic, there are defined some flows, depending basically on the number of vehicles driving on the roads per hour. Besides, these flows are based on their direction thus having two main flows directions: from north to south (and vice versa) and from east to west (and vice versa).

The number of vehicles per hour and its general flow type are represented in Table 3.1. Besides, it is indicated the range of values that can represent each kind of flow, for example, the traffic type “Very Low” is represented from one vehicle to five vehicles per hour, as the default number of vehicles is three and the range is plus/minus two.

Note: the defined range is calculated using 30% of the number of vehicles except for the case of the “Very Low” type, which is used the 60%. These percentages have been selected to add some randomness and uncertainty to the number of vehicles used per each traffic flow, being aware of the number of vehicles do not match or are too close to the adjacent flows types.

| Type | Number of vehicles | Range |
|---|---|---|
| Very Low | 3 | ±2 |
| Low | 20 | ±6 |
| Medium | 150 | ±45 |
| High | 500 | ±150 |

Table 3.1: Traffic generic flows types with number of vehicles per hour and its range.

Based on these general flows, there are defined eleven different types of traffic flows, depicted in Table 3.2, that are based on the different available directions on the scenario and their combinations:

| Traffic Flow NS | Traffic Flow WE | Type |
|---|---|---|
| Very Low | Very Low | 0 |
| Very Low | Low | 1 |
| Low | Very Low | 2 |
| Low | Low | 3 |
| Low | Medium | 4 |
| Low | High | 5 |
| Medium | Low | 6 |
| Medium | Medium | 7 |
| Medium | High | 8 |
| High | Low | 9 |
| High | Med | 10 |
| High | High | 11 |

Table 3.2: Traffic flow types based on directions.

### 3.1.3 Traffic light algorithms

Based on the existing traffic lights algorithms, there are described different variations of those algorithms, that will be used to study which one of them fits better to the different traffic type flows, considering the average waiting time of all the passing vehicles per lane.

In Eclipse SUMO there are implemented different traffic lights algorithms such as static and actuated by default, allowing to modify its phase duration and other parameters such as the distance to the sensor in the actuated algorithm. For this scenario, there will be only one traffic algorithm used, the static one, as the actuated one does not work as expected in SUMO (for more information see Section 6.2).

Main variations of the traffic light algorithms are based on its parameters such as the green phase time, being aware of not violating the main rule of a real traffic light: the sum of the different phases should not be greater than 90 seconds. Although, there are countries, like the United Kingdom, where the maximum of the sum of phase duration is 120 seconds, which is only used when there are no pedestrian crosses on the road.

As mentioned previously, the only kind of traffic light algorithm that is going to be used is the static algorithm, where both yellow phases have a duration of 5 seconds per direction and both green phases have a variable duration in the range from 20 seconds to 70 seconds with a 5 seconds step, having a total of 11 different algorithms shown in Table 3.3.

This is the first approach defined to be used in the traffic light controller but, as it is explained below, it has also been defined a more realistic approach, where the green phase time is calculated with the proportion of vehicles passing in each direction.

Before showing the defined algorithms, there are shown in Table 3.4 the proportion values per each one of the traffic flow types. It is worth mentioning that the values selected per traffic type are: Very Low $\rightarrow$ 1, Low $\rightarrow$ 2, Medium $\rightarrow$ 4, High $\rightarrow$ 8. Note that these values were defined based on the knowledge of the scenario, but they can be easily modified if required.

| Name | Green phase duration NS (s) | Green phase duration WE (s) |
|---|---|---|
| static_program_1 | 20 | 70 |
| static_program_2 | 25 | 65 |
| static_program_3 | 30 | 60 |
| static_program_4 | 35 | 55 |
| static_program_5 | 40 | 50 |
| static_program_6 | 45 | 45 |
| static_program_7 | 50 | 40 |
| static_program_8 | 55 | 35 |
| static_program_9 | 60 | 30 |
| static_program_10 | 65 | 25 |
| static_program_11 | 70 | 20 |

Table 3.3: Traffic Light algorithm variations based on time interval.

| Traffic Flow NS (Value) | Traffic Flow EW (Value) | Proportion NS/EW |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 0.5 |
| 2 | 1 | 2 |
| 2 | 2 | 1 |
| 2 | 4 | 0.5 |
| 2 | 8 | 0.25 |
| 4 | 2 | 2 |
| 4 | 4 | 1 |
| 4 | 8 | 0.5 |
| 8 | 2 | 4 |
| 8 | 4 | 2 |
| 8 | 8 | 1 |

Table 3.4: Proportion by traffic flow type.

As it can be seen with these proportions, when the flow is the same in both directions, the value is equal to 1, while if there is more traffic on north-south than in east-west the value is greater than 1, and if it is less, the proportion is lower than 1.

Once the proportions have been shown, it is described the algorithm used to calculate the phase time of the east-west direction for the traffic light algorithm that will fit the best the current situation. With this value it is possible to calculate the phase time in the north-south direction by $80 - phasetime_{ew}$. This algorithm is shown

in Equation 3.1, where 80 is the maximum time available to use in a phase.

$$phasetime_{ew} = \frac{80}{\text{Proportion NS/EW} + 1}(\text{s}) \tag{3.1}$$

3.1: EW direction phase time equation.

By using this equation with all the possible values of traffic proportions, it results in 5 different traffic light algorithms with defined time phases in each direction. All these algorithms are shown in Table 3.5.

| Name | Green phase duration NS (s) | Green phase duration WE (s) |
|---|---|---|
| static_program_1 | 16 | 64 |
| static_program_2 | 26 | 54 |
| static_program_3 | 40 | 40 |
| static_program_4 | 53 | 27 |
| static_program_5 | 64 | 16 |

Table 3.5: Traffic Light algorithm variations based on vehicles proportion.

Note that in the tables there are only shown the green phases for both NS and WE directions, as they are the only ones that are modified.

## 3.2 Requirements analysis

This section describes the requirements analysis, based on the main functionalities of the framework and some concepts directly related to its performance. This analysis is performed according to the goals previously defined in Section 1.2.

The main functionality of the framework is to adapt the current traffic light phases based on the contextual information on each of the adjacent lanes, in order to reduce the amount of time the different vehicles wait in that junction.

To achieve this, there have been defined several approaches using adaptation techniques based on contextual and date information, such as updating the time of a given phase when the current traffic flow is higher in one direction rather than the other. This real-time contextual information will be used by two different components to

analyze and predict the current traffic flow and select which is the traffic light algorithm that fits the best to that traffic situation.

As in real life both analyzer and predictor could fail at any time, it has been decided that the traffic light adapter will not perform any adaptation until at least one of these two components are available. In the case that there is no contextual information and the previously mentioned components are not available, the adapter will make the traffic light act as a static traffic light where the green phase time for both directions is the same.

Some considerations about this framework are related to its quality, but there are other ones very critical that are required to make it work as expected and with no failures. Based on the quality, we have the fault-tolerance; reusability and flexibility of the framework; its ease of use and maintainability; and the design of a loosely coupled architecture that is, in turn, highly distributive. Based on the critical considerations, there are two: the response time of the analyzer and the predictor must be very low in order to be applied to the real-time adaptation, as otherwise, the information they provide would be useless; data integrity and availability is very important as if the real-time traffic information is not valid or available, the whole framework will not work.

## 3.3 Design

Before going into the details of the SmartTLC architecture, it is worth mentioning that it was not designed from scratch, but from a base architecture, previously designed and developed also by the author of this Project (as part of his Bachelor Thesis). The following subsection briefly reviews this base architecture. The extensions and modifications included in it to cope with this Project requirements, are also detailed below.

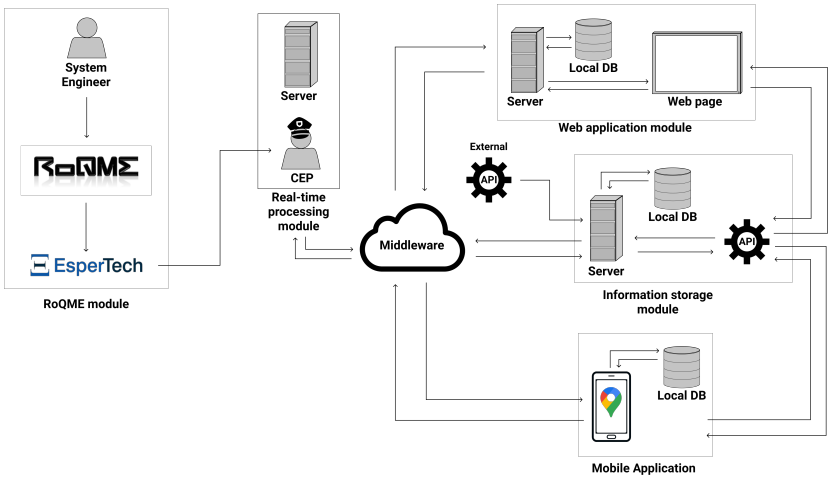
### 3.3.1 Base architecture

The base architecture on which SmartTLC has been built is outlined in Figure 3.2. It brings together different components all of which are briefly described next, even if some of them are not included in SmartTLC, as this will allow to better understand the base architecture and its behavior.

The main goal of that framework is the real-time monitoring of the society, in order to avoid crowds of people and hinder the expansion of the COVID-19 disease. Based on this goal, there are two sub-goals oriented to three different system actors, which has driven the framework's design and development. These are:

- Politicians and State security forces (such as the police) use a web application that allows them to real-time monitor the movement of the people in a given area; observing different movement patterns, crowded areas control and decision making based on this information.

- General population use a mobile application to observe the number of people around them, check their nearby allowed places by the authorities and receive relevant news related to the virus expansion.




*Source:* Documentation owner.
Figure 3.2: Bachelor Thesis architecture.

Once the base framework goals are defined and the architecture is shown, there are described its main components:

- **Real-time processing module**: it includes a complex-event processor (CEP) that is in charge of processing all the exchanged system information and detecting patterns on it.
- **Information storage module**: it represents the main server and stores all the relevant system information. This component retrieves that information from both an API and a middleware broker.
- **Web application module**: it is the web server that manages the relevant information for both politicians and security forces.
- **Mobile application module**: it is the mobile application that the general population will use.
- **Middleware**: this is the communication component used to exchange information between all the components of the architecture. This middleware is of the type publish-subscribe.
- **RoQME module**: used by the system engineer, it allows defining different data patterns that will be used by the CEP component.

### 3.3.2 Architecture evolution

As the main intended goal of the SmartTLC framework is related to the monitoring of a given system, it has been decided to use the previously described architecture as a basis, adding or removing new components that fit the defined goals. Moreover, some decisions have been made in order to define an easy-to-understand and loosely coupled architecture.

There are some components no longer required, and so on unused, that will be **removed** from the basic architecture. These components are (1) the mobile application,

(2) the real-time processing module, and (3) the RoQME module, used by the previous component.

Besides, some others will be **redesigned** as (1) the information storage module, where the API will be replaced by a metrics collector, (2) the database will be changed to fit better the data exchanged in the system; and (3) the web application, that will be used in this framework as a data visualization tool, removing its interactive role on the system. Furthermore, the contextual information will be provided by the Traffic Light Controller (TLC) component, instead of the mobile and web applications.

Once some of the components have been redesigned, there are some others that have been **added** to the base architecture:

- Recorder component for experiment storage.
- Player component for experiment replay.
- Traffic Light Predictor (TLP) component used for predicting the traffic type based on different kinds of information, such as date or traffic related to the junction context.
- Traffic Analyzer (TA) component to monitor the traffic flows and analyze them in order to determine the current traffic in the junction.
- Traffic Light Adapter (TLA) component that will adapt the traffic light algorithm, based on the traffic prediction and the traffic analysis, if available.

### 3.3.3 Additional considerations

At first sight, the proposed architecture could be thought to be divided into three different layers, each one of them based on its role in the system. These layers could be:

- **Producer layer**: composed of all those components that generate information that will be used along with the whole framework. This layer can also be named as context provider and its main component is the Traffic Light Controller, which

will be one of the most relevant components in the system, as it will be the one that will provide the contextual information. Besides, other components that could be fitted in this layer are the Traffic Light Predictor and the Traffic Analyzer.

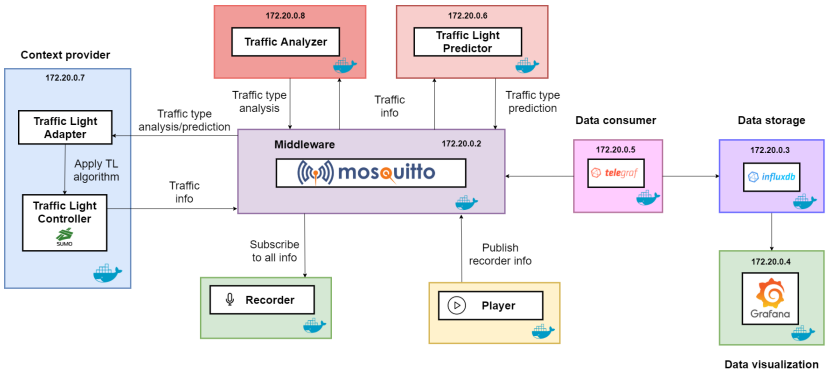
- **Middleware layer**: it is the main core of the framework as it is composed of the message information exchange component along with others, such as an experiment recorder and an experiment player.

- **Consumer layer**: this layer aims at storing and visualizing all the information exchanged into the system, in order to allow the system engineer to check the decisions taken and how these affect the traffic directly or indirectly. Its main components are a metric collector, database storage and a visualizing tool.

As the main context provider components (e.g., TLC, TA, TLA or TLP) also consume information from other components, their classification as ”Producers” or ”Consumers” is not trivial. Thus, for the sake of clarity, these layers have been removed.

### 3.3.4 SmartTLC architecture

Once all the evolution of the basic architecture has been depicted, it is described the final architecture used in the SmartTLC framework.




*Source:* Documentation owner.
Figure 3.3: Smart TLC architecture.

As shown in Figure 3.3, each architecture component, represented with a box, is marked with a Docker logo (), which means that this component is developed and deployed in the architecture as a Docker container. Besides, the IP addresses of each component (used by others components willing to establish a connection with them), is also displayed. All the architecture components are described in Section 3.3.5.

Note that not all the components have an IP address as some of them do not need it, due to the fact that they are only going to connect to the middleware in order to store or publish information. These components are the Recorder and the Player.

### 3.3.5 SmartTLC components

Once the architecture has been depicted, in this section there are explained the main components of the framework, its main purpose and behavior. These components are:

- **Traffic Light Controller** (TLC): this component is the traffic simulator that will monitor the contextual information of a given junction and will publish it into the middleware, in order to make it available for the rest of the framework's components. This component will use the traffic simulator Eclipse SUMO and the interface TraCI in order to access all the simulation information.

- **Traffic Analyzer** (TA): this component will obtain the contextual information from the middleware (previously published by the TLC) and will analyze the vehicle flows in each possible direction, based on a given temporal window, e.g. five minutes. This analyzed information will be published to the middleware in order to be used by the Traffic Light Adapter.

- **Traffic Light Predictor** (TLP): this component is composed of several machine learning models that will learn the behavior of the traffic flows based on two different approaches: (1) using contextual information such as the number of vehicles passing at a given moment and the date-time as the hour, day, month and year where is happening the simulation; and (2) using only information related to

the current date-time. Using the information provided by the TLC, these models will predict the current traffic flow and will publish it into the middleware.

- **Traffic Light Adapter** (TLA): this sub-component, inside the TLC component, is connected to the middleware in order to receive the traffic type predictions provided by the Traffic Light Predictor and the real-time traffic type analysis provided by the Traffic Analyzer. Based on this information and the conclusions obtained by the Traffic Light Study component (described below), the TLA component will update the traffic light algorithm to that algorithm that fits the best to the current traffic flows.

- **Middleware broker**: is the main component of the architecture as it is the one that allows the information exchange between all the others components. This middleware type is publish-subscribe as: (1) the components subscribe to the topics they are interested in, retrieving the information at the moment it is available on that topic; and (2) the components publishes its information into a specific topic, in order to provide valuable information to the others components. The broker used in this component is the Eclipse Mosquitto Middleware broker, using the MQTT protocol.

- **Recorder**: this component is in charge of storing all the information exchanged in the system, in order to reproduce, in the future, the different experiments previously done. It stores all information by subscribing to all the possible topics of the middleware broker, which allows it to obtain all the exchanged information. Although, it is possible to obtain only the information of a given component, by selecting its topic.

- **Player**: this component is used to reproduce the experiment stored by the Recorder component. It can replace some components that provide information to the system such as the Traffic Light Controller because it publishes the collected information into specific topics, representing a given component.

- **Data metrics consumer** (DMC): as all the information exchanged into the system is very important to outline its performance and to detect the motivation of the different adaptations done, there is a component related to gathering all those information and parsing it to a correct format in order to store it afterward. As the Recorder component, it receives all the information by subscribing to all the possible topics of the middleware broker. This component will use Telegraf technology, described below.

- **Data storage** (DS): once the information has been parsed, it will be stored on a database to persist over time. The additional main motivation for storing all the data is to visualize them and to see the real behavior of the system, which is done, in fact, by the Data Visualizer component. This component is directly connected with the DMC, as it is the one that parses all the system information and makes it valid to store in a database. In this case, the selected database is InfluxDB.

- **Data visualizer** (DV): last but not least is the data visualizer, which is in charge of providing tools for creating graphics and charts with all the information stored in the database. This component will be used by the systems engineers to check the behavior of the complete framework and to define new improvements, if necessary, based on this information. The visualizer tool defined is Grafana.

Note that the TLC and TLA are defined inside the same component, the context provider component, which is represented mainly by the TLC as it is the main one. This composition is motivated by the use of threads in Eclipse SUMO and TraCI, not allowing to access local simulation information from outside during the simulation.

There is also another component that is not shown in the architecture due to the fact that it is not connected with any other component. This component is the Traffic Light Study (TLS) and its main purpose is to perform different simulations using all the traffic flows defined along with all the possible traffic light programs, in order to check the waiting time per vehicle with these conditions. Using this information, it is

possible to conclude which is the best algorithm for each traffic flow, that in turn, is used by the Traffic Light Adapter.

Finally, it is important to mention that the main motivation for using Telegraf for metrics collection (instead of using any other technology), is that it is a part of the TICK stack [29], used mainly to monitor and alert about a system using Docker. This stack includes the following components:

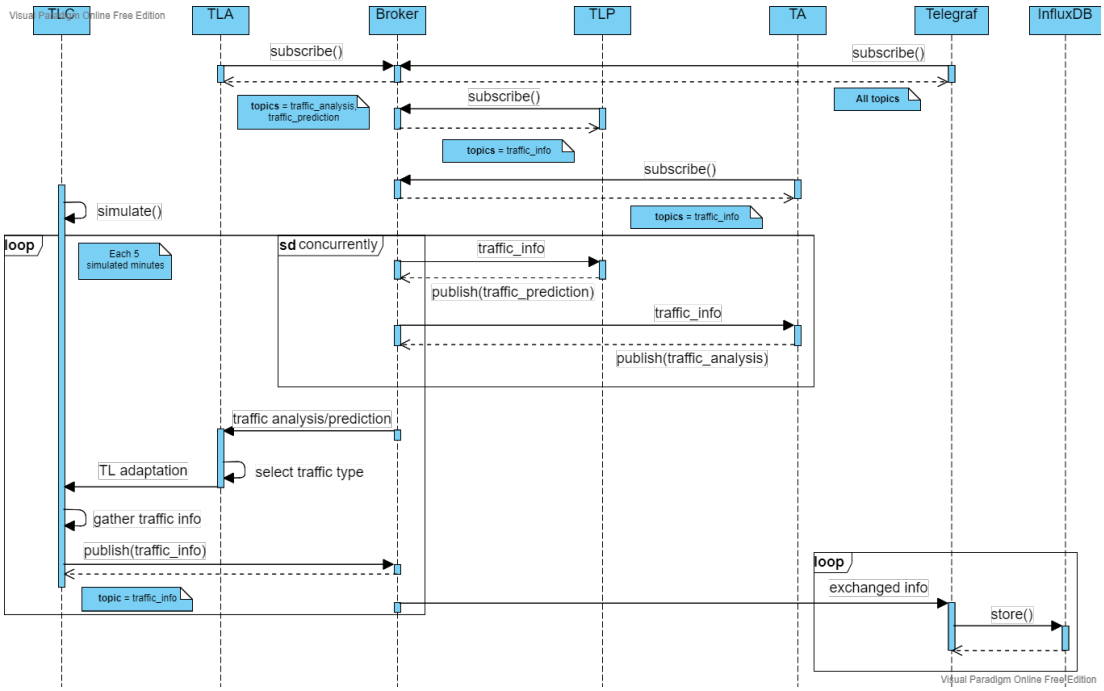
- **Telegraf**: it is a plugins collection that collects the system metrics and sends them into the InfluxDB database. It can monitor almost any kind of information from any source: Docker, scripts, services, hosts, middleware...
- **InfluxDB**: is a NoSQL time-series database oriented to metrics and real-time events.
- **Kapacitor**: is a tool for processing, monitoring and alerting real-time events. In this case, the Kapacitor is not used as there is no need for events alerting or even processing the stored data for this project, although it would be a good addition in future works.
- **Chronograph**: is a tool for visualizing and plotting the information storage on InfluxDB. It is not used as is more difficult to use than Grafana and has fewer features.

The use of the TICK stack (or at least of part of it) is a good option as the database selected for the SmartTLC framework is InfluxDB, which is at the core of this stack. Besides, the component of this stack are designed to work together in a very efficient way.

### 3.3.6 Sequence diagram

Following the component description, Figure 3.4 shows the SmartTLC sequence diagram including the main framework components and their actions in sequence. It is worth mentioning that the following diagram assumes that all the components are

deployed and working correctly (except the player and recorder, as they are not part of the framework core), so the deployment of all the components is not represented. Furthermore, the training process of the TLP models is not represented either, as it is an "off-line" process that is independently executed, always before deploying the framework. Please, note that all the subscription processes are done simultaneously but they are represented separately for the sake of clarity.




*Source:* Documentation owner.
Figure 3.4: Smart TLC full sequence diagram.

Next, the previous workflow is explained in detail:

1. All the component subscribes to its topics:
   - TLA: subscribes to topics: "traffic_analysis" and "traffic_prediction".
   - TLP and TA: subscribes to topics: "traffic_info".
   - Telegraf: subscribes to all topics: "#".
2. TLC component starts the traffic simulation based on a time pattern or a calendar date.

3. In a loop until the simulation ends:

    3.1 The TA retrieves the information from the TLP and TA components, published on the middleware broker.

    3.2 It selects the new traffic light algorithm that will use the TLC.

    3.3 TLC changes its current traffic light algorithm to the new one.

    3.4 Every 5 minutes, it publishes to the middleware broker ("traffic_info" topic) the retrieved information on that time window. This information is related to the number of vehicles that have passed the given junction and the accumulated waiting time of the vehicles per each one of the directions (North-South and West-East).

    3.5 Based on this published traffic information, both TA and TLP analyze and predict the current traffic type, and publish its value to "traffic_analysis" and "traffic_prediction", respectively.

4. While the simulation is running, the Telegraf component:

    4.1 When information is available on one topic, the component will retrieve it.

    4.2 The retrieved information will be parsed in order to be stored into the framework database.

    4.3 Parsed information is stored into the database.

It is also described the process followed by the Recorder and Player components, which are used in order to replace other components.

- Recorder:

    1. It is subscribed to all the possible topics of the middleware broker or only those selected by the system engineer, so when information is available on one topic, the component will retrieve it, just as Telegraf.

    2. Store the retrieved information in an external file, in order to replay it with the Player component.

- Player:

    1. The first step is to select which components it will “replace”, which means that the player will publish all the information stored related to that components.
    2. Once it is specified, the experiment will be loaded from the Recorder storage file and publish all those values related to the specified component, being aware of the timestamp of the published information and waiting if necessary, to behave exactly as the replaced components.

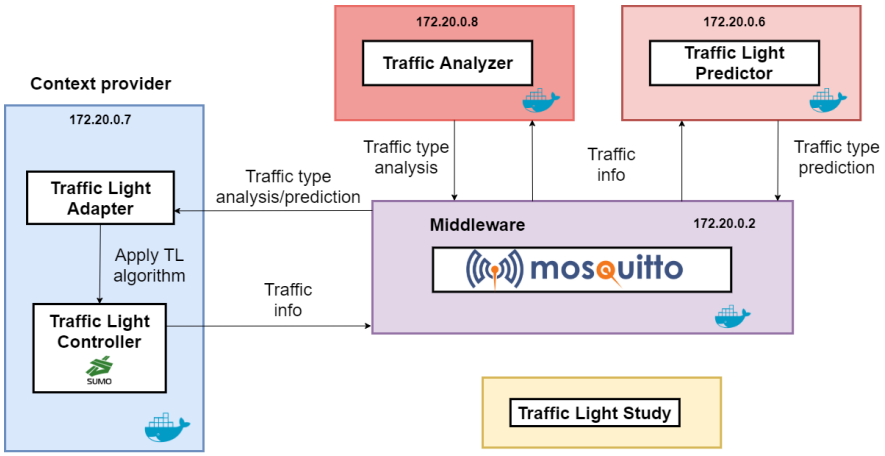
As the Traffic Light Study component is not connected to the middleware itself, the workflow followed is described below, but not shown in any figure as it is very easy.

1. Loop over all the possible traffic light algorithms and traffic type combinations.
2. Select the traffic light algorithm and traffic type used on the simulation.
3. Start the simulation process.
4. Gather traffic info as the waiting time and the number of passing vehicles per direction.
5. Restart the loop from 2.

# Chapter 4

# Implementation and Development

In this chapter, there are described deeply the main components developed for the framework, which are: (1) Traffic Light Study, (2) Traffic Light Controller, (3) Traffic Analyzer, (4) Traffic Light Predictor and (5) Traffic Light Adapter. These components and their relations are shown in the Figure 4.1 where it is also shown the middleware as it is the message broker of the framework. Lastly, it is described the deployment architecture strategy defined to be used in the framework, along with all the docker containers used in the framework. The full framework implementation is available on GitHub [30], its installation requirements and user manual on Appendix A and Appendix B, respectively.




*Source:* Documentation owner.
Figure 4.1: Smart TLC main components.

Note that the rest of the framework's components are not developed by me, as they are tools or technologies developed by third-party companies, so they are not relevant to be described in this section.

## 4.1 Traffic Light Study

The Traffic Light Study (TLS) component is used to perform different studies to select the best traffic light algorithms per each kind of traffic flow, based on the total amount of waiting time and the waiting time in each possible direction (north-south and east-west). All the possible traffic flow types and traffic light algorithms have been defined in Section 3.1.

As stated in Section 3.3.5, this component is independent of the architecture as it is not connected to any other component. Although both the study results and the conclusions obtained from it are the two most critical features of the system, as if it does not provide valid results or the conclusions are wrong, the relation between the traffic flow type and the algorithms will be wrong, propagating this error to the Traffic Light Adapter (the component that will use this kind of information) and therefore to the Traffic Light Controller, affecting to the real-time traffic adaptation.

In order to retrieve the traffic contextual information, it is performed an Eclipse SUMO simulation where the different kinds of traffic flows are generated and set into the scenario. It is important to mention that the methods provided by SUMO to calculate the waiting time of vehicles, does not work properly, so it has been developed an own function to calculate this value based on SUMO components, such as the induction detectors, that are activated when a vehicle passes by. This method is also used on other components as the Traffic Light Controller.

In this component are performed two different studies: (1) based on the first approach of time phases on the traffic light with intervals of 5 seconds, with a total number of 11 traffic light algorithms (see Table 3.3); and (2) based on the second approach that uses the proportion of traffic flows, to calculate the time phase of the traffic light, with a total number of 5 traffic light algorithms (see Table 3.5). In fact,

this second approach is the one used on the Traffic Light Adapter component to select the traffic light algorithm that better fits based on the current traffic proportion.

Once all the simulations and studies have been performed, the results are stored in an external file, in order to be processed and visualized by the system engineer, making it easier to acquire this information and draw its own conclusions.

Additionally, it is developed a visualization tool for loading the previously obtained results and plotting them. This tool is a command-line interface that allows creating different kinds of plots: (1) Box, (2) Scatter, (3) Pair, (4) Bar and (5) Line plot.

Lastly, it is worth mentioning that the results obtained from these studies are shown in Section 5.2 as it is where are all the results are gathered and discussed. These results are obtained using the visualizer tool and retrieving the best algorithms per traffic flow type.

## 4.2 Traffic Light Controller

The Traffic Light Controller (TLC) is the main component of the framework as it is the one that will simulate the traffic based on different time patterns (explained in Section 4.3) or even a range of days in a calendar. Besides, TLC is connected to a component called the Traffic Light Adapter (that is described in Section 4.5) from where it will receive the traffic light algorithm that it must update in real-time in order to adapt to the current traffic type flow, only if this component is available. In case the adapter is not available, the controller will use the same traffic light algorithm as in real life, a static traffic light with predefined green time phases, equally distributed for both directions, 40 seconds per each one, while the amber phases remain on 5 seconds.

The simulation process is the following one:

1. Start the simulation with the configuration files.

2. Update the traffic light algorithm by the adapter decision if it is available and there is information about traffic prediction, traffic analysis or both.

3. Retrieve simulation contextual information as the waiting time, current hour, day, month and year; in a temporal window of five minutes.

4. Publish this information into the middleware on the topic “traffic_info”.

5. Repeat from 2 until the simulation is done.

As it can be seen, the simulation process is very easy and simple due to the fact that the adaptation logic, the most difficult feature, is carried out by the Traffic Light Adapter component. In fact, this component only retrieves information from the traffic simulation, publishes them into the middleware and updates the traffic light algorithm when the TLA indicates.

The simulation ends when there are no more vehicles to add or the time to simulate has ended. Besides, it is worth mentioning that the defined temporal window can be easily modified to publish the traffic information at that given time range.

The followed data model, in order to publish the traffic information, is:

- **tl_id**: junction identifier that in this case, as there is only one junction, will have always the same value “c1”.

- **tl_program**: traffic light algorithm used in the simulation.

- **passing_veh_n_s**: integer number representing the number of vehicles that have passed from both north and south and on the two lanes per each one of them. It must be greater than 0.

- **passing_veh_e_w**: integer number representing the number of vehicles that have passed from both east and west and on the two lances per each one of them. It must be greater than 0.

- **waiting_time_veh_n_s**: integer number representing the amount of time the vehicles that have passed from both north and south and on the two lanes have been waiting in total. It must be greater than 0.

- **waiting_time_veh_e_w**: integer number representing the amount of time the vehicles that have passed from both east and west and on the two lanes per each one of them. It must be greater than 0.

- **hour**: it represents the current hour of the simulation. It only allows values that are half hours due to the fact that it is the time range defined in the different time patterns, e.g. the hour 10:05 will be published as 10:00. This fact does not affect the storage of the information in the database, as it is used a temporal database that stores the timestamp when the information has been stored, and this value is the used one instead of the "hour" to represent the time evolution in the simulation.

- **day**: it represents the name of the day it is currently on the simulation. Note that one day in the simulator is represented with a total of 86.400 timesteps, and its value is retrieved from Madrid's city calendar (described in Section 4.3). All the possible values are Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday. By default is Monday.

- **date_day**: integer number representing the day, where the minimum and maximum possible values are 1 and 31, respectively. Note that in some cases, as in February the maximum is 28 instead. By default is 2.

- **date_month**: integer number representing the month, where the minimum and maximum possible values are 1 and 12, respectively. By default is 2.

- **date_year**: integer number representing the year. In this case, the default value is "2021" as it is the year of the calendar used and the SmartTLC framework development.

## 4.3 Traffic Light Predictor

The Traffic Light Predictor component follows a basic machine learning training process starting from dataset generation (in this case as there is no available data that

fits the aims of the component), its clean-up, the model training process itself and the hyper-parameters tuning. Besides, this section it is also described the models' selection policy that will be used to predict the traffic type when there is more than one selected predictor.

### 4.3.1 Dataset generation

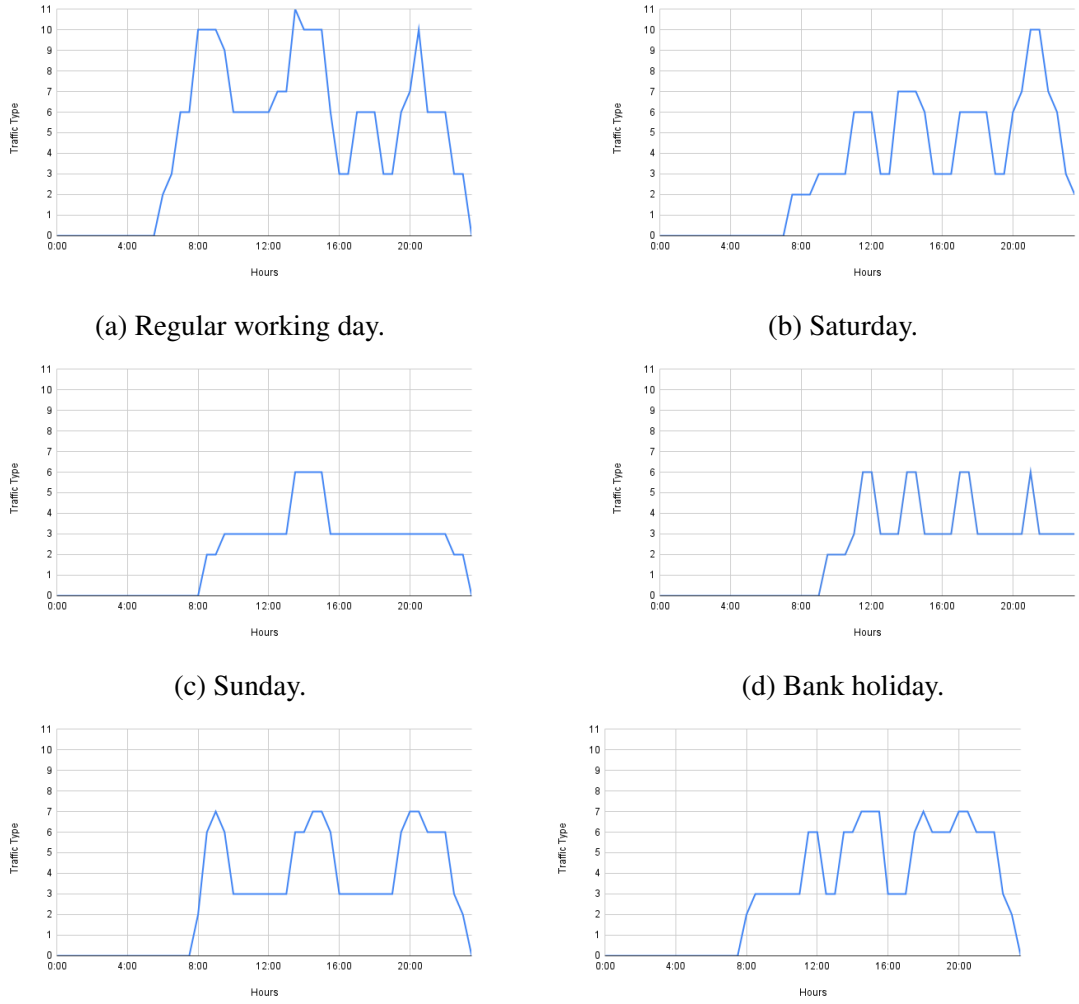
This step is related to the dataset generation with information associated with the accumulated number of vehicles per direction (NS and WE) in temporal windows of 30 minutes (that are 1800 time-steps in Eclipse SUMO). This dataset will be generated based on different traffic time patterns defined below and Madrid's city year calendar [31] (only 2021 year) with both working and bank holidays (defined as "festive" on the implementation). Moreover, there is defined a noise policy in order to represent the real-life randomness of the traffic as it never acts the same way.

#### 4.3.1.1 Traffic time patterns

In this section, there are described the main traffic time patterns defined to represent the variety of traffic flows, based on different week-days and situations such as bank holidays, summer or winter vacations days. All these time patterns are represented in Figure 4.2, and they will be used to generate the year dataset previously mentioned.

In order to ease the visualization of these patterns, all the working days have been represented by a single time pattern, as all of them follow the same pattern but with little variations that are not relevant enough to show the five kinds of working day time patterns.

As it can be seen, the working days have three different peak hours such as around 8:00, 14:00 and 20:00. The weekend follows a very different pattern as people are used to enjoying those days with family and not using a lot their vehicles. Note that in vacation and bank holidays, the patterns followed are very different, fitting as much as possible to real life. Furthermore, the condition to specify if a day is a vacation or not is based on the next clauses, where the year 2021 is selected as it the year used from




(a) Regular working day.

(b) Saturday.

(c) Sunday.

(d) Bank holiday.

(e) Summer vacation day.

(f) Winter vacation day.

*Source:* Documentation owner.
Figure 4.2: Time patterns.

the calendar to generate the dataset:

- If the day is between 20-7-2021 and 10-9-2021, it is a summer vacation day.
- If the day is between 21-12-2020 and 7-1-2021 or 21-12-2021 and 7-1-2022, it is a winter vacation day.
- Otherwise is a non-vacation day, meaning that it will follow the other types of time patterns.

It is important to mention that these traffic patterns have been hand-made defined

and are based only on the subjective perception of traffic flow. Despite this fact, these patterns are enough to train our models and to allow the system to adapt in terms of this traffic flow. Moreover, peak hours have been selected based on the Spanish society and its lifestyle, that is where the system has been designed and developed.

In fact, the definition of the patterns has been changed throughout the development of the framework due to its lack of reality, as in the first definitions, the traffic flow types perform sharp changes, e. g. passing from a "low-low" type to "high-high"; which can happen in real-life but not as usual as defined at first. This modification is explained in Section 6.2 and it has led to the development of a random noise policy to represent the uncertainty of the real traffic, which is explained below.

#### 4.3.1.2 Noise addition

Once the different traffic time patterns have been depicted, it is time to define a noise policy in order to provide randomness and uncertainty to the dataset generation, making it more realistic, as in real life there are some situations that can not be predefined due to the high traffic's complexity.

There are two kinds of noise policies, based on the type of modification it performs on the dataset:

- **Day replacement**: there are main considerations:
  - If the date is not a vacation day, meaning that are only considered working days and bank holidays:
    - Once every 50 days, replace the date with a summer or winter vacation day.
    - Once each 14 working days, swap it with another random working day. This swapping process does not affect bank holidays.
  - It the date is a vacation day, there are two considerations:
    - Once every fifteen days, swap the vacation day to a working day.

    - If the day is summer let it as predefined or swap it to a winter vacation day, once per every fifteen days.
    - If the day is winter let it as predefined or swap it to a summer vacation day, once per every fifteen days.

- **Traffic type replacement**: once every three days, modify a traffic type flow of a given slice (represented by a half four), by adding or subtracting a random value with a range of plus or minus two. In this case, the reason for the selected value is that, with a difference of two in traffic type, there is no unrealistic flow change, for example going from "verylow-verylow" to "high-high" in only a half-hour.

These noise policies are not excluded between them as they can be applied simultaneously on the same time pattern.

Note: this noise addition is performed on the process of parsing the Madrid calendar to a valid format that fits into Eclipse SUMO, which outputs the dataset with the simulation information.

#### 4.3.1.3 Data model

Once all the generation processes have been depicted, it is described the main data model used on the dataset. This data model contains the next fields, along with their meaning:

- **sim_id**: SUMO simulation identifier. It is stored as a key of the different rows of information but it is not used at all.

- **tl_id**: junction identifier that in this case, as there is only one junction, will have always the same value "c1". As the previous one, this field is not used but it is defined for future scenarios where there will be more than one junction.

- **tl_program**: traffic light algorithm used in the simulation. This field is always "static" as it is used in real life.

- **traffic_type**: traffic flow type representing all its possible values, from 0 to 11. This field is the one that will be predicted by the TLP component based on different kinds of information as is explained in the next section.

- **passing_veh_n_s**: integer number representing the number of vehicles that have passed from both north and south and on the two lanes per each one of them. It must be greater than 0.

- **passing_veh_e_w**: integer number representing the number of vehicles that have passed from both east and west and on the two lanes per each one of them. It must be greater than 0.

- **hour**: it represents the current hour of the simulation. This information is stored by default each half-hour, which in the simulator is a total of 1.800 timesteps.

- **day**: it represents the name of the day it is currently on the simulation. Note that one day in the simulator is represented with a total of 86.400 timesteps, and its value is retrieved from Madrid's calendar. All the possible values are Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday.

- **date_day**: integer number representing the day, where the minimum and maximum possible values are 1 and 31, respectively. Note that there are months as February where the maximum value is 28.

- **date_month**: integer number representing the month, where the minimum and maximum possible values are 1 and 12, respectively.

- **date_year**: integer number representing the year. In this case, the only possible value is 2021 as it is the simulated year.

It is worth mentioning that the results of the dataset generation are described and discussed in Section 5.2.2.

### 4.3.2 Model training process

Once the full dataset has been described and generated, it is explained the training process followed by the TLP component. Before starting this process, remind that there are two kinds of predictors: (1) based on date information and (2) based on contextual traffic and date information.

Before going deeper into the training process, it is important to mention that the selected machine learning algorithms that are going to be used on the predictor and the reason why they are selected, are described deeply in Appendix C. These algorithms are: (1) Naive Gaussian Bayes; (2) Support Vector Machines both linear and polynomial; (3) K-Nearest Neighbors; (4) Decision Trees; and (5) Random Forest. Besides, the training process is performed offline, previously to the real-time simulation.

Once the models have been selected, the training process can be divided into different steps, related to the process followed on any kind of data science project:

1. **Dataset generation**: this step has been described in the previous section.

2. **Data cleaning**: as the dataset is created by myself, there is no need to clean the dataset. This process would be relevant if the dataset is retrieved from any other source and it would contain invalid or empty values. Although it is not required, it has been done to check that the dataset is well constructed.

3. **Data exploration**: as in the previous step, the dataset has been created based on the requirements of the system and the exploration is no longer needed.

4. **Feature engineering**: this step is relevant as there will be two kinds of predictors that will be trained with different features. In this case, the selected features, described in the previous section, are:

    - For the predictor based on date information: hour, date, date_day, date_month and date_year;

- For the predictor based on both date and contextual information: previous features, passing_veh_n_s and passing_veh_e_w.

5. **Classifier modelling**: once the two kinds of predictors are described and selected its training features, it is described the training process where the dataset split followed is a K-fold process, explained in the appendix previously mentioned. In short, this process consists on splitting the dataset into different folds which will be used to train models with each one of them. For our scenario, the selected number of folds is 2 as the dataset is not very big.

6. **Model evaluation**: lastly, all the trained models will be evaluated in terms of its performance, based mainly on the F1 score (explained in the appendix previously linked). With this evaluation, the predictor component will only select those models that are the best ones to perform the traffic type predictions. The number of selected best models is indicated by parameter.

Besides, with some of these algorithms (K-Nearest Neighbors, Decision Trees and Random Forest), it has been done hyperparameter tuning, a technique that modifies the hyperparameters of a given model in order to retrieve the values that makes the model the one with the highest score of all. The values selected for the tuning are:

- **K-Nearest Neighbors**: the maximum number of neighbors is 15, starting at one with a step of 1.

- **Decision Trees**: the maximum depth of the decision tree is 20, starting at two with a step of two.

- **Random Forest**: the number of estimators (decision trees) is 20, starting at one with a step of 1; and the maximum depth of each one of these estimators is 10, starting at two and with a step of two.

Once the training process has been done and we have all the information related to the performance and the elapsed time on this process of each one of the models, we

can compare the results of these models. This results discussion will be done on the Section 5.2.3.

### 4.3.3 Model predictor selection

Lastly, there is explained the prediction process as it is possible to select one or more models to predict the given current traffic flow type. Based on the number of selected models, that are the ones with the best performance, there are several considerations:

- With a number of models equal to 1, the predicted value is the one that will be published.
- With a number of models greater than 1, there are different approaches:
  - Sum all the predicted types and divide by the number of models. This approach gives the same relevance to the models.
  - Weighted sum of all the predicted types and select the one with the best value, this means the predicted type multiplied by the model performance and select the one with the highest value. This approach gives more relevance to those models with the best performance.
  - Select the predicted type that selects the majority of the models, based on the number of appearances. This approach gives the same relevance to the models.

Note: all the approaches defined for more than one model have not been implemented yet as they are not going to be used in the example scenarios, but there are scheduled and defined in order to be implemented in the future.

Once the traffic flow type has been predicted by whether using a single model or a set of models, it is published into the middleware on the topic "traffic_prediction" with the next data model schema:

- **tl_id**: traffic light identifier. In this case, it will only be "c1" as there is only one traffic light in the scenario.

- **traffic_prediction**: traffic flow type prediction previously defined. It represents only one type of traffic flow. The possible values are from 0 to 11.

## 4.4 Traffic Analyzer

The main purpose of this component is to analyze the number of passing vehicles in each possible direction and to select which is the traffic type that fits the best to the current situation. This contextual information will be retrieved from the middleware, where the TLC has been published it in every five-minute temporal window (described in Section 4.2), and it will publish the traffic type analysis in order to be used by the Traffic Adapter.

In order to analyze the traffic flows, there have been defined the number of vehicles passing by each direction, which will be used to calculate the upper and lower bounds of the different traffic types. These bounds are calculated by:

- **Lower bound**: it is defined to be the upper bound of the previous kind of flow, except for the "very low" flow that is 0 as it is the lowest one.

- **Upper bound**: it is calculated by adding the number of vehicles per hour together with the maximum possible value in the range and dividing this sum by three. This factor has been selected as it is the one that best adjusts to the five minutes previously defined temporal window.

Based on these bounds and the number of passing vehicles on each direction, the different traffic types have been depicted, shown in the Table 4.1:

It is worth mentioning that the Traffic Analyzer does not require any kind of training process as it only processes the real-time contextual information and outputs the analyzed traffic type. In fact, this component does not require either the Eclipse SUMO simulator or the TraCI interface, as it can be considered as a single script that subscribes to a given topic from where receives the contextual information ("traffic_info"), processes that information based on a previously defined

| | Vehicles N-S (Bounds) | | Vehicles E-W (Bounds) | |
|---|---|---|---|---|
| **Traffic Flow Type** | **Lower** | **Upper** | **Lower** | **Upper** |
| 0 | 0 | 2 | 0 | 2 |
| 1 | 0 | 2 | 2 | 9 |
| 2 | 2 | 9 | 0 | 2 |
| 3 | 2 | 9 | 2 | 9 |
| 4 | 2 | 9 | 9 | 65 |
| 5 | 2 | 9 | 65 | 217 |
| 6 | 9 | 65 | 2 | 9 |
| 7 | 9 | 65 | 9 | 65 |
| 8 | 9 | 65 | 65 | 217 |
| 9 | 65 | 217 | 2 | 9 |
| 10 | 65 | 217 | 9 | 65 |
| 11 | 65 | 217 | 65 | 217 |

Table 4.1: Traffic analyzer vehicle bounds per traffic type.

algorithm and published its traffic type output into a specific topic of the middleware ("traffic_analysis").

Once the traffic flow type has been analyzed, it is published into the middleware on the topic "traffic_analysis" with the next data model schema:

- **tl_id**: traffic light identifier. In this case, it will only be "c1" as there is only one traffic light in the scenario.
- **traffic_analysis**: traffic flow type analysis previously defined. It represents only one type of traffic flow. The possible values are from 0 to 11.

## 4.5 Traffic Light Adapter

The Traffic Light Adapter is a component that is always deployed as it belongs to the Traffic Light Controller, which in fact, is required to be deployed to execute any kind

of simulation as it is the component that does the simulation itself. Furthermore, it is connected to the middleware, from where it subscribes to the specific topics of both the analyzer and the predictor ("traffic_analysis" and "traffic_prediction", respectively) in order to retrieve the information about the current traffic type.

As stated previously, both components can be deployed at the same time, only one of them or neither of them. This behavior makes the TLA must be aware of the information it receives, which in fact indicates if the components are deployed or not.

Based on this information, the component will select a traffic light algorithm that fits the best to the current traffic situation, which has been selected from the second study of the Traffic Light Study component. Moreover, there are some considerations to be aware of when adapting the traffic light algorithm:

- If there is only one traffic type related component (TA or TLP) deployed, the traffic type used to select the best traffic light algorithm will be the one retrieved by that component as the other one is down and does not provide any kind of information.

- If there is information about the two components:

  - If the prediction and the analyzed traffic type are very close, within a minimum predefined "distance", the analyzed traffic type will be the one that will be used to adapt the traffic light algorithm. By default, the value of this "distance" is three, as it represents the maximum traffic type change without being unrealistic.

  - Otherwise, the value used to adapt the traffic light is calculated by the third part of the distance between the analyzed and the predicted traffic types, getting closer to the analyzer value.

With these considerations, we assure that the real-time information is more valuable than the historical one, due to the fact that the last one would not perceive instant traffic variations while the real-time will.

As it can be seen, the behavior of this component is very simple to describe but it does not mean that it is not critical. In fact, it is one of the most relevant components of the full architecture, as it is the one that will make the full system retroactive and adaptive to the current traffic situation.

## 4.6 Component deployment

The next step, in order to perform several examples to test the framework performance, is to define the strategy that will be followed to deploy it fully or only some of its components.

As stated in Section 3.3, all the components of the framework (except the Traffic Light Study) are created within a Docker container, in order to be easily deployed lately.

On the one hand, some of these components are based directly on images provided by the Docker Hub community such as: “eclipse-mosquitto (middleware)”, “influxdb” (data storage), “grafana” (data visualization), “telegraf” (data consumer) y “python” (traffic analyzer).

On the other hand, those components related directly to the traffic simulation or the traffic type prediction it will use a hand-made defined Eclipse SUMO container, as there are no existing containers with the technologies required for this kind of simulation. This container is based on the Ubuntu 18.04 image and there are installed all the required libraries such as python, Eclipse SUMO and its tools (as TraCI); and configured all the dependencies and environment variables in order to work as expected. Besides it is used a special tool, called “Dockerize” [32], which allows the container to wait a given period of time or until another container has been deployed, which will be used to wait until other framework’s components are fully deployed.

### 4.6.1 Deployment strategy

The "docker-compose" tool is the one used for the deployment strategy, in order to deploy concurrently the required components as all of them are implemented as Docker containers. To fit the framework purpose and architecture, it has been developed a docker-compose file generator which takes as input the name of the different required containers (see below) that will be added to that deployment file, separated by comma.

In order to add both selected containers and their dependencies (if needed), the generator creates a dependencies matrix where it is stored if a container was previously created and the dependencies that need to be created in order to work as expected. In this generator there are also considered fields as image or folder for building the image, container name, restart policy, volumes associated and the network IP direction. Besides, some containers could have additional fields as the port, environment files, container users, links and dependencies to other containers, entry-points and commands to execute on the container when deployed.

Once all the containers and their dependencies have been added to the deployment file, it is added the network configuration used by all the components. Note that the components' IP addresses must be in the range of the defined network. Lastly, it is worth mentioning that the deployment of each one of the components themselves and of the full framework is described in the Appendix B.

### 4.6.2 Container list

On the previously mentioned script, there are stored all the components that will be used on the framework and that can be used on the generator. These components are:

- **mosquitto**: it is the middleware broker and it is based on the "eclipse-mosquitto" image from Docker Hub. Its port is the default one for the MQTT protocol (1883) and its IP address is 172.20.0.2.

- **influxdb**: it is the InfluxDB database and it is based on the "influxdb" image from Docker Hub. Its port is the default one (8086) and its IP address is

172.20.0.3.

- **grafana**: it is the visualizer tool (Grafana) and it is based on the "grafana" image from Docker Hub. Its port is the default one (3000) and its IP address is 172.20.0.4. Besides, it is linked to the "influxdb" container.

- **telegraf**: it is the data metrics consumer (Telegraf) and it is based on the "telegraf" image from Docker Hub. It does not have any available port and its IP address is 172.20.0.5. Besides, it is linked to and depends on the "influxdb" container.

- **traffic_light_predictor_date**: it is based on the own Eclipse SUMO created image and it represents the TLP component trained with date information. It is linked to the "mosquitto" container, as it is the middleware. Its IP address is 172.20.0.6 and its execution command is related to the required libraries installation, the waiting process until the database is enabled and the execution of the predictor with a single model.

- **traffic_light_predictor_context**: it is also based on the own Eclipse SUMO created image and it represents the TLP component trained with both contextual and date information. All the other fields are the same ones as the previous container, even the same IP as they are not going to be deployed at the same time.

- **traffic_light_controller**: it is also based on the own created image of Eclipse SUMO and it represents the TLC component. It is linked to the "mosquitto" container, as it is the middleware. Its IP address is 172.20.0.7 and its execution command is related to the required libraries installation, the waiting process until the database is enabled and the execution of the simulation from the creation of the configuration files to the selection of the time pattern or date interval.

- **traffic_analyzer**: it is the TA component and it is based on the "python" image from Docker hub. It is linked to the "mosquitto" container, as it is the

middleware. Its IP address is 172.20.0.8 and its execution command is related to the required libraries installation and the start of the analyzer script.

- **recorder**: it is the recorder component and it is based on the "python" image from Docker Hub, as the previous one. It has neither an available port nor an IP address, and it is linked to the "mosquitto" container. Its execution command is to start the recording process.

- **player**: it is the player component and it is also based on the "python" image from Docker Hub, as the previous one. It has neither an available port nor an IP address and it is linked to the "mosquitto" container. Its execution command is to start the replay process.

Note that not all the fields are represented in the previous list, only the most relevant ones.

# Chapter 5

# Result Presentation and Discussion

In this section, there are described the main examples that are going to be executed to test the framework performance and the adaptation behavior. Besides, there are stated the results obtained on these examples and the SmartTLC components, e.g. the Traffic Light Predictor; and discussed, to check if the system acts as expected. These two concepts (results and discussion) will be explained together in order to clarify the read.

## 5.1 Simulation examples

In this section, there are described the main examples that have been developed in order to test the behavior of the full framework by enabling or disabling some of the components and checking their performance. There have been described several different examples, each one of them is explained below.

Note that all the described examples will be performed with a given time pattern of a working day (e.g. Monday), with a range of days in the calendar (e.g. from 18/07/2021 until 20/07/2021, on summer vacation days: Sunday, Monday and Tuesday) and with a random time pattern generated by hand. Besides, the number of models used on the predictor will be one, in order to ease the system behavior.

First of all, it is defined the **"No adaptation"** approach where there are deployed the next components: "mosquitto", "influxdb", "grafana", "telegraf" and

"traffic_light_controller". This example consists of performing the simulation of the different traffic flows on a given time pattern or range of time patterns, using a static traffic light algorithm in which the green phase duration of each direction is the same, 40 seconds. It can be thought to be the base example as there is no real-time adaptation and the behavior of the traffic light is the same as in real life.

Secondly it is defined the **"Only predictions based on date adaptation"** approach where there are deployed the next components: "mosquitto", "influxdb", "grafana", "telegraf", "traffic_light_controller" and "traffic_light_predictor_date". This example is an improvement of the base example, the previous one, where the predictor based on the date information is enabled. This component will receive the date where the simulation is being performed and predict the traffic type based on it, which in turn is used by the Traffic Adapter (inside the Traffic Light Controller) for adapting the traffic light algorithm. This example can be considered a slight adaptive approach as it does not use contextual information related to the traffic flows, it only uses the historical date information, which in turn can not be as representative as the contextual due to the randomness of the traffic.

In third place it is defined the **"Only real-time analyzer adaptation"** approach where there are deployed the next components: "mosquitto", "influxdb", "grafana", "telegraf", "traffic_light_controller" and "traffic_analyzer". This example is also an improvement of the base example where the real-time analyzer is enabled. As stated previously, this component will analyze the traffic flow in real time and suggest the related traffic type, that will be used by the Traffic Light Adapter for adjusting the traffic light algorithm that fits the best. In this case, the example can be considered the higher adaptive approach than the previous one as it uses contextual information rather than historical date information.

Lastly it is defined the **"Both analyzer and contextual predictor adaptation"** approach where there are deployed the next components: "mosquitto", "influxdb", "grafana", "telegraf", "traffic_light_controller", "traffic_analyzer" and "traffic_light_predictor_context". This example is the last improvement performed

to the base example and it consists of the use of both real-time traffic analyzer and context-based predictor to retrieve the analyzed and predicted traffic types, respectively. With this information, the adapter will estimate the "real" traffic type by providing more relevance to the analyzer rather than the predictor as it is more aware of the current situation, which in turn will be used to select the best traffic light algorithm. This example can be considered the most advanced adaptive approach as it uses the real-time analyzer and both contextual and date information on the predictor.

It is worth mentioning that another possible example could be the use of only the contextual predictor, but as for this component is required the traffic contextual info and the analyzer also uses this kind of information, this example has been discarded. Besides, we assume that the number of examples defined is enough to check the improvements on the traffic flows and the waiting time of the vehicles, by using several adaption approaches.

## 5.2 Results and discussion

Once all the system components and examples have been described deeply, we are going to show their results and discuss them. In this section, there are shown the following topics: (1) Traffic Light Study component studies, (2) Traffic Light Predictor dataset generation, (3) Traffic Light Predictor model training performances and (4) simulation examples results.

### 5.2.1 TLS component studies

Remember that in the TLS component there have been defined two different studies/approaches: (1) create eleven different static algorithms, where the difference between them is the duration of the green phases on each direction, starting at a minimum of 20 and a maximum of 70 seconds, by adding/subtracting slices of 5 seconds, respectively; and (2) create five different static algorithms where the phase time is calculated using the proportion of traffic flows.

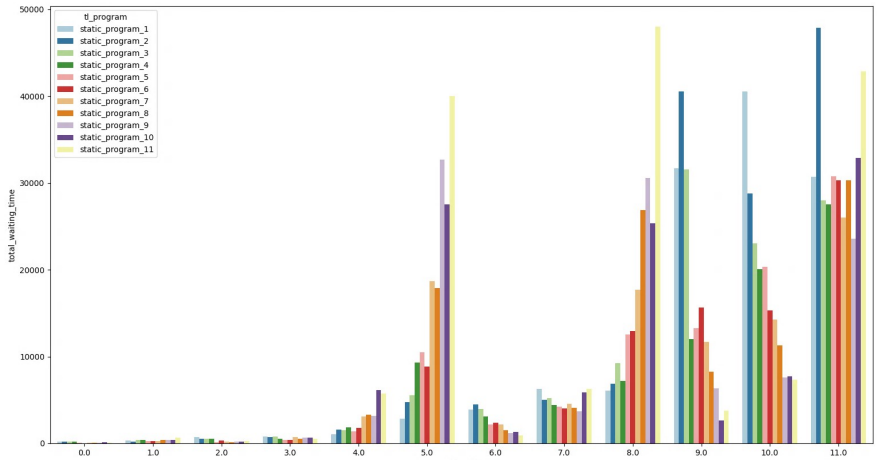
The study itself performs a simulation per each traffic light algorithm and each traffic flow type, resulting in a slow process that is not recommended to be executed several times as the results will not be very different on each execution. Note that both studies can be executed directly to obtain the results with the scripts available on the folder "study", for more information see Appendix B. These scripts generate the output files that will be used to represent the data on different plots, from where the conclusions will be obtained.

In the **first study** the traffic light algorithm phases will use an interval of five seconds, resulting in a total of 11 different algorithms (shown in Table 3.3). Its results can be visualized in Figure 5.1 where there are represented the total waiting time per algorithm and traffic flow type, but as the scale on the traffic flow types is too different, it is also obtained the best traffic light algorithms per traffic flow type in textual mode (only the best one), that are shown in Table 5.1. Lastly, it is worth mentioning that the time elapsed on generating the simulation results is around **13 minutes and 30 seconds**.

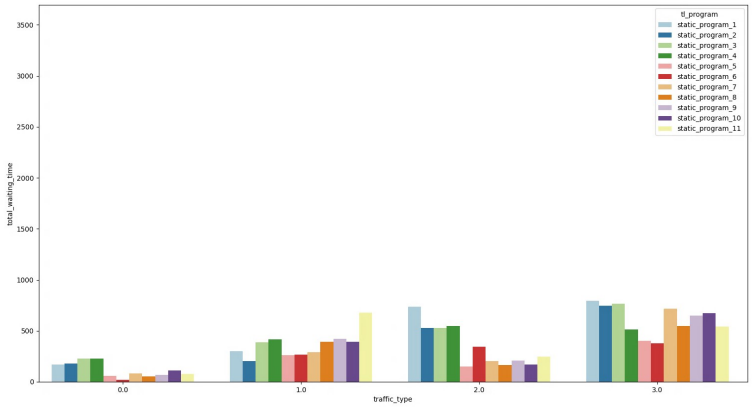
| Traffic Type | Best TL algorithm |
|---|---|
| 0 | static_program_6 |
| 1 | static_program_2 |
| 2 | static_program_10 |
| 3 | static_program_6 |
| 4 | static_program_1 |
| 5 | static_program_1 |
| 6 | static_program_11 |
| 7 | static_program_9 |
| 8 | static_program_1 |
| 9 | static_program_10 |
| 10 | static_program_11 |
| 11 | static_program_9 |

Table 5.1: Best traffic light algorithm (5 seconds interval) per traffic type.

In the **second study** the traffic light algorithm phases will use the traffic flows proportion, resulting in a total of 5 different algorithms (shown in Table 3.5). The results can be visualized in Figure 5.2 where there are represented the total waiting




(a) All traffic types.




(b) Zoomed in the lower waiting time traffic types.

*Source:* Documentation owner.

Figure 5.1: Plot with the traffic light study with an interval of 5 seconds.

time per algorithm and traffic flow type, but as the scale on the traffic flow types is too different, it is also obtained the best traffic light algorithms per traffic flow type in text (only the best one), as in the previous study. The time elapsed on generating the simulation results is around **5 minutes**, much less than the previous one as there are fewer algorithms to simulate. The results of this study are shown in Table 5.2:

Note that both studies' results are very similar and the conclusions obtained on both can be described together, as the behavior is the same. This is the reason why both of them have been described their results and behavior at once.

As can be seen in both Figures 5.1 and 5.2, the total waiting time follows a "logic" pattern that can be divided into to different approaches:

| Traffic Type | Best TL algorithm |
|---|---|
| 0 | static_program_2 |
| 1 | static_program_3 |
| 2 | static_program_4 |
| 3 | static_program_3 |
| 4 | static_program_1 |
| 5 | static_program_1 |
| 6 | static_program_5 |
| 7 | static_program_3 |
| 8 | static_program_1 |
| 9 | static_program_5 |
| 10 | static_program_5 |
| 11 | static_program_3 |

Table 5.2: Best traffic light algorithm (proportion) per traffic type.

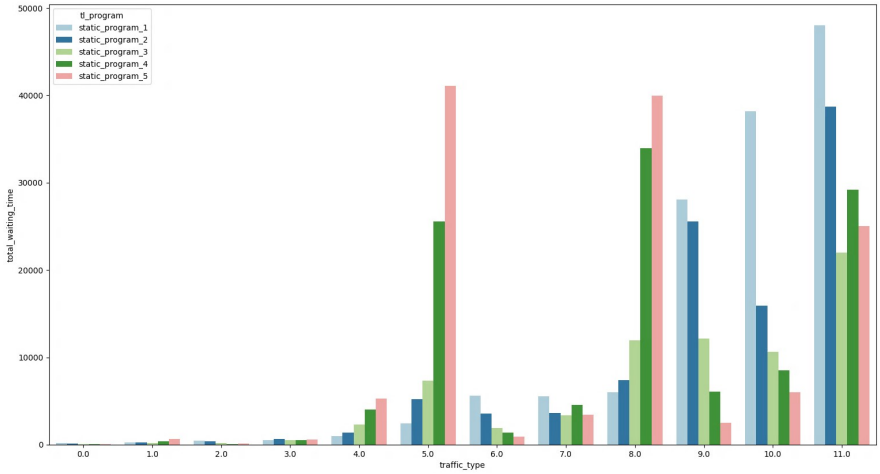
- **Balanced traffic flows type** (such as 0, 3, 7 and 11), always follows the same pattern by having similar total waiting times on those algorithms that are phase time equally distributed, the central ones, and higher time at the edges.

  This behavior is easily seen in traffic type 11, as it is the one with higher traffic flow and, so on, the total time is the higher of all.

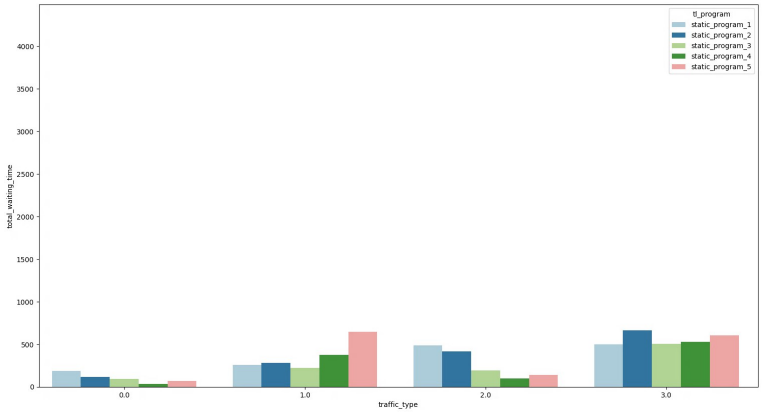
- **Unbalanced traffic flows type** (the rest of them), follows two different patterns that at glance can be depicted as the same. This pattern is related to the curve of total waiting time, being rightward inclined (highest value on the right side) when the traffic is higher on the secondary direction (EW) and the phase time is the minimum allowed on that direction (last traffic lights algorithms); and left-oriented (highest value on the left side) when the traffic is higher on the main direction (NS) and the phase time is the minimum possible on this direction (first traffic lights algorithms).

  This behavior is easily seen in the traffic types 5 and 8, and 9 and 10, respectively, as they are the ones with the higher difference of traffic flows between the two directions with a high number of vehicles per hour.

It can be concluded that the best algorithms are either the central one, with equally distributed time phases, or the edges ones, with higher traffic flow in one direction




(a) All traffic types.




(b) Zoomed in the lower waiting time traffic types.

*Source:* Documentation owner.
Figure 5.2: Plot with the traffic light study with traffic flow proportion.

than the other and with the minimum allowed time phase duration in that direction. These results are obvious as the vehicles that are in one direction will wait less time (on average) if the green phase time duration in this direction is higher.

Moreover, in the first four traffic types, the difference between the total waiting time is almost the same, which means that the selected algorithm will not affect it too much directly. This can be explained as the number of vehicles per hour in these traffic types are either “very low” or “low”, that is a little amount of them in an hour. This fact does not happen when there is more traffic as from the fifth traffic type.

Another relevant fact is that the fewer number of traffic lights algorithms, the easier

the adaptation will be, as long as these algorithms represent a wide variety of time phases and there is a noticeable difference on them. This is the main reason why the selected study to be used in the framework is the second one, as the time phases difference of the first study is only 5 seconds with the adjacent algorithms, that in real life is not enough to notice an actual difference, while in the second is around 12 seconds.

Lastly, it is important to mention that this study will not always output the same results as it depends on the randomness of the traffic generation in each direction. This is why sometimes the best algorithm is not the closest one to the edges or the exact center one, but they are very close to the expected results. Despite this, it is a good starting point to define the best possible traffic light algorithms per traffic type and to check the traffic behavior in different scenarios.

### 5.2.2 Dataset generation

Once the whole dataset generation process has been described in Section 4.3.1, it is shown the dataset output on both textual and graphical approaches, in order to see if the generation was performed successfully and the noise policy has been applied as expected.

Before showing the results of the generated dataset, it is important to mention that the elapsed time on the generation process is around 11 hours, depending on the simulation time and the number of vehicles defined per each traffic flow type. The number of dataset rows is 17.520 (48 rows per day $\times$ 365 days) so the dataset is not very big in terms of training different machine learning models, this problem is solved on the training process explained in Section 4.3.2.

Besides, this dataset has been to be regenerated several times along the development of the framework for the next reasons: (1) at first sight we defined that the number of vehicles per each traffic type was bigger, based on our conception of traffic density, but once the dataset was generated and tested along with the other components, we realized that this number was not realistic and so on, we should adjust it; (2) as

mentioned previously, the time patterns have been redesigned to be more realistic and to avoid extreme changes on the traffic flows type, this redesign implied to regenerate again the full dataset; and lastly (3) the improvement of the noise addition policy, that at first was very basic and did not represent the actual randomness of the real traffic.

At first, it is going to be indicated the time patterns that have been swapped based on the noise policy, in order to check in the resulting dataset if it is successfully generated. The number of swapped time patterns days is 45, which is 12,3% of the year. This swapping process is shown in Table 5.3, where it is indicated the original and the swapped time pattern day, and the date the swapping process occurs.

| Original day | Swapped day | Date |
|---|---|---|
| Friday | Thursday | 08/01 |
| Tuesday | Summer vacation day | 02/02 |
| Saturday | Monday | 13/02 |
| Friday | Saturday | 26/02 |
| Tuesday | Summer vacation day | 02/03 |
| Friday | Monday | 05/03 |
| Friday | Thursday | 12/03 |
| Tuesday | Summer vacation day | 23/03 |
| Thursday | Monday | 25/03 |
| Wednesday | Monday | 31/03 |
| Monday | Summer vacation day | 05/04 |
| Saturday | Summer vacation day | 17/04 |
| Wednesday | Winter vacation day | 12/05 |
| Thursday | Summer vacation day | 27/05 |
| Wednesday | Friday | 02/06 |
| Tuesday | Sunday | 22/06 |
| Tuesday | Friday | 06/07 |
| Wednesday | Thursday | 07/07 |

| | | |
|---|---|---|
| Tuesday | Summer vacation day | 13/07 |
| Sunday | Winter vacation day | 18/07 |
| Summer vacation day | Thursday | 22/07 |
| Summer vacation day | Saturday | 24/07 |
| Summer vacation day | Winter vacation day | 27/07 |
| Summer vacation day | Thursday | 29/07 |
| Summer vacation day | Winter vacation day | 03/08 |
| Summer vacation day | Monday | 16/08 |
| Summer vacation day | Winter vacation day | 20/08 |
| Summer vacation day | Winter vacation day | 29/08 |
| Summer vacation day | Tuesday | 31/08 |
| Summer vacation day | Thursday | 02/09 |
| Summer vacation day | Winter vacation day | 10/09 |
| Thursday | Saturday | 16/09 |
| Thursday | Summer vacation day | 23/09 |
| Sunday | Tuesday | 26/09 |
| Sunday | Summer vacation day | 03/10 |
| Friday | Sunday | 22/10 |
| Saturday | Saturday | 23/10 |
| Wednesday | Thursday | 27/10 |
| Sunday | Summer vacation day | 31/10 |
| Saturday | Summer vacation day | 06/11 |
| Friday | Monday | 19/11 |
| Saturday | Wednesday | 27/11 |
| Thursday | Friday | 09/12 |
| Monday | Friday | 13/12 |
| Winter vacation day | Summer vacation day | 26/12 |

Table 5.3: Swapped days on the generated dataset.

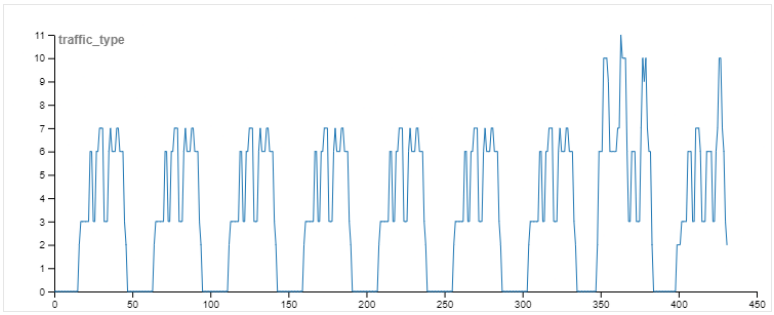
In Figure 5.3 it can be seen a part of the generated dataset, where are depicted the main time patterns involved and the noise added to each one of them, if added. In fact, this figure represents different intervals of days in order to show the most valuable information of the dataset itself: working days, noise addition, vacation days and randomly swapped days. These patterns are described below, indicating one by one their meaning. Besides, remember that the swapped days are represented in Table 5.3

a) The first time pattern is Friday but, until the seventh time pattern (day), it is followed the same pattern but with little variations, for example between the fourth and fifth, as these days are associated with winter vacations (ending at 7 of January). The eighth day in fact should be a Friday but it is swapped to a Thursday.

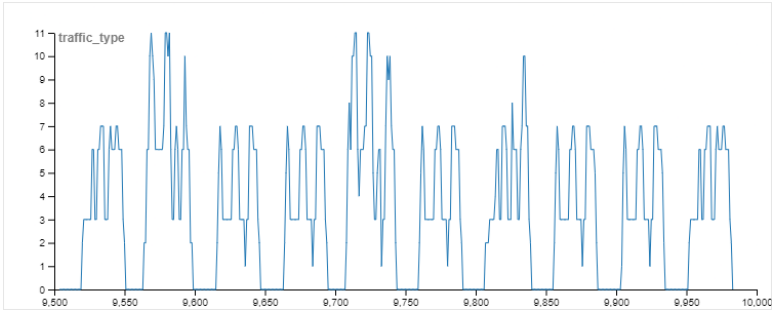
b) The first time pattern is Sunday but it is swapped to a winter vacation day. Due to the fact that from the 20th of July starts the summer vacations, the second time pattern is a usual Monday time pattern and the next ones are summer days with some little modifications. Other days that have been swapped too are the 22nd, 24th and 27th of July to Thursday, Saturday and winter vacation day, respectively.

c) The first time pattern is Wednesday and, in this case, almost all the time patterns are set by default with the predefined strategy of working days, with some little modifications on any of them, except for the days 22th (Friday), 23th (Saturday), 27th (Wednesday) and 31th (Sunday) of October, that have been swapped to Sunday, Saturday, Thursday and a summer vacation day, respectively.

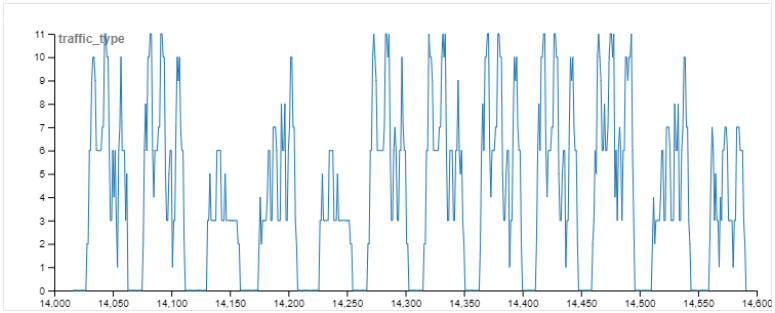
d) The first time pattern is Friday and from the 21st of December until the 31st, all the time patterns are winter vacation time patterns as they are in their defined range. Before this day, all the time patterns follow the working days' predefined strategy with some noise, and the only time pattern that is swapped in this case is the 26th with a summer vacation day.



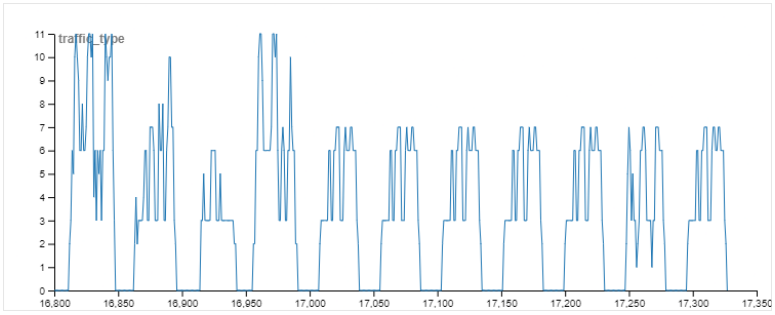

(a) From 01-01-2021 to 09-01-2021




(b) From 18-07-2021 to 27-07-2021




(c) From 20-10-2021 to 31-10-2021




(d) From 17-12-2021 to 27-12-2021

*Source:* Documentation owner.
Figure 5.3: Different generated calendar patterns.

Note that the different time patterns can be easily visualized as there is always a starting traffic type of zero and then the actual values, representing that at night hours

there is no traffic at all. It is also important to mention that the randomness of the noise policy, generates each time a different dataset and so on the results will be completely different if the calendar is regenerated. In fact, some of the time patterns used may not represent the real calendar time patterns as it is a totally subjective definition that follows, in fact, a hand-made noise strategy.

### 5.2.3 Predictor models training

This section depicts and compares both the performance and the elapsed time of all the different *Machine Learning* algorithms used in the Traffic Light Predictor component.

Note that not all the models are included on the section tables as there have been trained a total number of 204 models per each kind of predictor (based on date information and based on context and date information). This number is that high because of the use of the k-fold training process and the hyperparameter tuning.

In fact, this huge number of models affects directly the elapsed time of the training process even if the training time on each one of the models is very low. The elapsed time on the training process of the date-based predictors is around 51 seconds, while in the context-based predictors is around 24,4 seconds. These values have been calculated by adding the training elapsed time per each model.

It is important to mention that the model weights, along with its performance values are stored in two different folders, one per each kind of predictor:"classifier_models_date" and "classifier_models_context", respectively.

The computer used to perform the training process has the following specifications:

- Processor: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- RAM memory: 16 GB DDR4.
- OS: Ubuntu 18.04 LTS.
- Graphics Card: NVIDIA GeForce GTX 1650 4GB.

In Table 5.4 there are shown the average elapsed time and F1 score, the performance measure, per type of algorithm; and as there are two kinds of predictors, it is also divided by them.

| | Date-based model | | Context-based model | |
|---|---|---|---|---|
| **Model** | Elapsed time | F1 score | Elapsed time | F1 score |
| **Naive Bayes** | 0.030150 | 0.143621 | 0.007067 | 0.997105 |
| **SVM Linear** | 8.087447 | 0.148330 | 0.062826 | 0.999649 |
| **SVM Polynomial 2** | 9.386143 | 0.059206 | 4.466585 | 0.630051 |
| **KNN** | 0.302031 | 0.296457 | 0.313135 | 0.999621 |
| **Decision Tree** | 0.011950 | 0.488837 | 0.009619 | 0.887136 |
| **Random Forest** | 0.048436 | 0.216213 | 0.042495 | 0.618791 |

Table 5.4: Average of time elapsed and F1 score in TLP models.

As can be seen, the context-based models achieve a great performance in terms of both elapsed time and F1 score, where all the model algorithms improve their performance in comparison with the date-based models.

On the one hand, the algorithm with the highest F1 score in the date-based models is the Decision Tree (0.488837) while in the case of the context-based models is Linear Support Vector Machine (0.999649) followed too close by the KNN (0.999621), doubling the score.

On the other hand, the algorithm with lowest elapsed time on the training process in the date-based models is the Decision Tree (0.011950 seconds), while in the context-based models the lowest value is related to the Naive Bayes algorithm (0.007067 seconds). Note that these values are the average of all the models using such algorithms, and they only are used to check their overall performance.

As in the TLP component there are going to be used the best models in terms of F1 score, it is going to be indicated in Table 5.5 and Table 5.6 the top ten best F1 score models per kind of information used (date and context), in order to see the algorithm and its hyperparameters. Based on these tables, it is possible to observe the following

facts:

- In **date-based models**, the Decision Tree algorithm is the best one, and the highest values of maximum depth, the higher F1 score achieved. In this case, the best model is the Decision Tree with a maximum depth of 16 levels using the second fold (represented by the number "1") with a total F1 score of 0.692461. Another relevant fact to mention is that the elapsed time on the training process is almost the same on each one of them, being a very low value of near 0.01 seconds.

| Model | Fold | Hyperparameter | F1 score | Elapsed time |
|---|---|---|---|---|
| Decision Tree | 1 | Max Depth: 16 | 0.692461 | 0.016153 |
| Decision Tree | 1 | Max Depth: 18 | 0.690532 | 0.015352 |
| Decision Tree | 1 | Max Depth: 14 | 0.681481 | 0.014087 |
| Decision Tree | 0 | Max Depth: 18 | 0.661954 | 0.016140 |
| Decision Tree | 0 | Max Depth: 16 | 0.660982 | 0.016574 |
| Decision Tree | 0 | Max Depth: 14 | 0.656836 | 0.014378 |
| Decision Tree | 0 | Max Depth: 12 | 0.647193 | 0.015273 |
| Decision Tree | 0 | Max Depth: 10 | 0.624881 | 0.013551 |
| Decision Tree | 1 | Max Depth: 12 | 0.614707 | 0.011710 |
| Decision Tree | 1 | Max Depth: 10 | 0.575008 | 0.010465 |

Table 5.5: Best TLP date-based models based on F1 score.

- In **context-based models**, the KNN algorithm is the best one on average as it is the one that appears the most, independently of its number of neighbors. It is worth mentioning that as all the models indicated achieve the same F1 score, 0.999881, the main feature to compare between them would be the elapsed time. In this case, the best algorithm is the Decision Tree with a maximum depth of 6 levels and a elapsed time of 0.014560 seconds to train. Moreover, it can be seen that the KNN models last much more time on the training process, which means that it is slower and therefore, worse than the Decision Tree model option.

Lastly, the conclusions, divided into date and context approaches along with additional ones, that could be drawn from all this information are:

| Model | Fold | Hyperparameter | F1 score | Elapsed time |
|---|---|---|---|---|
| Decision Tree | 1 | Max Depth: 6 | 0.999881 | 0.014560 |
| KNN | 1 | Neighbors: 5 | 0.999881 | 0.276917 |
| KNN | 1 | Neighbors: 14 | 0.999881 | 0.288099 |
| KNN | 1 | Neighbors: 9 | 0.999881 | 0.290509 |
| KNN | 1 | Neighbors: 6 | 0.999881 | 0.292982 |
| KNN | 1 | Neighbors: 8 | 0.999881 | 0.301028 |
| KNN | 1 | Neighbors: 12 | 0.999881 | 0.301704 |
| KNN | 1 | Neighbors: 11 | 0.999881 | 0.301838 |
| KNN | 1 | Neighbors: 7 | 0.999881 | 0.315319 |
| KNN | 1 | Neighbors: 13 | 0.999881 | 0.319673 |

Table 5.6: Best TLP context-based models based on F1 score

- In **date-based** models:
  - On the one hand, the algorithm with the best F1 score average is the Decision Tree with a score of 0.488837, while the next best one is the KNN but with a much lower value of 0.296457. On the other hand, the model with the worst F1 score average is the Polynomial Support Vector Machine with 0.059206.
  - The algorithm with the best elapsed time on the training process, on average, is the Decision Tree with a total time of 0.011950 seconds, followed by the Naive Bayes with 0.030150 seconds; while the one with the worst average is the Polynomial Support Vector Machine with 9.386143 seconds followed too close by the Linear Support Vector Machine with 8.087447 seconds, a huge difference between the best and the worst.
  - The best possible algorithm is the Decision Tree in this case, as it achieves in average the best F1 score and the lower elapsed time on the training process. In fact, this algorithm is the one used on the top ten best models, shown in Table 5.5. The difference in the elapsed time is very low, even being inestimable between them, as their elapsed time is around 0.01 seconds. Besides, it is possible to notice that in this case, the higher the maximum depth value the higher the F1 score is.

  - The main reason why both Support Vector Machines algorithms last that amount of time would be that they do not find a hyperplane that fits fine the relation between the date and the traffic type without any additional information such as the number of vehicles.

- In **context-based** models:

  - The algorithm with the best F1 score average is the Linear Support Vector Machine with a score of 0.999649, followed too close by the KNN algorithm with 0.999621; while the one with the worst average is the Random Forest with 0.618791.
  - The algorithm with the best elapsed time on the training process, on average, is the Naive Bayes with a total time of 0.007067 seconds, followed by the Decision Tree with 0.009619 seconds; while the one with the worst average is the Polynomial Support Vector Machine with 4.466585 seconds, a huge difference between them.
  - The best possible algorithm is the Linear Support Vector Machine in this case, as it achieves in average the best F1 score and one of the lowest elapsed training time, but in this case, this algorithm is surpassed by the hyperparameter tuning on the Decision Tree and KNN algorithms as can be seen in Table 5.6.
  - The difference on the elapsed time is very low between the KNN models with values around 0.28 seconds, but all of them are surpassed by the Decision Tree as it only lasts 0.014 seconds on the training process.
  - The KNN number of neighbors hyperparameter does not affect too much to the F1 score as all the values are exactly the same, 0.999881, while this parameter is being modified on each one of the models.

- It is possible to see that the use of context information increases a lot the performance of the different algorithms as the elapsed time on the training process.

- The low F1 score on the date-based models would be caused, maybe because there is not implicit relation between the date and the traffic type, as it occurs with the contextual information, due to the randomness and uncertainty of the traffic.

- Context-based models do have better results because of the use of more features, and due to the fact that the pattern used to define the traffic types is directly related to the context information, as the number of vehicles, which means that it is easier to "learn" the function to predict its value.

- Even the worst context-based model is better, in average, than the best date-based model in terms of both F1 score and elapsed training time.

- Hyperparameter tuning is a very useful process as it allows to train the same model with different hyperparameters and achieve better results.

### 5.2.4 Examples execution

Lastly, there are shown the main examples described previously, in Section 5.1, along with several graphics used to compare the framework's performance and adaptation behavior between them.

This section is divided by: one day (e.g. Monday) time pattern example, date interval (from 18-07-2021 to 20-07-2021) time patterns example and a random time pattern. In all the examples that are described in this section, there are going to be shown the total waiting time per direction, the number of vehicles per direction and lastly the average waiting time per vehicle in each direction. Besides, it is also shown the behavior of the Traffic Analyzer and Traffic Light Predictor components, in order to check if they analyze/predict, respectively, the current traffic type successfully, when they are deployed.

It is worth mentioning that the randomness of the traffic makes that each simulation outputs different results in terms of the number of vehicles per direction, but this will not affect directly the results and conclusions that will be obtained from these

simulations, as the "pattern" followed is the same and so on, the behavior would be the same with little variations that can be considered as noise. Besides, remember that the time interval used to monitor the contextual information is 5 minutes.

The order of describing and discussing the results of the different adaptation approaches is: (1) "No adaptation", (2) "Only predictions based on date adaptation", (3) "Only real-time analyzer adaptation" and (4) "Both analyzer and contextual predictor adaptation", starting from the most basic adaptation and ending with the most complex one. This order is followed by all the example time patterns described previously.

Before starting the execution of the examples, note that the explanations given in the next sections might be very similar between them, especially on the same adaptive approach, as the behavior of these approaches are well defined and will not change significantly between executions, in fact in most of them it only changes the value results.

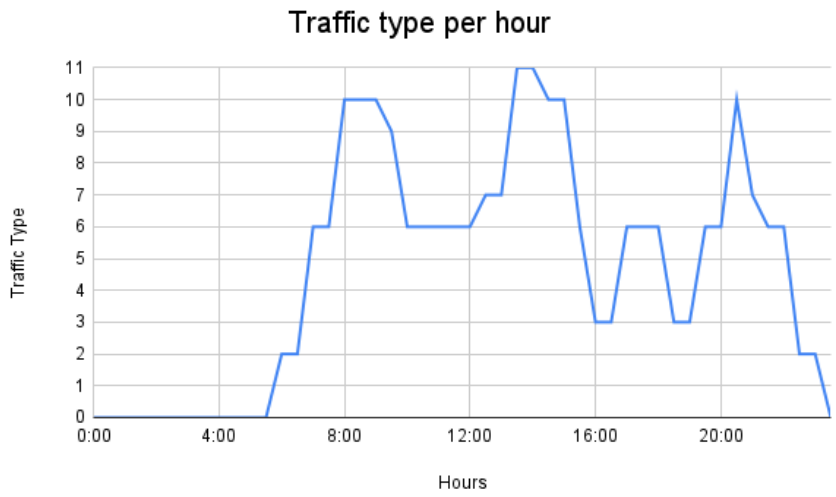
#### 5.2.4.1 One day simulation

Before starting, it is worth mentioning that the time pattern used in this example, is shown in (a) on Figure 4.2, despite it being named as regular working day, the pattern is very similar to the Monday one. Despite this, in order to see this pattern easy to compare it with the analysis and predictions performed next, it is shown in Figure 5.4.

The time elapsed on the execution of this example is around 8 minutes per adaptation approach, as the deployment of the components and installation of the required libraries on each one of them last around 2 minutes; and the simulation itself lasts around 6 minutes.

In Figure 5.5 it is possible to see all the plots related to the "No adaptation" approach, which will be considered as the base example in order to compare the results between the different adaptations as this simulation is like the real-life approach, with static time phases and no adaptation.

On the one hand, it is possible to see that the plots (a) and (b) are directly related



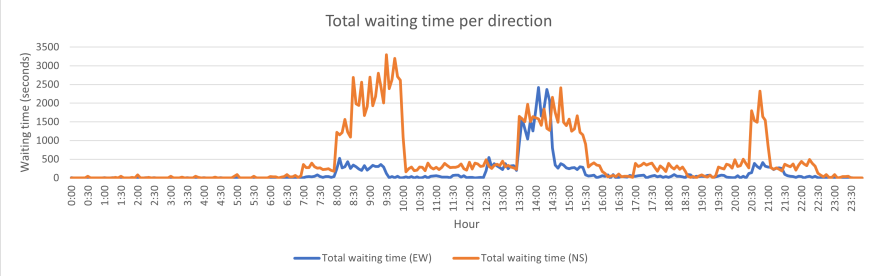

*Source:* Documentation owner.
Figure 5.4: Monday simulation - Time pattern.

as the waiting time per direction is based on the number of vehicles that pass in that direction, in fact, the pattern is very similar on both plots and it is adjusted to the input time pattern.

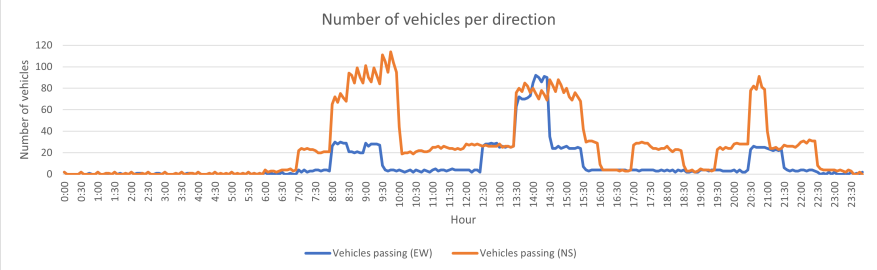
On the other hand, the (c) plot shows the average waiting time per vehicle, where it is possible to conclude that until the first "peak" hour (around 7:00), the NS direction has some "peaks", which indicates that there are vehicles that wait a lot of time, a random fact that may occur and the only possible solution would be using an actuated traffic light, where the green phase is switched when it is detected a vehicle on that road. This approach is out of the scope of this project. In fact, this situation happens on all the examples developed, so it is only explained here in order to not be repeated and make reading clearer and easier.

In summary, the highest value of total waiting time on a direction is around 3.200 seconds on NS and 2.500 seconds on EW, while the highest number of cars are 110 and 90, respectively. On average, the waiting time per vehicle is almost equally distributed, unless the first time range (explained in the previous paragraph), being higher on NS direction with 40 seconds per vehicle.

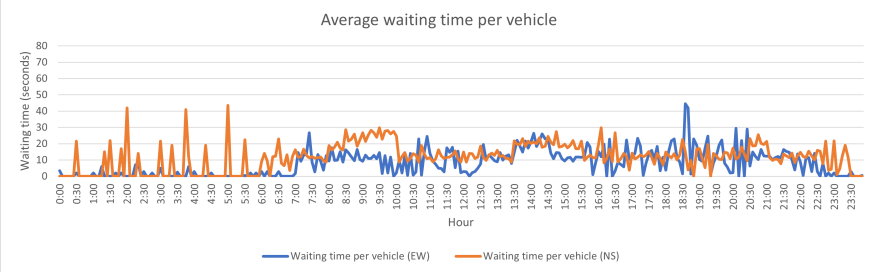
The next step is to use the most basic adaptation, using the traffic light predictor




(a) Total waiting time per direction.




(b) Number of vehicles per direction.

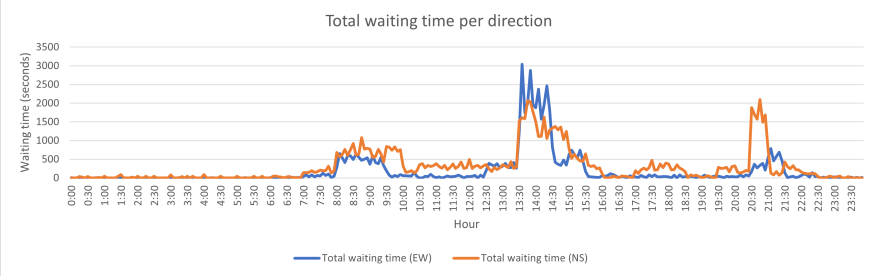



(c) Average waiting time per vehicle.

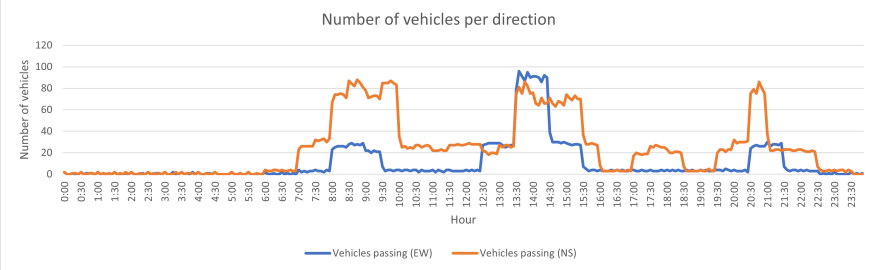
*Source:* Documentation owner.
Figure 5.5: One day simulation - No adaptation.

based on the date information, the "**Only predictions based on date adaptation**" approach. In this case, the plots shown in Figure 5.6 are the same as the previous one, along with the prediction of the traffic type, in order to compare it with the input time pattern (Monday).

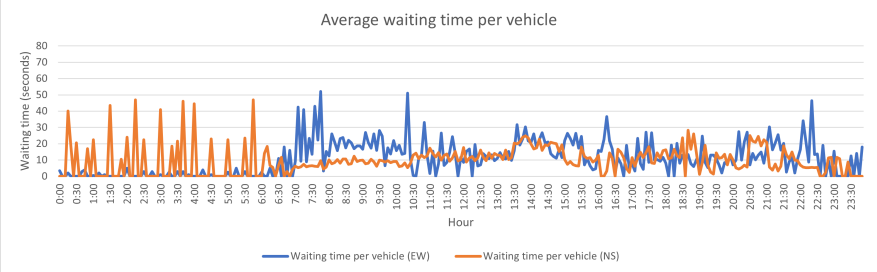
As can be seen on (a), the curve on the two first "peak" hours (8:00 and 13:30) are flatten and balanced highly in the first range (reducing from 3.000 seconds on NS to 1.000 and letting the EW direction almost as same as previously) and even a little bit on the second one, but not as good as the first one (in this case increasing from 2.500 seconds to 3.000 on EW and smoothing the curve on NS direction reducing it from 2.500 to 2.000 seconds). The last "peak" hour (20:30) does not have any kind of



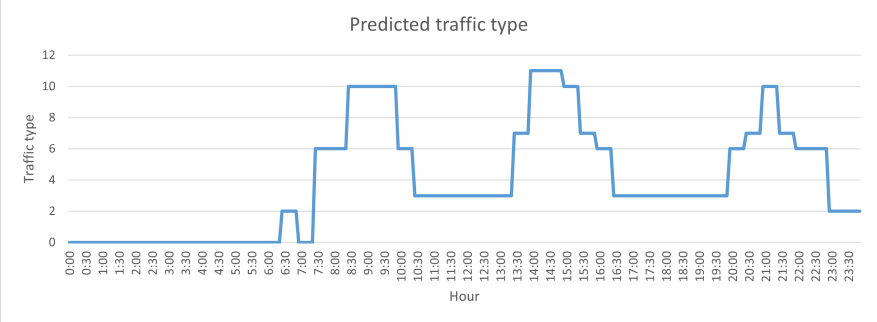

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Predicted traffic type pattern.

*Source:* Documentation owner.

Figure 5.6: One day simulation - Only predictions based on date adaptation.

improvement and, in fact, it acts very similarly to the base example. In this case, the number of vehicles (b) simulated is not exactly the same as in the base example, but on average, the numbers are very close, following the same pattern.

Based on the average waiting time per vehicle (c), it is possible to see that the starting waiting time follows the same pattern as in the base example. It can also be seen that the average time on the NS direction is decreased while the EW time is increased a little bit. In fact, notice that this improvement only happens on the range from 7:00 to 10:30, which matches with the hours where the traffic flow is much higher on NS rather than EW direction.

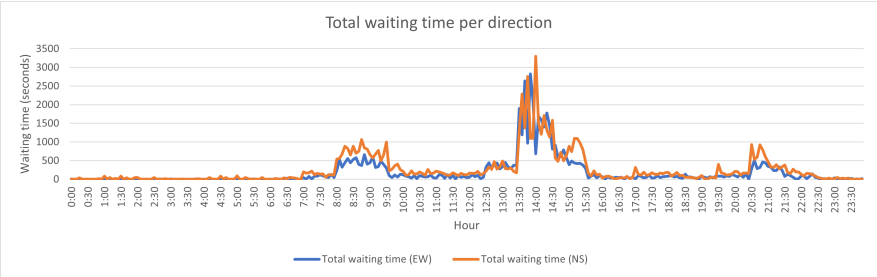
The predicted traffic type (d) is similar to the predefined time pattern but is much "squarer", making the prediction more inaccurate. Even it is possible to see that at 6:00 it predicts a traffic type of 2 and then the next value is 0 again, a pattern not defined in the original time pattern and that will be considered as a bad prediction. Moreover, the curve existing in the original pattern at 18:00 is not well predicted, as there is a single value on this range, a traffic type of 3, not evaluating the real flow variation at all.

In terms of values, the highest value of total waiting time on a given direction is reduced to 2.000 seconds on NS and increased to 3.000 seconds on EW, while the highest number of cars are 110 and 90, respectively. On average, the waiting time per vehicle is reduced on the NS direction (around to its half, 50%) on NS, and increased (around two times, 200%) on EW on the range from 7:00 to 10:30. Before and after these ranges, the average waiting time is almost the same in both examples.

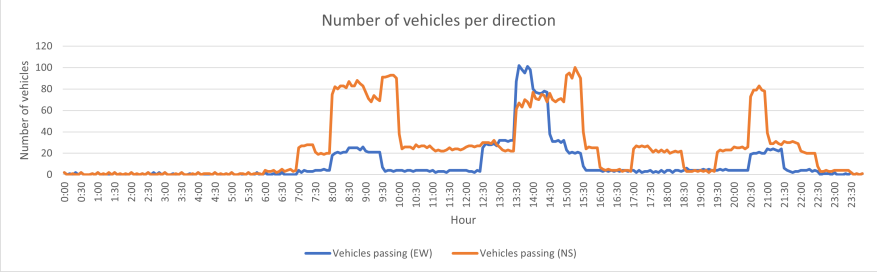
Based on all the plots and values indicated previously, it can be concluded that the use of a predictor based on date, in this case, reduces the overall waiting time on NS direction, which is the main direction on the scenario, while increasing a little bit the waiting time on EW as there are much fewer vehicles on that direction on the range of hours indicated previously. Despite that the predictor based on date is not very accurate and sometimes it would not predict the traffic type correctly, it is noticeable that it is a better approach than the "No adaptation" one, even if the predictor is not adapting as well expected.

The next approach is related to the use of a real-time analyzer with traffic contextual information (**"Only real-time analyzer adaptation"** example). In this case, the analyzed traffic type is also shown in the plots in Figure 5.7, along with the other plots

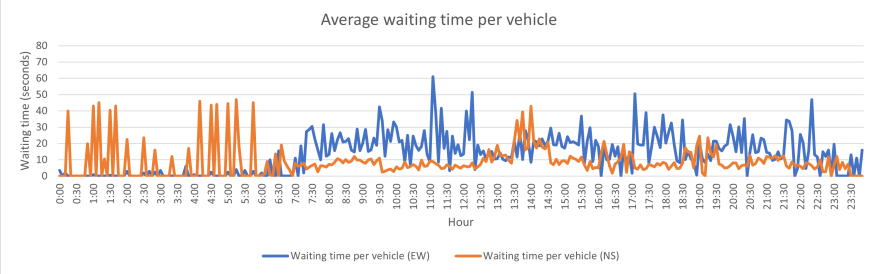
mentioned previously. It is worth mentioning that this approach will be compared directly with the previous one (date-based predictor), not with the base one, as it improves its behavior.



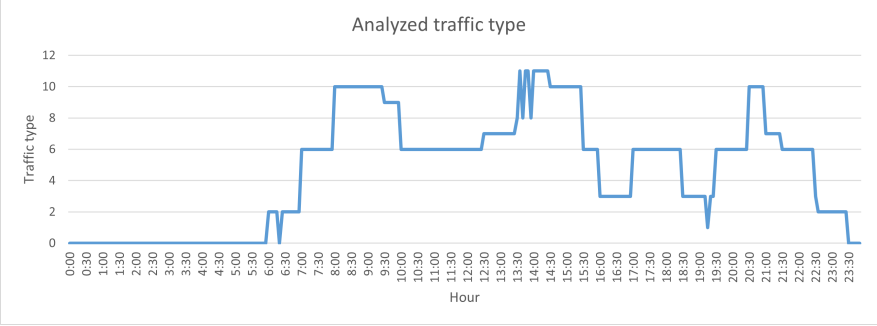

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Analyzed traffic type pattern.

*Source:* Documentation owner.

Figure 5.7: One day simulation - Only real-time analyzer adaptation.

On (a) it is possible to see that both lines, representing each one of them a single direction, are very similar between them, even with the same results on some hour ranges. The most relevant fact to mention is that the waiting time one both directions is almost the same and the curves (in the NS direction) of all the "peak" hours are flattened and reduced to the ones in the EW direction. Besides, the ranges from 10:00 to 12:00 and 15:30 to 20:30 are reduced, from 500 to 200 seconds. In fact, using the analyzer it is possible to see that the last "peak" hour (20:30 to 21:30) is highly flattened, while with the predictor based on date information, this fact does not happen as it remained the same as with no adaptation. So, it can be concluded, there is not much difference with the date-based predictor example, except for this previously mentioned range and from 9:30 to 10:00, where it is also flattened on NS, which coincides with the bad models' predictions.

On (b) there are represented the number of vehicles per direction, being very similar in each example. As can be seen, the number of vehicles is not reduced on the example but the total waiting time per direction is reduced and it does not follow the same pattern as it occurs on the base example. Therefore, it can be said that the adaptation is working fine and as expected.

The average waiting time per vehicle (c), is very similar to the previous example as it reduces the waiting time on the NS direction while increasing the time in the EW direction. In this case, the waiting time on EW is a little bit higher than in the previous example but this might be caused by the traffic noise, so is not considered to achieve a real change.

Lastly, the analyzed traffic type (d) is much more accurate than the date-based prediction as the pattern analyzed is almost the same as the predefined time pattern, although there are some values with "noise", that indicates that the analyzer does not fit perfectly to the real traffic time pattern but it is too close. One of the most relevant differences with the date-based prediction example is that in this case, the curve from 17:00 to 18:30 is well represented.

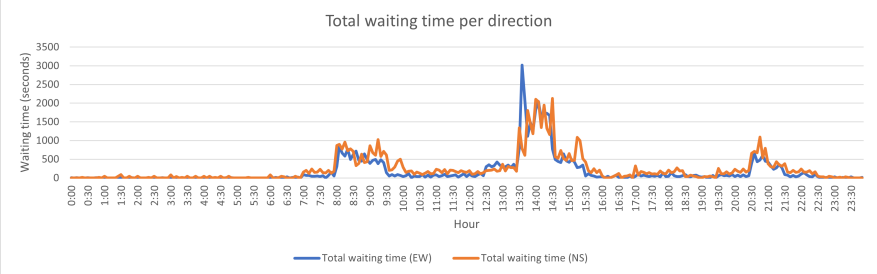
In terms of values, the highest value of total waiting time on a given direction is

increased to 3.200 seconds on NS and remains the same (3.000 seconds) on EW, while the highest number of cars is 100 in both directions. Despite that the waiting time on NS is higher than the date-based predictor example, it is only a single value and the rest of the values are slightly lower. On average, the waiting time per vehicle is almost the same on the NS and EW direction as in the previous approach. Based on all the plots and values shown previously, it can be concluded that the use of a real-time analyzer improves the adaptation behavior of the system compared to the date-based predictor, as it reduces the overall waiting time on NS direction, which is the main direction on the scenario, while increasing a little bit the waiting time on EW; even the analyzer fits better to the real-time conditions, showing that it works better than the predictor. Besides, it flattens, even more, the total waiting curves, reducing also the last "peak" hour curve, the one that the predictor was not able to do it.

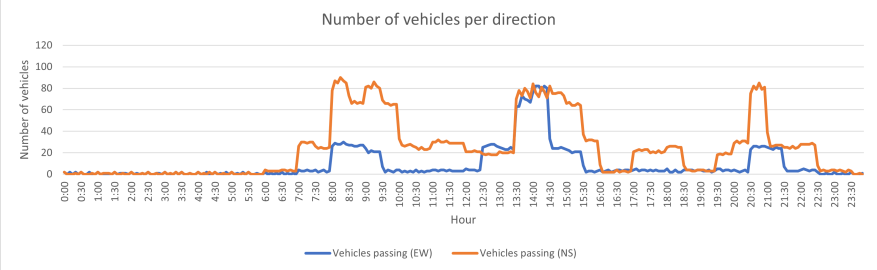
The last approach is a combination of both the real-time analyzer and a context-based predictor, which in fact is an improvement of the date-based predictor, by using traffic contextual information (**"Both analyzer and contextual predictor adaptation"** approach). In this case, the analyzed and predicted traffic types are also shown in the plots in Figure 5.8, along with the other plots mentioned previously. It is worth mentioning that this approach will be compared directly with the previous one (real-time analyzer), not with the base one.

On (a) it is possible to see that both lines, representing each one of them a single direction, are very similar between them, even with the same results on some hour ranges as in the previous example. In fact, the main difference with that example is that the highest curve, on range from 12:30 to 15:30, is slightly reduced in both directions. Even the "peak" value on EW direction at 13:30 could be considered as a noise value, due to the adaptation process and the uncertainty of the traffic.

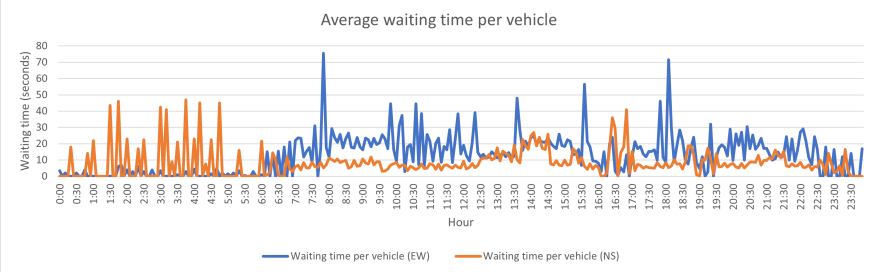
This slight improvement can be thought to happen because of the randomness of the traffic and the number of vehicles used, but as it is shown on (b), this number is lower in both directions at the beginning (EW) and at the end (NS) of the mentioned time range, while the rest of vehicles is almost the same on both examples, meaning



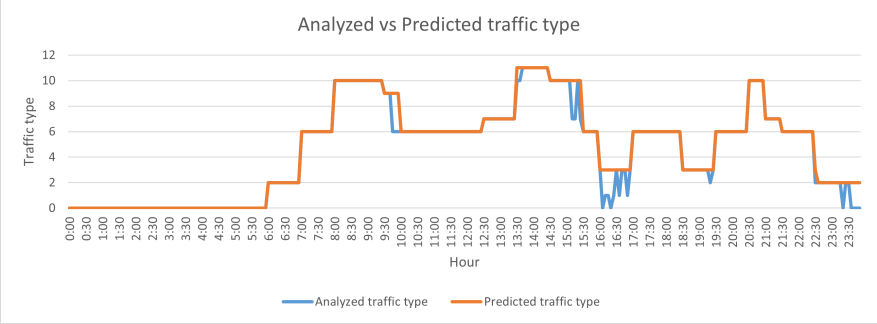

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Analyzed traffic type pattern.

*Source:* Documentation owner.

Figure 5.8: One day simulation - Both analyzer and context predictor adaptation.

that the adaptation behavior can be affected by the number of vehicles but the global behavior will remain the same. Besides, as in the previous approach, the number of vehicles is not reduced on the example but the total waiting time per direction is

reduced and it does not follow the same pattern as it occurs in the previous example, so it can be concluded that adaptation is working fine and improving slightly the previous approach, but this improvement does not mean that it will work better than the previous one in all the cases.

The average waiting time per vehicle (c), is almost the same on both EW and NS than in the previous approach. This occurs because the real difference between the two approaches is not even noticeable. Moreover, in the EW direction, there are some higher "peaks" than in the previous example, but the mean remains almost equal.

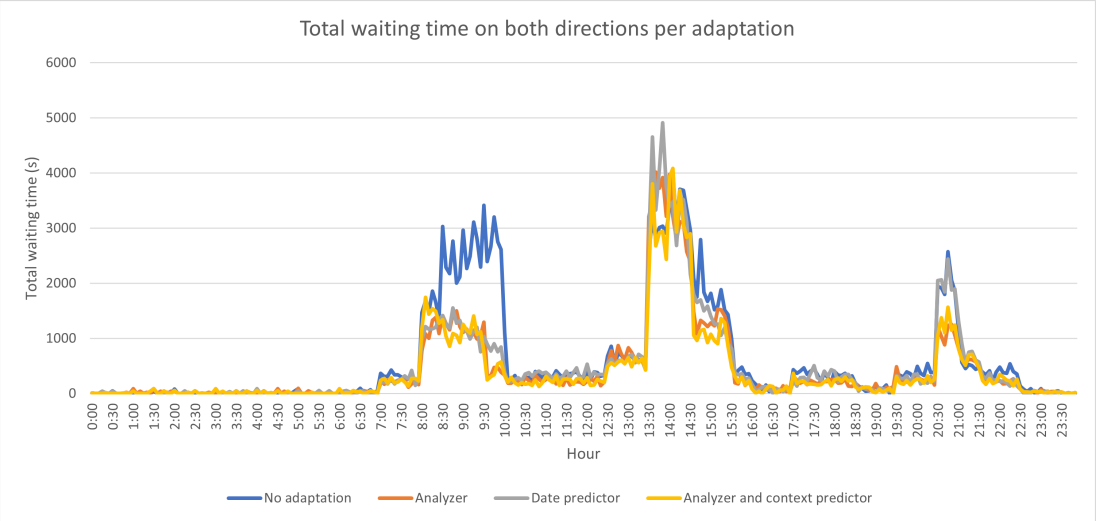
Lastly, the analyzed and predicted traffic types (d) are very similar between them but the analyzer does not adapt perfectly to the real traffic type in comparison to the context-based predictor. The most relevant fact to mention about this plot is that the context-based predictor is much better than the date-based one as it fits better to the real traffic types, predicting even better than the analyzer itself on some hours (e.g. from 16:00 to 17:00). Moreover, the difference of results between the analyzer and the predictor is not high enough to notice real variations on the adaptation, as their outputs are almost the same and the adapter does not need to apply the difference algorithm mentioned in Section 4.4.

In terms of values, the highest value of total waiting time on a given direction is reduced to 2.500 seconds on NS and increased to 3.000 seconds on EW, while the highest number of cars are 90 and 80, respectively. On average, the waiting time per vehicle is almost the same on the NS direction as in the previous example but reduced at the half on the range from 13:30 to 14:30, while it is doubled on EW on some hours (7:30 or 18:30). Before and after these ranges, the average waiting time is almost the same in both examples, and even these "peak" values are not relevant at all as they are "noise".

Based on all the plots and values shown previously, it can be concluded that the combination of both analyzer and context-based predictor could increase a little bit the adaptation performance but the real difference between using this approach and the previous is not easily noticeable. In fact, the most relevant improvement of this

approach is that the context-based predictor performs much better than the date-based one, as it uses contextual information. Furthermore, this approach does not flatten more any curve compared to the previous approach, indicating that in fact, is not possible to improve even more the adaptation behavior.

Once all the adaptation examples have been described and shown their results, there are all gathered into a single plot (on Figure 5.9) in order to **compare directly** between them and to see which one is the best adaptation approach for this time pattern. Note that the information compared is the sum of the total waiting time in both directions.



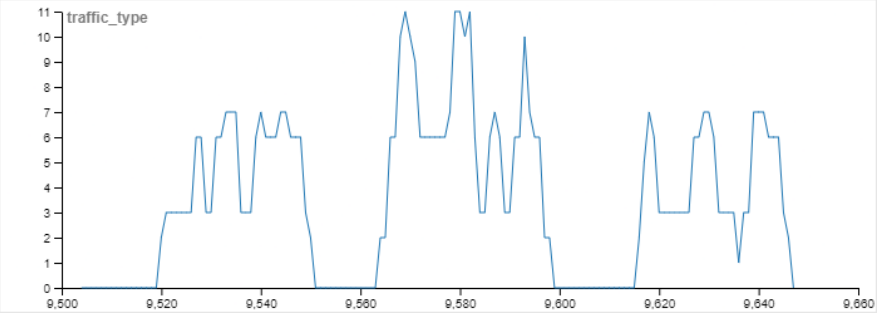

*Source:* Documentation owner
Figure 5.9: One day simulation - Total waiting time summary.

The first fact to mention is that all the adaptation approaches reduce significantly the total waiting time on the first "peak" hours (from 8:00 to 10:00) where the maximum traffic flow is "high" on NS and "medium" on EW. This means that the adaptations prioritize the direction with high vehicle density. On the second "peak" hours (from 13:30 to 15:30), the difference between the no adaptation approach and the rest of them does not exist as the total waiting time is very similar on all the cases, except for the date-based predictor which, in fact, is higher than the no adaptation approach. In fact, this range of hours only indicates that the traffic with the high flow cannot be easily improved globally as one direction will be improved while the

other will worsen. This does not mean that the approach does not work fine, it only shows that the time phase balance is not performed as well as in the other approaches because the traffic is balanced. On the third "peak hour" it is possible to see that the no adaptation and date-based predictor are exactly the same, while the contextual information-based approaches improve the overall waiting time. The rest of the hours are almost the same on all the approaches as the real number of vehicles used on them is not high enough to see a substantial improvement.

#### 5.2.4.2 Date interval simulation

In this simulation, the defined pattern is retrieved from the three first days from the range shown on (b) in Figure 5.3. In order to clarify the time patterns used, it is also shown in Figure 5.10. This date interval has been selected as it has three different time patterns with additional noise and one of them, the first one, is swapped from its original pattern. The defined time patterns are related to a winter vacation day, Monday and a summer vacation day. This noise will show the true relevance of the contextual information rather than the date information. In this case, the total elapsed time on the execution of the example is around 20 minutes as the deployment of the components and installation of the required libraries on each one of them last around 3 minutes; and the simulation lasts around 17 minutes.

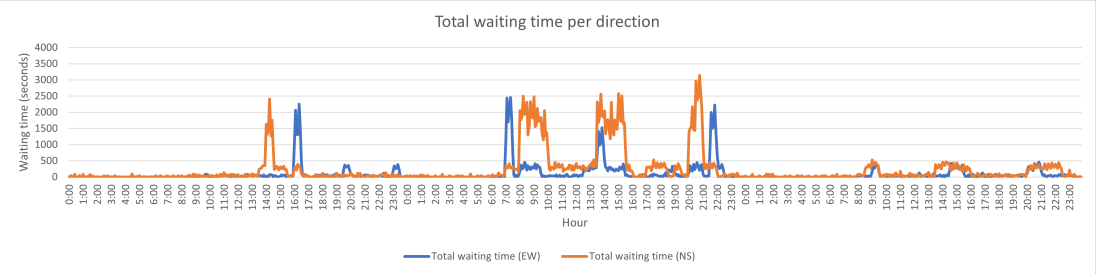



*Source:* Documentation owner.
Figure 5.10: Date interval simulation - Time pattern.

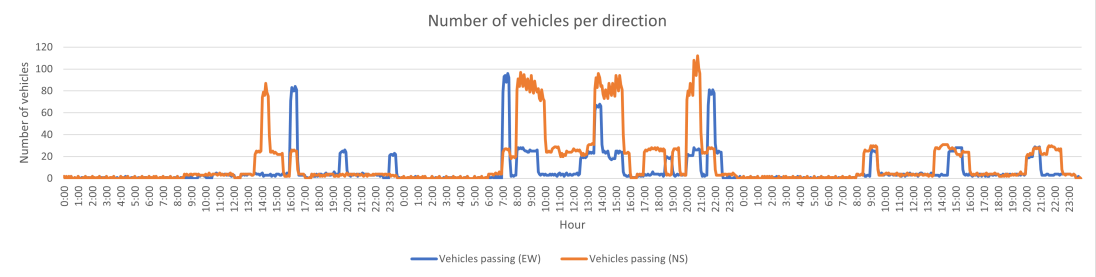
The process followed in this simulation example is the same as in the previous one, starting from the "no adaptation", followed with the "date-based predictor" and

"real-time analyzer", concluding with the "combination of the real-time analyzer and context-based predictor" approaches.

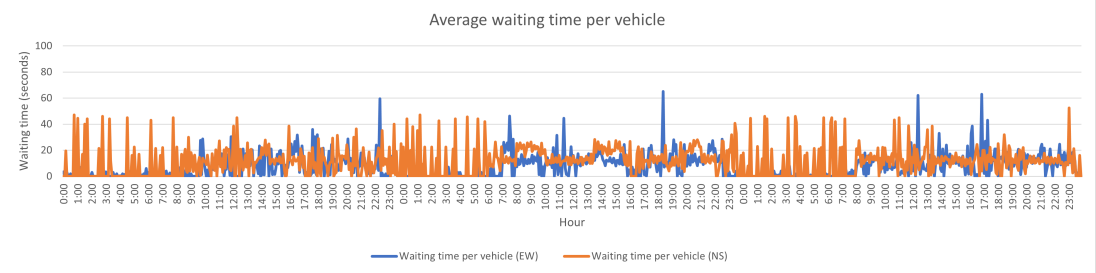
In Figure 5.11 it is possible to see all the plots related to the "No adaptation" example, which will be considered as the base plots used to compare the results between the different adaptations as this simulation is like the real-life approach, with static time phases and no adaptation.




(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.

*Source:* Documentation owner.
Figure 5.11: Date interval simulation - No adaptation.

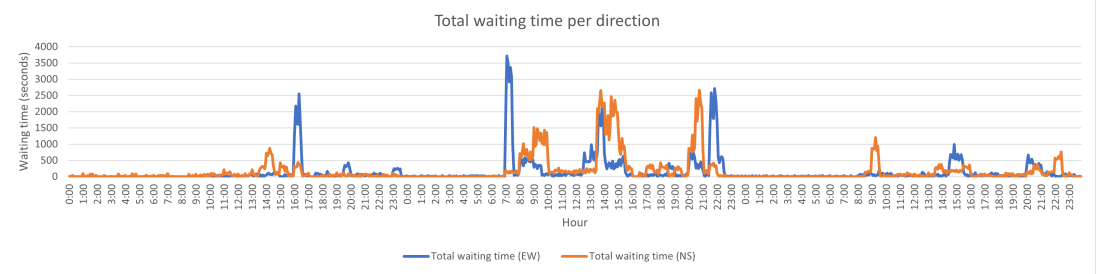
The first noticeable fact is that all the plots show not only one time pattern but the concatenation of the three mentioned time patterns. This can be a little bit conflicting as the amount of information shown on them is much more than in the previous simulation example. This is the main reason why the explanations are going to be more general.

On the one hand, it is possible to see that the plots (a) and (b) are directly related as the waiting time per direction is based on the number of vehicles that pass on a given direction following the same pattern, as in the previous simulation example. Besides, it can be shown that the day with more traffic vehicles and higher waiting time per direction is the second one, which is a regular working day. This is obvious as the other two days are winter and summer vacations, respectively, which are defined to have fewer vehicles. Besides, on (b) it is possible to see that the maximum waiting time in both directions is the almost same, around 2.500 seconds, located on the second day.

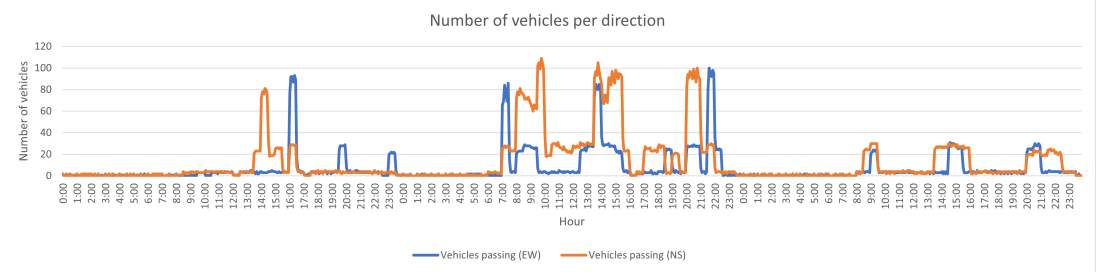
On the other hand, the (c) plots show the average waiting time per vehicle, where it is possible to see that until the first "peak" hour (from 0:00 to 7:00) the NS direction has some "peaks" which indicates that there are vehicles that wait a lot of time. This fact has also been described in the previous simulation example, and occurs on the three days, with a high variance from 0 to 40 seconds. In summary, it can be seen that the average waiting time per vehicle is almost stable for the rest of the day hours, being the higher difference the "peak" values on the EW direction, about 30 seconds more.

The next step is to use the most basic adaptation, using the traffic light predictor based on the date information, the "Only predictions based on date adaptation". In this case, the plots shown, in Figure 5.12, are the same as the previous one, along with the prediction of the traffic type, in order to compare it with the predefined time patterns.

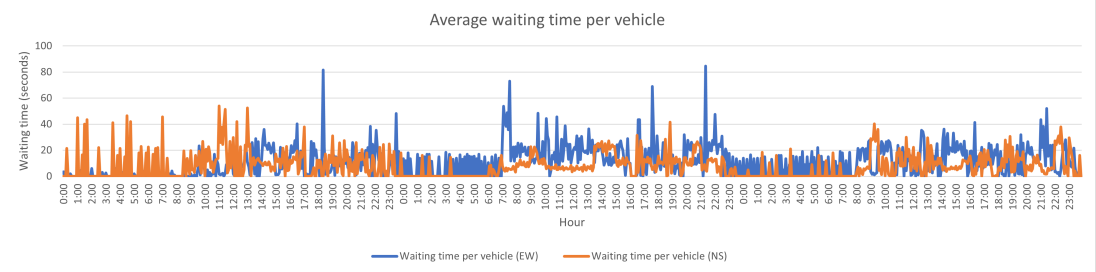
As can be seen on (a), in the first day the total waiting time of the NS direction on the range from 13:00 to 15:00 is highly reduced, more than half, while the time of the EW direction on range from 16:00 to 17:00 is even a little bit greater. On the second day, it can be seen that globally, all the curves on the NS direction have been smoothed and reduced, at least, to its half while the ones at EW direction are a little bit higher at 7:00, 13:00 and 22:00, adding on average 500 seconds on these "peaks". Note that the hour range that has been reduced (on NS) the least is the one with the high number of vehicles in both directions. On the third day, the only relevant fact to mention is that the little curve on NS from 9:00 to 10:00 is slightly increased as the curve on EW from



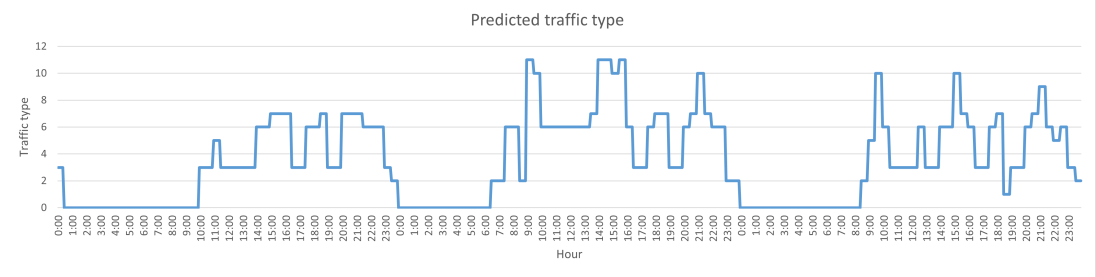

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Predicted traffic type patterns.

*Source:* Documentation owner.

Figure 5.12: Date interval simulation - Only predictions based on date adaptation.

14:00 to 15:00, but this fact could be happening by the noise generation.

In this case, the number of vehicles (b) simulated is not exactly the same as in the base example, but in summary, the numbers are very close, following the same pattern which is the most relevant fact. Although the number is not the same, the average

number of vehicles per hour is almost the same, as it balances with higher and lower values e.g. on the second day on NS direction from 8:00 to 10:00 or from 20:00 to 21:00.

Based on the average waiting time per vehicle (c), it is possible to see that the starting values of each day follow the same pattern as in the base example. It can also be seen that the average time on the NS direction is decreased while the EW time is increased a little bit. In fact, notice that this improvement only happens on the range from 7:00 to 10:30 of the second day, which matches with the hours where the traffic flow is much higher on NS rather than EW direction. Another fact is that on the second and third days, the starting values on the direction NS are switched by those on the direction EW, meaning that these vehicles are randomly generated and sometimes they wait more time than the average because of the randomness of the generation.

The predicted traffic type (d) is similar to the predefined time pattern but is much "squarer", making the prediction more inaccurate. In fact, there are predictions going from a very low to a much higher value, that affects directly the adaptation of the traffic light algorithm and so on the results, e.g. on the second day at 8:00, that in fact is the highest value of the total waiting time on EW direction on all the simulation. Note that there are some predicted traffic types that are not realistic as the first prediction made or the variation on the third day at 12:00.

In terms of values, the highest value of total waiting time on a given direction is reduced to 2.700 seconds on NS and increased to 3.500 seconds on EW, while the highest number of cars is 100 on both. Note that as there is more than one time pattern, these values are not the only ones relevant to the simulation but the highest. On average, the waiting time per vehicle is reduced at half on the NS direction (around 50%) on NS, and increased (around a half, 150%) on EW on the range from 7:00 to 10:30 of the second day. Before and after these ranges, the average waiting time is almost the same on both days and approaches.

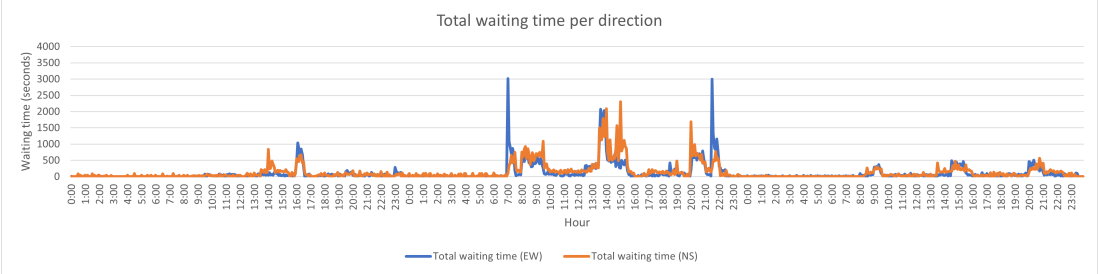
Based on all the plots and values shown previously, it can be concluded that the use of a predictor based on date, in this case, reduces the overall waiting time and balances

the waiting time curves on NS direction, which is the main direction on the scenario, while increasing a little bit the waiting time on EW as there are much fewer vehicles on that direction on the range of hours indicated previously; although the predictor based on date is not very accurate and sometimes it would not predict the traffic type correctly, and then, not adapting as well as expected in real-time. In fact, there is a bad prediction of the traffic type that makes the vehicles from EW direction waits for around 3 times more than the average, on the second day at 8:00, showing that a bad prediction can have a huge negative impact on the adaptation.

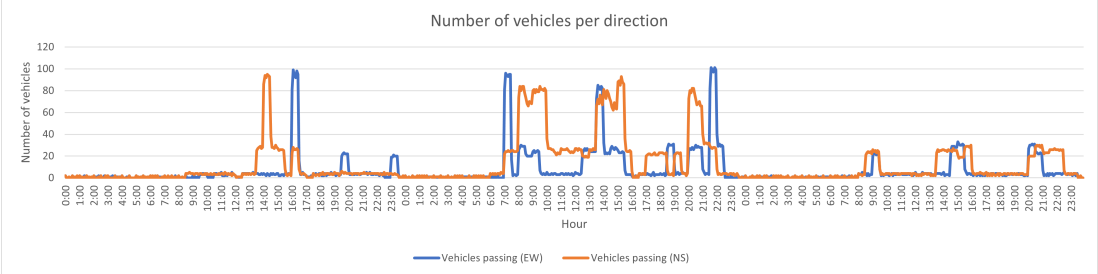
The next approach is related to the use of a real-time analyzer with traffic contextual information (**"Only real-time analyzer adaptation"** approach). In this case, the analyzed traffic type is also shown in the plots in Figure 5.13, along with the other plots mentioned previously. It is worth mentioning that this example will be compared directly with the previous one (date-based predictor), not with the base one, as it improves the behavior of that example.

On (a) it is possible to see that both lines, representing each one of them in a single direction, are very similar, even with the same results on some hour ranges. The most relevant fact to mention is that the waiting time one both directions is almost the same and the curves (in the NS direction) of all the "peak" hours are flattened and reduced to the ones in the EW direction. In this case, the use of the analyzer also makes the EW direction curve to be reduced highly even making it proximate to 0 as in the first day (at 19:00 and 23:00). In general, this approach reduces both waiting time on NS and EW, as it tries to balance the traffic the best possible based on real-time information. The second day is the one that is the most affected by the use of this approach as the two curves are almost the same compared to the previous example where they were very different. It is also worth mentioning that the "peak" hours (7:00 and 22:00) are smoothed a little bit but there are not as reduced as expected. So, it is possible to state that, in this case, the use of the analyzer reduces the total waiting time on both directions considerably and smooths the waiting time curves.

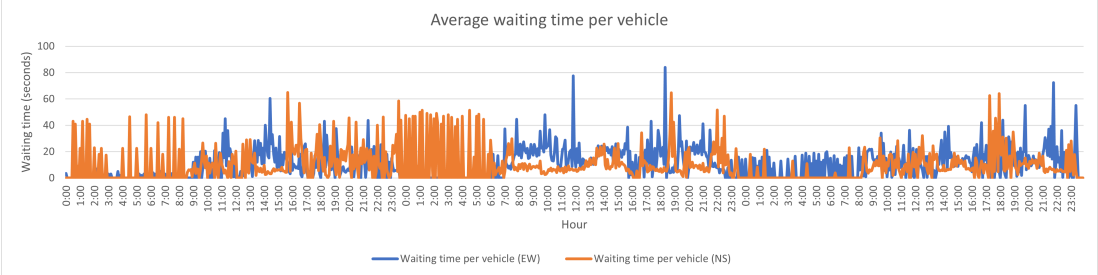
On (b) there are represented the number of vehicles per direction, which is very



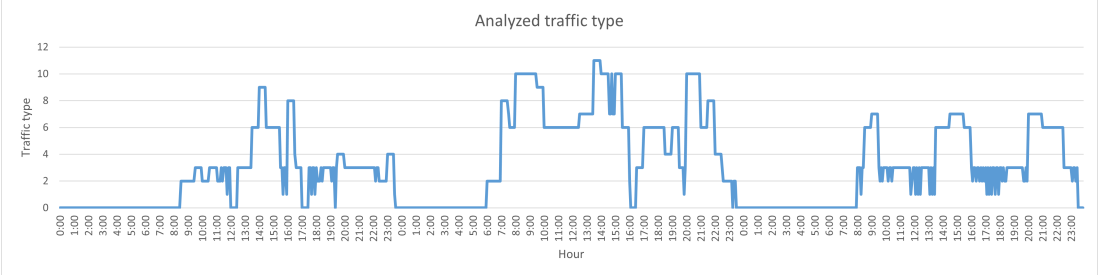

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Analyzed traffic type patterns.

*Source:* Documentation owner.

Figure 5.13: Date interval simulation - Only real-time analyzer adaptation.

similar to each example on average. As can be seen, the number of vehicles is not reduced considerably, as its average is almost the same in both examples, but the total waiting time per direction is reduced and it does not follow the same pattern as it occurs on the base example. Therefore, it can be said that the adaptation is working fine and

as expected.

The average waiting time per vehicle (c), is very similar to the previous example, in fact, the value is almost the same and the only relevant fact to mention is that there is a little more variation on this approach than in the previous one, especially on the last day from 17:00 to 19:00; this might be caused due to a higher number of modifications on the traffic light algorithm. Besides, in this case, the "noise" at the beginning of the second day is the same as in the first day (NS) while in the previous example it was from EW direction.

Lastly, the analyzed traffic type (d) in this case is less accurate than the date-based predictor on the first day while in the other two the traffic type analyzed is better than the predicted. This can be caused due to the randomness of the generated vehicles on low type ranges. Although the analyzer gets more realistic traffic type values, there are some values with "noise" or too much variance in a given range (e.g. fro 10:00 to 13:00 on the third day), that indicates that it does not fit perfectly to the real traffic and it could be, somehow, improved.

In terms of values, the highest value of total waiting time on a given direction is reduced to 2.200 seconds on NS and reduced to 3.000 seconds on EW, while the highest number of cars are 90 and 100, respectively. Despite that the waiting time on EW is not too reduced compared to the date-based predictor example, it is only a single value and the rest of the values are slightly lower; and on the NS direction, there is a great improvement as the waiting time is much less than in the previous approach. On average, the waiting time per vehicle is almost the same on the NS direction as in the previous example but with a higher variance of around 10 seconds, on some punctual hours (last day from 17:00 to 19:00) but not globally.

Based on all the plots and values shown previously, it can be concluded that the use of a real-time analyzer improves the adaptation behavior of the system compared to the date-based predictor, as it reduces the overall waiting time on both NS and EW direction, but it can not reduce the "peak" values as good as expected; although the analyzer fits better to the real-time conditions (in fact it works better than the predictor),

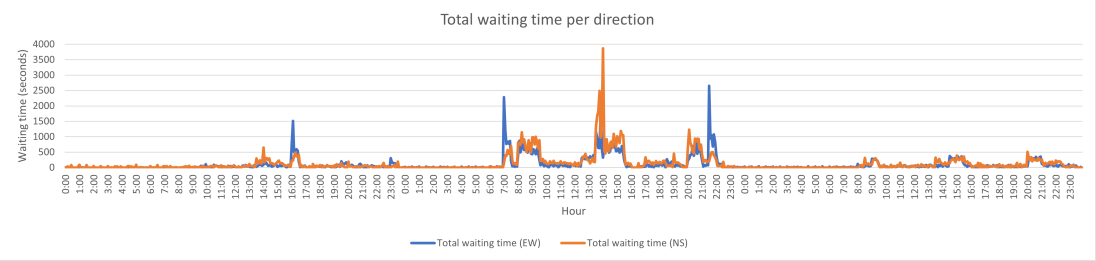
is not very stable and it should be considered to be improved. Besides, it flattens, even more, the total waiting curves, reducing also the lowest curves, those that the predictor was not able to reduce.

The last approach is a combination of both the real-time analyzer and a context-based predictor, which in fact is an improvement of the date-based predictor, by using traffic contextual information (**"Both analyzer and contextual predictor adaptation"** approach). In this case, the analyzed and predicted traffic types are also shown in the plots in Figure 5.14, along with the other plots mentioned previously. It is worth mentioning that this example will be compared directly with the previous one (real-time analyzer), not with the base one.

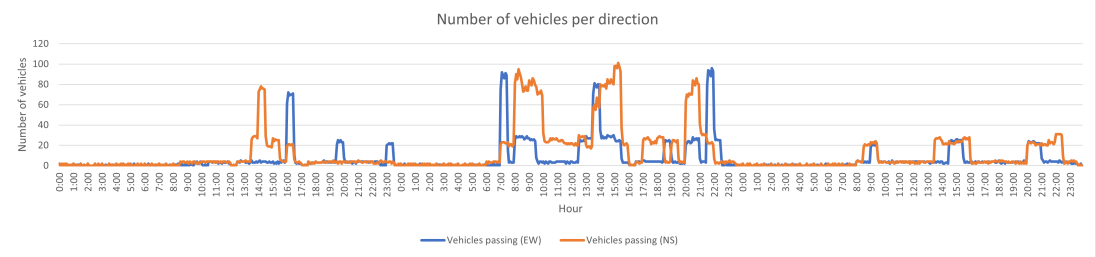
On (a) it is possible to see that both lines, representing each one of them a single direction, are very similar between them, even with the same results on some hour ranges as in the previous example. In fact, the main differences with that example are that the "peaks" of waiting time on the EW direction are reduced and that there is another "peak" on the NS direction on the second day at 14:00, that could be even considered as a random noise because it only occurs only time on all the simulation.

This behavior can be thought to happen because of the randomness of the traffic and the number of vehicles used but as it is shown on (b), this number is higher on both directions (EW and NS) on the second day, while the rest of vehicles is almost the same on both examples, meaning that the adaptation behavior can be affected by the number of vehicles but the global behavior will remain the same. Besides, as in the previous example, the number of vehicles is not reduced but the total waiting time per direction is reduced and it does not follow the same pattern as it occurs on the base example, so it can be concluded that adaptation is working fine and improving slightly the previous approach, although there are some randomly generated "peaks" that could be used to define some improvements. This slight improvement does not mean that it will work better than the previous one in other cases, in fact, sometimes the previous adaptation approach would be better.

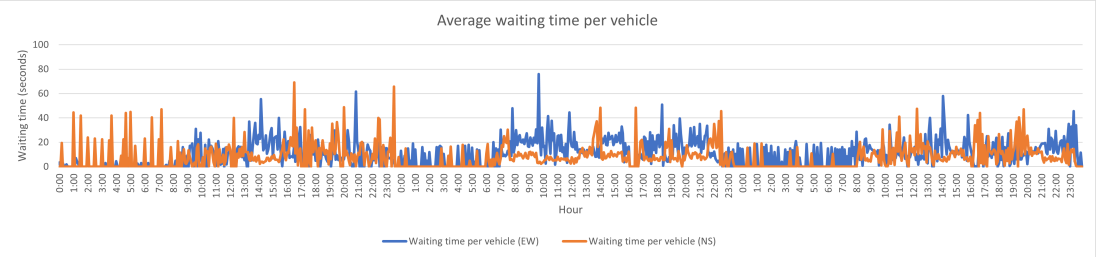
The average waiting time per vehicle (c), is a little bit higher in the EW direction



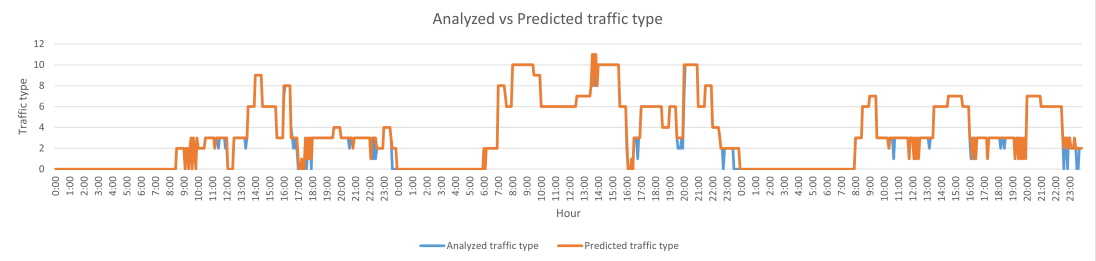

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Analyzed traffic type patterns.

*Source:* Documentation owner.

Figure 5.14: Date interval simulation - Both analyzer and context predictor adaptation.

while remaining almost the same in the NS direction. Despite this, in general, it is possible to see that the variance of both directions is less than in the previous example, but the real difference between the two approaches is not noticeable. Moreover, on the second day, the random "noise" is again from EW rather than from NS as in the

previous example.

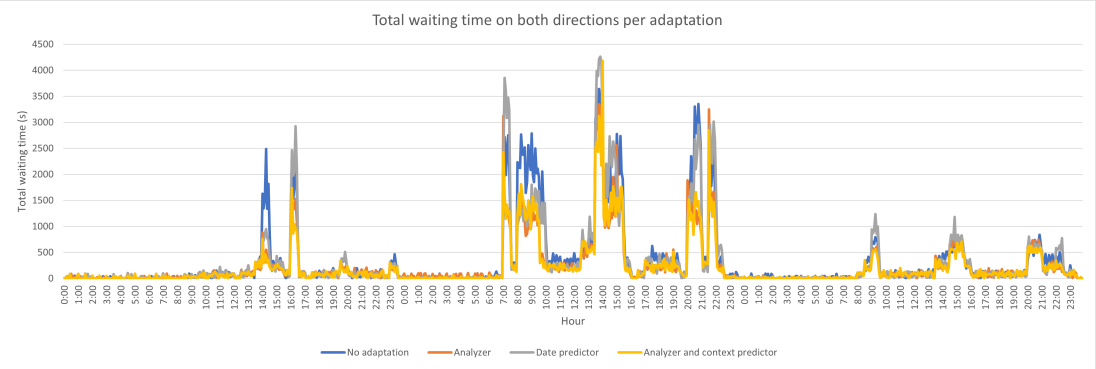
Lastly, the analyzed and predicted traffic types (d) are very similar between them, even retrieving the same value in most of the situations, where those different values are slight variations of the real ones. In fact, it is possible to see that the use of contextual information, in this case, can affect positively the predictor as in the date-based predictor, the values predicted were much worse than in this approach. The difference between the analyzer and the predictor is not high enough to notice real variations on the adaptation. Furthermore, the analysis/prediction output is more similar to reality than the output of the analyzer standalone.

In terms of values, the highest value of total waiting time on a given direction is reduced on average to 1.300 seconds but the "peak" value turns into 3.800 seconds on NS and is reduced to 2.500 seconds on EW, while the highest number of cars is 100, on both directions. On average, the waiting time per vehicle is almost the same in both NS and EW directions where the difference is the reduction of the variance of the values and is not even noticeable.

Based on all the plots and values shown previously, it can be concluded that the combination of both analyzer and context-based predictor increases a little bit the adaptation performance but the real difference between using this approach and the previous is not easily noticeable, unless if they do not appear random values as it appears on this example. In fact, the most relevant improvement of this approach is that the context-based predictor performs much better than the date-based one, as it uses contextual information, and even improves the analyzer itself. In fact, this approach does not flatten more any curve compared to the previous approach, indicating that in fact, is not possible to improve even more the adaptation behavior, unless the analyzer is better adjusted, as it does not fit very well the real-time traffic patterns.

Once all the adaptation examples have been described and shown their results, there are all gathered into a single plot (on Figure 5.15) in order to **compare directly** between them and to see which one is the best adaptation approach for this time pattern. Note that the information compared is the sum of the total waiting time in

both directions.



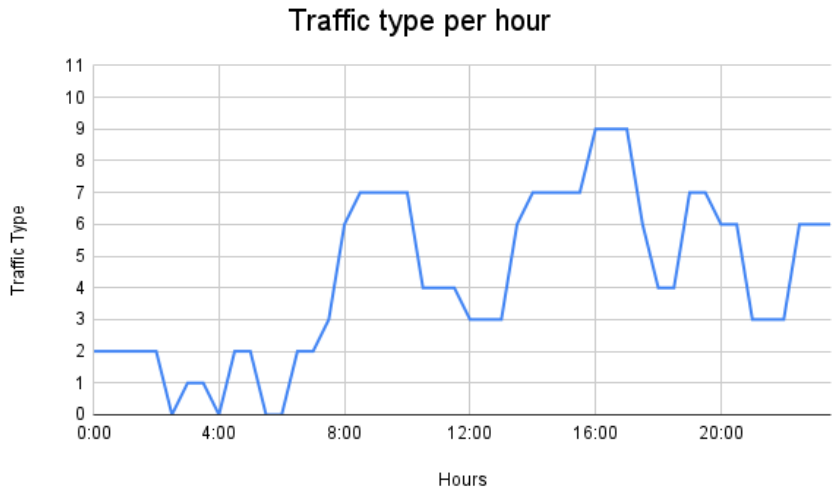

*Source:* Documentation owner
Figure 5.15: Date interval simulation - Total waiting time summary

As in the previous simulation example, all the adaptation approaches reduce significantly the total waiting time on the "peak" hours on the different days where the maximum traffic flow is "high" on NS and "medium" on EW. This means that the adaptations prioritize the direction with high vehicle density. On all the days it is possible to see that the analyzer-use approaches are the best ones, and even the date-based predictor approach is worse than the no-adaptation approach. In fact, it is possible to see that in some cases is not relevant the approach used as the number of vehicles on that hours is low enough to affect the overall waiting time, this occurs mainly on the first and third days, where there are not high traffic densities. On the second day, the best approaches are both the analyzer and the combination of the analyzer and context-based predictor in general, as they reduce the curve compared to the other approaches.

#### 5.2.4.3 Random day simulation

Firstly, the defined pattern to be used in this simulation is randomly generated by hand, not following any kind of real behavior, just setting different traffic types over a day. This pattern is shown in Figure 5.16 and it has been defined this way in order to check the behavior of the predictors when there is used a non-existing time pattern on the training dataset. In this case, the total elapsed time on the execution of the example is

around 9 minutes as the deployment of the components and installation of the required libraries on each one of them last around 3 minutes; and the simulation lasts around 6 minutes.

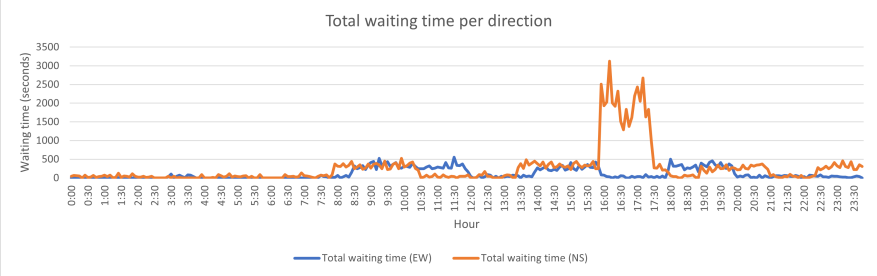



*Source:* Documentation owner
Figure 5.16: Random day simulation - Time pattern.

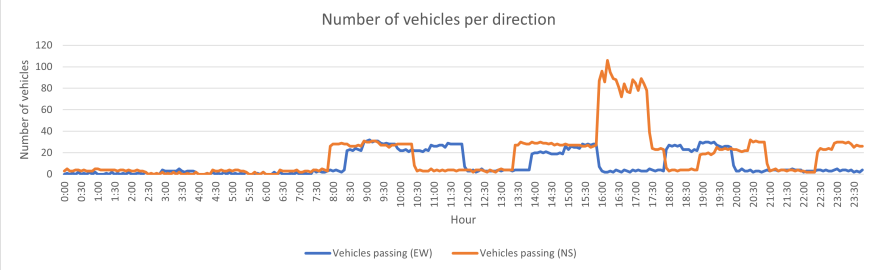
The process followed in this simulation example is the same as in the previous ones, starting from the "no adaptation", followed by the "date-based predictor" and "real-time analyzer", ending with the "combination of the real-time analyzer and context-based predictor" approaches.

In Figure 5.17 it is possible to see all the plots related to the "No adaptation" approach, which will be considered as the base example in order to compare the results between the different adaptations as this simulation is like the real-life approach, with static time phases and no adaptation.

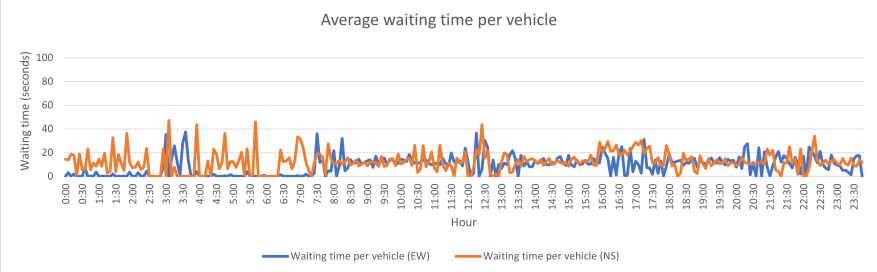
On the one hand, it is possible to see that the plots (a) and (b) are directly related, as in all the previous simulation examples, due to the fact that the waiting time per direction is based on the number of vehicles that pass on a given direction, in fact, the pattern is very similar on both plots and it is adjusted to the input time pattern. The most remarkable fact is that there is only one range when the waiting time is much higher on NS than on EW, from 16:00 to 18:00. Furthermore, note that the traffic is




(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.

*Source:* Documentation owner.
Figure 5.17: Random pattern simulation - No adaptation.

mostly balanced except for this range.

On the other hand, the (c) plot shows the average waiting time per vehicle, where it is possible to see that until the first "peak" hour (from 0:00 to 7:00) the NS and EW directions has some "peaks", which indicates that there are vehicles that wait a lot of time. This fact has also been described in the previous simulation example. In summary, it can be seen that the average waiting time per vehicle is almost stable on the rest of the hours, being the higher difference from 16:00 to 18:00, where the NS vehicles wait almost half of the EW vehicles.

In summary, the highest value of total waiting time on a given direction is around 2.500 seconds on NS and 500 seconds on EW, while the higher number of cars are

100 and 30, respectively. On average, the waiting time per vehicle is almost equally distributed.

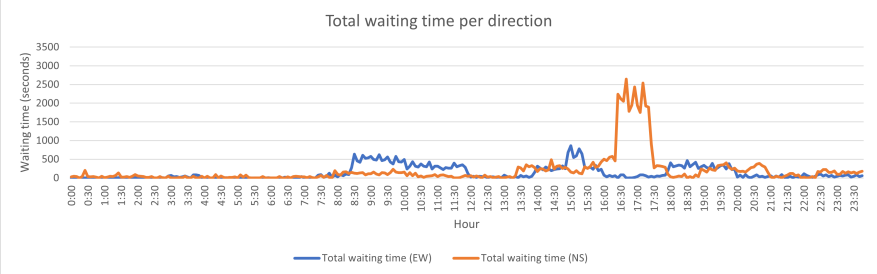
The next step is to use the most basic adaptation, using the traffic light predictor based on the date information, the "Only predictions based on date adaptation". In this case, the plots shown in Figure 5.18 are the same as the previous one, along with the prediction of the traffic type, in order to compare it with the predefined time patterns.

As can be seen on (a), the NS waiting time curve is almost flattened to 100 seconds in the range from 8:00 to 10:30 and from 22:30 to 23:00, not affecting at all the waiting time in the EW direction. The main NS "peak" (from 16:00 to 17:30) is not even reduced while there are some hours where the EW waiting time is increased slightly as at 15:00, but this can be caused by the traffic noise. In this example, the number of vehicles (b) simulated is almost the same as in the base example and they follow the same pattern.

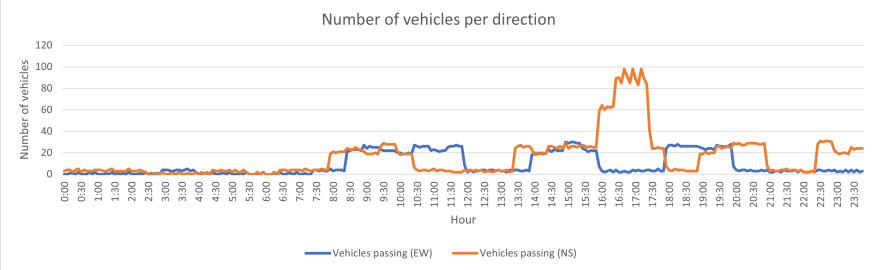
Based on the average waiting time per vehicle (c), it is possible to see that the starting waiting time follows the same pattern as in the base example. It can also be seen that the average time on the NS direction is decreased while the EW time is increased a little bit. In fact, notice that this improvement only happens on the range from 8:00 to 10:30, which matches with the hours where the waiting time on NS has been flattened. The rest of the average waiting time is almost the same as in the previous example but with a higher variance.

The predicted traffic type (d) is not even similar to the input time pattern, which means that the prediction is not valid, even if there is a little improvement on the waiting time of the NS direction as this could have happened randomly and not by the use of the predictor. In fact, this behavior is expected as the defined new pattern, or even something similar is never seen by the predictor. Furthermore, as the default value of date, when there is not used the calendar, is defined to be a Monday, the predicted traffic type coincides with that traffic time pattern, as can be seen.

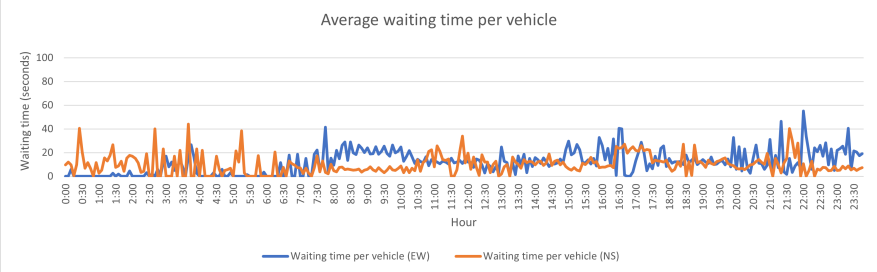
In terms of values, the highest value of total waiting time on a given direction is the same, around 2.500 seconds on NS and increased to 800 seconds on EW, while the



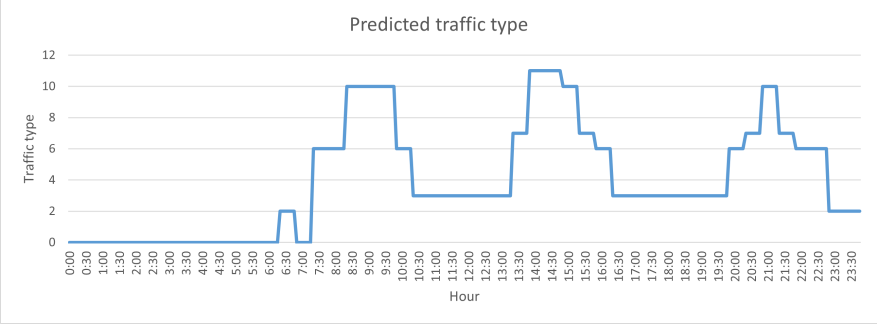

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Predicted traffic type pattern.

*Source:* Documentation owner.

Figure 5.18: Random pattern simulation - Only predictions based on date adaptation.

highest number of cars are 100 and 30, respectively. On average, the waiting time per vehicle is reduced on the NS direction and increased a little bit on the EW direction, within the range from 8:00 to 10:30. Before and after these ranges, the average waiting

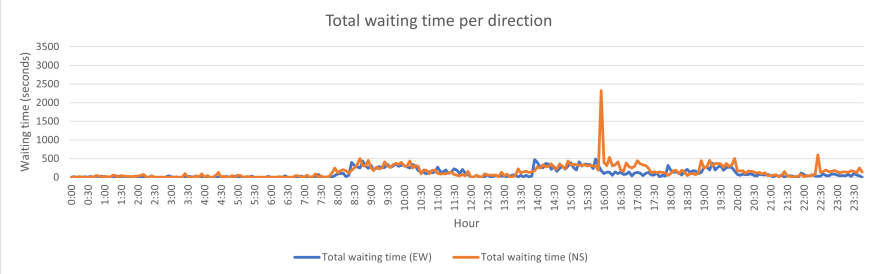
time is almost the same in both examples.

Based on all the plots and values indicated previously, it can be concluded that the use of a predictor based on date, in this case, reduces a little bit the waiting time on NS direction in some ranges but these improvements are more related to the randomness of the traffic rather than the predictor itself, as it does not even get close to the real traffic types evolution. In summary, this approach is not useful as it does not improve the overall waiting time in any direction and the adaptation performed is not based on good traffic predictions as the predictor outputs another time pattern.

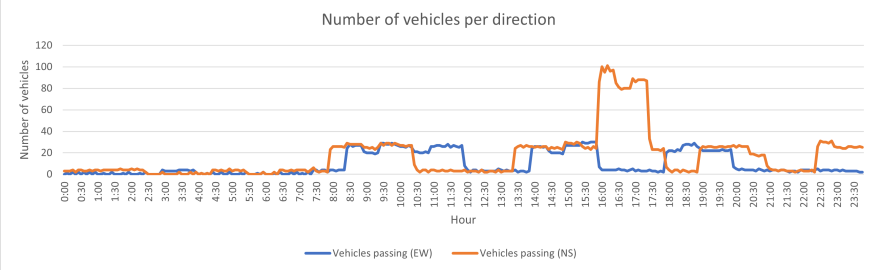
The next approach is related to the use of a real-time analyzer with traffic contextual information (**"Only real-time analyzer adaptation"** approach). In this case, the analyzed traffic type is also shown in the plots in Figure 5.19, along with the other plots mentioned previously. It is worth mentioning that this example will be compared directly with the previous one (date-based predictor), not with the base one, as it improves the behavior of that example.

On (a) it is possible to see that both lines are very similar between them, even with the same results on most of the hours. The most relevant fact to mention is that the waiting time one both directions is almost the same and the curves (in the NS direction) are reduced to the ones in the EW direction, except for the "peak" hour value at 16:00. In this case, the use of the analyzer also makes the EW direction curve to be smoother. In general, this approach reduces a lot the waiting time on NS, while affecting slightly to the EW direction as it tries to balance the traffic the best possible based on the real-time information. In this example, the number of vehicles (b) simulated is almost the same as in the base example but the total waiting time per direction is reduced and it does not follow the same pattern as it occurs on the base example. Therefore, it can be said that the adaptation is working fine and as expected.

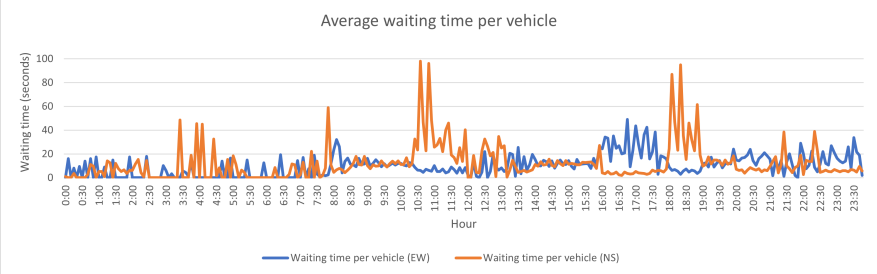
The average waiting time per vehicle (c) is very different compared to the previous one. In fact, the only range that seems to be similar is until 7:00, which is random as is explained in the other examples. In this case, most of the time the average waiting time is almost the same in both directions, while there are some ranges, which coincides



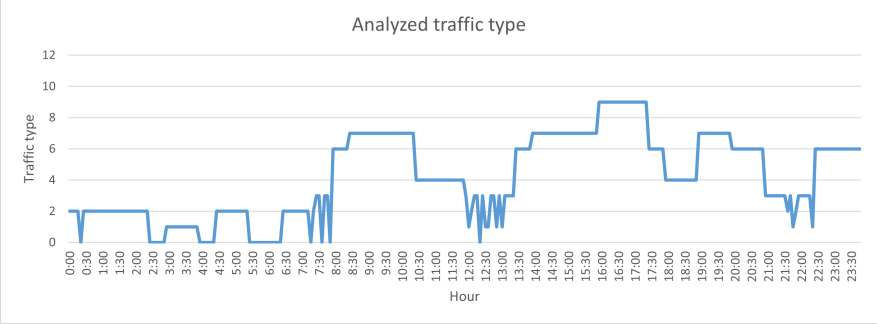

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Analyzed traffic type pattern.

*Source:* Documentation owner.

Figure 5.19: Random pattern simulation - Only real-time analyzer adaptation.

when the traffic on EW is higher than in NS, where the average waiting time on the NS direction is much higher (e.g. from 10:30 to 12:00). This behavior is not happening in any of the previous time pattern examples as in none of them, the traffic on EW is

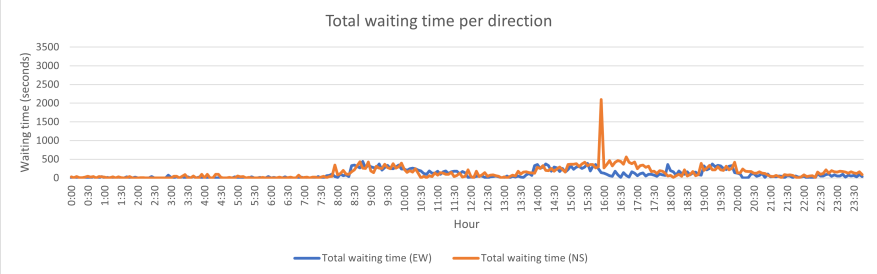
higher than on NS. This condition occurs the same way but is reversed from 16:00 to 17:30 where the traffic is higher on NS than on EW. This is very important because it means that the real-time analyzer will always try to reduce to its minimum the waiting time of the direction with higher traffic, not only the main direction.

Lastly, the analyzed traffic type (d) is much more accurate than the date-based prediction as the pattern analyzed is almost the same as the predefined time pattern, although there are some values with "noise", that indicates that the analyzer does not fit perfectly to the real traffic time pattern but it is too close.

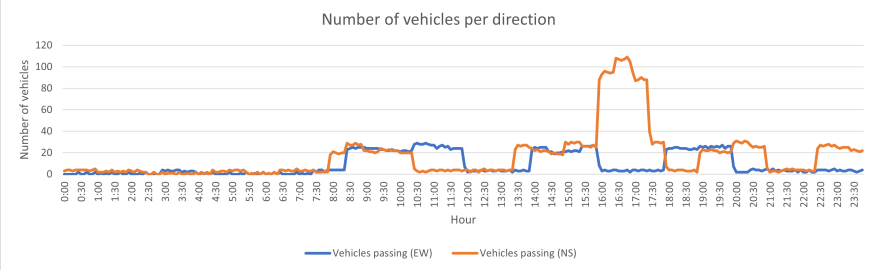
In terms of values, the highest value of total waiting time on a given direction is the same, around 2.500 seconds on NS and decreased to 400 seconds on EW, while the highest number of cars are 100 and 30, respectively. Note that the highest value on NS is not representative as it only appears one time on the simulation and it can be considered, in fact, a "noise" error, and the second-highest value is 500 seconds. On average, the waiting time per vehicle is almost the same in both directions, except for the ranges indicated previously.

Based on all the plots and values shown previously, it can be concluded that the use of a real-time analyzer improves the adaptation behavior of the system compared to the date-based predictor as in this case the predictor is not useful due to its lack of correctness when predicting a random time pattern. Besides, the analyzer flattens all the curves, reducing the waiting time considerably, especially when the traffic is much higher in one direction than the other (e.g. from 16:00 to 17:30). Therefore, this approach is much better than the date-based predictor.

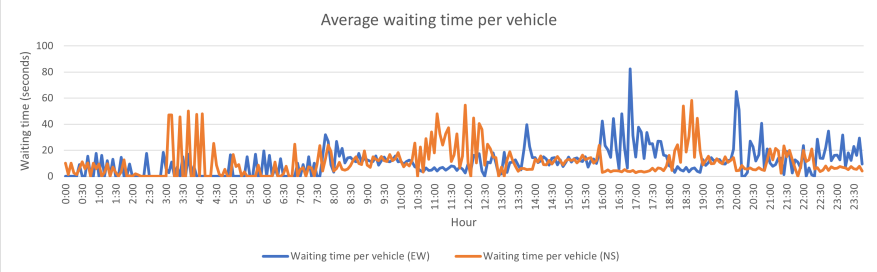
The last approach is a combination of both the real-time analyzer and a context-based predictor, which in fact is an improvement of the date-based predictor, by using traffic contextual information (**"Both analyzer and contextual predictor adaptation"** approach). In this case, the analyzed and predicted traffic types are also shown in the plots in Figure 5.20, along with the other plots mentioned previously. It is worth mentioning that this approach will be compared directly with the previous one (real-time analyzer), not with the base one.



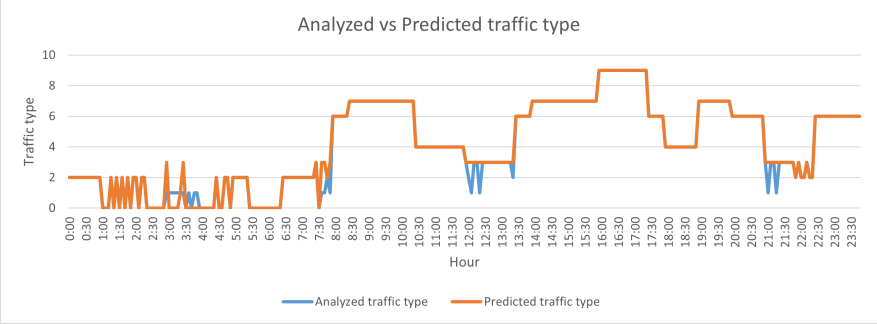

(a) Total waiting time per direction.




(b) Number of vehicles per direction.




(c) Average waiting time per vehicle.




(d) Analyzed traffic type pattern.

*Source:* Documentation owner.

Figure 5.20: Random pattern simulation - Both analyzer and context predictor adaptation.

On (a) it is possible to see that both lines, representing each one of them a single direction, are very similar between them, even with the same results on most of the

hours as in the previous example. In fact, the main difference with that example is that the highest "peak", at 16:00, is slightly reduced on NS.

This slight improvement can be thought to happen because of the randomness of the traffic and the number of vehicles used but as it is shown on (b), this number is almost the same in both examples and directions. Besides, as in the previous approach, the number of vehicles is not reduced on the example but the total waiting time per direction is reduced and it does not follow the same pattern, so it can be concluded that adaptation is working fine and improving slightly the previous approach, but this improvement does not mean that it will work better than the previous one in other the cases.

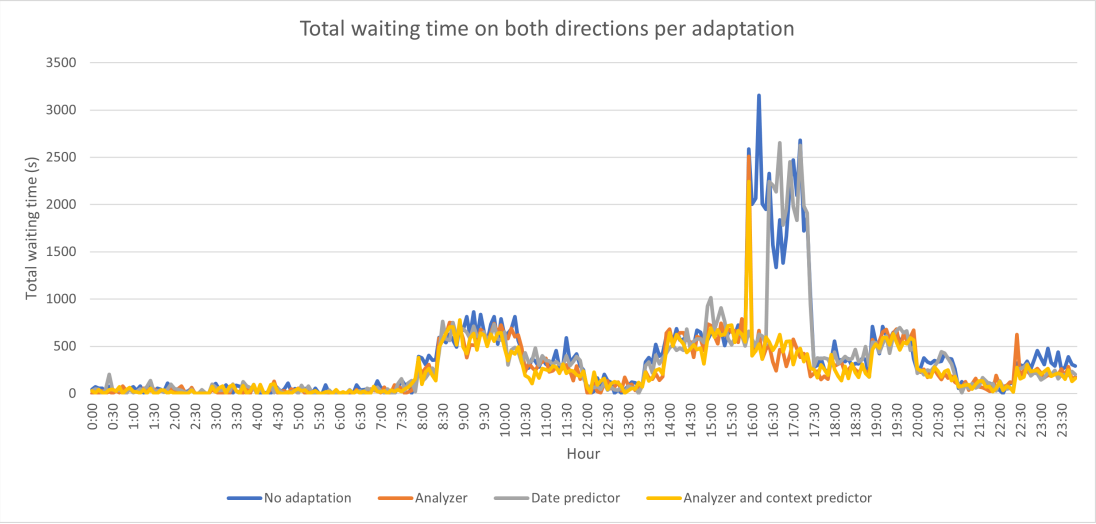
The average waiting time per vehicle (c) follows the same pattern as in the previous example, but in this case, the "peak" ranges are much lower in the NS direction while a little bit higher in the EW direction. These ranges are defined in the previous examples.

Lastly, the analyzed and predicted traffic types (d) are very similar between them but the analyzer does not adapt perfectly to the real traffic type in comparison to the context-based predictor, even if the analyzer itself in the previous simulation retrieve better the traffic type of the range of hours from 1:00 to 2:30. As it occurs in all the previous simulation examples, the most relevant fact to mention about this plot is that the context-based predictor is much better than the date-based one as it fits better to the real traffic types. In fact, the date-based predictor is not useful as its prediction is always wrong, at least for random time patterns. Moreover, the different results between the analyzer and the predictor are not high enough to notice real variations on the adaptation, as their outputs are almost the same.

In terms of values, the highest value of total waiting time on a given direction is reduced to 2.000 seconds on NS and remained on 400 seconds on EW, while the highest number of cars are 100 and 30, respectively. As in the previous example, the second-highest value on NS is 500. On average, the waiting time per vehicle is almost the same in both directions and the only noticeable fact is that the "peak" ranges are reduced a little bit.

Based on all the plots and values shown previously, it can be concluded that the combination of both analyzer and context-based predictor could increase a little bit the adaptation performance but the real difference between using this approach and the previous is not easily noticeable. In fact, the most relevant improvement of this approach is that the context-based predictor performs much better than the date-based one, as it uses contextual information. Furthermore, this approach does not flatten more any curve compared to the previous approach, indicating that in fact, is not possible to improve even more the adaptation behavior.

Once all the adaptation examples have been described and shown their results, there are all gathered into a single plot (on Figure 5.21) in order to **compare directly** between them and to see which one is the best adaptation approach for this time pattern. Note that the information compared is the sum of the total waiting time in both directions.




*Source:* Documentation owner.
Figure 5.21: Random pattern simulation - Total waiting time summary.

Firstly, all the analyzer-based adaptation approaches reduce significantly the total waiting time on the "peak" hour (from 16:00 to 17:30) where the maximum traffic flow is "medium" on NS and "high" on EW. This means that the adaptations prioritize the direction with high vehicle density. In general, both non-adaptive and date-based

predictor adaptation obtains almost the same results.

Secondly, except for the previous range, the difference between the no adaptation approach and the rest of them does not exist as the total waiting time is very similar in all the cases. In fact, this means that the total waiting time is not reduced at all, it is only adapted to be divided into directions based on its traffic density.

Concluding with this example, it can be seen that the date-based predictor is useless as the used pattern is a randomly generated one and that the contextual information is even more useful on these kinds of patterns than the first adaptive approach, the previously mentioned, as it does not perform any valid adaptation.

#### 5.2.4.4 Results discussion

Once all the simulation examples have been described, executed and discussed its results, it is going to be summarized in this section the main conclusions obtained from these, based on the different approaches. These are:

- The **"no-adaptation"** approach is the one used as the basic example as there is no adaptation and therefore it is the best one to represent the real-life scenarios. Sometimes this approach can be even better than the next one, due to its bad predictions.

- The **"date-based prediction"** approach is useful when the input time pattern is one of the used in the training dataset generation as it, more or less, predicts valid traffic values due to the fact that it is "known" by the predictor. In fact, it reduces, overall, the total waiting time on the direction with more traffic density, even if it is not the best approach but if there is no real-time contextual information, this approach is good enough to replace the no adaptation approach. On those time patterns "unknown" by the predictor, as occurs in the last simulation example, it results in a useless component and that even can make the system behave worse than the no adaptation approach.

- The **"real time analyzer"** approach is one of the best approaches as it adjusts

the highest waiting time direction curve to the other one, meaning that the traffic light algorithm has a green time phase larger in that direction than in the other. With this approach is easily observable the relevance of the contextual information. Another relevant fact is that this approach does not require a previous training process of any of its components, which makes it an "on the fly" approach and usable in almost any scenario.

- The **"both context-based predictor and analyzer"** approach is the other approach that can be the best one, as its results are almost the same as the previous approach. In fact, this approach truly shows the power of contextual information, due to the fact that even the context-based predictor improves the behavior of the real-time analyzer. On the contrary to the previous approach, this one requires a previous "off-line" training process, which in fact requires a data-gathering process that might not be possible to perform in a real scenario.

# Chapter 6

# Conclusions and Future Works

This section outlines the main conclusions, both technical and personal, from this work, along with the problems found while analyzing, designing and implementing the SmartTLC framework. In addition, this section also draws some future research lines planned to be covered as part of my PhD Thesis.

## 6.1 Technical conclusions

This section outlines the main technical conclusions derived from the development of the Project and also from the results obtained during the experimental phase.

Firstly, the results presented and discussed in Chapter 5 demonstrate that, even using history-based predictions build only on dates and hours of the day, it is possible to adapt the traffic light control system to achieve slightly better results than using a classical non-adaptive strategy.

Secondly, the results also show that training the traffic pattern prediction component also with historic traffic-based information (in addition to dates and hours of the day) achieves much better results and enables a more accurate traffic-light adaptation.

Thirdly, the use of real-time context data to predict the current traffic pattern, has demonstrated to significantly improve the results achieved by the two former

approaches. This clearly indicates that, as expected, recent data is more relevant than historic data to predict what is coming immediately next. However, it is worth noting that real-time traffic data is not always available (e.g., the monitoring infrastructures may fail or, simply, may not exist). In these cases, the possibility of predicting traffic evolution based only on (real-world or simulated historic data) can achieve a good enough improvement.

Finally, the experiments show that combining both real-time and historic context data achieves the best predictions and traffic light control adaptation.

It is worth noting that all the experiments have been executed locally with all the components running on a single machine (not really distributed). This results in almost instantaneous data communication and actuation. However, in a real-world scenario, delays in context data availability or adaptation decision execution may occur. These delays may cause a sub-optimal performance and thus, should be minimized in as much as possible. In fact, the predictor models should be regularly (but not too frequently) re-trained always offline, to avoid introducing any extra-computational load which may cause additional delays.

It is also worth mentioning that all the features required for SmartTLC have been implemented (actually, some additional ones were included during the Project lifetime) and the framework has been successfully tested on all the scenarios initially considered.

## 6.2 Problems found

The research nature of the Project required facing some novel challenges, which implied adjusting some of the initial goals and tasks. This sections reviews these problems, dividing them into two blocks:

- **Definition problems**
  - When the traffic scenario was initially outlined, we defined the number of vehicles to be simulated per hour for each traffic pattern. However, after simulating the initially generated dataset (which took around 16 hours), we

realized that some vehicles had waiting times of more than 20 minutes, which was not realistic at all. Thus, we had reduced the number of vehicles initially considered and regenerate the dataset.

- The minimum value of a green/red time phase was initially defined to be 5 seconds. However, after executing the traffic light study we realized that only one vehicle was allowed to cross the junction in each phase. Thus, we decided to increase this value to 20 seconds.
- We realized that it was not realistic to allow certain sequences of traffic patterns (e.g., from "very low" on both NS and EW directions to "high" and "medium" on NS and EW, respectively.
- We initially considered using a Reinforcement Learning (in fact, Deep Reinforcement Learning) approach for optimizing the traffic flows. However, as our vision and comprehension of problem evolved, we realized that the problem was better shaped for a Machine Learning approach. Thus, we decided to consider several Machine Learning algorithms to be able to compare and select the one producing better results.
- Some of the components of the architecture were redesigned several times. For instance, the Traffic Light Controller, which initially included the Traffic Light Predictor, the Analyzer, the simulator and the Traffic Light Adapter (these modules were progressively implemented apart, as independent components).
- Initially, the traffic light control algorithms were defined to use a static time phase with intervals of 5 seconds (first TLS study), but this was modified to better fit the adaptation strategy by using the proportion of vehicles (second TLS study), which in fact, offers much better results.
- We also considered using temporal windows of 30 minutes to retrieve the contextual traffic information, but the results were not good because the greater the window, the more time it takes to adapt the traffic light

algorithm, making it useless (too late adaptation). Based on this results we decided to use a 5-minute window instead.

- **Technical problems**
  - The Eclipse SUMO and the TraCI documentation is too brief and based on non-easy to understand generic explanations. Even some of the concepts mentioned in the documentation are not available on the stable version of these tools, meaning that it is not up to date.
  - The available examples of interaction with SUMO and TraCI are not rich enough to explain some concepts and the TraCI data model is not clear and not referenced in the documentation.
  - There is not any dataset related to the information that is used to test this framework, meaning that it has to be manually generated.
  - There is no existing Docker image for Eclipse SUMO and TraCI tools, so it had to be manually created.
  - The deployment of SUMO docker container instances sometimes requires a lot of the previous configuration in order obtain the expected behavior. There is a lot of information about Docker itself, but for the containers used on this framework, there was not much configuration information.
  - The Traffic Light Predictor dataset has been regenerated several times due to "definition problems", as reported before, but also due to the fact that the trained models achieved an F1 score of 1.0, meaning that they were overfitted and that the dataset did not include noise enough to behave like a real traffic scenario.
  - Eclipse SUMO supports actuated traffic light algorithms which can be configured through several parameters (e.g., distance or time gap between vehicles). However, this algorithm did not work as expected and, as the documentation is not clearer at all, this algorithm was rejected to be used in the framework, at least as part of the work developed on this Thesis.

## 6.3 Future Works

The duration of this Project was quite limited in time and thus, there were some concepts and features (both technical and design-related) that could not be (fully) developed. The most important ones are listed next:

- The scenarios defined for this Project only consider private family-size cars. However, Eclipse SUMO allows to define all kinds of agents including pedestrians and vehicles (both private and public, individual and collective, emergency, motorbikes, bicycles, etc.). Enriching the scenarios with new types of vehicles will allow us to run experiments closer to real-world situations. It would be particularly interesting to consider special adaptation policies for emergency vehicles (fire trucks, ambulances, police cars, etc.) which, on emergency, should reach their destination as soon as possible.

- It would also be interesting to consider self-driven vehicles, as their number is (and will be) increasingly growing in the near future. SUMO currently does not support this kind of vehicles, but tools such as FLOW (described in Section 2.2) do and it can be easily connected with SUMO.

- Supporting additional traffic light control algorithms along with new scenarios as those outlined in Section 3.1. Enabling more complex topologies with more than one junction, and allowing drivers to perform additional actions (e.g., turn both left and right, overtaking, etc.). All these improvements would significantly enrich the framework enabling the simulation of more realistic situations.

- Extend the Traffic Light Predictor with Neural Networks and check their behavior compared to the Machine Learning algorithms already implemented. This could help improve the overall performance of the component, but requires more training data. This is not a big issue in this case, as the dataset is self-generated and it can be extended to be big as needed.

- Improve the Analyzer by defining a formula that fits better with the real-time information, giving more importance to recent data than to the information retrieved longer ago. The inclusion of different Reinforcement Learning algorithms is also an option.
- Use different theorems to better balance the weight of the predictions made by the analyzer and the predictor.
- In case a experimental pilot could be performed in a real-world scenario, the inclusion of security and privacy aspect would become essential.

As already mentioned, this Project lays down the foundations for the Doctoral Thesis I plan to start as soon as I complete the Master Degree. Thus, I hope I can address (at least) some of these features in the near future.

## 6.4 Personal conclusions

This section gathers my personal conclusions regarding this Master Thesis Project.

First, I would like to remark that the development of the SmartTLC framework has been both interesting and challenging, the later due to my lack of background on traffic-related issues.

Besides, using the software architecture I developed for my Bachelor Thesis as an starting point for SmartTLC has taught me that, even the software specifically developed to be easy to extend and adapt, requires quite a bit redesign and reimplemention effort, searching for possible failures (not even foreseen when initially developed), rethinking some features from scratch, etc.

This Project has given me the opportunity to use some of the technologies and tools covered in some of the Master courses, such as Docker or different Machine Learning algorithms. Besides, I have realized how important it is to spend some time selecting those that better fit the problem and the system goals.

I found the methodology followed during the Project very useful. The regular meetings with my supervisor, and sometimes also with other of her Bachelor, Master

and Doctoral students have always been very interesting and helpful. Besides, I have had the opportunity to start supervising myself some Bachelor students and playing both roles (advisor and advisee) during the meeting, has given me a new perspective I really appreciate. I have also met with other teachers and colleagues, also belonging to the Quercus/SPILab research group. These meetings have also helped me a lot as I have discovered new solutions and interesting points of view. Last but not least, meeting non-technology-related friends has also helped me a lot, as trying to explain them what my Project was about required me to do a cognitive exercise in which I also discovered interesting things.

In summary, the design and development of the SmartTLC framework, along with the experimental results and the conclusions obtained from the Project, have made me enjoy it a lot, especially when everything finally worked fine.

I do not consider this Project as the conclusion of my Master Degree but as the beginning of a Doctoral Thesis on adaptive traffic light control strategies for Smart Cities, which I plan to build on the results presented here.

# Appendices

# Appendix A

# Installation guide

In this appendix, it is described the required tools and libraries to successfully deploy and execute the framework architecture and examples. Furthermore, it is explained the installation guide of these tools.

Due to the fact that all the architecture components have been developed as Docker containers, one of the required tools to deploy both architecture and execution examples is Docker, and the other one is Anaconda, which allows executing the components itself locally. Therefore, the installation guide is aimed only at these tools on both Windows 10 and Ubuntu 18.04 LTS operating systems, but before this, it is explained what is Docker, Docker images, Docker containers and Anaconda.

## A.1 What is Docker?

Docker [33] [34] [35] [36] is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages known as containers. The main functionality of Docker is that it can package any application and its dependencies in a virtual container that can run on almost any OS.

The main components of Docker are: (1) the software daemon, which manages Docker containers; (2) objects as containers, images or services; and (3) registries acting as repositories of Docker images. Furthermore, Docker provides several

applications as: (1) Docker Compose to deploy multi-container Docker applications, as the proposed framework; (2) Docker Swarm allowing native clustering; and (3) Docker Volumes to facilitate the independent persistence of data.

As stated previously, one of the most relevant concepts of Docker are Docker containers and Docker images, which are described briefly below:

- **Docker image** [37]: it is a file with a set of instructions to build a Docker container, acting as a template. This image contains application code, libraries, tools, dependencies and other kinds of files required to execute the application. Usually, these images are based on other images and its special feature is that they can be customized easily. Furthermore, they are represented in a Dockerfile, where it is defined the different steps required to create the image and execute it. In the framework there are used both registry (from third-parties) and own images; e.g. influxdb and eclipse-sumo on TLC component, respectively.

- **Docker container** [38]: a container is a standard unit of software that gathers all the code and dependencies required to execute an application; in summary, it is just a runnable instance of an image. These containers can be created, started, stopped, moved or deleted using the Docker API or CLI. Even it is possible to connect the container to one or more networks and even attach persistent storage. The main difference between a container and a virtual machine is that the first one virtualizes the operating system instead of the hardware; making it portable on almost any operating system (Linux, Windows, Cloud, etc.) and more efficient.

## A.2 What is Anaconda?

Anaconda [39] is an open-source Python distribution platform widely used, oriented mainly to data science processes; but it is also available to other programming languages as R. One of its main advantages is that it has a huge number of different libraries and packages preinstalled and available, as the community is growing

day-by-day. It is based on the environment concept, a directory that contains a specific collection of packages installed, which is isolated from other environments and even projects.

## A.3 Installation

The installation process is shown in both Windows 10 and Ubuntu 18.04 LTS, as those are the main operating systems used in the development of the framework. As the docker-compose tool is also used on the framework, it is explained its installation too.

### A.3.1 Windows 10

On Windows 10, Docker [40] is installed along with other tools (as docker-compose) within an application called Docker Desktop. It is worth mentioning that these tools can only be installed on Windows Home, Pro, Enterprise or Education as they allow virtualization. The installation steps are described below:

1. Download the Docker Desktop installer from Docker Hub (linked on [40]).
2. Execute the Docker Desktop Installer.exe by double-clicking it.
3. Follow the instruction on the installation wizard; authorize the installer and proceed with the installation process.
4. Close the installation process when it is successfully ended.

As can be seen, this process is very easy as it uses an installation wizard. In the case of Anaconda, the process followed is almost the same as the previous one, but the main difference is the installation wizard used (available on [41]).

### A.3.2 Ubuntu 18.04

On Ubuntu 18.04 there are several ways to install Docker, but in this case, it is only explained the one that uses the Docker repository [42]. In this case, it is not installed

by default the “docker-compose” tool, so it will be installed alone. The steps required to install both tools are:

1. Update the “apt” package index and install the packages required to use a repository over HTTPS with the following commands:

```
$ sudo apt-get update
$ sudo apt-get install \
ca-certificates \
curl \
gnupg \
lsb-release
```

2. Add Docker’s PGP key:

```
$ curl -fsSL \
https://download.docker.com/linux/ubuntu/gpg \
| sudo gpg --dearmor -o \
/usr/share/keyrings/docker-archive-keyring.gpg
```

3. Set up the stable repository with the next command:

```
$ echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" \
| sudo tee /etc/apt/sources.list.d/docker.list \
> /dev/null
```

Once all these steps have been performed, the Docker tool will be installed on the OS. Then, the next step is the “docker-compose” installation [43], which has only two steps:

1. Download the current stable release of the tool:

```
$ sudo curl -L \
"https://github.com/docker/compose/releases/download
/1.29.2/docker-compose-$(uname -s)-$(uname -m)" \
-o /usr/local/bin/docker-compose
```

Note: the file URL is separated in order to fit in this document, when executing remember to merge it by deleting white spaces.

2. Grant executable permissions on the downloaded file:

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

In this case, the Anaconda installation [44] process is very easy:

1. Download the Anaconda installer linked previously.

2. Execute the next command in a shell, in the folder where the installer is downloaded:

```
bash~/Downloads/Anaconda3-2020.02-Linux-x86_64.sh
```

3. Click “Enter” and type “Yes” to accept the license terms.

4. Select the desired Anaconda installation folder and press “Enter”.

5. Install Anaconda by typing “Yes” when appears the following message: “Do you wish the installer to initialize Anaconda3 by running conda init?”.

6. Restart the shell and Anaconda will be available

Once Anaconda is installed, we are going to create an environment with all the libraries required to execute the SmartTLC framework locally, stored in the “requirements.txt” file in the root folder. This can be done by executing the next commands:

```
# Create environment
$ conda create --name smarttlc --file ./requirements.txt
# Activate environment
$ conda activate smarttlc
```

After these steps, it will be installed both Docker, docker-compose and Anaconda, along with the “smarttlc” environment, in our OS.

# Appendix B

# User manual

In this appendix, it is described the main execution features that provide the SmartTLC framework, along with the full architecture deployment guide. Note that these execution features are related directly to the generation of configuration files e.g. SUMO or Docker configuration files. Although, the Docker configuration files are described in the deployment guide, as they are directly related to the deployment process itself. It is important to mention that all the execution commands shown in this appendix are required to be executed on the "smarttlc" conda environment, described and installed in the Appendix A.

## B.1 Generation features

In order to make the framework itself more configurable and allow to create different scenarios easily, there have been developed several configuration files generators that are described below, along with its execution commands. As some of the generators are used on several components, these will be mentioned in order to clarify their use. These generators are:

- **Traffic Light Programs generator**: it generates the XML file in which are stored all the different traffic lights algorithms that will be used on the SUMO simulation. This generator is available on the "config_generator.py" of the TLS,

TLC and TLP components, and it has the following parameters:

- **–tl-program-file** or **-t**: which indicates the output directory where the file is going to be stored.
- **–interval** or **-i**: indicates the interval used to calculate the phases of the different traffic light algorithms. This approach is related to the first TLS study. This parameter can not be used along with the next one.
- **–proportion** or **-p**: that indicates that the traffic lights phases will be calculated based on the traffic flows proportion defined previously (0.25, 0.50, 1, 2, 4). This approach is related to the second TLS study.

In order to execute this generator use the following commands on any of the previously mentioned components folders, firstly the interval example and secondly with proportions.

```
$ python config_generator.py --tl-program-file \
../net-files/topology/topology.tll.xml \
--interval <seconds>
or
$ python config_generator.py --tl-program-file \
../net-files/topology/topology.tll.xml \
--proportion
```

- **SUMO Configuration generator**: it generates an XML file following the SUMO configuration file schema in which are indicated values or external files as the network topology, traffic lights algorithm, GUI settings and additional components as detectors. The only parameter of this generator is **–sumo-config-file** or **-s**, which indicates the directory where the SUMO configuration file is going to be stored. Note that all the configuration values used for this generator are set as default values, in folder “topology”. In order to execute this generator use the following command, on the previously mentioned

components folders, as this generator is also used by those components and, so on accessed with the "config_generator.py" file.

```
$ python config_generator.py --sumo-config-file \
../net-files/config/simulation.sumocfg
```

- **Vehicles Flows generator**: it generates the vehicle flows file that will be used later in the SUMO simulation indicating the beginning and ending flow times and the number of vehicles per hour defined for that flow. This generator is used only in the simulation process, so it does not have an execution command, and it generates the flows based on the defined traffic type on the time patterns. By default this generator outputs the given XML flows file into the folder "../net-files" of the given component folders, in this case, TLP, TLC and TLS as they all use these flows in its simulation, the same as with the previous generators.

- **Calendar generator**: it parses and creates a ".csv" file where it is stored the calendar that will be used to generate the simulation dataset. In the generated calendar, there is used the noise policy, defined in Section 4.3, and the different hand-made time patterns to generate the calendar based on an existing calendar with bank holidays and working days; which will be used by the SUMO simulator to generate the output training dataset. This generator is only available from the Traffic Light Predictor component and the only parameter used is the **–calendar** or **-c** option, which indicates the input calendar file. The output calendar is generated on a default directory ('../time_patterns/generated_calendar.csv'). The command used is the following one:

  ```
  $ python config_generator.py --calendar \
  ../time_patterns/calendar.csv
  ```

Note: there are other SUMO configuration files such as the network topology that do not have a generator due to the fact that they have been hand-made.

## B.2 Deployment guide

Once all the generators have been described, it is shown the deployment steps followed to execute each one of the architecture components for each kind of simulation example, indicating also its main parameters if apply. This deployment is carried out with the "docker-compose" tool and the steps can be found on the "examples" folder and are:

1. Remove and stop Docker containers if there are any deployed. This can be done with a created script, available on the folder "docker-utils/docker_generator", named as "clean_up_containers.sh".

```
# Move to docker generator folder
$ cd ../../docker-utils/docker_generator || exit


# Clean up existing containers
$ ../clean_up_containers.sh > /dev/null 2>&1
```

2. Select the components that are going to be used on the framework.

3. Generate the docker-compose file by using the docker-compose generator script, indicating the parameters on the components that require them, described below.

```
# Create docker-compose file mosquitto, influxdb and
# grafana for example
$ python main.py -o ../../docker-compose.yml \
-c mosquitto,influxdb,grafana
```

4. Execute the command to deploy all the docker containers:

```
$ sudo docker-compose up
```

5. Wait until all the components all deployed and working successfully.

### B.2.1 Components execution

In this section, there are described the execution commands and parameters of the main architecture components, such as:

- **Recorder**: the main parameters of this component are:
  - **–output-file** or **-o**: indicates the output file where it is stored the experiment. By default, it is named as "record_" plus the current date of the experiment execution.
  - **–topics** or **-t**: indicates the topics that will be subscribed to, in order to store all its information from the middleware. By default, it records all the topics ("#").
  - **–broker-url** or **-b**: indicates the middleware broker url. By default it is defined the middleware docker container IP direction ("172.20.0.2").
  - **–broker-port** or **-b**: indicates the middleware broker port. By default it is defined the middleware docker container IP port ("1883").
- **Player**: in this component, there are three repeated parameters ("broker-url", "broker-port" and "topics") from the recorder, which the first two are related directly to the middleware connection and the third one to the kind of information published. The parameter changed is **–input-file** or **-i**, which indicates the input file from where it is retrieved the experiment data. Note that in both recorder and player, the parameter related to the output/input file, respectively, is defined on the docker-compose generator script with the character ":", but this feature is not used in the system itself.
- **Traffic Light Controller**: in this component, there are four parameters: two related to the SUMO configuration and another two related to the input time pattern used in the simulation. These time pattern parameters are provided in the deployment guide by using the characters ":" and "#", to indicate the type of parameter and the current value of this parameter, respectively.

- **–nogui**: flag to use either the SUMO GUI or not in the simulation process. By default is set to True, which means that the GUI is not used.
- **–config** or **-c**: indicates the location of the SUMO configuration file. By default is "../net-files/config/simulation.sumocfg".
- **–time-pattern** or **-t**: indicates the file where the input time pattern is located. It can not be used along with the next parameter.
- **–dates** or **-d**: indicates the range of dates, retrieved from the previously generated calendar, that will be simulated. The format is dd/mm/yyyy-dd/mm/yyyy, where the first date is the start and the second one is the end, both included.

- **Traffic Light Predictor**: in this case, there are two different execution programs:
  - "dataset_generator.py": related to the generation of the simulation training dataset and its visualization. Its main parameters are:
    - **–num-sim** or **-n**: indicates the number of simulations that will be performed to generate the dataset. By default is 0. This parameter can not be used with the next one.
    - **–time-pattern** or **-t**: indicates the directory where the time pattern, that will be used to generate the dataset, is stored.
    - **–output-file** or **-o**: indicates the directory where the generated dataset will be stored. By default is "../output/simulation_calendar.csv".
    - **–cli-visualize**: enables the command line interface visualization developed to create plots about the generated dataset. By default is disabled.

    Note that there are two arguments ("–nogui" and "–config") that are also used in this generator, but there are not described because they have the same meaning as in the previous component. The command used to execute this generator is:

```
$ python dataset_generator.py --time-pattern \
../time_patterns/generated_calendar.csv \
-c ../net-files/config/simulation.sumocfg \
-o ../output/simulation_calendar.csv --nogui
or
$ python dataset_generator.py --num-sim 2\
-c ../net-files/config/simulation.sumocfg \
-o ../output/simulation_calendar.csv --nogui
or
$ python dataset_generator.py --cli-visualize
```

- "ml_trainer.py": it is related to the training process of the models, a basic prediction feature, and the execution of the full component. In fact, this is the division followed to explain them:
  - Training process: (1) **–train** or **-t**: starts the training process and indicates the input dataset file used to train the models; (2) **–clean** or **-c**: it deletes the models stored previously if enabled. By default is disabled.
  - Prediction process: **–predict** or **-p**: it uses all the models to predict the traffic type from the indicated input, split by comma. Default format is: "passing_veh_e_w, passing_veh_n_s, hour, day, date_day, date_month, date_year". If "date" flag (described below) is active, the fields are: "hour, day, date_day, date_month, date_year".
  - Full component: (1) **–component**: flag to deploy the full component with the trained models. By default is enabled; (2) **–num-models** or **-n**: number of models used to make the traffic type prediction. By default is 1; (3) **–middleware_host**: indicates the middleware broker url. By default it is defined the middleware docker container IP direction ("172.20.0.2"); and (4) **–middleware_port**: indicates the middleware broker port. By default it is defined the middleware docker

container IP port (“1883”).

* Besides, there are is a global argument called **–date** or **-d**, which enables the date-based predictor. By default is disabled, which means that the models will be context-based.

The command used to execute this file is:

```
$ python ml_trainer.py --train \
../output/simulation_calendar.csv -c
or
$ python ml_trainer.py --predict \
10,10,10:30,Monday,01,02,2021
or
$ python ml_trainer.py --component -n 3
```

Note that there are only shown execution examples with the context-based predictors, as the flag “-d” is not used.

- **Traffic Light Study**: as it performs the study itself of the different traffic lights, there are explained both the study script along with its parameters and the full study execution. In the **first script**, the parameters are:

  – **–interval** or **-i**: indicates the interval used to calculate the phases of the different traffic light algorithms. This is the first study approach. It is worth mentioning that this parameter can not be used with the next one.

  – **–proportion** or **-p**: that indicates that the traffic lights phases will be calculated based on the traffic flows proportion (0.25, 0.50, 1, 2, 4). This is the second study approach.

  – **–output-file** or **-o**: indicates the directory where the study results will be stored. Default value is “../output/simulation.csv”

Note that there are three arguments (“–nogui”, “–config” and “–cli-visualize”) that are also used in this generator, but there are not described because they have

the same meaning as in the previous component. In fact, this generator uses similar arguments as the Traffic Light Program generator described previously: "–proportion" and "–interval".

The process followed on the **full study execution** is the next one:

1. Generate the traffic lights algorithms based on the study type (interval or proportion).

```
# Move to traffic light study folder
$ cd ../traffic_light_study/tl_study || exit

# Generate the Traffic Light programs
# by interval of 5 seconds
$ python config_generator.py -t \
../net-files/topology/topology.tll.xml -i 5
```

2. Generate SUMO simulation configuration file.

```
# Generate the SUMO configuration file
$ python config_generator.py -s \
../net-files/config/simulation.sumocfg
```

3. Execute the SUMO simulation with those configuration files and store the output file.

```
# Execute the SUMO simulation
$ python main.py -c \
../net-files/config/simulation.sumocfg -i 5 \
--nogui -o ../output/simulation_interval_5.csv
```

4. Initialize the cli-visualizer to check the study results.

```
# Execute the CLI visualizer
$ python main.py --cli-visualize
```

But, before executing these studies, it is required to have installed all the required libraries, explained in Appendix A, as stated previously, using the conda "smarttlc" environment. Once all the libraries are installed the command used to execute each one of the studies is, on folder "study":

```
$ ./study_interval_5.sh
or
$ ./study_proportion.sh
```

Note that the execution of these components, except for the TLP component, is done with the following command, on the component folder:

```
$ python main.py <parameters>
```

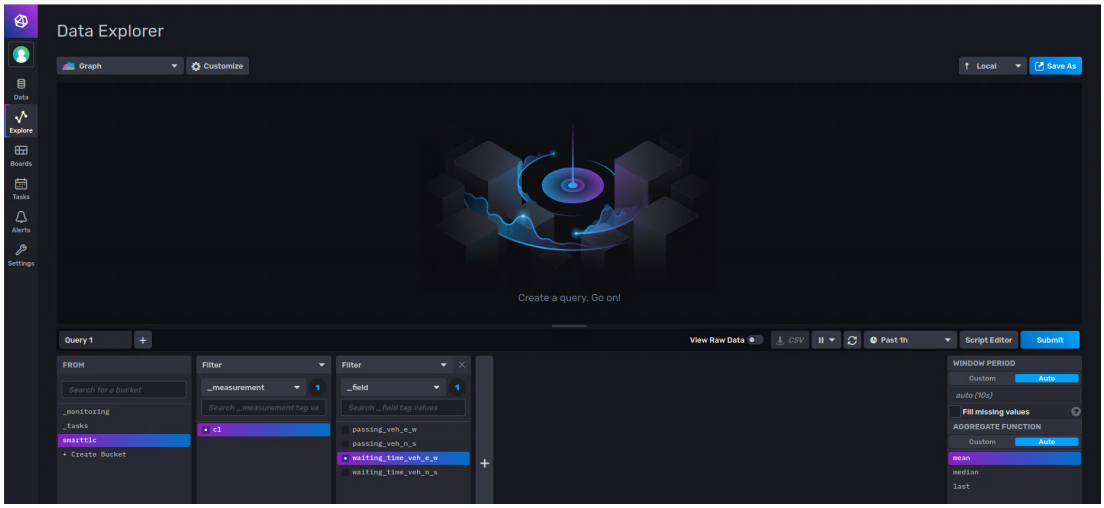
### B.2.2 InfluxDB queries

Once the InfluxDB component is deployed, it is possible to access its web page, allocated on "172.20.0.3:8086", which can allow us to perform queries over the database, in order to check if the information is stored correctly. This web page also provides a plot tool to show all the information based on its timestamp (axis X). The credentials to access the InfluxDB tools are:

- User: sumo-user
- Password: root-secret-password

Once the user accesses the web page with its credentials, it can perform any kind of query on the database by selecting the option "Explore" as can be seen in Figure B.1. Furthermore, it is possible to download the output query with the "csv" button.

In this query builder tool, it is possible to execute several queries in order to obtain different information. In this case, there have been developed three queries which provide three different types of information, each one used for the discussion of the results. These are:




*Source:* Documentation owner.
Figure B.1: InfluxDB web page.

- **Total waiting time**: it retrieves the total waiting time per direction (NS and EW):

```
from(bucket: "smarttlc")
    |> range(start: v.timeRangeStart, stop:
    v.timeRangeStop)
    |> filter(fn: (r) => r["_measurement"] == "c1")
    |> filter(fn: (r) => r["_field"] ==
    "waiting_time_veh_n_s"
    or r["_field"] == "waiting_time_veh_e_w")
```

- **Total number of vehicles**: it retrieves the total number of vehicles per direction (NS and EW):

```
from(bucket: "smarttlc")
    |> range(start: v.timeRangeStart, stop:
    v.timeRangeStop)
    |> filter(fn: (r) => r["_measurement"] == "c1")
    |> filter(fn: (r) => r["_field"] == "passing_veh_n_s"
    or r["_field"] == "passing_veh_e_w")
```

- **Average waiting time**: it retrieves the average waiting time on the NS. In order to get the average of EW direction, the character's "n_s" must be changed to "e_w":

```
from(bucket: "smarttlc")
    |> range(start: v.timeRangeStart, stop:
    v.timeRangeStop)
    |> filter(fn: (r) => r["_measurement"] == "c1")
    |> pivot(rowKey:["_time"], columnKey: ["_field"],
    valueColumn: "_value")
    |> map(fn: (r) => ({ r with _value:
    if r.passing_veh_n_s > 0 then r.waiting_time_veh_n_s
    / (r.passing_veh_n_s) else 0.0}))
```

### B.2.3 Grafana configuration

Once the Grafana component is deployed, it is possible to access its web page, allocated on "172.20.0.4:3000". The credentials to access the Grafana visualization tools are:

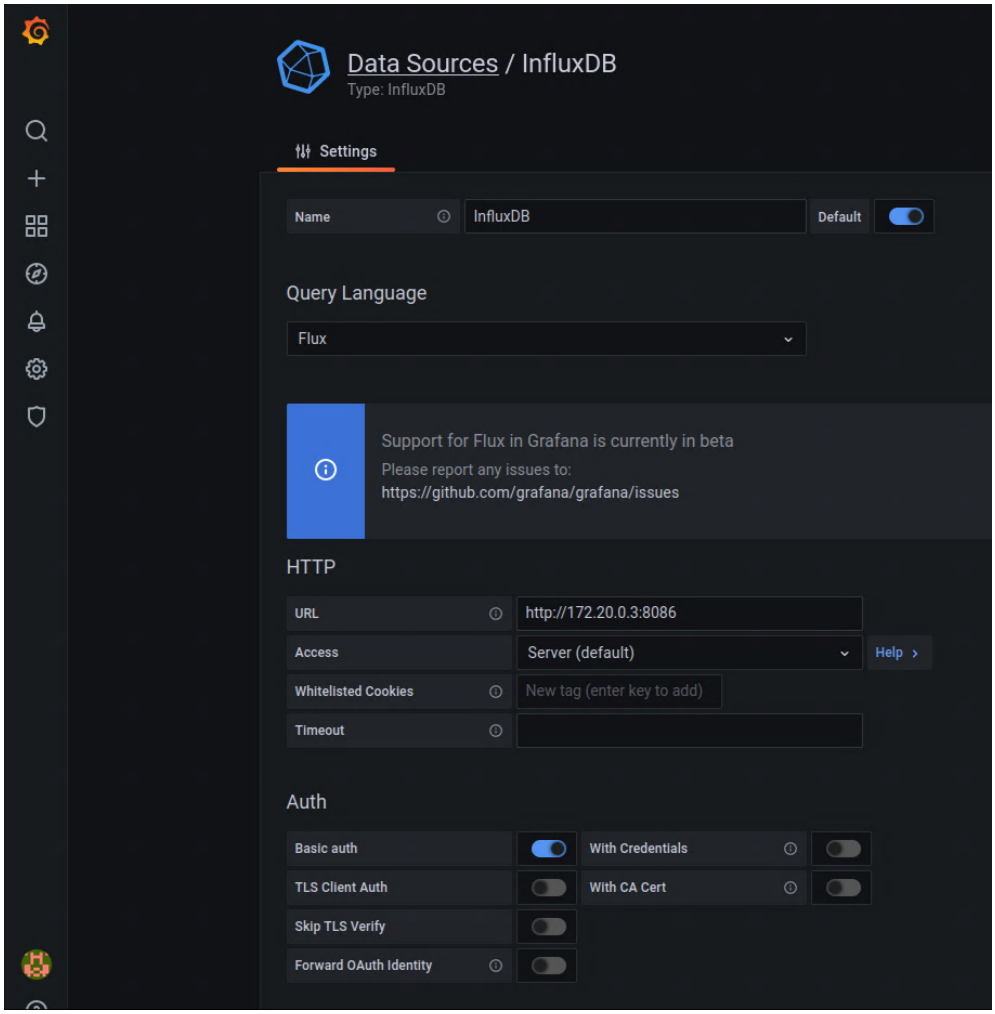
- User: admin
- Password: admin

The next step is to define a data source on Grafana, by selecting the option "Configuration" and "Data sources". The data source that will be selected is "InfluxDB" and the configuration values are described below and shown in Figure B.2:

- **Query language**: Flux.
- **Basic auth details**: the user is "sumo-user" and the password is "root-secret-password".
- **Organization**: smarttlc.

- **Token**: my-super-secret-auth-token.
- **Default bucket**: smarttlc.

Note that all the not-mentioned fields are set by their default values.




*Source:* Documentation Owner.
Figure B.2: Grafana add data source.

Lastly, when all these configuration values are set, it is possible to test the connection using the "Save and test". If the connection is enabled, the database will be available to create the desired dashboards.

# Appendix C

# Machine Learning concepts used in SmartTLC

This appendix outlines the different *Machine Learning* algorithms that have been studied to use in the TLP component. Inside the Machine Learning scope, the used paradigm will be **Supervised Learning**, due to the fact that our generated dataset is labeled. Moreover, the kind of Supervised Learning used will be the classification kind as the aim is to classify the traffic type depending on different kinds of information. Additionally, there are also described different model performance metrics and their training process.

## C.1 Machine Learning Algorithms

The algorithms [45] [46] [47] that are described in this section are: Linear Regression, Logistic Regression, Naive Bayes, KNN, Decision Trees and ensemble methods such as Random Forest. Besides, it is indicated which one of them is used in the proposed framework.

### C.1.1 Linear Regression

This algorithm is one of the most fundamental used to represent the relationship between different variables. In fact, the main purpose of this algorithm is to find the "line that fits the best" variable representation, which is calculated by minimizing the squared distance between different points. This algorithm is represented by the Equation C.1 and the Figure C.1 along with the Logistic Regression, described below.

$$Y = a + bX \tag{C.1}$$

C.1: Linear regression where **x** is the variable, **b** is the slope and **a** is the intercept.

This algorithm does not fit the proposed framework due to the fact that it only allows binomial classification.

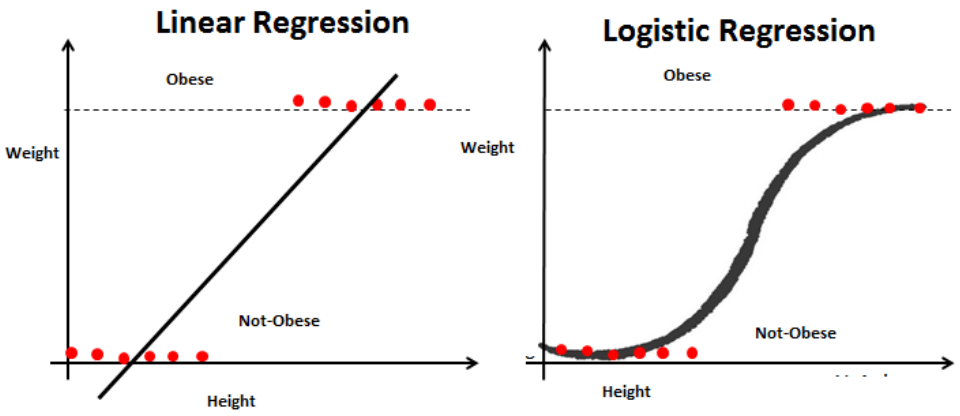
### C.1.2 Logistic Regression

The Logistic Regression [48] is like a linear regression that is used when the target value to predict is a class instead of a number. It is used for the prediction of a binary output, e.g. the input data is related to a "dog" or a "cat". One type of this algorithm is the Logistic Sigmoid Function, which provides the probabilities of belonging to either class. Using these probabilities, it can classify the input data as the class with a higher probability. The sigmoid function is shown in Equation C.2 and in Figure C.1.

$$P = \frac{e^y}{1 + e^y} \tag{C.2}$$

C.2: Logistic sigmoid regression where **y** is the linear regression function.

As the linear regression, the logistic regression does not fit the proposed framework due to the fact that it only allows binomial classification too.




*Source:* Analytics Vidhya (`http://shorturl.at/fhCER`)
Figure C.1: Linear vs Logistic Regression.

### C.1.3 Naive Bayes Classifier

The Naive Bayes Classifier [49][50] is a probabilistic classifier based on the Bayes' theorem [51] and some additional simplified hypotheses. This theorem provides the **independence hypothesis** between the different predictor variables, stating that the presence of a given feature is not related to the presence of any other feature from the same dataset.

The model is easily built providing a way of calculating the next probability of a certain event to occur, based on previous events. This probability can be calculated with the Equation C.3, which states how often A happens given that R happens $P(A|R)$; knowing how often R happens given that A happens $P(R|A)$, how likely A is $P(A)$ and how likely R is $P(R)$.

$$P(A|R) = \frac{P(R|A)P(A)}{P(R)} \tag{C.3}$$

C.3: Bayes Theorem equation.

On the one hand, this kind of classifier has the next advantages: (1) it provides a fast and quick way to predict classes on both binary and multi-class classification; (2) it works better than other algorithms if there is a valid independence hypothesis, even with less number of training data; and (3) each distribution can be estimated as a single

dimension. This fact improves global performance.

On the other hand, the disadvantages of the algorithm are: (1) the probabilities calculated are not very valid, as they are very poor estimators; (2) real-life data may not have the independence hypothesis; and (3) not known features will have a probability of zero, and then the predictions will be useless.

There exists some **variants** related to the Naive Bayes classifier:

- **Gaussian Naive Bayes** (GNB): the likelihood of features is assumed to follow a Gaussian distribution [52], represented with the Equation C.4, where the parameters $\sigma_y$ and $\mu_y$ are estimated using maximum likelihood.

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{C.4}$$

C.4: Gaussian Naive Bayes formula.

- **Multinomial Naive Bayes** (MNB): oriented to multinomially distributed data and used in text classification. This distribution is an extension of the binomial distribution.

- **Complement Naive Bayes** (CNB): this algorithm is an adaptation of the previously defined standard Multinomial Naive Bayes for those situations where the dataset is not balanced. As it is based on MNB, the CNB is used for text classification.

- **Bernoulli Naive Bayes** (BNB): this algorithm is used for data that is distributed according to a multivariate Bernoulli distribution [53], where all the features are assumed to be binary-valued. BNB differs from the MNB in the way that it penalizes the non-occurrence of a feature while MBN simply ignores it. This algorithm is also used for text classification.

- **Categorical Naive Bayes**: this algorithm is used for data that is categorically distributed and it assumes that each feature has its own categorical distribution.

Besides, this algorithm assumes that the sample matrix is encoded by representing the different possible categories as a number from 0 to $n_i - 1$, where $n_i$ is the number of available categories of the feature $i$.

Once all the Naive Bayes variation algorithms have been described, in Table C.1 it is shown the difference between them, selecting those that fit the aims of our problem. As it is possible to see, the only algorithm that fits our defined dataset and the problem aimed is the Gaussian Naive Bayes. Another option should be the Categorical Naive Bayes, but this option is rejected as it requires that all the dataset features must be categorical, which in our case is not possible for the number of passing vehicles features as it is "continuous". The MNB, CNB and BNB algorithms have been rejected as they are oriented to be used in text classification.

| | GNB | MNB | CNB | BNB | Categorical NB |
|---|---|---|---|---|---|
| Equally distributed | ✗ | ✓ | ✗ | ✓ | ✗ |
| Categorical | ✓ | ✓ | ✓ | ✓ | ✓ |
| Multi-use | ✓ | ✗ | ✗ | ✗ | ✗(*) |
| Easy of use | ✓ | ✓ | ✓ | ✓ | ✓ |
| Multiclass | ✓ | ✓ | ✓ | ✓ | ✓ |

Table C.1: Comparison between the different Naive Bayes algorithms.
(*) Only usable when all the features are categorical.

The only one algorithm that will be used on the framework is the Gaussian Naive Bayes.

### C.1.4 Support Vector Machine

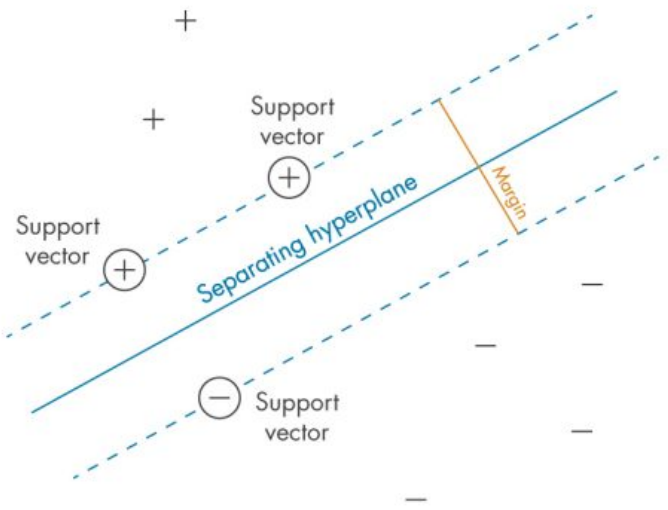
Support Vector Machine (SVM) [54] [55] can be used for both regression and classification but in this case, it is described only the classification-related features. It is based on the concept of decision planes (hyper-plane) that defines decision boundaries and split between a set of objects belonging to different classes.

An SVM creates a hyperplane or set of hyperplanes in a high or infinite-dimensional space. A way to define the best hyperplanes is to find the one that maximizes the margin between two different classes using a support vector (which are

the data points with the minimum distance to a hyperplane), since the larger margin the lower the generalization error of the classifier; moreover these support vectors influence higher than other data points to the position of the hyperplane. In order to find this hyperplane, the problem is transformed using linear algebra.

A variant of the SVM algorithm is the Support Vector Classification (SVC), which allows performing both binary and multi-class classification on a given dataset. In multi-class classification there are different approaches to use:

- **One-versus-one**: where a total of `num_classes * (num_classes - 1) / 2` classifiers are constructed and each one of them train data for two different classes. This means that it performs multiple binary classification cases.
- **One-versus-rest**: the number of total classifiers constructed and trained is `num_classes`. This means that the data points are divided in a given class and the rest, where a class is distinguished from all other classes.




*Source:* MathWorks - Support Vector Machine (`http://shorturl.at/nFJ26`).
Figure C.2: Support Vector Machine.

Sometimes, the problem is not linearly separable and the support vectors are the samples within the margin boundaries. Furthermore, based on the dimensionality (number of features) of the data, there are different kinds of SVM kernels functions [56] :

- **Linear**: creates a linear separability between each variable. The function is presented as $\langle x, x' \rangle$.

- **Polynomial**: it allows curved lines to be used in the input space and the degree of the polynomial kernel must be defined previously. The function is presented as $(\gamma \langle x, x' \rangle + r)^d$, where $d$ is the degree and $r$ is the coefficient.

- **Radial basis function** (RBF): used for non-linearly separable variables but the use of typical values can lead to overfitting in the data. The function is presented as $\exp(-\gamma \|x - x'\|^2)$ where $\gamma$ is the influence of a training example.

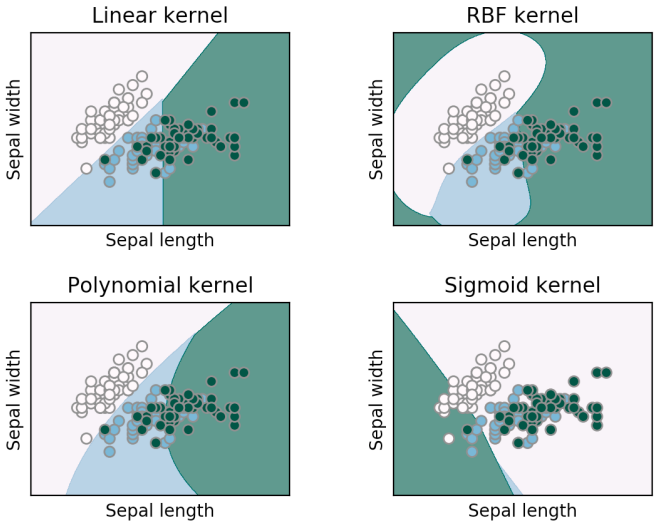
- **Sigmoid kernel**: it is used for binary classification, as in the logistic regression. The function is presented as $\tanh(\gamma \langle x, x' \rangle + r)$ where $\gamma$ is the influence of a training example.

- **Custom kernel**: some libraries allow defining custom kernels that can be more adjusted to the problem itself.

The kernel itself takes two data points and calculates the distance score between them, where the higher score means they are close and the lower score means they are far away. The use of these kernels helps transforming the data points to a higher-dimensional mapping which, in fact, reduces the time and computational effort.

One relevant disadvantage of this kind of algorithms is its **complexity** because the computation and storage requirements increase rapidly as the number of training vectors increase. In this way, the core of an SVM algorithm is a quadratic programming problem and its solver scales between $O(n_{features} \times n_{samples}^2)$ and $O(n_{features} \times n_{samples}^3)$ which means that it is dataset size strong-dependent.

This algorithm will be used in the proposed framework on both linear and polynomial with degree 2 kernels.




*Source:* Towards Data Science (`http://shorturl.at/oBJT0`).
Figure C.3: Support Vector Machine kernel function.

### C.1.5 K-Nearest Neighbors

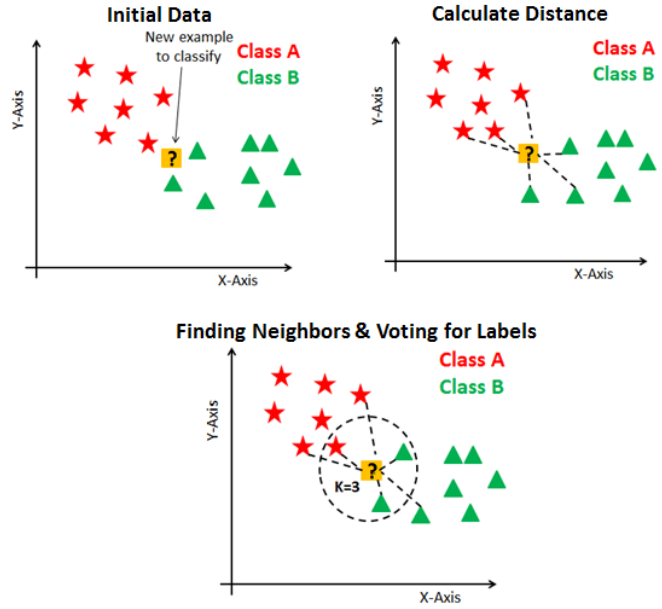
K-NN algorithm [57] [58] is one of the simplest classification algorithms and is used to identify the data points that are separated into several classes to predict the class of a new sample, but it can also be used for regression. K-NN is a non-parametric lazy learning (using the majority of computation to the consultation time) algorithm as it classifies the new samples based on similarity functions such as distance or proximity, and not learning a classification model. As it calculates the distance between the different classes, it does not scale to large data well. Its workflow is very simple and its represented in Figure C.4:

1. Calculate the distance between the data point and the full dataset.

2. Select those "K" closest elements based on the distance, which must be defined. The most famous distance functions are Euclidean Distance or Cosine Similarity.

3. Perform a "voting" of those data neighbors and the class with more "votes" will be the input data point class.

The K value means the number of neighbors to consider when classifying the input




*Source:* DataCamp (`http://shorturl.at/ckwAG`).
Figure C.4: KNN algorithm main steps.

information to a given class from all the possible classes. This value should be odd as if it is even it can produce situations where the number of neighbors belonging to two different classes is the same, and thus the algorithm could not select one of them easily. Some considerations about the number of neighbors are related to its value: (1) if K is small, there will be a low bias but a high variance; and (2) if K is large, there will be a high bias but a low variance. Based on these statements, the best option is to try different K values and select those values that better fit the input dataset.

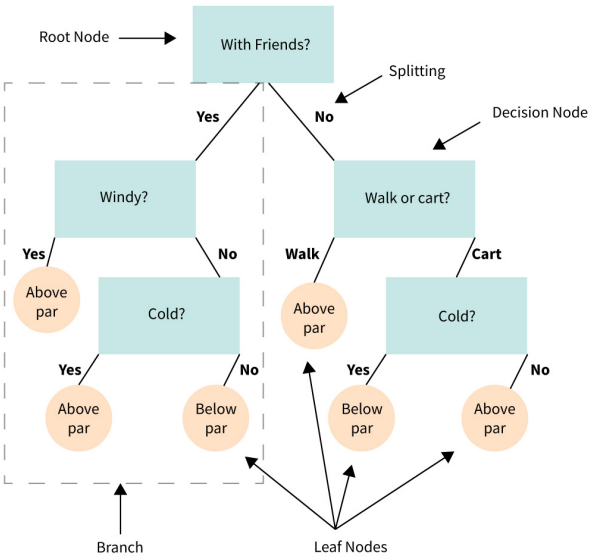
On the one hand, its main advantages are: (1) simple and easy to implement, (2) no need to build a model, (3) the algorithm is versatile as it can be used on classification and regression. On the other hand, its disadvantages are: (1) uses all the datasets to train each data point, therefore using a lot of processing resources as CPU; and (2) it gets slower as the dataset size increases.

This algorithm will be used in the framework along with the hyperparameter tuning which is explained below.

### C.1.6 Decision Trees

Decision Trees [59] [60] [61] is a non-parametric supervised algorithm that allows building both classification and regression models based on a tree structure, by breaking down the input dataset into smaller subsets, while the tree is incrementally developed upside down with the root at the top. The result of this procedure is a tree with decision nodes and leaf nodes. In this way, the process involves deciding which features to choose, what conditions to use and knowing when to stop, besides the trim of the tree to perform better.

Its goal is to create a model (tree) that predicts the value of a target variable by learning simple decision rules based on the data features. In the case of the classifiers, they can perform a multi-class classification, not only predicting a single value. An example of a decision tree can be seen in Figure C.5.




*Source:* Master's in Data Science (`http://shorturl.at/foMZ0`).
Figure C.5: Decision Tree example.

The main concepts used to construct the decision tree are:

- **Entropy**: degree of uncertainty in the randomness of elements, in other words, it allows to know about the predictability of a certain event. Moreover, it calculates

the homogeneity of a sample, being zero if it is fully homogeneous or one if the sample is equally divided.

- **Information gain**: measures the relative change in entropy with respect to the independent attribute. The tree construction is based on finding the attribute with the highest information gain, with the most homogeneous branches. It ranks filtering attributes at a given node in the tree.

On the one hand, the advantages of these algorithms are: (1) simple to understand and interpret as trees can be visualized, (2) require less data preparation compared to other algorithms, (3) logarithmic cost by using a tree, (4) it can handle both numerical and categorical data, (5) is able to handle multi-output problems, (6) it uses a white box model, which means that the explanation of a condition is easily explained by boolean logic, (7) the possibility of validating the model using statistical tests.

On the other hand, the disadvantages are: (1) its learners can create over-complex trees that do not generalize the data well, causing overfitting; (2) they can be unstable because small variations might result in a completely different generated tree; (3) not good as extrapolation as the predictions are neither smooth nor continuous; (4) the creation of an optimal decision tree is an NP-complete problem, that is why the used decision trees are created by using heuristic algorithms; (5) some concepts such as XOR, parity or multiplexer are hard to learn with decision trees; and (6) learners creates a biased tree if there are classes that dominate, which can be solved by balancing the dataset. Some of the previously mentioned disadvantages can be solved by using ensemble methods, which are explained in the next section.

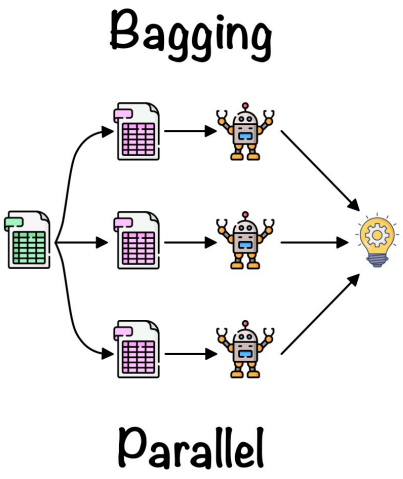
This algorithm will be used in the framework along with the hyperparameter tuning which is explained below.

### C.1.7 Ensemble methods

An ensemble model [62] [63] is a group or team of models that are individually trained and the results are comprised and merged in different ways to achieve a final prediction.

The result obtained by using these methods is higher than any of the individual models used, which means that the combination of learning models increases the overall result. Note that the combination of models can be used with different kinds of classifying models as it does not need to be the same, for example, an ensemble of Naive Bayes and Decision Trees.

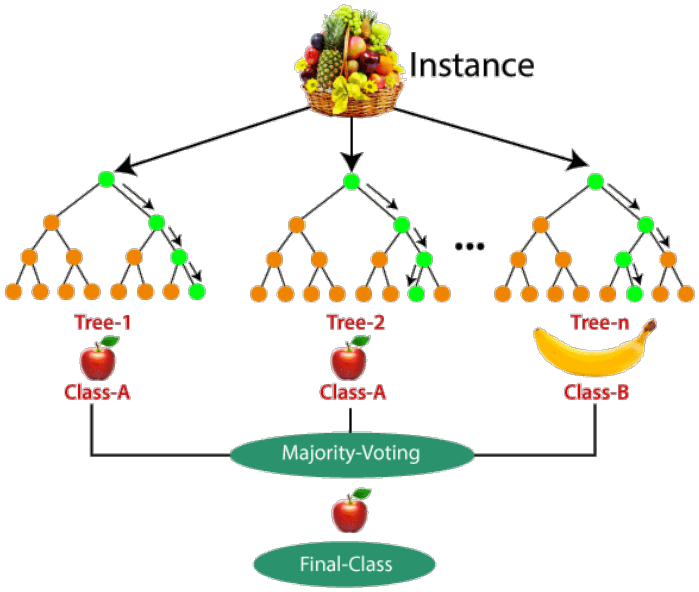
One of the most used ensemble methods is the Random Forest algorithm, based on the **Bagging** technique, which improves stability and accuracy of a machine learning algorithm, reducing variance and helping to avoid overfitting. It is used on both classification and regression algorithms. The parallel process followed is that from a given training set, it generates new training sets with a given size by sampling from the original one uniformly and with replacement (where some observations may be repeated); then fit each model with a given bootstrap sample and process their results. This process can be seen in Figure C.6.




*Source:* Towards Data Science (`http://shorturl.at/ewxDJ`).
Figure C.6: Bagging example.

Random Forest [64] [65] is an ensemble algorithm used for classification or regression which is based on bootstrap aggregating (also known as bagging) meta-algorithm. This algorithm is composed of a number of decision trees (also known as estimators) that are created using random subsets of features and bootstraps of

data, each one of them being uncorrelated to the rest. Once the decision trees have been created, each one of them "votes" which class the input data point belongs to by predicting it. Finally, all the votes are tallied to reach the final prediction. In classification, the output of the random forest is the class selected by most of the trees while in regression, the output is the mean average prediction of all the decision trees. An example of a random forest can be seen in Figure C.7.




*Source:* Analytics Vidhya (`http://shorturl.at/lwFMU`).
Figure C.7: Example of random forest simplified.

The overfitting caused by the decision trees is solved in the random forest by creating trees on random subsets and taking the average of all the predictions, which cancels out the biases. Besides, random forest adds additional randomness to the model while growing the forest, searching the best feature among a random subset of features instead of searching the best feature while splitting the node.

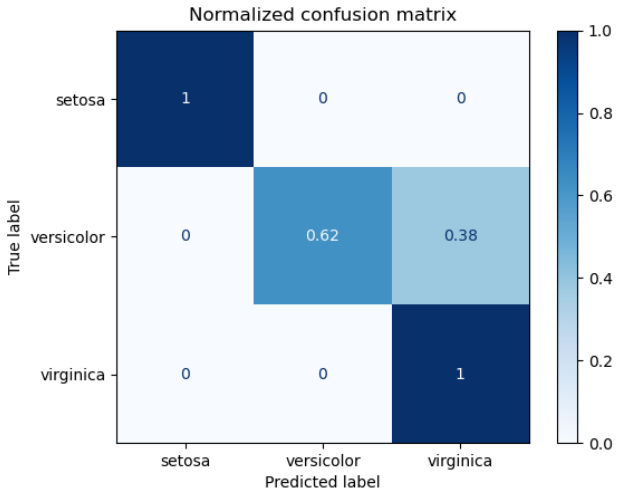
This algorithm will also be used in the framework along with the hyperparameter tuning which is explained below.

## C.2 Classification metrics for model performance

In this section, there are described the main features used to check the performance for different classification models. These metrics allow knowing if the training process of a model was good or even if there is either overfitting or underfitting.

### C.2.1 Confusion Matrix

The confusion matrix [66] is a table that is used to describe the performance of a model on a set of test data for which the true values are known, allowing to see the model predicted values and the actual values. It can be used in both binary and multi-class classification, which can provide information about mistaken patterns. An example of a confusion matrix can be seen in Figure C.8.




*Source:* Scikit-Learn (`http://shorturl.at/awFJK`).
Figure C.8: Example of a normalized confusion matrix with three classes.

Confusion matrix is based on 4 different concepts, described below:

- **True positive**: model correctly predicts the positive class.
- **True negative**: model correctly predicts the negative class.

- **False positive**: model incorrectly predicts the positive class. It is the rejection of a true null hypothesis but it is not as bad as the false negative.

- **False negative**: model incorrectly predicts the negative class. It is the acceptance of a false null hypothesis, which is very dangerous.

In this way, the more values in the main diagonal, the better the model is, as it represents that the predictions are right while the off-diagonal values represent the mislabeled predictions by the model. Based on this matrix, it is possible to infer the next concepts:

- **Accuracy**: fraction of predictions the model predicts right. It is important to mention that accuracy is not a good metric when the dataset is class-imbalanced. To calculate the accuracy it is used the Equation C.5:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{C.5}$$

C.5: Accuracy equation.

In the case of binary classification, the accuracy can also be calculated using the positives and negatives terms using the Equation C.6 where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{C.6}$$

C.6: Accuracy equation in binary classification.

- **Precision**: it indicates how much the model has predicted correctly from all the possible classes, taking into count only the positive values such as true and false positives. In fact, this measure is able to not label a true negative as positive. In this case, the precision is a good metric when the dataset is class-imbalanced and it has to be the higher value possible. To calculate the precision it is used the Equation C.7:

$$Precision = \frac{\text{True Positives}}{\text{True Positives + False Positives}} \tag{C.7}$$

C.7: Precision equation.

- **Recall**: it indicates how much the model has predicted correctly from only the positive classes, taking into count only the true positives and false negatives. It is also called sensitivity or true positive rate (TPR). In fact, this measure expresses the ability of the classifier to find positive examples. As with precision, the recall is a good metric when the dataset is class-imbalanced and it has to be the higher value possible. To calculate the recall it is used the Equation C.8:

$$Precision = \frac{\text{True Positives}}{\text{True Positives + False Negatives}} \tag{C.8}$$

C.8: Recall equation.

- **F1 score**: it is the combination of both precision and recall, calculating their harmonic mean (a mathematical function that gives much more weight to low values) by punishing the extreme values more) which will get only a high value if both precision and recall are high. It is used basically to compare different classifiers using the same measure and its bounds are 0 and 1, representing a bad and a good value respectively. To calculate the F1 score it is used the Equation C.9:

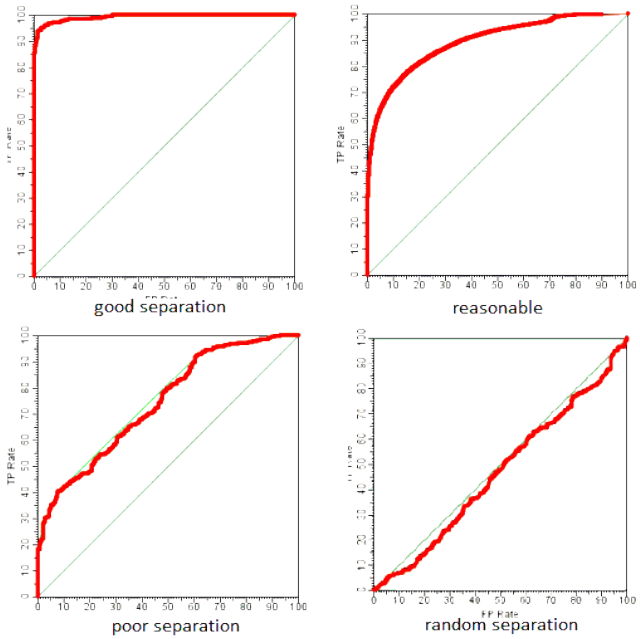
$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{C.9}$$

C.9: F1 score equation.

All the previously described measurements are the ones that will be used on the framework to test the models' performance, especially the F1 score.

### C.2.2 Receiver Operator Curve

The Receiver Operator Curve (ROC) [67] states how well the model has accurately predicted with a curve that shows the sensitivity of the classifier by plotting the true positive rates against the false positives rate. The better area under the curve (AUC), the better the model is. Some curves are shown in Figure C.9. This measure is used in binary classification and for using it on multi-class classification it is required to binarize the output, that is the main reason why it is not used on the system.




*Source:* ML Wiki (`http://shorturl.at/rGV12`)
Figure C.9: Different ROC curves

## C.3 Training process

This section outlines the two main model training processes: basic train-test split and k-fold split. This last one will be the one used in the framework.

### C.3.1 Basic Train-Test split

The basic train-test split [68] [69] is used to evaluate the performance of several machine learning models for both classification and regression. Although this process is very simple and is used widely, there are some situations where it should not be used e.g. when the dataset used is too small.

This process consists of dividing a given dataset into two splits, randomly, usually selecting 70% and 30%, respectively:

- **Train dataset**: used to fit the model.
- **Test dataset**: used to evaluate the trained model.

As the generated dataset is not big enough, this approach has been rejected.

### C.3.2 Cross-validation (K-Fold split)

The cross validation [70] [71] [72] is a statistical method that is used to estimate the performance of different machine learning models. In fact, this validation is a resampling procedure used with a limited length dataset. Its main parameter is the $k$ value, which indicates the number of "folds" or groups of data that the original dataset will be divided into.

One of its main advantages is that it provides better results e.g being less biased than other methods as the one previously mentioned, the basis train/test split.

The main process of this cross-validation process is the following one and it is represented in Figure C.10:

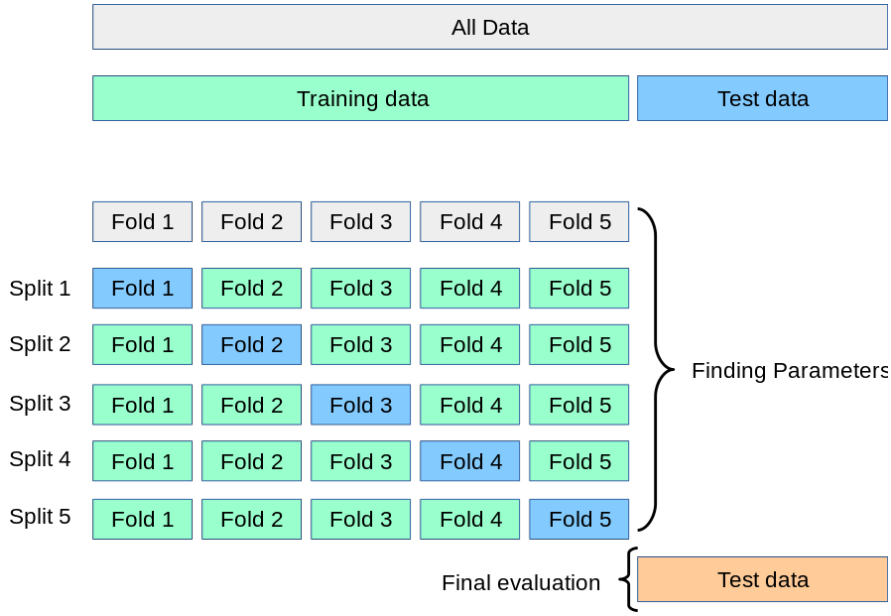
1. Shuffle the input dataset randomly.
2. Split the dataset into $k$ folds.
3. Per each fold:
    3.1 Retrieve the training dataset, 70%.

3.2 Retrieve the test dataset, 30%.

3.3 Fit the model with the training dataset.

3.4 Calculate perform values with the test dataset and store them.

4. Evaluate the performances and retrieve those models with better performance.




*Source:* Scikit-Learn (`http://shorturl.at/dmtDK`).
Figure C.10: K-Fold process.

In fact, the most relevant step of this process is the selection of a valid $k$ value, because this number affects directly the training process of the models. The three main tactics are the next ones:

- **Representative value** on which the train and test folds are large enough. In the framework, it is used a fold of 2 due to the fact that this number of folds fits the best to the generated dataset.
- $k = 10$ as this value is tested to generally result in a model skill estimate with low bias.
- $k = n$, being $n$ the size of the dataset. This approach gives each test sample an opportunity to be used in the training dataset, and so it is called "leave-one-out" cross-validation.

# Bibliography


[1] European Institute of Innovation and Technology (EIT) for Urban Mobility. *Solving the mobility challenges facing our cities together*, 2021. Available at: https://www.eiturbanmobility.eu/.

[2] U. S. Environmental Protection Agency. *Fast Facts on Transportation Greenhouse Gas Emissions*. Available at: https://www.epa.gov/greenvehicles/fast-facts-transportation-greenhouse-gas-emissions.

[3] U. S. Energy Information Administration. *International Energy Outlook 2021*, 2021. Available at: https://www.eia.gov/outlooks/ieo/index.php.

[4] Hedges Company. *How many cars are in the world in 2022: Market Research*, 2021. Available at: https://hedgescompany.com/blog/2021/06/how-many-cars-are-there-in-the-world/.

[5] Bob Pishue (INRIX). *INRIX Global Traffic Scorecard*. Available at: https://inrix.com/scorecard/.

[6] Jon Glasco. *Smart mobility: challenges and solutions in Smart Cities*. Available at: https://hub.beesmart.city/en/solutions/smart-mobility/smart-mobility-challenges-and-solutions-in-smart-cities.

[7] Thales Group. *Secure, sustainable smart cities and the IoT*. Available at: https://www.thalesgroup.com/en/markets/digital-identity-and-security/iot/inspired/smart-cities.

[8] Institute for Transportation and Development Policy. *Basic concepts*. Available at: `https://brtguide.itdp.org/branch/master/guide/intersections-and-signal-control/basic-concepts`.

[9] Oxford Learner's Dictionaries. *Traffic definition*. Available at: `https://www.oxfordlearnersdictionaries.com/definition/english/traffic_1`.

[10] TBF Traffic company. *What is Traffic Management*, 2022. Available at: `https://www.tbftraffic.com/what-is-traffic-management/`.

[11] Oxford Learner's Dictionaries. *Traffic Light definition*. Available at: `https://www.oxfordlearnersdictionaries.com/definition/american_english/traffic-light`.

[12] PIARC World Road Association. *What is ITS*, 2021. Available at: `https://rno-its.piarc.org/en/intelligent-transport-systems/what-its`.

[13] Suresh Chavhan, Deepak Gupta, Chandana Nagaraju, Rammohan A, Ashish Khanna, and Joel J.P.C. Rodrigues. *An Efficient Context-Aware Vehicle Incidents Route Service Management for Intelligent Transport System*. *IEEE Systems Journal*, 2021. Available at: `https://ieeexplore.ieee.org/abstract/document/9402882`.

[14] Md Ashifuddin Mondal and Zeenat Rehena. *An IoT-Based Congestion Control Framework for Intelligent Traffic Management System*. *Advances in Intelligent Systems and Computing*, 1133:1287–1297, 2021. Available at: `https://ieeexplore.ieee.org/abstract/document/9402882`.

[15] Anoop Vidyadharan (Medium). *Intelligent Traffic Management System (ITMS) — Smart Solution for Future Cities*. Available at: `https://medium.com/@anoopvidyadharan6/intelligent-traffic-management-system-itms-dec9a8fcca9a`.

[16] Mustafa Bani Khalaf, Mohammad Ashraf Ottom, Mamoon Obiedat, and Nabhan Hamadneh. *A Framework for fog virtual traffic light system. Journal of Theoretical and Applied Information Technology*, 98:279–289, 01 2020. Available at: `http://www.jatit.org/volumes/Vol98No2/9Vol98No2.pdf`.

[17] Rusheng Zhang, Frank Schmutz, Kyle Gerard, Aurélien Pomini, Louis Basseto, Sami Hassen, Akihiro Ishikawa, Inci Ozgunes, and O.K. Tonguz. *Virtual Traffic Lights: System Design and Implementation. IEEE 2018 Fall Vehicular Technology Conference (VTC)*, 06 2018. Available at: `https://www.researchgate.net/publication/325813596_Virtual_Traffic_Lights_System_Design_and_Implementation`.

[18] Songqing Chen, Tao Zhang, and Weisong Shi. *Fog Computing. IEEE Internet Computing*, 21(2):4–6, 2017. Available at: `https://ieeexplore.ieee.org/abstract/document/7867739`.

[19] Viswacheda Duduku. V, Ali Chekima, Farrah Wong, and Jamal Ahmad Dargham. *A Study on Vehicular Ad Hoc Networks. IEEE Internet Computing*, pages 422–426, 2015. Available at: `https://ieeexplore.ieee.org/document/7604612`.

[20] Pampa Sadhukhan and Firoj Gazi. *An IoT based intelligent traffic congestion control system for road crossings. IEEE Internet Computing*, pages 371–375, 03 2019. Available at: `https://link.springer.com/chapter/10.1007/978-981-15-3514-7_96`.

[21] Cathy Wu, Abdul Rahman Kreidieh, Kanaad Parvate, Eugene Vinitsky, and Alexandre Bayen. *Flow: Architecture and Benchmarking for Reinforcement Learning in Traffic Control*. 10 2017. Available at: `https://www.researchgate.net/publication/320441979_Flow_Architecture_and_Benchmarking_for_Reinforcement_Learning_in_Traffic_Control`.

[22] Nishant Kheterpal, Eugene Vinitsky, Cathy Wu, Aboudy Kreidieh, Kathy Jang, Kanaad Parvate, and Alexandre Bayen. Flow: Open Source Reinforcement Learning for Traffic Control. Available at: https://flow-project.github.io/papers/33f6dea4d284de3e316bcd76ddf6e0c29434aa5f.pdf.

[23] Eugene Vinitsky, Kanaad Parvate, Abdul Rahman Kreidieh, Cathy Wu, and Alexandre Bayen. *Lagrangian Control through Deep-RL: Applications to Bottleneck Decongestion*. pages 759–765, 11 2018. Available at: https://www.researchgate.net/publication/329618515_Lagrangian_Control_through_Deep-RL_Applications_to_Bottleneck_Decongestion.

[24] Eclipse Foundation. *Simulation of Urban MObility*. Available at: https://www.eclipse.org/sumo/.

[25] Eclipse SUMO Docs. *TraCI*. Available at: https://sumo.dlr.de/docs/TraCI.html.

[26] anylogic. *Road Traffic Simulation Software*. Available at: https://www.anylogic.com/road-traffic/.

[27] Aimsun. *aimsun.next*. Available at: https://www.aimsun.com/es/aimsun-next-2/.

[28] Doug Mak. *Are most cars left or right hand drive?* Available at: https://www.quora.com/Are-most-cars-left-or-right-hand-drive.

[29] influxdata. *Components of the TICK Stack*. Available at: https://www.influxdata.com/time-series-platform/.

[30] Owner. *SmartTLC - GitHub*. Available at: https://github.com/JoserraLP/SmartTLC.

[31] Portal de datos abiertos del Ayuntamiento de Madrid. *Calendario laboral*. Available at: https://datos.madrid.es/portal/site/egob/menuitem.c05c1f754a33a9fbe4b2e4b284f1a5a0/?vgnextoid=9f710c96da3f951

0VgnVCM2000001f4a900aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD&vgnextfmt=default.

[32] jwilder GitHub. *dockerize*. Available at: https://github.com/jwilder/dockerize.

[33] Docker. *Docker*. Available at: https://www.docker.com/.

[34] Open Source. *What is Docker?* Available at: https://opensource.com/resources/what-docker.

[35] Red Hat. *What is Docker?*, 1 2018. Available at: https://www.redhat.com/en/topics/containers/what-is-docker.

[36] Docker Docs. *Docker overview*. Available at: https://docs.docker.com/get-started/overview/.

[37] Alexander TechTarget S. Gillis. *Docker image*. Available at: https://searchitoperations.techtarget.com/definition/Docker-image.

[38] Docker. *Use containers to Build, Share and Run your applications*. Available at: https://www.docker.com/resources/what-container.

[39] Anaconda. *Anaconda*. Available at: https://www.anaconda.com/.

[40] Docker Docs. *Install Docker Desktop on Windows*. Available at: https://docs.docker.com/desktop/windows/install/.

[41] Anaconda Docs. *Install on Windows*. Available at: https://docs.anaconda.com/anaconda/install/windows/.

[42] Docker Docs. *Install Docker Engine on Ubuntu*. Available at: https://docs.docker.com/engine/install/ubuntu/.

[43] Docker Docs. *Install Docker Compose*. Available at: https://docs.docker.com/compose/install/.

[44] Anaconda Docs. *Install on Linux*. Available at: https://docs.anaconda.com/anaconda/install/linux/.

[45] Badreesh Built-in Shetty. *An in-depth guide to supervised machine learning classification*, 11 2021. Available at: https://builtin.com/data-science/supervised-machine-learning-classification.

[46] Sin Towards Data Science Terence. *All Machine Learning Algorithms You Should Know in 2021*, 11 2020. Available at: https://towardsdatascience.com/all-machine-learning-algorithms-you-should-know-in-2021-2e357dd494c7.

[47] Sin Towards Data Science Terence. *All Machine Learning Algorithms You Should Know in 2022*, 11 2021. Available at: https://towardsdatascience.com/all-machine-learning-algorithms-you-should-know-in-2022-db5b4ccdf32f.

[48] Saishruthi Towards Data Science Swaminathan. *Logistic Regression — Detailed Overview*, 3 2018. Available at: https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc.

[49] Scikit-Learn. *Naive Bayes*, 2021. Available at: https://scikit-learn.org/stable/modules/naive_bayes.html.

[50] Victor Medium Roman. *Algoritmos Naive Bayes: Fundamentos e Implementación*, 2019. Available at: https://medium.com/datos-y-ciencia/algoritmos-naive-bayes-fudamentos-e-implementacion-4bcb24b307f.

[51] Math is Fun. *Bayes Theorem*, 2021. Available at: https://www.mathsisfun.com/data/bayes-theorem.html.

[52] Probability Course. *Normal (Gaussian) Distribution*, 2021. Available at: https://www.probabilitycourse.com/chapter4/4_2_3_normal.php.

[53] Statistics How To. *Bernoulli Distribution: Definition and Examples*, 2021. Available at: https://www.statisticshowto.com/bernoulli-distribution/.

[54] Scikit-Learn. *Support Vector Machines*, 2021. Available at: https://scikit-learn.org/stable/modules/svm.html.

[55] MathWorks. *Support Vector Machines (SVM)*, 2021. Available at: https://la.mathworks.com/discovery/support-vector-machine.html.

[56] Hucker Towards Data Science Marius. *Multiclass Classification with Support Vector Machines (SVM), Dual Problem and Kernel Functions*, 06 2020. Available at: https://towardsdatascience.com/multiclass-classification-with-support-vector-machines-svm-kernel-trick-kernel-functions-f9d5377d6f02.

[57] Onel Towards Data Science Harrison. *Machine Learning Basics with the K-Nearest Neighbors Algorithm*, 9 2018. Available at: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761.

[58] Scikit-Learn. *K Neighbors Classifier*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

[59] Master´s in Data Science. *What is a Decision Tree?* Available at: https://www.mastersindatascience.org/learning/introduction-to-machine-learning-algorithms/decision-tree/.

[60] Scikit-Learn. *Decision Trees*. Available at: https://scikit-learn.org/stable/modules/tree.html.

[61] Prashant Towards Data Science Gupta. *Decision Trees in Machine Learning*. Available at: https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052.

[62] Scikit-Learn. *Ensemble nethods*. Available at: https://scikit-learn.org/stable/modules/ensemble.html.

[63] Vihar Paperspace Blog Kurama. *Introduction to Bagging and Ensemble Methods*, 2018. Available at: https://blog.paperspace.com/bagging-ensemble-methods/.

[64] Scikit-Learn. *Random Forest Classifier*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.

[65] Tony Towards Data Science Yiu. *Understanding Random Forest*, 6 2019. Available at: https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

[66] Scikit-Learn. *Confusion matrix*. Available at: https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html.

[67] Scikit-Learn. *Receiver Operating Characteristic (ROC)*. Available at: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html.

[68] Jason Machine Learning Mastery Brownlee. *Train-Test Split for Evaluating Machine Learning Algorithms*, 7 2020. Available at: https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/.

[69] Scikit-Learn. *Train-Test Split*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[70] Jason Machine Learning Mastery Brownlee. *A Gentle Introduction to k-fold Cross-Validation*, 5 2018. Available at: `https://machinelearningmastery.com/k-fold-cross-validation/`.

[71] Scikit-Learn. *K-Fold Split*. Available at: `https://scikit-learn.org/stable/modules/cross_validation.html`.

[72] Krishni Medium Data Driven Investor. *K-Fold Cross Validation*, 12 2018. Available at: `https://medium.datadriveninvestor.com/k-fold-cross-validation-6b8518070833`.