POLITECNICO DI TORINO

Master Degree Thesis

Mechanical Engineering

AI Solutions for Soot Prediction in CI Engines

Supervisor

Prof. Daniela Anna Misul

Assistant Supervisor

Dott. Ing. Alessandro Falai

Candidate

Stefano Marchetta

July 2021

Table of Contents

Table of Figures

Table of Tables

Introduction

In 2018, more than half of world crude oil consumption came from the transportation sector[1]. Despite the Covid-19 pandemic and consequent reduction of world oil demand experienced by every sector (especially the transportation one) in 2020, it is expected that the above-mentioned trend will strengthen, increasing over the next years, as the growth rebound is already visible.

Talking about road transportation, the immense world fleet of land vehicles, equipped with internal combustion engines (ICE), contributes primarily to the deterioration of the air quality due to the polluting emissions characteristic of this kind of thermal volumetric machine. The emitted pollutants lead to a general increase in health problems and a higher incidence of cardiovascular and respiratory diseases and tumours. Pollutants are chemical species that have a direct negative (carcinogenic or poisonous) effect on human health, and they can be classified in primary and secondary pollutants. The former ones are direct products of the combustion process, while the latter ones are formed later starting from the primary pollutants which can be subjected in the atmosphere to chemical reactions between them and with other chemical species present in the environment. The main primary pollutants are carbon monoxide (CO), unburnt hydrocarbons (HC), nitrogen oxides (NO_x) and particulate matter (PM). On the other hand, the main secondary pollutants are acid rains (which contain H_2SO_4 and HNO_3) and "photochemical smog", that is a mixture of harmful substances including mainly ozone (O_3) and volatile organic compounds (VOC).

Particulate matter is a mixture of fine powders of different origin, size, and composition which, being very small, tend to remain suspended in the air and are mainly produced by CI engines. Medical evidence has shown how these suspended particles are among the major causes of respiratory diseases. For this reason, stringent legislations have been adopted, both nationally and internationally, in order to place severe limits on both the mass and the size of the particles emitted by the vehicles. These regulatory restrictions have fostered the resumption of research activities aimed at deepening the mechanisms of formation and oxidation of PM. Furthermore, new devices have been adopted over the years, with specific reference to after-treatment technologies, as the main consequence of the tightening in the emissions limitation imposed by the standards.

This thesis work stems from the desire to find a valid alternative to the use of physical sensors, called soot sensors, for measuring the quantity of soot emitted (carbonaceous fraction of the particulate) by the engine. While the emission limits set by the legislation are defined in relation to the specified driving cycles, the engine output emissions can vary significantly due to the

[1] "World oil final consumption by sector, 2018", IEA, World Energy Balances, 2020

behaviour of the driver and the different operating conditions of the engine itself. So, the main objective of this thesis is to apply and validate a virtual model that accurately estimates the engine-out soot mass. For this purpose, predictive models, based on Machine Learning and Deep Learning algorithms, will be designed and developed using the main engine parameters processed in the ECU (Engine Control Unit).

Chapter 1

Theorical Background

In order to better understand this thesis work, a briefly introduction is needed about the compression ignition engines combustion model and the polluting emissions formation mechanisms characterizing this type of ICE operations. These theorical concepts will help the reader to deep dive more easily into this thesis work.

1.1 Combustion in Compression Ignition Engines

In CI engines fuels with relatively low ignition delays are used, that are fuels characterized by high reactivity levels (examples are Diesel, biodiesel). Therefore, unlike what happens in spark ignition engines, fuel cannot be premixed with air and then compressed in the combustion chamber because it would burn up well before this process completes. Thus, to allow a good combustion control, the fuel should be injected in the cylinder at the compression stroke end, when the piston is reaching the TDC. Multi-hole electronic injectors release fuel at very high pressures (values up to 2000 bar are quite common) and high speeds (around 100 m/s) allowing the liquid fuel jet to be atomized in a cloud of microscopic droplets (with diameters of 10 micron circa). The fuel gets in touch with the hot (at around 900 K) and compressed air (at 20-30 kg/m^3) in the combustion chamber, vaporizes and mixes with it, then it ignites spontaneously without the need of an external spark.

Thanks to the fuel high reactivity, in CI engines the air-fuel mixture can ignite also with air to fuel ratios quite far from the stoichiometric conditions (with low oxygen concentrations). This behaviour causes a fuel molecules dehydrogenation which results in carbon skeletons. They can then agglomerate giving rise to solid carbon particles (i.e., soot) which are accountable for particulate emissions of this type of engines.

The oxidation process (combustion) of fuels like gasoline and Diesel can be represented by the following chemical reaction between reactants and products:

$$C_xH_y + \left(\frac{y}{4} + x\right)(O_2 + \Psi N_2) \longrightarrow xCO_2 + \frac{y}{2}H_2O + \Psi\left(x + \frac{y}{4}\right)N_2$$

where Ψ represents the ratio between nitrogen and oxygen moles in the air, x indicates the carbon atoms in the fuel molecule, and y is the relative number of hydrogen atoms.

The reaction shown above describes an ideal (complete) and stoichiometric combustion where carbon dioxide, water vapour and molecular nitrogen (it should not react during the process) are the products obtained. Unfortunately, the fuel oxidation process takes place through a series of

intermediate reactions whereby it could happen that not all fuel molecules are completely oxidized, resulting in the previously mentioned polluting emissions. The CO_2 is a product of the ideal combustion process previously mentioned and it is not considered a pollutant because it has no effects on human health However, it is a greenhouse gas with a global scale impact, so it is one of the major causes of the global warming.

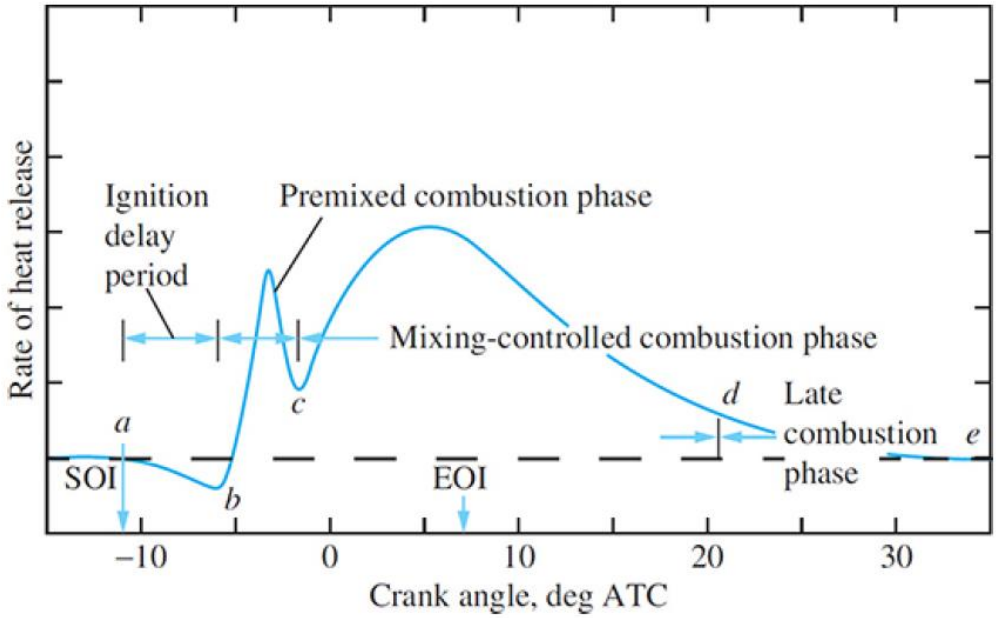

Figure 1. Typical CI engine HRR diagram highlighting the different combustion phases [10]

Observing the trend of the Heat Release Rate (HRR) curve versus the crank angle reported in Figure 1, it is possible to identify four different phases in the overall duration of the injection and combustion processes:

- Ignition delay period: this phase begins with the start of fuel injection and ends with the start of combustion
- Premixed combustion phase: fast combustion of the fuel accumulated in the cylinder during the previous phase and mixed with air (fuel rich conditions), causing a great pressure increasing and a spike in the energy released
- Mixing-controlled combustion phase: start and development of the diffusive flame at the jet periphery
- Late combustion phase: the combustion is completed with the burning of the fuel injected lastly.

We will see in detail phase by phase, identifying for each one the contribution made to the global combustion process and underlining the most important aspects that characterize it.

Before that, it is necessary a brief consideration regarding the amount of fuel that accumulates inside the combustion chamber before the process begins. This accumulation must be kept under

control, within certain limits, to reduce the premixed combustion peak. This can be accomplished by modifying the injection rate, that is instant flow rate introduced into the cylinder. However, a considerable reduction of this parameter could lead to a combustion efficiency drop due to the widening of the process on a greater crank angle range. This imply having the last part of the combustion in the expansion phase. To solve this issue the main alternatives are:

- Injection rate shaping – It consists in the modulation of the fuel flow rate by keeping it relatively low in the early stages of injection, so that a small amount accumulates in the combustion chamber and increasing it in the subsequent phase.
- Pilot injection – It is a small amount of fuel (about 5-10 % of the total quantity) injected before to the main injection. Trough properly timing the pilot injection, we can ensure that the accumulated quantity burns when the main injection starts. Thereby the temperature raises locally and the oxidation of the fuel subsequently introduced into the chamber accelerates, reducing the accumulation during the main event. This solution makes it possible to considerably reduce the pressure peaks which are responsible for the characteristic noise of CI engines.

1.1.1 Ignition Delay

The ignition delay in a CI engine is defined as the timeframe (in the order of ms) between the Start of Injection (SOI) and the Start of Combustion (SOC), which does not occur simultaneously for all the fuel molecules injected. This parameter is very important because it is accountable for the subsequent combustion at almost constant volume. This process is advantageous in terms of thermal efficiency, but harmful from the stresses and vibrations (noise) point of view. The fundamental reasons of this delay are the following:

- physical causes: phenomena that modify the aggregation state of the fuel molecules and regulate mixing processes between different substances
- chemical causes: oxidation reactions that modify the chemical structure of the fuel molecules.

A duration is associated with each of the foregoing causes, whose sum gives the overall time corresponding to the ignition delay:

$$\tau_{total} = \tau_{physical} + \tau_{chemical}$$

The phenomena associated with the two causes never occur strictly in series but can overlap each other. Furthermore, the physical rate is predominant, as the fuel is highly reactive, resulting in a high reaction rate and, therefore, reduced time required by the chemistry of the phenomenon.

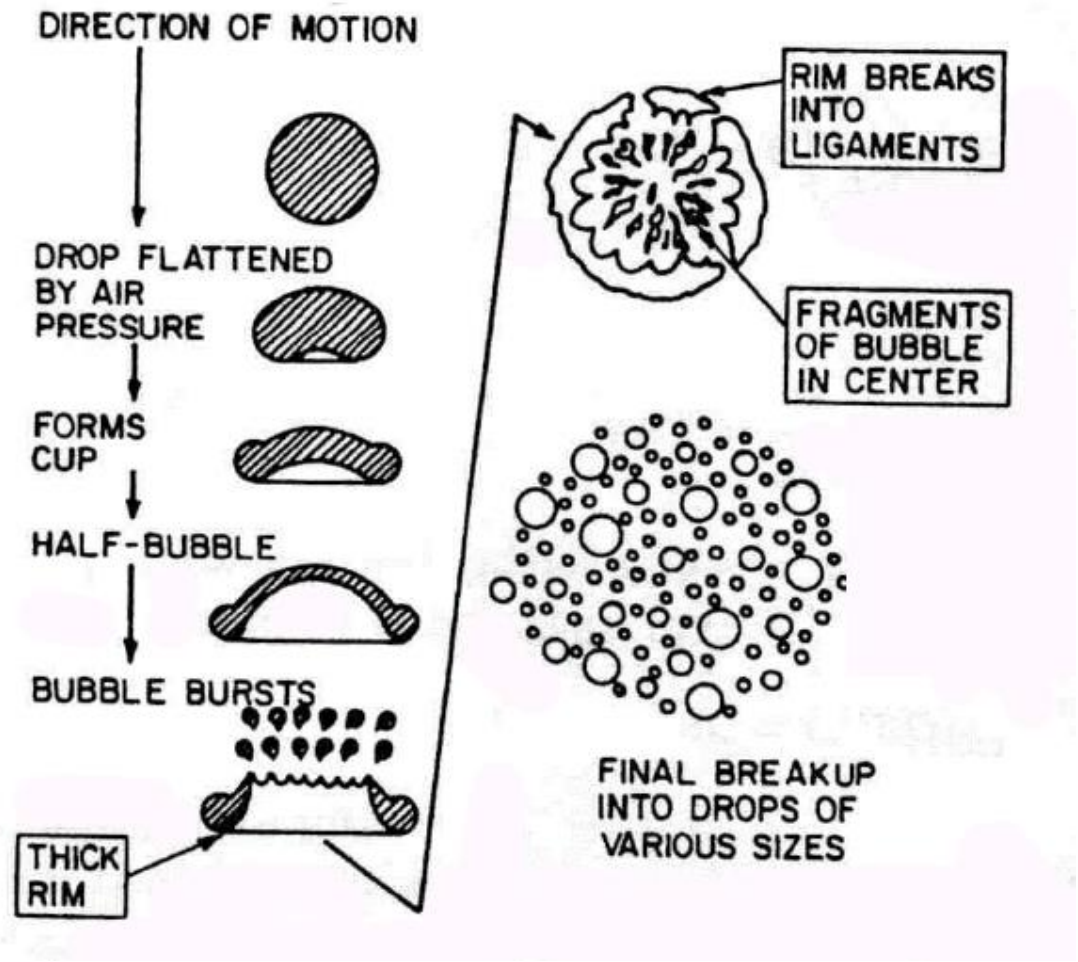

Figure 2. Fuel droplet break-up process schematization [15]

By going into details, the physical phenomena can be divided into a set of phases that characterize the changes occurring in the fuel jet penetrating the combustion chamber:

- Atomisation (pulverisation) – The liquid fuel jet must vaporize so that it can mix with the air. This phase change occurs through the breaking process of the liquid column and is mainly governed by fuel physical characteristics, such as the surface tension and the intensity of aerodynamic interaction with the surrounding high-density air. The aerodynamic forces tend to pulverize the fuel jet in a two-step process. Firstly, they act on the outer surface of the jet, causing the formation of fuel drops (primary break up), and they subsequently act on these droplets formed, which lose their spherical shape and divide into myriads of microscopic size drops (secondary break up). The entire process is represented in Figure 2. Therefore, this physical interaction is controlled by the opposition of the aerodynamic forces, which push toward the atomisation of the jet, and

surface tension forces, which tend to keep the liquid droplets cohesive. The parameter that expresses the opposition of these two contributions is represented by the Weber number:

$$We = \frac{\rho_g v^2 D}{\sigma}$$

where ρ_g represents the fuel density, v the injected liquid velocity, D the droplets diameter and σ the fuel surface tension. The value of 12 for the Weber number is taken as the limit below which no pulverisation of the liquid jet occurs.

- Droplets evaporation – Due to the heat received from the surrounding air, the fuel drops heat up, causing a simultaneous speed-up in the speed of the vaporization process. The consequent reduction of the droplets size and their deceleration produced by the aerodynamic resistance cause a decrease in the heat flux and a consequent slowdown of the process. Hence, the speed of the evaporation process is not constant but decreases over time.
- Fuel vapours mixing with air – It is the last phase of the ignition delay and strongly depends on the engine operating point. In fact, the higher the rotational speed of the engine, the shorter the time that will be available for the mixing process. Therefore, the turbulence in the combustion chamber must be stronger in order to accelerate the mixing between fuel vapours and air, that is to avoid an efficiency drop.

On the other hand, the chemical phenomena mainly concern:

- formation reactions of intermediate compounds, such as radicals and peroxides: highly reactive fuels are made of long and flexible molecules (alkanes with high molecular weight) which favour the formation of intermediate compounds, such as peroxides and hydroperoxides
- strongly exothermic phenomena, such as thermal cracking reactions, preceding the oxidation process.

1.1.2 Premixed Combustion Phase

One of the main problems encountered in the design of a diesel engine is to be able to obtain a correct mixing between the injected fuel and the air. Once the ignition delay of the injected nuclei has ended, their combustion causes a sudden increase in temperature and this allows an extreme acceleration of the oxidative process of the accumulated fuel which burns almost simultaneously, i.e. in an almost isochoric way, giving rise to a sudden surge in pressure. This increase, although advantageous in terms of thermodynamic efficiency, is responsible, in

addition to the noise previously mentioned, for the onset of vibrations and stresses of important entity, resulting in more robust mechanical structures which, resulting in high inertia of the engine, limit the rotation speeds of an engine of this type.
During this phase, a further drawback is added: reaching the high final temperatures creates suitable conditions for the formation of nitrogen oxides (better known as NO_x). In fact, their creation does not occur during premixed combustion because there are still rich mixture conditions (λ values of 0.25- 0.5).

1.1.3 Mixing-controlled Combustion Phase

Once the rapid combustion of the accumulation is exhausted, the process is regulated by the speed with which the new fuel fractions are available to burn, that is, by the speed with which they evaporate, mix with the combustion air and oxidize. Usually in this phase the maximum pressure of the cycle is reached, and the speed with which the energy is released can be controlled by acting on the injection rate. As the process progresses, the new injected fuel finds less and less oxygen available to react due to the progressive increase in the combustion gases inside the combustion chamber. Following processes of dehydrogenation, condensation and pyrolysis, unburnt carbonaceous nuclei, or soot, can form, consisting of solid particles containing a high number of C atoms (with H/C equal to 0.1 circa). This phase corresponds to diffusive combustion, in which about 90 % of the total fuel is burned.

1.1.4 Late Combustion Phase

The injection is now complete, but the chemical reactions are still progressing, gradually fading. Combustion can involve the carbonaceous nuclei that formed in the previous phase, allowing their oxidation. This last part is promoted by the turbulent motions that mix the gases inside the chamber, but it is necessary that this is not prolonged excessively so as not to reduce the efficiency of the engine heat release around the TDC).
After the brief description of the combustion process, we proceed by analyzing the polluting emissions produced by diesel engines, in particular we focus our attention on what concerns the causes and formation of the soot.

1.2 CI Engines Polluting Emissions

One of the main defects of internal combustion engines is linked to the production of pollutants during the combustion phase. In fact, they suck in air from the atmosphere, subsequently

discharging the combustion products into it, thus altering their natural composition, generating the phenomenon more commonly known as atmospheric pollution.

Pollutants can be divided into three macro-categories:

- products of incomplete combustion (CO, unburnt hydrocarbons, peroxides, etc.), which are toxic and capable of producing different physiological damage depending on their chemical composition
- complete oxidation products of substances present in the fuel such as sulphur and nitrogen
- complete oxidation products, already present in the atmosphere as CO_2 and H_2O, but for which the natural production cycles alternate, helping to modify the balance between absorbed and released energy, generating the phenomenon more commonly known as the greenhouse effect.

Road traffic has always been the main cause of nitrogen oxide emissions and a significant source of fine dust, as well as particulate matter, produced by diesel engines. Despite the sharp increase in traffic, emissions have decreased over the years thanks to technological innovations, necessary for the increasingly stringent limitations imposed by the European directives, in the automotive sector. In 2016, the percentages of emissions from world road transportation compared to the total were[2]:

- over 50 % for nitrogen oxides (NO_x)
- over 25 % for carbon monoxide (CO)
- 15 % circa for sulphur dioxide (SO_2)
- 7 % circa for fine powders ($PM_{2,5}$).

These values are undoubtedly alarming, given the effects they have on human health. Following studies conducted on diesel engines, it has been seen that for this kind of engines, the presence of unburnt hydrocarbons (HC) and CO is much less relevant than that of NO_x and particulate matter (PM).

1.2.1 Pollutants Formation Mechanisms

Old models for the description of the combustion phenomenon were based exclusively on the pressure and heat release signals and hypotheses and assumptions were advanced on the behaviour of the fuel jet in the chamber. Thanks to the advent of instruments for optical analysis, more recent models based on scientific evidence have spread. The traditional combustion process in a compression ignition engine can be described by conceptual model proposed by Dec, who

[2] "Special report on Energy and Air Pollution, 2015", IEA, WEO 2016

conducted experiments on engine equipped with optical access. To understand the formation of polluting particles, it becomes essential to analyze the fuel jet behaviour and its transformations through the multi-hole injector. In Figure 3 evidence from Dec's experiments is disclosed and it is possible to see what happens to the injected fuel during the combustion process. The liquid phase penetration inside the chamber is limited by the evaporation of the jet itself, which tends to occur rapidly following the conditions of high pressure and temperature. During the experiments, the injection pressure is kept as constant as possible.

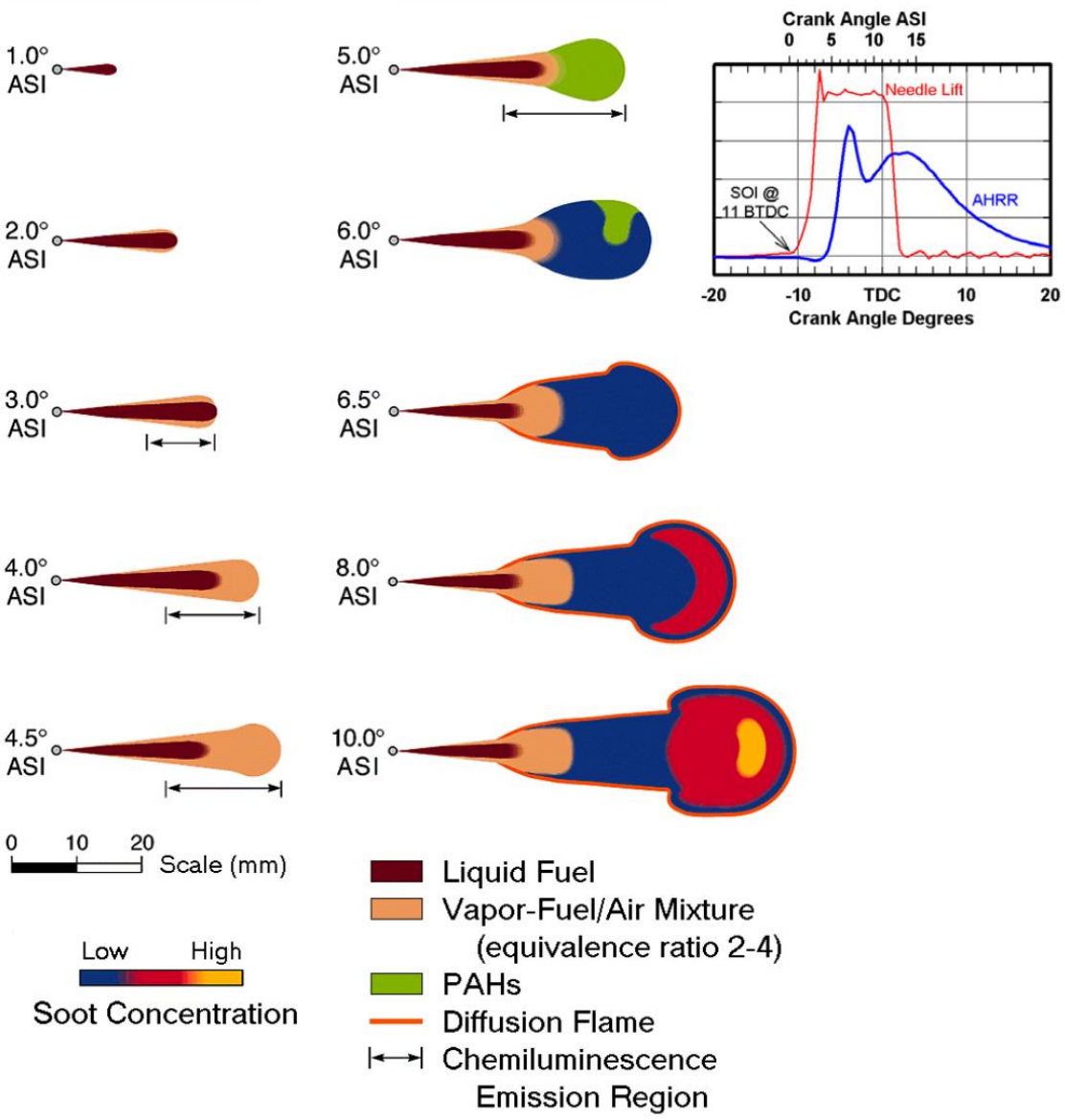

Figure 3. Schematic temporal sequence of a fuel spray in CI engines combustion [5,7]

Some injectors today are of the electromechanical indirectly operated type: thus, there is a delay between the electric control and the moment in which the injector opens (hydraulic injection start). The first phase corresponds to the penetration of the jet inside the chamber up to 3° ASI (After the Start of Injection, it refers to the actual injection start, not the electric command)

where it stops and subsequently becomes smaller, simultaneously with the formation of fuel vapours around the jet itself. The evaporation phenomenon determines the area in which we no longer have liquid penetration and depends on the thermodynamic conditions present in the chamber. After just 5° chemiluminescence appears: in correspondence with the creation of the mixture zone, a weak luminosity begins to come, not due to combustion, but to the initial reactions of the fuel with oxygen forming some radicals, without an important release of energy and increase of pressure. In this phase there is a rich combustion and, following the descent below the flash point, fragments of fuel are found: the fuel molecules do not undergo oxidation but breakage, so the $C - C$ bond gives rise to the formation of polycyclic aromatic hydrocarbons (PAH), which correspond to the precursors of soot. As the process continues, PAHs grow into quite large particles: this represents the so-called growth mechanism. Subsequently, the diffusive flame can be observed, characterized by a substantial release of chemical energy and an approximately unitary dosage value. Proceeding with the injection, the spray continues to enlarge and the temperature increases, continuing to create conditions suitable for the formation of carbonaceous particles, which are more numerous at the tip of the jet thanks to the longer time available and the higher thermodynamic conditions. If the diffusive flame intersects the bowl, obtained on the head of the plunger, it goes out due to the heat exchange and the lack of oxygen, causing the non-oxidation of the carbonaceous particles that we will find at the discharge, but only partially after having participated in soot burnout. Nitrogen oxides NO_x are formed under conditions of high temperatures and, unlike soot, in the presence of oxygen. In fact, the diffusive flame zone is the best one for their formation.

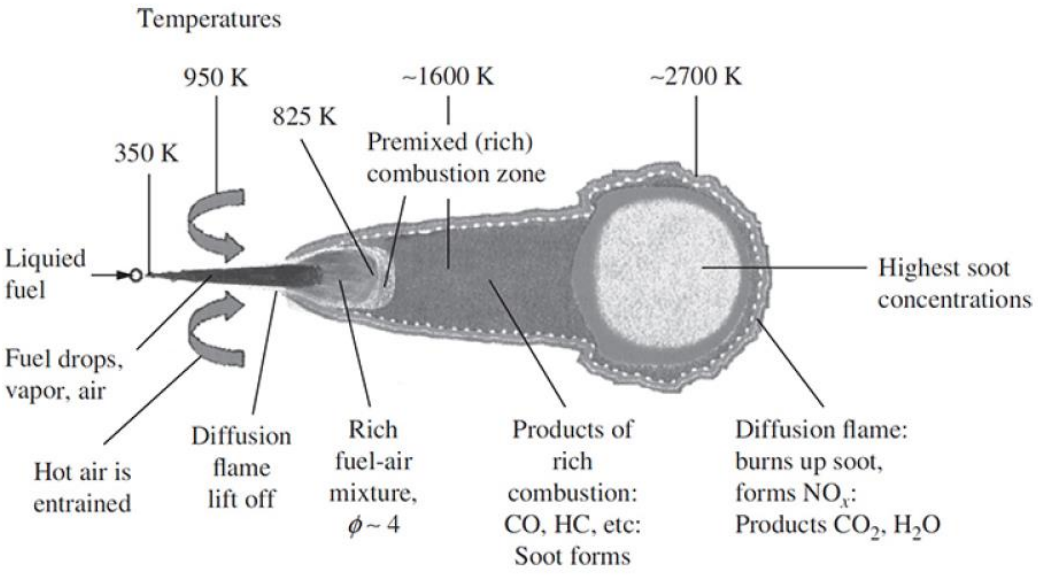

Figure 4. Schematic picture of CI engine fuel spray [7,10]

In Figure 4 the description of the pollutant formation in CI engines exposed above is summarized. Briefly, the soot begins to form as soon as the premixed combustion (rich combustion products such as fuel fragments, HC and PAH) has been overcome and with temperatures of about 1600 K. Once reached the level of the diffusive flame, the temperatures increase (up to at 2700 K) and the concentration of oxygen, thanks to its diffusion from the outside, allow the oxidation of the soot and any unburnt hydrocarbons. However, these conditions facilitate the formation of NO_x. The right compromise between the two pollutants, determining the well-known soot-NO_x trade-off, is described by the Kamimoto-Bae diagram, reported in Figure 5. It represents a map of the equivalence ratio ϕ (the inverse of the relative air-fuel ratio λ) as a function of the temperature and it highlights the formation zones of the two pollutants, hence also the conditions (from thermodynamic and chemical composition points of view) that lead to their development.

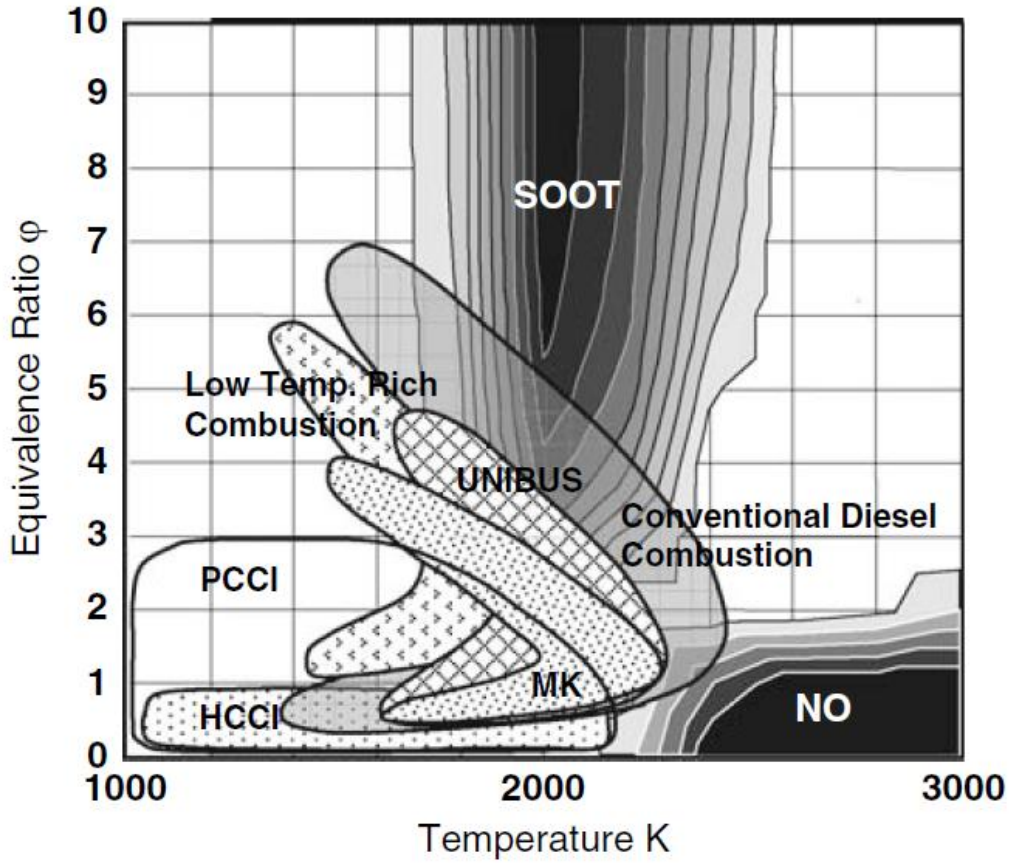

Figure 5. Kamimoto-Bae diagram with conventional and advanced combustion processes [13]

In Figure 5, together with the pollutant formation zones, the typical areas where advanced and conventional combustion processes develop are highlighted. The portion of the diagram with new generation HCCI and PCCI combustions is a coveted research objective as it shows the absence of both pollutants but is still difficult to achieve.

1.2.1.1 Nitrogen Oxides Formation

In CI engines, nitrogen oxides (NO_x) are formed for the most part by NO (70-80 %) and the remainder by NO_2. The main formation mechanism is the thermal one, which occurs within the combustion products, and consists of the Zeldovich model, according to which the formation of nitrogen monoxide is described by the following reactions:

$$N_2 + O \leftrightarrow NO + N$$
$$N + O_2 \leftrightarrow NO + O$$
$$N + OH \leftrightarrow NO + H$$

The first reaction is the one that requires the highest activation energy, available thanks to the high temperatures and presence of oxygen. Compared to the combustion process, the formation of nitrogen oxides is slow, but with times compatible with the duty cycle of the engine. This reaction takes place in the combustion products and not in the diffusive flame, remaining at a high temperature for a long time to allow the reactions to take place. The process is strongly influenced by chemical kinetics, but this is beyond the scope of this thesis work. As regards, instead, the formation of NO_2, we refer to the reactions:

$$NO + HO_2 \leftrightarrow NO_2 + OH$$
$$NO_2 + O \leftrightarrow NO + O_2$$

In Diesel engines the first reaction always takes place, while the second one is easily interrupted due to the lower temperatures in the combustion products compared to spark ignition engines, therefore the conversion into carbon monoxide stops, allowing the presence of carbon dioxide and nitrogen to the exhaust.

1.2.1.2 Particulate Matter Formation

Particulate Matter (PM) derives from carbon particles (soot) developed during the incomplete combustion of the fuel and brought into suspension by the exhaust gases. This pollutant can be present both in SI engines with direct injection and, above all, in CI engines. PM is usually referred to using an operational definition which indicates as particulate matter everything that can be collected on a filter through which the exhaust gases (diluted with air to bring their temperature below 52 $°C$) are passed. A PM particle model is schematically drawn in Figure 6.
The development of the particulate begins during the combustion process in the cylinder and continues along the path that brings the exhaust gases into the atmosphere, thus passing through the ATS (After Treatment System). PM precursors develop immediately after the start of

combustion at the central parts of the jet which are unable to undergo an optimal mixing process with oxygen. These areas are in fact characterized by a rich mixture (low λ) with a low presence of oxygen.

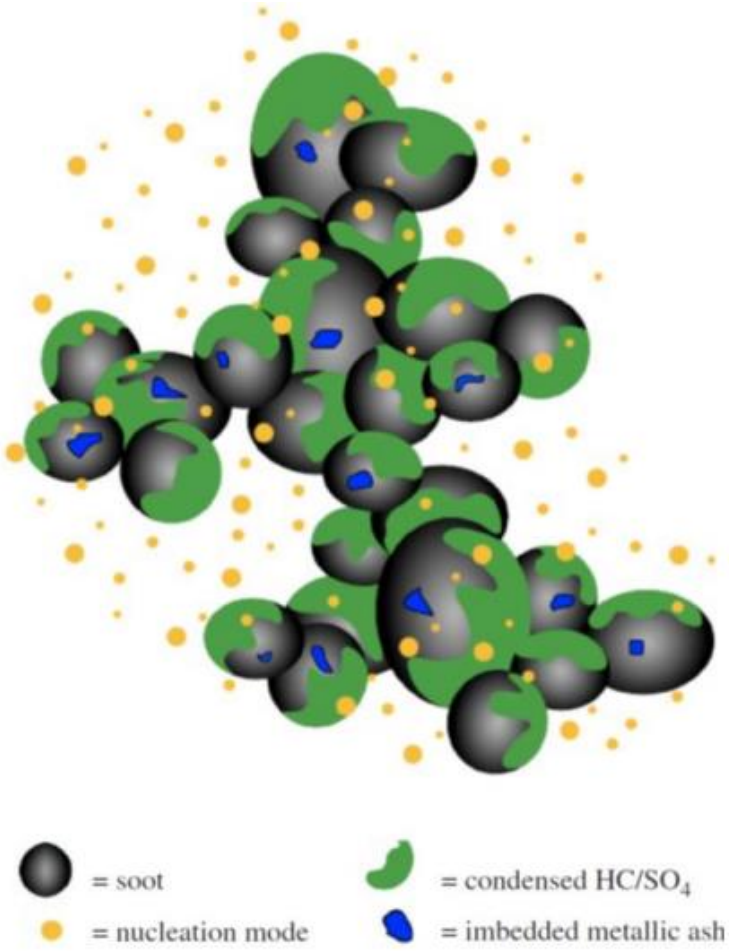

Figure 6. Schematic picture of a PM particle [14]

In this phase there is a rich combustion and, following the drop below the ignition point, fragments of fuel are found: the fuel molecules, not undergoing oxidation but breaking, generate polycyclic aromatic hydrocarbons (PAH), which correspond to the precursors of the soot. In this area the hydrocarbon molecules (having a H/C slightly lower than 2) decompose due to the thermal effect, progressively losing hydrogen due to a phenomenon called pyrolysis. Molecular agglomerates of carbon atoms very poor in hydrogen (with H/C equal to 0.1 circa) are thus generated.

Subsequently, the precursors interact with the molecules of unsaturated hydrocarbons and radicals poor in hydrogen, thus forming the primary constituent elements of PM with dimensions in the order of nanometers. There is thus a surface growth caused by the attachment to the surface of the precursors of chemical species with a high carbon content, generating nuclei, also called spherules. Part of these nuclei are put into suspension by the exhaust gases, while others

become the constituent elements of larger particles, resulting from the agglomeration of several nuclei by collision, coagulation, or aggregation.

In each of the various stages of the PM formation process, oxidation phenomena can develop (in the presence of high temperatures and concentration of oxidizing species) which lead to the combustion of the particulate, obtaining CO_2 and CO between the burnt gases. Consequently, the final emission of particulate matter from the engine will depend on the balance between the oxidation and formation processes.

As a result of its development, the particulate is put into suspension by the exhaust gases by strongly decreasing its temperature. If the temperature is below 500 $°C$, the carbonaceous particles are coated by low vapor pressure substances, such as high molecular weight hydrocarbons, sulphates, and water, which are absorbed due to the high porosity of the surface.

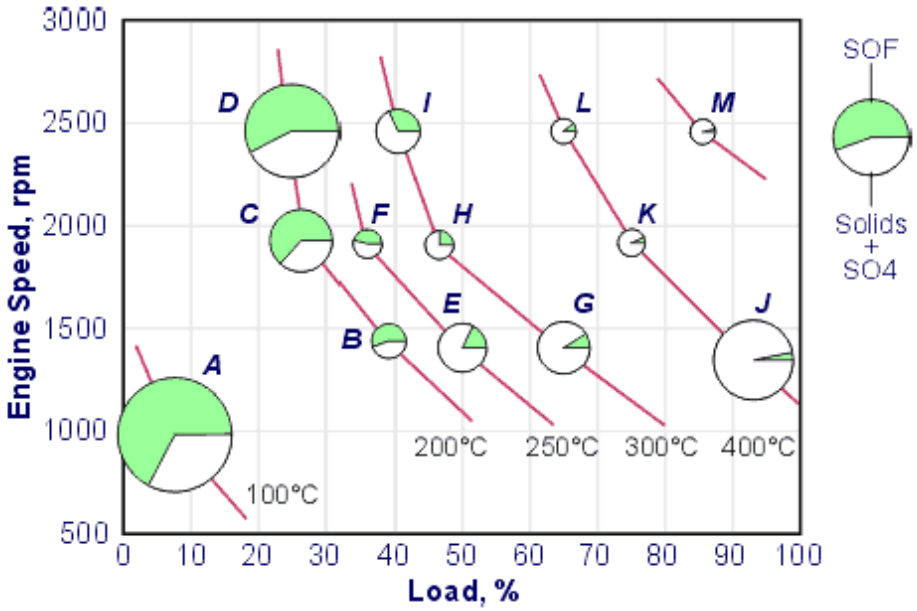

Figure 7. PM composition changes by varying engine speed and load conditions [15]

The PM is made up of:

- A solid part (SOL) – Made up of carbonaceous elements and ashes, it is the main component of PM and is known as soot. During the combustion process, the particulate precursors generate these carbonaceous particles which grow and aggregate. Initially they have hexagonal structures (order of nanometers) that can stack on top of each other to give lamellar structures that sometimes aggregate into more or less spherical particles. These particles are not compact and have non-negligible interstices and spaces. The ashes are instead incombustible substances (due to metal additives, iron oxides) that can form in the combustion chamber or in the exhaust manifolds.

- A soluble part (SOF) – It consists of organic compounds (hydrocarbons) that may derive from the lubricating oil or fuel. The largest organic fraction derives from the former one and it is mainly rich in compounds such as C_{20}.
- Sulphates (SO_4): they can derive from water or sulfuric acids and are formed downstream of the combustion process, when temperatures decrease, therefore in the exhaust pipe.

The percentages of these three categories present in the particulate matter depend on the operating conditions of the engine, that is the load and the rotational speed. Figure 7 reports a real-world example of the PM composition for a particular engine.

The problem of particulate emissions has become increasingly important for manufacturers of vehicles that mount Diesel engines since the latest anti-pollution regulations no longer require only a limitation of the mass emissions of PM, but also that of the number of particles emitted by the engine. In fact, the particulate particles can have different sizes, and for this reason, the smallest have been neglected for a long time; however, recent studies have shown that also these are harmful to human health. In this sense, a classification of the PM particles can be carried out with reference to the relative size:

- Nuclei mode – It is composed of ultrafine particles and nanoparticles, which are characterized by a different quality. The finer ones are carbonaceous compounds surrounded by organic elements that have been adsorbed by the solid particle; most are SOF molecules condensed to form larger droplets. This does not happen during combustion, but at the exhaust or outside when temperatures are particularly low. These types of particles are very unstable.
- Accumulation mode – Fine particles of this type are mostly all carbonaceous and, on them, during the discharge phase, organic compounds are deposited.
- Coarse mode – This type is formed by wear of the exhaust parts and are the largest particles. They are low both in number and in mass, so they have a low incidence, but some of these can also form downstream of the particulate filter.

In Figure 8, the trend of the normalized concentration of PM particles as a function of the diameter shows that the distinction (pursued in current regulations) between the number of PM particles and the total mass of PM is relevant. In fact, profoundly different distribution curves can be noted from the figure. The particle number distribution shows that a large number of very small and light particles will be emitted from an exhaust from a compression ignition engine; vice versa, the distribution relative to the mass of PM shows that, despite being emitted in a decisively lower number, the particles in accumulation mode or coarse mode are those of greater weight and size.

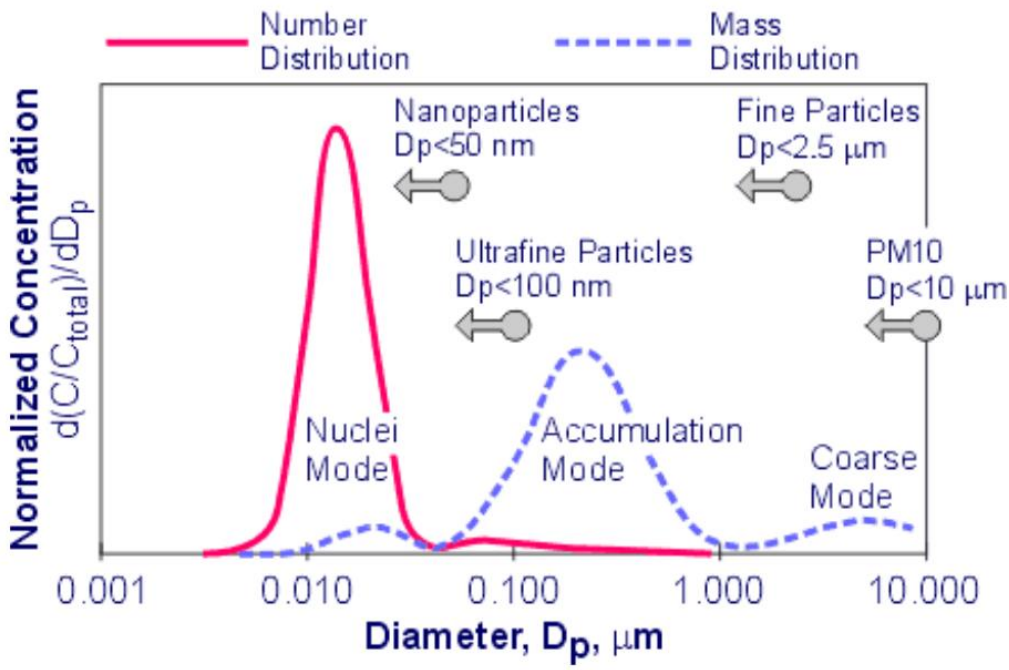

Figure 8. PM particles number and mass (normalized) distributions versus the particle diameter [15]

1.2.2 Regulatory Framework

To reduce polluting and climate-altering gas emissions, car manufacturers must comply with regulations that require the engines emissions to be assessed in order to obtain vehicle approval. For what concerns European legislation, the emissions standards are highly changing over time and have the aim of reducing emissions from internal combustion engines in an increasingly stringent trend.

It started from the Euro 1 directive, adopted for all homologated cars until December 1991. For now, the last step arrived in September 2014, the year in which the Euro 6 legislation has been published, which must be applied for all new models approved from that year on. Figure 9 reports the full emission standards history for SI and CI engines (referred to passenger cars).

Subsequently, starting from January 2016, different versions of the Euro 6 legislation are introduced:

- Euro 6a, compulsory for vehicles registered since January 2016. The limits for CO emissions are 0.5 g/km for diesel and 1 g/km for petrol, those of NO_x 0.080 g/km for diesel and 0.060 for SI engines, those of particulate matter (PM) 0.005 g/km for both types of engines.
- Euro 6b, PM emission limits are reduced to 0.0045 g/km.
- Euro 6c, mandatory for homologations since September 2017, requires petrol cars to reduce PM from 6000 billion particles per km to 600 billion per km. In addition, a new WLTP (Worldwide harmonized light vehicles test procedure) test cycle was introduced,

which replaces the NEDC (New European driving cycle) and the RDE (Real Drive Emission) road test only for monitoring.

- Euro 6d-temp, for homologations from September 2018, also introduces the measurement of on-road emissions known as RDE. In this test the differences between the emissions in the cycle in the laboratory (limit for NO_x equal to 80 mg/km for Diesel) and in that on the road can reach a maximum of 110 % (limit of 168 mg/km for Diesel).
- Euro 6d complete, for registrations from January 2021, establishes that the difference detected between the WLTP emissions and those measured in the RDE test cannot exceed 50 % (limit for diesel 120 mg/km).

Stage	Date	CO	HC	HC+NOx	NOx	PM	PN
		g/km					*#/km*
Positive Ignition (Gasoline)							
Euro 1†	1992.07	2.72 (3.16)	-	0.97 (1.13)	-	-	-
Euro 2	1996.01	2.2	-	0.5	-	-	-
Euro 3	2000.01	2.30	0.20	-	0.15	-	-
Euro 4	2005.01	1.0	0.10	-	0.08	-	-
Euro 5	2009.09[b]	1.0	0.10[d]	-	0.06	0.005[e,f]	-
Euro 6	2014.09	1.0	0.10[d]	-	0.06	0.005[e,f]	6.0×10^{11} [e,g]
Compression Ignition (Diesel)							
Euro 1†	1992.07	2.72 (3.16)	-	0.97 (1.13)	-	0.14 (0.18)	-
Euro 2, IDI	1996.01	1.0	-	0.7	-	0.08	-
Euro 2, DI	1996.01[a]	1.0	-	0.9	-	0.10	-
Euro 3	2000.01	0.64	-	0.56	0.50	0.05	-
Euro 4	2005.01	0.50	-	0.30	0.25	0.025	-
Euro 5a	2009.09[b]	0.50	-	0.23	0.18	0.005[f]	-
Euro 5b	2011.09[c]	0.50	-	0.23	0.18	0.005[f]	6.0×10^{11}
Euro 6	2014.09	0.50	-	0.17	0.08	0.005[f]	6.0×10^{11}

* At the Euro 1..4 stages, passenger vehicles > 2,500 kg were type approved as Category N_1 vehicles
† Values in brackets are conformity of production (COP) limits
a. until 1999.09.30 (after that date DI engines must meet the IDI limits)
b. 2011.01 for all models
c. 2013.01 for all models
d. and NMHC = 0.068 g/km
e. applicable only to vehicles using DI engines
f. 0.0045 g/km using the PMP measurement procedure
g. 6.0×10^{12} 1/km within first three years from Euro 6 effective dates

Figure 9. EU pollutant emissions standards for passenger vehicles (source: dieselnet.com)

1.3 After-treatment Systems

The combustion gases coming out of compression ignition engines, before being released into the atmosphere, are treated through a system of devices designed to reduce polluting emissions, bringing the values within the limits imposed by the regulations. Two ways are used to meet this requirement, the first is to adopt techniques for reducing pollutants in the combustion chamber through various techniques such as:

- the use of a fuel with better characteristics
- optimization of the geometry of the intake ducts and the combustion chamber to ensure appropriate mixing in the chamber
- the development of advanced injection systems, with appropriate injection laws
- control of the engine thermal state, which influences the fuel vaporization rate.

However, although these techniques are effective, they do not allow to reach the complete achievement of the stringent regulatory limits. For this purpose, exhaust gases after-treatment systems (ATS) are then used downstream of the engine exhaust manifold to reduce the concentrations of the polluting species produced.

The ATS system is generally composed of:

- DOC (Diesel Oxidation Catalyst), which is used to convert HC, CO and the PM carbonaceous fraction into CO_2 and H_2O
- DPF (Diesel Particulate Filter), that has the task of filtering the emitted particulate through a ceramic monolith with a honeycomb structure, drastically reducing its quantity
- SCR (Selective Catalytic Reduction), which is used to reduce NO_x through a series of chemical reactions that involve the use of a reducing agent (generally ammonia or urea) to break down nitrogen oxides into N_2 and H_2O.

Since the core subject of this work is represented by the soot (the solid fraction of PM), reference will be made to the functioning of the DPF, as its main abatement system.

1.3.1 Diesel Particulate Filter

As mentioned in the previous paragraph, the actions developed in the combustion chamber are not sufficient to significantly reduce the products of the combustion product, and especially the PM. To reduce this pollutant, it is necessary to use a device, known as Diesel Particulate Filter (DPF), that allows to filter the gases by partially removing the soot particles, which remain trapped by the filter itself.

This system is a mechanical filter that consists of a certain number of blind channels: in particular, they are blind towards the entrance or exit in an alternating manner, to force the flow of gas to pass through the walls equipped with a certain porosity, as shown in Figure 10. The separation between gas and particulates occurs mechanically and can occur in two ways (Figure 11):

- Deep bed filtration – Particles which have smaller diameters than the pores of the filter matrix are trapped passing through it thanks to electrostatic actions.

- Soot cake filtration – Particles that have slightly larger diameters than the porous matrix, instead of passing through it, settle on the matrix itself and, consequently, the following ones will deposit on the previous ones, forming layers and generating what is commonly called soot cake. As its thickness increases, the filtering property of the device increases and therefore also its filtering efficiency.

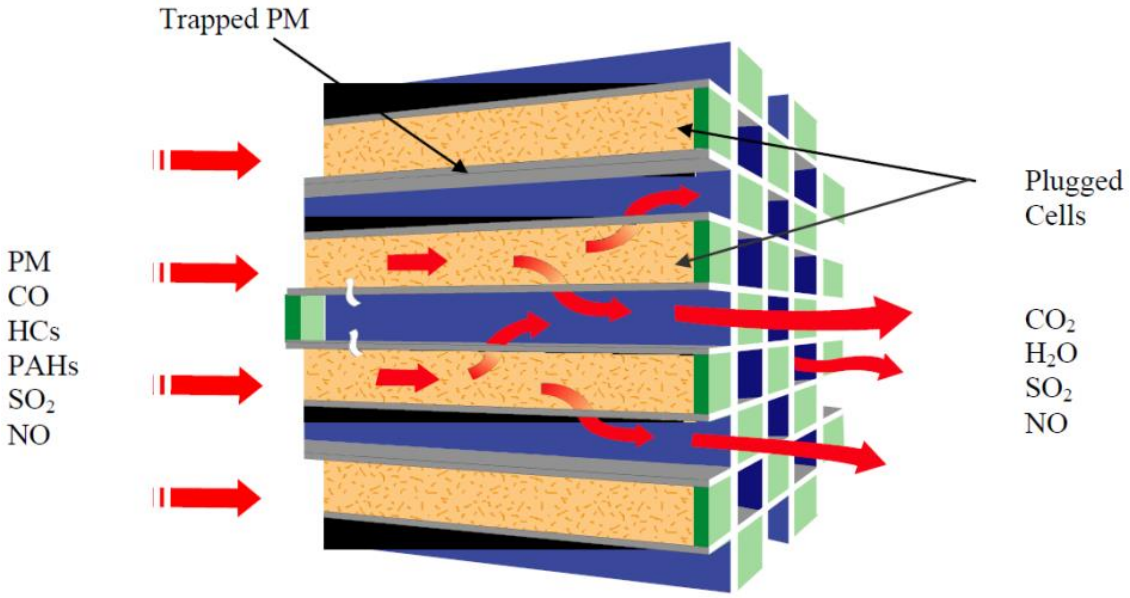

Figure 10. DPF structure and operating principle (source: Emission Control Technologies for Diesel-Powered Vehicles, White paper, MECA)

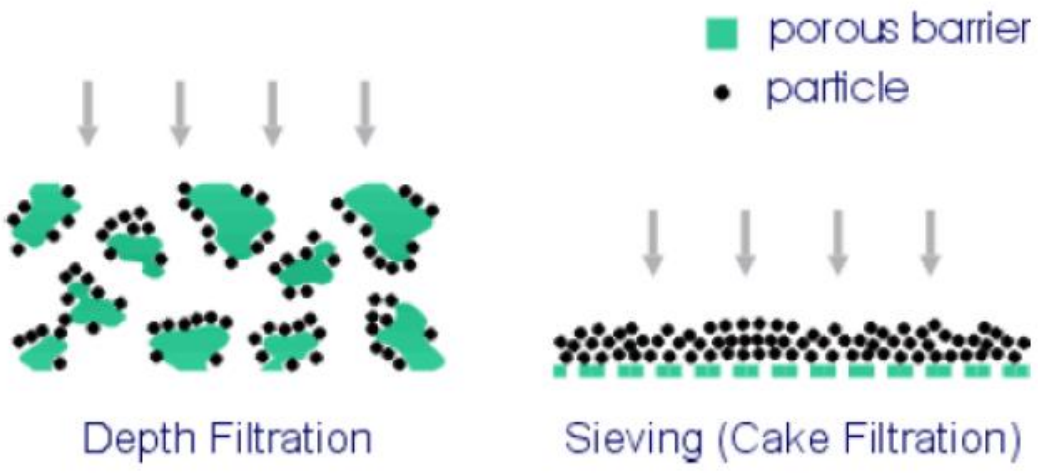

Figure 11. DPF soot filtration principles [15]

The advantages of using a particulate filter are here disclosed: high filtering efficiency; low pressure drop (approximately 2 kPa with clean filter, 20 kPa with dirty filter); high thermal and mechanical resistance; good accumulation power of carbonaceous particles.

However, as the accumulation of soot inside the filter increases, there is a risk of its clogging with the consequent increase in the pressure drop across the filter (Δp_{DPF}). This has a negative effect in terms of engine performance, since in a turbo-diesel engine upstream of the ATS there is a turbine, whose operation is strictly linked to the pressure upstream of the ATS. Therefore, the increase in Δp_{DPF} causes a back pressure downstream of the turbine, reducing its performance and consequently those of the engine, also causing an increase in fuel consumption. Cleaning the filter is therefore necessary to eliminate accumulated soot particles. To do this, the regeneration of the particulate filter is carried out through the oxidation of the PM. This process can be developed in different ways depending on the following factors:

- oxidizing agent used (O_2, NO_2)
- periodicity (continuous, intermittent)
- cause producing the regeneration event (active, passive).

DPF regeneration can thus be classified as:

- Periodic – It refers to a forced regeneration that uses O_2 as an oxidizing agent and to which we must supply heat. Since the use of oxygen involves high temperatures (650 $°C$), a catalyst is used to lower them.
- Continuous: NO_2 is used as an oxidizing agent and, since it reacts at low temperatures, it does not require a catalytic agent.

The regeneration process can be of two types:

- Active – The oxidation reactions of the PM are activated by a controlled external energy supply. Appropriate fuel injection strategies, such as delayed ("after") injections, or the development of heat through a separate device (burners, electric heaters, etc.) are used.
- Passive – The use of suitable catalytic reactants allows the PM oxidation reactions to take place at the temperatures typically reached by the exhaust gases in normal engine operating conditions. The catalysts are added to the fuel in small percentages or are used to coat the filter channels, distributing them over a wider surface area (platinum and palladium are usually employed). In the oxidizing catalyst placed upstream of the DPF, the oxidation of CO, HC and NO occurs, causing the exhaust gas temperatures to increase by about 50 $°C$, thus managing to reach the threshold of about 250 $°C$, sufficient for the DPF passive regeneration start. The oxidation reactions that take place are the following:

$$[C] + 2NO_2 \longrightarrow CO_2 + 2NO$$

$$[C] + O_2 \longrightarrow CO_2$$

Chapter 2

Introduction to Machine Learning

The use of Artificial Intelligence algorithms and in particular neural networks are widely used in most industrial and research sectors; only a few years ago these algorithms began to be used in the automotive field to make improvements in terms of safety, control and emissions of cars.
The field of Artificial Intelligence (AI) is quite complex and vast, it includes several categories including that of Machine Learning (ML), which refers to the science of programming computers so that they can learn from data. A sub-category of ML is Deep Learning, which includes neural networks. A schematic representation of the relationship between these Data Science fields is reported in Figure 12.

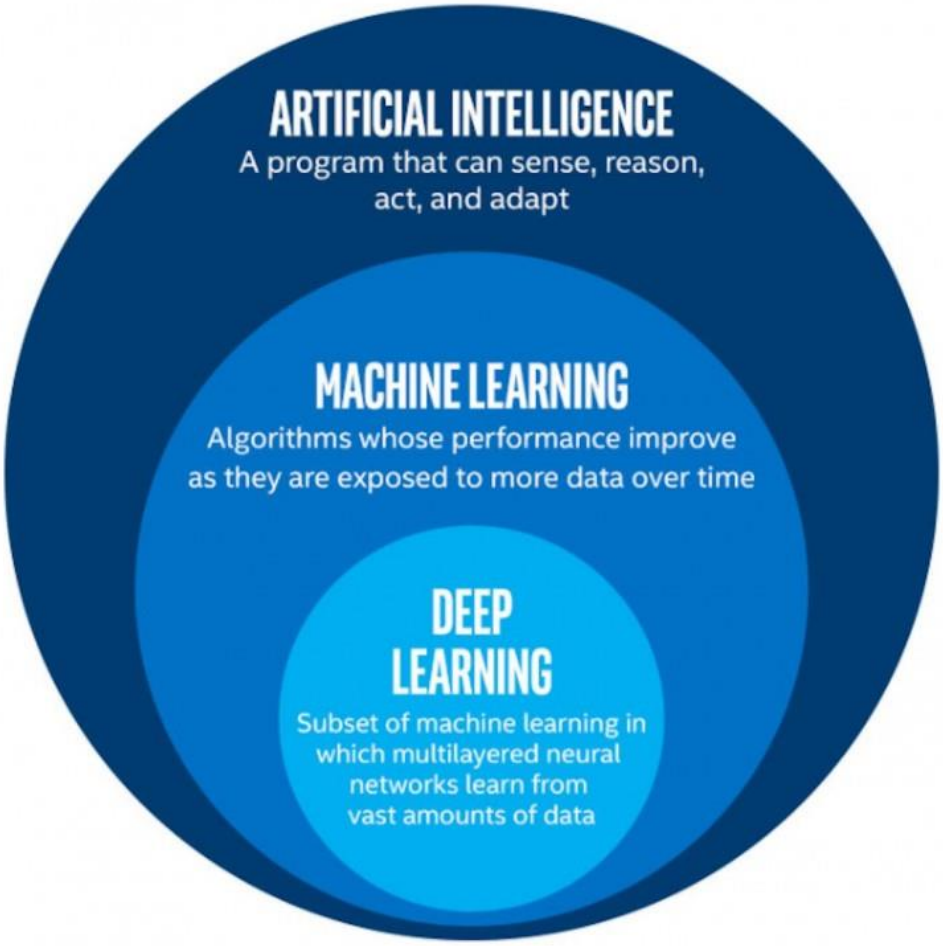

Figure 12. Schematization of the relationship between the different Data Science fields (source: interestingengineering.com)

In the following paragraphs we will develop a brief description of the Machine Learning and Deep Learning fields, in order to provide the definitions and basic concepts on these topics.

In this thesis work, scripts are used, written in Python language, which exploit various Machine Learning algorithms. We will see later in detail how these algorithms are constituted and how they work.

2.1 Algorithms Categorization

There are different types of Machine Learning algorithms, and they can be divided into multiple categories according to the following criteria:

- they may or may not be trained with human supervision (Supervised, Unsupervised, Semi-supervised and Reinforcement Learning)
- they may or may not learn incrementally during application
- whether they work simply by comparing the new data to those already known, or they detect the models through the training data and then build a predictive model.

These criteria are not exclusive but can be combined with each other to compose a more complex model. ML algorithms can be classified according to the type and degree of supervision they can obtain during training.

2.1.1 Supervised Learning

Currently, the most common types of algorithms are those that automate decision-making processes by generalizing from known examples.
In this setting, the user provides the algorithm with a series of input (features) and output (label) data on which to develop the learning phase. Subsequently, the algorithm will then be able to generate an output starting from an input never seen before.
A typical task of supervised learning is the Classification (it is schematically represented in Figure 13), in which the algorithm is trained on different known data, each with their own class, and then, subsequently, learn how to classify the new ones. A different type of task is constituted by the Regression (Figure 13 reports a schematic example of it) in which the algorithm itself must predict a certain numerical target value, given a certain set of features, called predictors. To train this system it is necessary to provide, in general, a lot of data containing both predictors and target values (labels) which can therefore be considered respectively inputs and outputs of the system. Hence, solving a regression problem corresponds to learning an approximating function of the given input-output pairs.
It can be seen how some regression algorithms can be used for classification, as well as the other way around. The most important and well-known supervised learning algorithms are the following ones: Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector

Machines, Decision Trees and Random Forests, Artificial Neural Networks. Of these, only the algorithms applied in designing the models used in this thesis work will be dealt with in detail.

2.1.2 Unsupervised Learning

In unsupervised learning algorithms, the training set is not labelled, that is, it does not contain the output values through which to train the model and, therefore, the classes for the data used for training are not known.
Among the algorithms of this type the most important are:

- Clustering – It involves identifying groups (clusters) of observations with similar characteristics in which the problem classes are not known and the data not labelled (without output). Often the number of clusters is not known a priori, but those identified in learning can then be used as classes. The unsupervised nature of the problem makes them more complicated than the classification ones. A schematic representation of this learning task is reported in Figure 13.
- Dimensionality reduction – It consists in reducing the number of dimensions of the input data, without losing too much information. The operation involves a loss of information, but the goal is to keep the important ones for the case, therefore dependent on the application. It is therefore very useful for making problems with very high dimensionality treatable, discarding redundant and unstable information, thus allowing the code to calculate faster thanks to the smaller disk space occupied by the data and, sometimes, to have higher performance.

2.1.3 Semi-supervised Learning

Semi-supervised learning algorithms work with only partially labelled training sets and distributing patterns with no output can help optimize the classification rule. Many such algorithms are combinations of supervised and unsupervised ones. A schematic representation of this kind of learning is shown in Figure 13.

2.1.4 Reinforcement Learning

A Reinforcement Learning algorithm aims to learn optimal behaviour from past experiences. The acquisition of knowledge is dedicated to an agent, who observes the surrounding environment and performs actions to modify it, causing changes from one state to another and receiving rewards which can also be penalties in the sense of negative rewards. The goal is to learn the

optimal action in each state, in order to maximize the sum of the rewards obtained in the long run.

It is considered important to underline an aspect concerning ML in general: since the main purpose is to train a learning algorithm on a certain set of data, its selection is extremely important as we could end up with a "bad algorithm" or "bad data", in the sense that they may not be compatible, or the chosen algorithm is not compatible with the application itself, or even there is only a small number of data available for training.

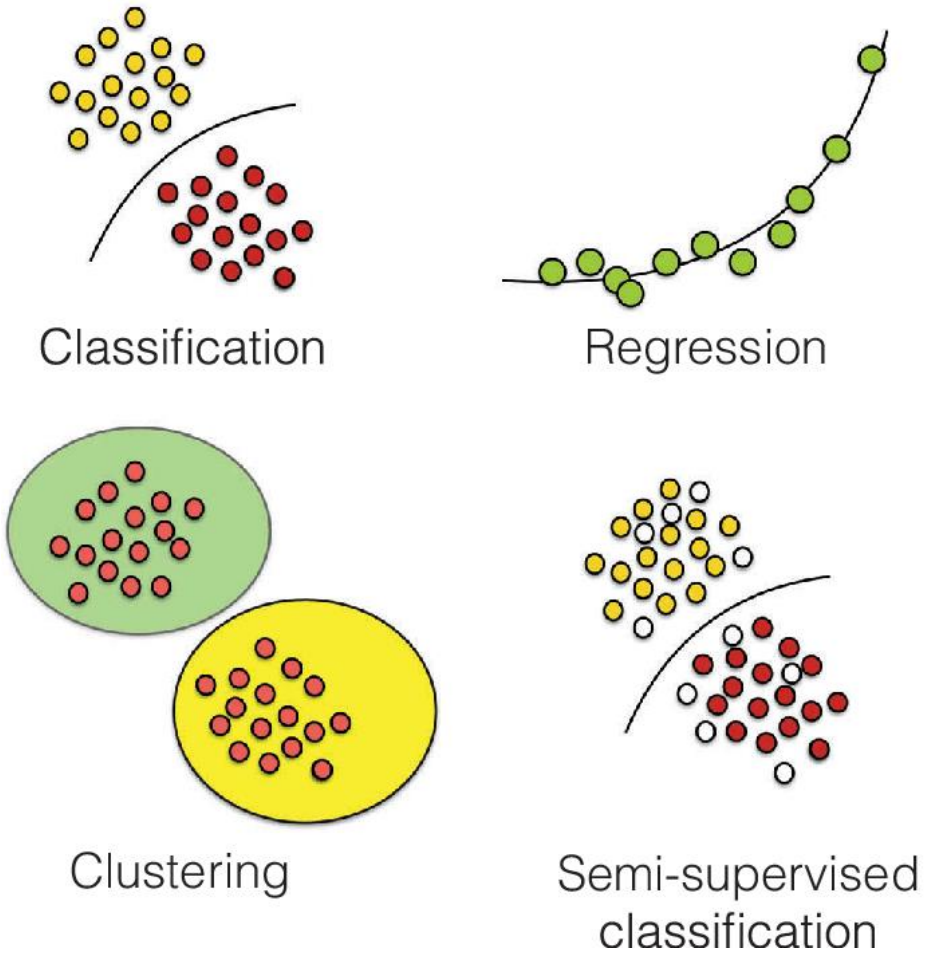

Figure 13. Schematic example of different Machine Learning tasks (source: Carrasquilla J., Machine learning for quantum matter, Advances in Physics: X, 5, 1797528, 2020)

2.2 Algorithm Training and Evaluation

Generally, the behaviour of a ML algorithm is governed by a set of parameters that characterize it. Learning an algorithm consists in determining the optimal value of these parameters.

Therefore, given a training set and a set of parameters, the objective function f (training set, parameters) can indicate:

- the optimality of the solution to be maximized
- the error or loss function to be minimized.

This function f can be optimized explicitly, with methods that operate starting from its mathematical definition (for example through partial differential calculus), or implicitly, using heuristic methods that modify the parameters coherently with the function itself.

Most of the algorithms ask to define, before the actual learning, the value of the so-called hyperparameters. These are the parameters that characterize the structure, behaviour and performance of the algorithm, such as the degree of a polynomial used in a regression, the type of loss function, the number of neurons or layers used in a neural network, and so on.

2.2.1 Dataset Subdivision and Cross-validation

To train the chosen algorithm and to know how it will behave on new data, the available dataset is divided into two parts: the training set and the testing set, in turn the first is divided into the actual training set and the validation set:

- Training set: observations through which to train the algorithm to recognize the patterns present in the data
- Validation set: observations on which the system calibrates the hyperparameters
- Testing set: data never seen before by the algorithm on which its performance is evaluated

To avoid using too much training data in the validation set and therefore not having enough for the algorithm training, a common technique to estimate the performance of a machine learning algorithm is that of cross-validation: the validation set is not separated from the entire training dataset, but the latter is split into k complementary subsets, after which the model is trained on $k-1$ subsets and validated on the remaining one. This operation is repeated so that each subset is given a chance to be the held back test set ending up with k different performance scores that you can be averaged. After running cross-validation, the hyperparameters values are selected based on those showing the best performances. Then, the final model is trained on the complete training set and, subsequently, applied to the test set to provide the generalization error. Cross-validation approach is characterized by less variance than a single train-test set split. It gives more reliable estimates of the model performance on new data. This technique is also more accurate because the model is trained and evaluated multiple times on different data. A complete schematization of cross-validation procedure is reported in Figure 14.

One of the first objectives to be pursued during the training phase is the convergence of the model on the training set and on the testing set. One of the main methods for studying this

convergence of a model is to evaluate the trend of the loss function obtained during the training and validation phases. Convergence is achieved if the output of the loss function has a decreasing trend compared to the number of iterations performed, however this is not always enough.

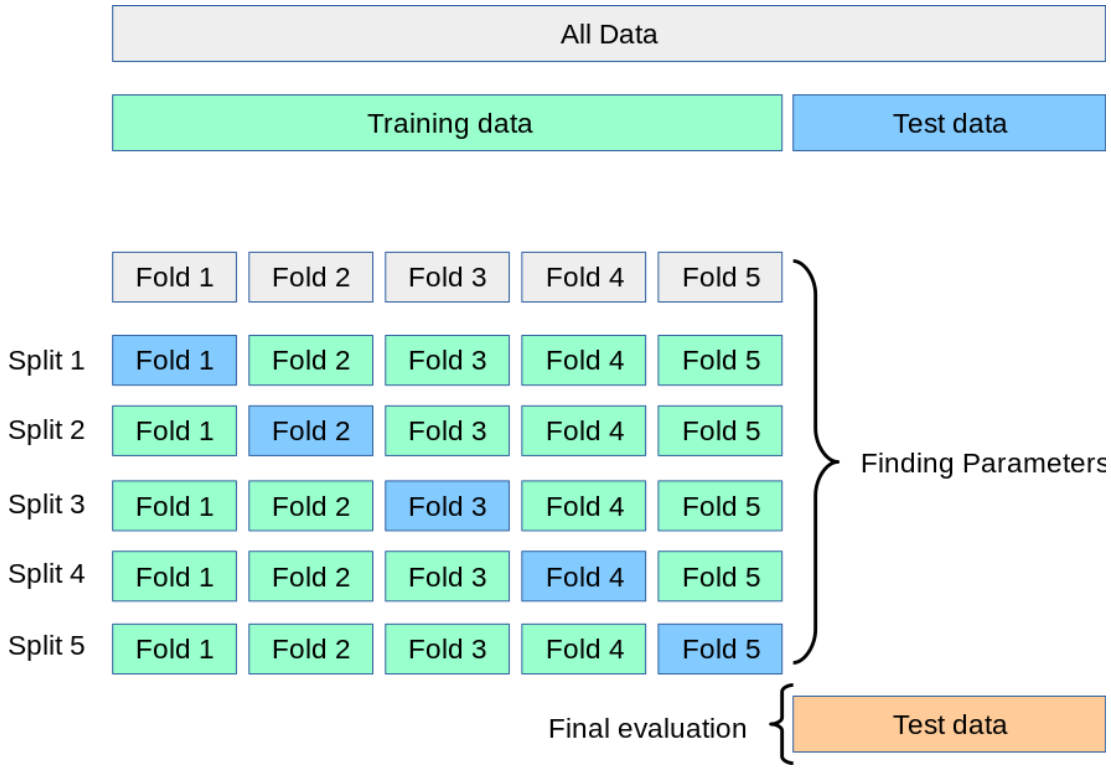

Figure 14. Cross-validation schematic representation (5-fold example case) [18]

If the loss function does not decrease (or oscillates) the system does not converge and the causes can be many: the optimization method is not effective, the hyperparameters are out of range (mainly an inadequate learning rate), there can be implementation errors, and so on.

A valid and widely used tool to verify the performance/behaviour of a machine learning model is the learning curve, which reports the evolution/trend of the learning capabilities of a model.

The training learning curve gives an idea of how the model learned from the training set, while the validation learning curve gives insights about how much the model is able to generalize. Generalization refers to the ability of the model to maintain the high performance achieved on training data even when applied to new data. Now, a complete machine learning pipeline can be drawn, as represented in Figure 15.

2.2.2 Performance Evaluation

The performance of an algorithm can be evaluated in two different ways: the first consists in directly using the objective function to quantify the performance, while the second consists in the

use of mathematical parameters of the field statistics and probability, more closely related to the characteristics of the problem.

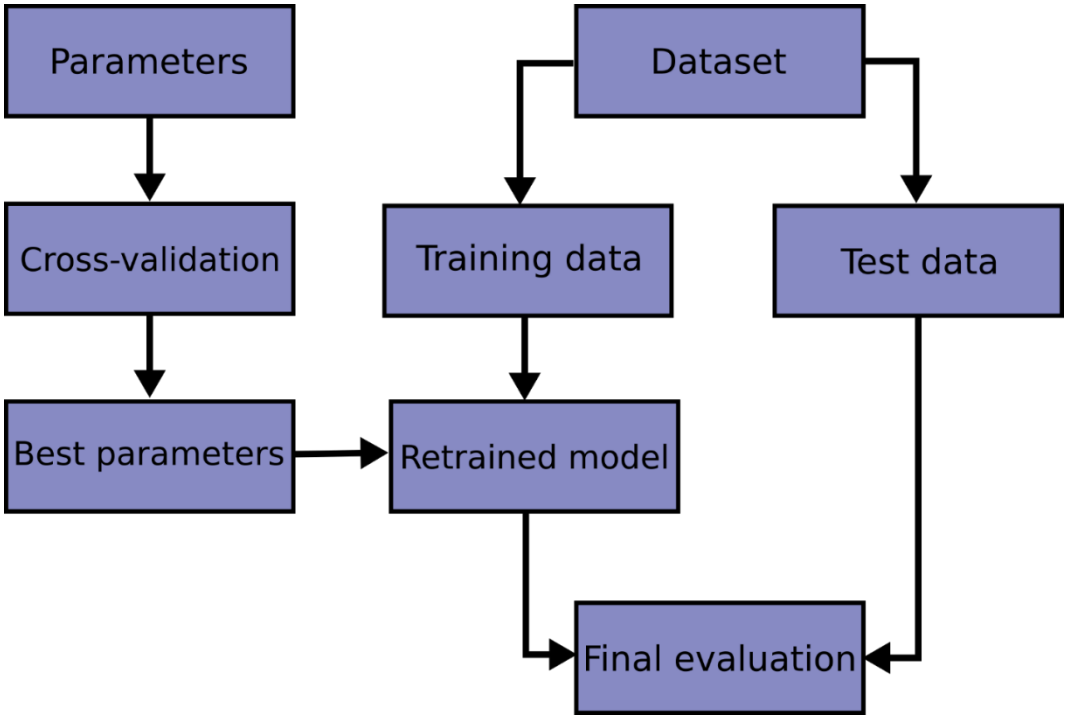

Figure 15. Typical machine learning pipeline [18]

With reference to a classification problem, accuracy is defined as the percentage of correctly classified data, while the error is the complement of this:

$$Accuracy = \frac{correctly\ classified\ samples}{classified\ samples} \cdot 100 \quad [\%]$$

$$Error = 100 - Accuracy \quad [\%]$$

In the case of a regression problem (object of the thesis), reference is mainly made to MSE (mean squared error). It is a quality measure of an estimator and it is always a positive value. This error embodies both the variance of the estimator (how wide the estimated values are spread among the observations) and its bias (how far the average of the estimated values is from the true mean value). The MSE represents the average of the squared errors, that is the average of the squared differences between the predicted and actual values, and is defined over N samples as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 ,$$

where $\hat{y}_i$ is the estimated value for the i^{th} sample and y_i is its corresponding actual value. It is also widely used in regression problems as loss function for algorithms fit or optimized using the

least squares framing. It is worth noting that the squaring has the effect of enlarging magnifying large errors because the larger the difference between the predicted and real values, the larger the resulting squared positive error. When MSE is used as a loss function this effect "punishes" models more for larger errors, when the error is used as a metric the same models are also penalized by inflating the average error score.

The RMSE (root mean squared error) is an extension of the mean squared error. It is obtained by calculating the square root of MSE, which means that the units of the RMSE are the same as the original units of the target value that is being predicted. This can give us an idea of the proportion of the error magnitude with respect to our problem scale. It is defined as the square root of the mean of the squares of the differences between target value and predicted value and it is scale dependent. RMSE then provides an estimate of how far the model is deviating from the real data and the relative formula for a dataset of N samples is:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}\ ,$$

where $\hat{y}_i$ is the estimated value for the i^{th} sample and y_i is its corresponding actual value.

MAE (mean absolute error) is a popular metric because, as we have described for RMSE, the units of the error score is equal to the units of the actual value that is being predicted. Importantly, the changes in MAE are linear and intuitive, unlike what we have seen for the previous metrics. In fact, MSE and RMSE punish larger errors more than smaller errors, magnifying the mean error score due to the squaring of the error value. Instead, MAE scores increase linearly with increasing errors. It evaluates the average dispersion present between the real and predicted values and is defined as the average of the absolute error values. The MAE is a scale-dependent metric whose mathematic formulation for a dataset of N samples is:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|\ ,$$

where $\hat{y}_i$ is the estimated value for the i^{th} sample and y_i is its corresponding actual value.

While RMSE is generally preferable as a measure of performance in a regression algorithm, sometimes MAE (also called average absolute deviation) can be used.

Furthermore, it is possible to evaluate the coefficient of determination, usually referred to as R^2. It indicates a proportion between the variability of the data and the correctness of the statistical model used. In fact, it represents the proportion of output values variance that can be explained

by the features (i.e., independent variables) in the model, also known as explained variance. It is very useful to understand how close the predicted values are to the real target value because it provides an estimate of the goodness of the model fit. The relative formulation is the following:

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2},$$

where y_i is the observed value for total n samples, $\bar{y}$ is the mean of the true values, $\hat{y}_i$ is the value predicted for the i-th sample. The best possible score for the coefficient of determination is one and it can be negative (that means the model can be arbitrarily worse). Instead, a constant model that always predicts the mean value (i.e., the expected value) for the output, disregarding the input attributes, would get a R^2 score of zero. In other words, it is a measure of how well the model will perform in predicting output values for unseen samples. Note that comparing coefficients of determination across different datasets may not be meaningful because the variance is dataset dependent.

2.2.3 Overfitting and Underfitting

The purpose of an ML algorithm is to maximize performance on the test set, thus ensuring good generalization skills. After training, as said before, the learning algorithm should be able to predict the outputs for input data never seen before, that is, it should be able to generalize.

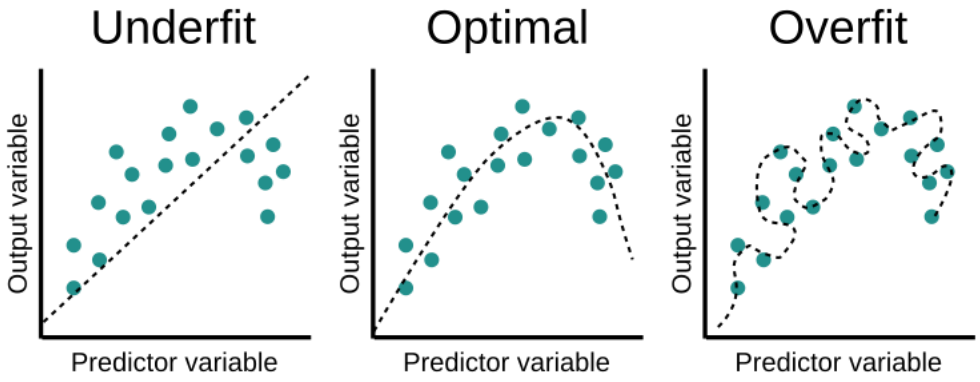

Figure 16. Illustration of underfitting and overfitting phenomena, and an optimal fit example (source: fastaireference.com)

However, especially in cases where the training develops over an excessive number of iterations or when a limited number of training examples are available, or if the choice of hyperparameters is not optimal, the model could adapt to specific characteristics of the training set, but which are not reflected in the rest of the cases. When this happens, because for example the model is characterized by an excessive number of parameters compared to the number of observations, we speak of overfitting, that is the model achieves high accuracy on the training set but not on the

validation one (poor generalization). It is therefore good practice to start with a few degrees of freedom, controllable through the hyperparameters, and gradually increase them, simultaneously monitoring the performance on the training and validation sets.

The opposite concept to overfitting is underfitting, which occurs when the model is too simple to learn from the structure of the data provided during the training phase, failing to obtain a sufficiently low error for the training set. This can happen when the parameters that define the algorithm are too few compared to the complexity of the data used for learning. The main strategies to solve this problem are here disclosed: to choose a model with more parameters (i.e., with greater learning capabilities, the degrees of freedom of the model are not sufficient to handle the complexity of the problem); to provide a higher number or more significant features to the algorithm; to reduce model constraints.

Examples of both underfitting and overfitting phenomena are shown in Figure 16, together with an optimal fit of the data. A model affected by underfitting can be diagnosed through the training curve, as it remains constant or decreases without ever coming to an end. An inconsistency in the training or validation dataset might also occur if the samples used are not representative of the entire dataset. To solve this problem, it is often necessary to use larger training and validation sets to have samples that are more representative of the problem.

2.3 Virtual Sensoring Applications

One of the most important applications in the field of Machine Learning is virtual sensoring, that is the use of software and applications (virtual sensors) to monitor certain physical parameters of interest or to optimize predictive maintenance techniques. In recent years there has been an increasing use of this type of sensors to replace real sensors. In fact, many times it is not possible to measure a certain quantity through physical instruments for various reasons, such as cost, feasibility, reliability, convenience, and so on. In addition, real sensors can be affected by natural aging with consequent lowering of reliability and performance.

To overcome these drawbacks, the use of virtual sensors is thus used. By virtual sensor we mean a software that, starting from a series of quantities (system variables) that are easy to measure, returns an accurate estimate of values that are difficult to acquire or measure directly.

Chapter 3

Introduction to Deep Learning

Machine learning algorithms perform well on a large number of tasks with excellent results. However, it is often necessary to use algorithms belonging to the Deep Learning category. This is partly due to the failure of some types of algorithms to perform the tasks imposed by artificial intelligence. In particular, one of the main reasons is related to the difficulty that simpler algorithms encounter in solving problems with multidimensional data (for example, observations characterized by tens or hundreds of thousands of features). This problem is known as the 'curse of dimensionality': it is a phenomenon that often occurs in many fields of computer science posing important challenges, difficult to solve with high accuracy through traditional algorithms. As the complexity of the problems increases, the number of features also increases, which in turn entails a sustained growth in the number of observations necessary to train the algorithm.
Formally, Deep Learning (DL) is a sub-field of machine learning that is based on algorithms that attempt to learn several levels of representations, corresponding to different levels of abstraction. These levels can be thought as a hierarchy of features or concepts, where higher-level concepts are defined from lower-level ones, and in turn they can help to identify other higher-level concepts. Deep learning is part of a broader family of machine learning methods based on learning representations and it typically uses artificial neural networks (ANN) for supervised or unsupervised feature extraction and transformation, and for pattern analysis, regression and classification. The capability of learning features at multiple levels of abstraction allow a system to learn complex functions mapping the input to the output directly from data, without depending too much on features provided by user. An observation can have many representations, but some of them make it easier to learn interesting patterns from fed examples, and research in this area attempts to define what makes better representations and how to learn them. Deep learning achieves great power and flexibility by learning to represent the world in the above-described nested hierarchy of concepts because it allows to reduce the computational cost and to reduce the huge use of memory required in the case of multilayer structures. The main applications of Deep Learning are image, speech and text recognition, automatic text generation and machine translation.

3.1 Artificial Neural Networks

Neural Networks are a pretty old algorithm that was originally motivated by the goal of having machines that can mimic the brain. The origins of Neural Networks were as algorithms that try to

mimic the brain, perhaps the most amazing learning machine we know about, to build artificial learning systems. Neural Networks came to be very widely used throughout the 1980's and 1990's and for various reasons as popularity diminished in the late 90's. But more recently, Neural Networks have had a major resurgence. One of the reasons is that Neural Networks are computationally very expensive algorithms and so, it was only more recently that computers became fast enough to really run large scale Neural Networks and because of that (as well as a few other technical reasons) modern Neural Networks today are the state-of-the-art technique for many applications. The idea of mimicking comes from the fascinating hypothesis (provided with many evidence) that the way the brain does all of the amazing different things it does is just using a single learning algorithm. And instead of needing to implement a thousand different programs or algorithms to do the thousand wonderful things that the brain does, maybe if we can figure out what the brain's learning algorithm is and implement it or some approximation to that algorithm on a computer, maybe that would be our best shot at making real progress towards the AI, the artificial intelligence dream of someday building truly intelligent machines.

So, artificial neural networks (ANNs) are born with the aim of reproducing the activities typically developed by the human brain. Their structure is in fact inspired by the network generated by neurons, synapses and dendrites. In reality, this sort of biological inspiration does not bring out from the operational logic behind this computing system called neural network.

An ANN consists of a group of connected units called neurons or nodes, which loosely mimic the biological neurons in the brain. Each link, known as edge, between these neurons can transmit signals or information, like the synapses do in the biological counterpart. Every neuron can have multiple input and output connections to other neurons. Each edge of the network generally is assigned a parameter, called weight, that represents its relative importance (can modify the strength of the signal by increasing or decreasing it) and that changes as learning proceeds. In ANN each neuron receives signals (that are real numbers, products of other neurons' outputs and relative weights) in input, then processes them and send them to neurons connected with it. To compute the input of a neuron from the outputs of neurons connected to it a propagation function is used that performs a weighted sum, sometimes called the activation. It usually includes also a bias term. The output of each neuron is produced by applying an activation function (often nonlinear) to the above-mentioned weighted sum. Neurons may have a threshold that allows the output signal to be sent only if the activation function output crosses that threshold. In deep learning, neurons are arranged to form multiple layers in a way such that nodes of a layer are connected only to neurons of the immediately previous and following layers. Information travels from the input layer (i.e., the first one) that receives external data, to the output layer (that is the last one) which produces the final result, after passing through a variable

number of intermediate layers, called hidden layers. Different layers may perform different transformations on their inputs. A number of patterns are possible for the connections between two layers: they can be fully connected (i.e., each neuron in a layer is connected to each one in the next layer), or they can be of the pooling type (that means a group of neurons in a layer is linked to a single neuron in the successive layer). Networks presenting only these types of connections are known as feedforward networks. On the other hand, networks presenting links connecting neurons in a layer between them or with previous layers are known as recurrent networks.

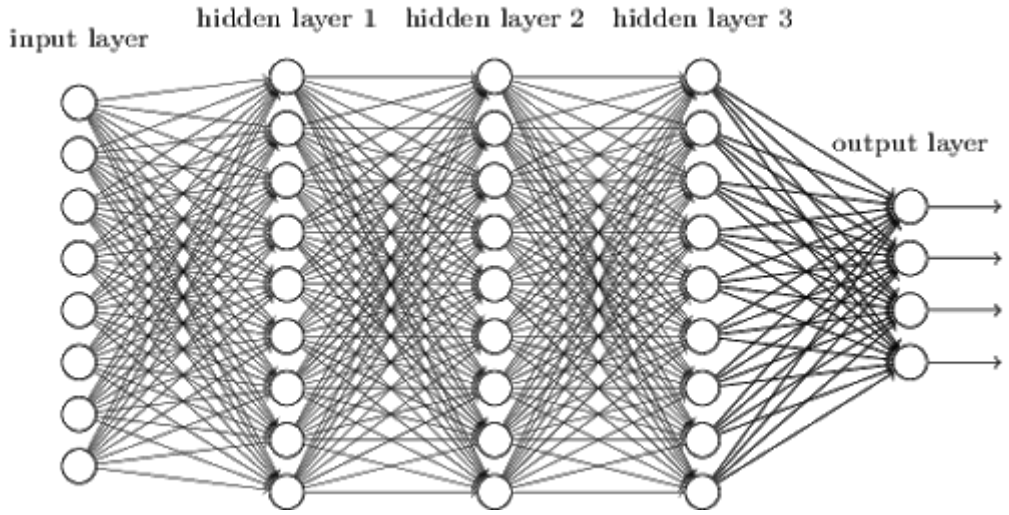

Figure 17. Deep Fully Connected Feedforward Neural Network architecture (source: neuralnetworksanddeeplearning.com)

Learning represents the modification of the network to better perform a task by considering sample observations. Learning implies tuning network weights (and thresholds if there are any) to improve the results and this is done by minimizing the measured error. Practically a cost (loss) function is defined and evaluated periodically during learning. This function is defined as a statistics metric and its choice can be determined by some function's desirable properties (such as convexity). Learning attempts to reduce the total of the differences across the observations, that is the training error. To this end, connection weights should be adjusted, and backpropagation is the applied method. It computes the gradients (partial derivatives) of the cost function associated with a given network configuration with respect to the weights. Their updates can be done via gradient descent or other optimization methods. If the training error continues to decline, learning continues. Learning is complete when the error rate does not meaningfully reduce feeding the network with additional observations. If the error remains too high after learning, the network should be redesigned. A parameter, known as learning rate, defines the size of the steps to be taken to decrease errors in each observation. A high value shortens the training time, at the cost of lower final score, while a lower learning rate takes longer time, but with the potential for

greater results. The goal of optimization is primarily speeding up error minimization. An adaptive learning rate which increases or decreases as appropriate can be used to avoid oscillation (in the weight values) inside the network and to improve the rate of convergence.

There are different neural networks architectures, among these the most important are the following:

- Deep Feed-forward Neural Networks. They are multi-layer perceptrons and are made of multiple layers of neurons. They are interconnected in a feed-forward way: each neuron of a layer has direct connections to the neurons of the subsequent layer.
- Convolutional Neural Networks. They are the most used networks for image classification and derive their name from the layers they are made of, that are multiple convolution layers which oversee the extraction of relevant features from the input object, such as an image. The earlier layers address low-level, simple details while the later layers are in charge of detect high-level features. The convolution operation principle uses a custom matrix, also called filter, to convolute over the input image and produce maps. These filters are randomly initialized and during training are updated through backpropagation. After the convolution layer, there is a pooling one which is responsible for the aggregation of the maps produced from the previous layer. For regularization purposes, CNNs allow for adding dropout layers which drop or make certain neurons inactive to reduce overfitting and speed up convergence of the solution. The output layer of CNNs have a fully connected dense layer.
- Recurrent Neural Networks. They are used when handling predictions using sequential data, such as a sequence of images, words, etc. The RNN structure is similar to that of a Feed-Forward Network, except that the layer neurons also receive a time-delayed input of the previous output prediction. This is stored in the network cell which is used as a second input for every prediction. The main disadvantage of RNN the difficulty to remember earlier layers' weights due to the vanishing gradient problem.
- Long Short-Term Memory Neural Networks. This type of network overcomes the above-mentioned issue of vanishing gradient by adding a particular memory cell that can store information in the network for long periods of time. LSTM uses three gates to define which output prediction should be retained or forgotten: input gate, output gate and forget gate. The first one tells what data should be kept in memory. The second controls information given to the next layer and the forget gate controls when to dump useless data not required. LSTMs are used in a wide range of applications such as gesture, speech recognition and text prediction.

In this thesis, only Deep Feedforward Neural Networks will be used, in particular those of the fully connected type, as shown in Figure 17.

Chapter 4

Predictive Models for Soot Emissions Evaluation

In the previous chapters a brief overview of the theory underlying this thesis has been presented. The phenomenological part of diesel combustion, the mechanisms of pollutant formation and a brief description of the main Machine Learning and deep learning algorithms were discussed. In this chapter we want to enter into the specifics of the actual experimental work carried out.
The models that are used in this paper, as already mentioned above, exploit different supervised learning ML and DL algorithms called k-Nearest Neighbors, Support Vector Machines and. These can perform both classification and regression tasks (the latter corresponding to our case) and are implemented in an open source integrated development environment using the Python programming language. For this purpose, specific open-source machine learning libraries are used, such as Scikit-Learn, Keras (Tensorflow) and XGBoost, and within which the previously mentioned algorithms are present. Other libraries for data processing and visualization are also used, such as NumPy, SciPy, Pandas and Matplotlib.

4.1 Nearest Neighbors

The *Nearest Neighbors* algorithm is a simple, easy-to-implement ML algorithm that provides functionalities for both unsupervised and supervised learning tasks. Unsupervised nearest neighbors is the foundation of many other algorithms, such as manifold learning and spectral clustering. Instead, supervised neighbors-based learning is mostly used to solve both classification (data with discrete labels) and regression (data with continuous labels) problems.
The principle behind nearest neighbor methods is to store in memory observations given during training and predict the label of a new unseen point from those of a group of training samples found by the algorithm closest in distance (in the multidimensional space of the features) to the above point. Essentially, the nearest neighbors algorithm assumes that similar things exist in close proximity, that is similar things come near to each other. The number of samples can be a constant defined by the user (k-nearest neighbor learning) or can vary based on the local density of points (radius-based neighbor learning). The distance can be any metric measure: standard Euclidean distance is the most common choice, but also the Manhattan, Chebyshev and the more general Minkowski distances are used. In other words, nearest neighbors capture the idea of similarity (called also proximity, or closeness) with some basic mathematics.
Neighbors-based algorithms are known as non-generalizing machine learning methods, since they simply keep track of ("remember") all the training dataset (possibly transformed into a

faster-accessible structure). This type of learning is also known as instance-based learning: it does not try to learn a general internal function to be used in prediction, but simply stores instances of the training data for their later use for the evaluation of a new query point. This approach is also referred to as 'lazy learning' because of the algorithm behaviour which takes almost zero time to learn.

Although it is disarmingly simple, nearest neighbors algorithm has been successful in many problems, both in classification and regression settings, including handwritten digits and image scenes.

For classification tasks a simple majority vote of each new sample nearest neighbors is computed, that is a query point is assigned the data class which has the most representatives within the nearest neighbors (saved training samples) of the point. In regression tasks, instead, the query point target is predicted by local interpolation of the labels associated to the nearest neighbors in the training set. A query point is a new observation (unseen by the algorithm during training) whose label has to be predicted by the model.

Learning can be implemented based on the k nearest neighbors of each query point, where k is an integer value specified by the user, or based on the neighbors within a fixed radius r of the query point, where r is a floating-point value specified by the user. The optimal choice of the value k is highly data-dependent: a larger k generally suppresses the effects of noise (outliers), but makes the approximation less accurate. A value of k extremely large will result in underfit, on the contrary choosing it too small gives rise to overfitting as the algorithm will be more sensitive to outliers. In cases where the data is not uniformly sampled, radius-based neighbors regression can be a better choice. The user specifies a fixed radius r, in order that points in sparser neighbourhoods use fewer nearest neighbors for the regression. For high-dimensional attribute spaces, this method becomes less effective due to the so-called "curse of dimensionality".

Generally, uniform weights are used, that is each point in the local neighborhood contributes uniformly to the approximation of a query point. Under some circumstances, it can be better to weight points in a way that closer points contribute more to the regression than further points. This can be accomplished through assigning weights that are proportional to the inverse of the distance from the query point.

4.1.1 Brute-force Algorithm

As already said, making a prediction with nearest neighbors involves calculating the distance in the multidimensional feature space between a query point and the stored training observations,

and finding the nearest neighbors. Fast computation of these neighbors is still an area of research in machine learning and the most artless neighbor search implementation involves the "brute-force" calculation of distances between all pairs of points in the dataset. This approach scales as $O(DN^2)$ for data consisting of N samples with D features (dimensions): thus, brute-force neighbors searches can be very efficient for small datasets, however, as the number of observations grows, this approach becomes computationally infeasible very quickly.

4.1.2 KD Tree Algorithm

To solve the inefficiencies of the brute-force approach, a particular type of tree-based data structures has been introduced. These structures generally try to reduce the number of required distance computations by efficiently handling aggregate information for the sample about distance, that is deducing distances between points (by knowing information about relative position) without having to explicitly calculate them. In this way, the computational cost of a nearest neighbors search can be significantly improved to $O(DN \log N)$ or better, for large N.
A first approach adopted to take advantage of the above-described concept was the KD (K-Dimensional) tree data structure. This is a binary tree structure which iteratively partitions the feature space along the data axes, creating nested orthotropic regions where data points are arranged. Since this partitioning is performed only along the data axes and no multidimensional distances computations are required, the construction of a KD tree is very fast. After that, the nearest neighbor of a query point can be determined with a computational cost of only $O(\log N)$. Although the KD tree approach is very fast for low-dimensional neighbors searches, it becomes computationally inefficient as features (dimensions) number grows very large: this is an evidence of the so-called "curse of dimensionality" previously mentioned.

4.1.3 Ball Tree Algorithm

A new data structure, the ball tree, was developed to cope with the inefficiencies of KD trees in higher dimensions. This new method partitions data in a series of nesting hyper-spheres whereas KD trees partition data along Cartesian axes: this makes the construction of ball trees more costly than that of KD trees, but results in a data structure which can be really efficient even on highly structured data (i.e., high dimensional). A ball tree divides in an iterative way the data into a series of nodes, each defined by a centroid C and radius r, and data points in the node stay within the hyper-sphere defined by C and r. The following triangle inequality is used to reduce the number of candidate points for a neighbour search:

$$|x + y| \leq |x| + |y|$$

In this way, only a single distance calculation between a test point and the centroid is needed to determine lower and upper bound values on the distance to all examples within the node. The spherical geometry of the ball tree nodes allows them to outperform KD trees in high dimensions, even if the actual performance is strictly dependent on the structure of the training data.

4.1.4 Choice of Nearest Neighbors Algorithm

As previously mentioned, the optimal algorithm for a given dataset can be a complicated choice, it will strongly influence the query time and depends on many factors:

- Number of samples N and number of features (dimensions) D. In the case of brute-force approach query time grows as $O(DN)$, while it grows approximately as $O(D \log N)$ for the ball tree algorithm. Instead, KD tree query time changes significantly with D: for small values, the cost scales as roughly $O(D \log N)$ and it can result to be very efficient, on the other hand, for large values of D the computational cost increases nearly as $O(DN)$ and the overwork due to the tree construction can lead to queries which are slower than brute-force approach. Regarding the dataset size, for small values $\log N$ is similar to N and brute-force algorithm can compete and even do better than tree-based approach. This problem can be solved by introducing the leaf size hyperparameter: it controls the dataset sizes at which a query switches to brute-force method.
- Number of query points. Both the ball tree and the KD tree require a construction phase and its cost in terms of time required becomes negligible when amortized over many points. However, in the case a small number of queries will be executed, the construction can make up a significant fraction of the total time required and only if very few query points will be required, then brute-force method is better than a tree-based one.
- Number of nearest neighbors k requested for a query point. Brute-force approach query time is quite unaffected by the value of k. Rather, ball and KD tree query time will intuitively become slower as k increases, mainly because a larger value leads to the need to search a larger portion of the parameter space. Moreover, as k becomes large compared to the number of samples, the ability to prune branches in a tree-based query is reduced and brute-force queries can result more efficient.

4.2 Support Vector Machines

Support vector machines (SVMs) consist of a set of popular machine learning tools for classification, regression, and other learning tasks, and in their present form have been largely developed by Vapnik and his colleagues from 1992 onwards. Today, SVMs show comparable outcomes to neural networks and other statistical models on the most popular benchmark problems.

SVMs belong to the supervised learning algorithms class and are considered a *nonparametric* technique because it relies on kernel functions. This does not mean that the SVMs models do not have parameters at all, but the parameters are not predefined, and their number depends on the data used during training. To put it another way, model parameters are data-driven such that they match model capacity to data complexity.

Like neural networks, SVMs possess the well-known ability to be universal approximators of any multivariate function to any desired degree of accuracy. Therefore, they are of particular interest for applications to unknown, highly nonlinear, complex problems.

Initially developed for solving classification problems, Support Vector techniques can be successfully applied in regression, as stated before. In this chapter, the theory behind SVM regression will be presented briefly describing both linear and nonlinear problems.

4.2.1 Overview of SVM Regression

In the case of SVM's regression, we measure the *error of approximation* instead of the margin used in classification. There is also a difference in respect to classic regression, that is the use of the Vapnik's *linear ε-insensitivity loss function*, which defines an ε tube. This loss function ignores errors that are within ε distance of the target value (the predicted value is within the tube) by treating them as equal to zero (i.e. the loss or cost is equal to zero if the difference between the predicted f(xi,w) and the measured value is less than ε). In contrast, for all other predicted points outside the tube, the loss or error is measured based on the distance between observed value y and the ε boundary: if the difference is larger than ε, the loss equals the magnitude of the difference between the predicted value and the radius ε of the tube. This is formally described by:

$$L_\varepsilon = \begin{cases} 0 & if\ |y - f(x)| \leq \varepsilon \\ |y - f(x)| - \varepsilon & otherwise \end{cases}$$

In other words, in linear epsilon-insensitive SVM (ε-SVM) regression the goal is to find a function $f(\boldsymbol{x})$ that deviates from yn by a value no greater than ε for each training point $\boldsymbol{x}$, and at the same time is as flat as possible.

4.2.2 Linear SVM Regression

Given a set of training data with N observations where each one present a predictors vector $\boldsymbol{x}_i \in R^n$ with the relative observed response value $y_i \in R$.
To find parameters $\boldsymbol{w}$ and b of the linear approximation function

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$$

and ensure that it is as flat as possible, meaning to find $f(\boldsymbol{x})$ with the minimal norm value $(\boldsymbol{w}^T\boldsymbol{w})$. This is formulated as the following objective function (convex optimization problem):

$$\min\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} = \min\frac{1}{2}\|\boldsymbol{w}\|^2 ,$$

subject to all residuals (differences between predictions and target values) having a value less than ε, that is in equation form:

$$\forall i : \ |y_i - (\boldsymbol{w}^T\boldsymbol{x}_i + b)| \leq \varepsilon .$$

It is assumed that the convex optimization problem is feasible, that means such a function f really exists that approximates all pairs $(\boldsymbol{x}_i, y_i)$ with at least ε precision. Sometimes, however, this may not be the case, or we also may want to allow for some errors. To deal with otherwise infeasible constraints, slack variables ξ_i and ξ_i^* are introduced for each point. This approach is analogous to the "soft margin" loss function concept in SVM classification, because the slack variables allow regression errors to exist up to the value of ξ_i and ξ_i^*, yet still satisfy the required conditions.
Including these slack variables leads to the objective function:

$$\min \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$$

subject to:

$$\forall i : \ y_i - (\boldsymbol{w}^T\boldsymbol{x}_i + b) \leq \varepsilon - \xi_i ,$$
$$\forall i : \ (\boldsymbol{w}^T\boldsymbol{x}_i + b) - y_i \leq \varepsilon - \xi_i^* ,$$
$$\forall i : \ \xi_i, \xi_i^* \geq 0 .$$

The constant C is a design parameter chosen by the user and is the box constraint, a positive numeric value that controls the penalty imposed on observations that lie outside the ε margin and helps to prevent overfitting (regularization function). This value determines the trade-off between the flatness of $f(\boldsymbol{x})$ (that means the weight vector norm $\boldsymbol{w}$) and the amount up to which deviations larger than ε are tolerated (approximation error). An increase in C penalizes large errors i.e., it forces ξ_i and ξ_i to be small. This leads to an approximation error decrease which is achieved only by increasing the weight vector norm $\boldsymbol{w}$, however, this increase does not guarantee a small generalization error of a model. The choice of a 'suitable' value of C is always done experimentally by using cross-validation techniques.

Another design parameter selected by the user is the required precision, through the ε value that defines the size of the ε-tube. The choice of this hyperparameter is easier than the choice of C because can be given as either a maximum allowed value or some desired percentage of the output values y_i.

The above objective function is a constrained quadratic optimization problem which is solved by the following primal variables Lagrangian:

$$L_p = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{N}(\xi_i + \xi_i^*) - \sum_{i=1}^{N}(\beta_i\xi_i + \beta_i^*\xi_i^*) - \sum_{i=1}^{N}\alpha_i(\boldsymbol{w}^T\boldsymbol{x}_i + b - y_i + \varepsilon + \xi_i) - \sum_{i=1}^{N}\alpha_i^*(y_i - \boldsymbol{w}^T\boldsymbol{x}_i - b + \varepsilon + \xi_i^*)$$

where α_i, α_i^* are the nonnegative Lagrange multipliers. It can be shown that this function has the saddle point at the optimal solution ($\boldsymbol{w}_o$, b_o, $\xi_{i,o}$, $\xi_{i,o}^*$) to the original (optimization) problem. This primal objective function can be solved in the space of the parameters $\boldsymbol{w}$, b (primal space) or in the space of the multipliers α (dual space): the second approach is chosen because it gives meaningful results.

At the saddle point the partial derivatives of the Lagrangian with respect to the primal variables should vanish due to optimality so the following Karush-Kuhn-Tucker conditions for finding the optimum of a constrained function are used:

$$\boldsymbol{w}_o = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\boldsymbol{x}_i\,,$$

$$\sum_{i=1}^{N}(\alpha_i - \alpha_i^*) = 0\,,$$

$$C = \alpha_i + \beta_i\,,$$

$$C = \alpha_i^* + \beta_i^* .$$

Substituting these expressions into L_p the problem of the maximization of a dual variables Lagrangian is obtained:

$$L_d = -\frac{1}{2}\sum_{i,j=1}^{N}(\alpha_i - \alpha_i^*)\left(\alpha_j - \alpha_j^*\right)\boldsymbol{x}_i^T\boldsymbol{x}_j - \varepsilon\sum_{i=1}^{N}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{N}(\alpha_i + \alpha_i^*)y_i$$

subject to constraints:

$$\sum_{i=1}^{N}(\alpha_i - \alpha_i^*) = 0 ,$$
$$0 \le \alpha_i, \alpha_i^* \le C , \;\; i = 1, \dots, N .$$

After the calculation of Lagrange multipliers, the desired weight vector of the regression hyperplane can be found as:

$$\boldsymbol{w}_o = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\boldsymbol{x}_i .$$

The so-called Support Vector expansion is obtained, that is the parameter $\boldsymbol{w}$ can be completely described as a linear combination of the training examples $\boldsymbol{x}_i$.

The best regression hyperplane (i.e., the decision function used to make new predictions) is obtained by:

$$f(\boldsymbol{x}, \boldsymbol{w}, b) = \boldsymbol{w}_o^T\boldsymbol{x} + b = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\boldsymbol{x}_i^T\boldsymbol{x} + b$$

It is worth noting that the complete algorithm can be written in terms of inner products between the training samples: even when evaluating $f(\boldsymbol{x})$ it is not necessary to explicitly compute the parameter vector $\boldsymbol{w}$. These remarks will be used for the formulation of the nonlinear case.

At the optimal solution, the KKT complementary conditions reported below must be met:

$$\forall i : \; \alpha_i(\boldsymbol{w}^T\boldsymbol{x}_i + b - y_i + \varepsilon + \xi_i) = 0 ,$$
$$\forall i : \; \alpha_i^*(y_i - \boldsymbol{w}^T\boldsymbol{x}_i - b + \varepsilon + \xi_i^*) = 0 ,$$
$$\forall i : \; \xi_i(C - \alpha_i) = 0 ,$$
$$\forall i : \; \xi_i^*(C - \alpha_i^*) = 0 .$$

These expressions state that for all samples strictly inside the ε-tube the Lagrange multipliers α_i, α_i^* vanish. Therefore, the Support Vector expansion written before is sparse because we do not need all $\boldsymbol{x}_i$ to describe the weight vector $\boldsymbol{w}$.

The observations that come with nonvanishing coefficients (i.e., either α_i or α_i^* is not zero) are called support vectors: in particular, data points for which $0 \leq \alpha_i \leq C$ or $0 \leq \alpha_i^* \leq C$ are identified as free (unbounded) support vectors (observations located at the boundary of the ε-tube) and samples for which α_i or α_i^* equals C are the so-called bounded support vectors (data points outside of the ε-tube).

To summarize, after the learning stage, the number of support vectors is equal to the number of nonzero multiplier pairs (α_i, α_i^*) and this number does not depend on the dimension of input space (i.e., training data set size). In other words, the complexity of a function represented by support vectors is independent of the input space dimensionality and depends only on the number of SVs. This property will be particularly important when working in very high dimensional spaces.

Free support vectors as defined above allow also computing the value of the bias term b by exploiting the KKT conditions just mentioned.

4.2.3 Nonlinear SVM Regression

Most of real-world regression problems cannot adequately be described using a linear model, so more useful and more interesting would be trying to solve nonlinear regression tasks. Thus, the next step is to make the SV algorithm nonlinear.

The idea at the base of designing nonlinear SVMs is to map the input vectors $\boldsymbol{x}_i$ into vectors $\Phi(\boldsymbol{x}_i)$ of a high dimensional feature space S (where Φ represents a mapping, i.e. a fixed transformation function chosen in advance) and then apply the linear standard SVM regression algorithm in this feature space:

$$x \in R^n \rightarrow \Phi(\boldsymbol{x}) = [\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \ldots, \phi_s(\boldsymbol{x})]^T \in R^s \, .$$

By performing such a step, it is hoped that in a Φ-space, the learning algorithm will be able to identify a linear regression hyperplane by applying the linear regression SVM formulation discussed above. It is expected to again obtain a quadratic optimization problem with inequality constraints to solve, this time in the feature space. The solution for the regression hyperplane $f = \boldsymbol{w}^T \Phi(\boldsymbol{x}) + b$, linear in the feature space, will create a nonlinear regression hypersurface (function) in the original input space.

Unfortunately, there are two main problems when using the feature mapping Φ although it allows to deal with nonlinear dependency: firstly, the choice of the mapping should result in a "well-populated" class of decision hyperplane (can be overcome by the fact that a linear regression hyperplane always exists for a whole class of mappings); secondly, performing the scalar product $\Phi^T(\boldsymbol{x})\Phi(\boldsymbol{x})$ can be computationally very expensive if the dimensionality of the feature space (i.e., the number of features) is very large.

The second problem is related to the well-known 'curse of dimensionality' issue and clearly a computationally cheaper way has to be found. The key consideration to avoid this explosion in dimensionality is that the SVM algorithm, both in the quadratic optimization problem and in the final expression, only depends on scalar products $\boldsymbol{x}_i^T\boldsymbol{x}_j$ between training samples. These products will be replaced by $\Phi^T(\boldsymbol{x})\Phi(\boldsymbol{x}_i) = [\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \dots, \phi_n(\boldsymbol{x})][\phi_1(\boldsymbol{x}_i), \phi_2(\boldsymbol{x}_i), \dots, \phi_n(\boldsymbol{x}_i)]^T$ in a feature space S, which in turn will be expressed by using the kernel, which is a function such that $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \Phi^T(\boldsymbol{x}_i)\Phi(\boldsymbol{x}_j)$.

It is worth noting that a kernel $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a function in the input space, so the required inner products in the feature space are calculated directly by computing the kernel function for the given training data in the input space. In this way, it is not needed to work in the possibly very high dimensionality of the feature space S because using the kernel function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ it is avoided performing a mapping $\Phi(\boldsymbol{x})$ at all, it is not even necessary to know what the actual mapping $\Phi(\boldsymbol{x})$ is. This is very well known under the popular name of the *kernel trick* and it allows to construct SVMs that operate even in an infinite dimensional space.

Kernel Type (Name)	***Kernel Function***
Linear	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T\boldsymbol{x}_j$
Gaussian	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\gamma\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2\right)$
Sigmoid	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh\left[\gamma(\boldsymbol{x}_i^T\boldsymbol{x}_j) + r\right]$
Polynomial	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left[\gamma(\boldsymbol{x}_i^T\boldsymbol{x}_j) + r\right]^d$

Table 1. Most popular and widely used kernel functions

There are many possible kernels, and the most popular ones are given in Table 1. Each of these functions builds a different nonlinear regression hypersurface in input space, but all of them should fulfil the so-called Mercer's theorem. It states that $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a valid kernel function if and only if for any given training data point the corresponding kernel matrix is positive semidefinite (meaning all its eigenvalues are greater or equal to zero). The kernel matrix is a

symmetric n-by-n matrix whose elements are the inner product of the training examples vectors as transformed by the mapping Φ.

Coming back to the learning stage, the nonlinear regression problem can be again formulated as the maximization (i.e., optimization objective) of the following dual Lagrangian:

$$L_d = -\frac{1}{2}\sum_{i,j=1}^{N}(\alpha_i - \alpha_i^*)\left(\alpha_j - \alpha_j^*\right)K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) - \varepsilon\sum_{i=1}^{N}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{N}(\alpha_i + \alpha_i^*)y_i$$

subject to:

$$\sum_{i=1}^{N}(\alpha_i - \alpha_i^*) = 0\,,$$
$$\forall i:\ 0 \le \alpha_i, \alpha_i^* \le C\,.$$

After calculating Lagrange multiplier vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}^*$, we can compute an optimal weight vector as

$$\boldsymbol{w}_o = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\Phi(\boldsymbol{x}_i)\,.$$

Note here that, in a nonlinear SVM regression, the optimal weight vector $\boldsymbol{w}_o$ could be of very high or even infinite dimension. Consequently, $\boldsymbol{w}_o$ is neither computed nor given explicitly (i.e., expressed in closed form). The function used to predict new values may be written as

$$f(\boldsymbol{x}, \boldsymbol{w}, b) = \boldsymbol{w}_o^T\Phi(\boldsymbol{x}) + b = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\Phi^T(\boldsymbol{x}_i)\,\Phi(\boldsymbol{x}) + b = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\,K(\boldsymbol{x}_i, \boldsymbol{x}) + b$$

The last result follows from the very setting of the learning (optimizing) stage in the feature space where in all the equations shown for the linear case $\boldsymbol{x}_i$ is replaced by the corresponding feature vector $\Phi(\boldsymbol{x}_i)$.

The KKT complementarity conditions are

$$\forall i:\ \alpha_i(f(\boldsymbol{x}_i) - y_i + \varepsilon + \xi_i) = 0\,,$$
$$\forall i:\ \alpha_i^*(y_i - f(\boldsymbol{x}_i) + \varepsilon + \xi_i^*) = 0\,,$$
$$\forall i:\ \xi_i(C - \alpha_i) = 0\,,$$
$$\forall i:\ \xi_i^*(C - \alpha_i^*) = 0\,.$$

The bias term b, for a nonlinear SVM regression, can be explicitly calculated from the upper or lower SVs.

It can be note that in the nonlinear setting the solution to the optimization problem corresponds to finding the flattest function in the feature space, not in the input space.

It is worth to highlight that the way in which the kernel trick was applied in designing a nonlinear SVM can also be used in all other algorithms that depend on the inner product $\boldsymbol{x}_i^T \boldsymbol{x}_j$.

To summarize, a few learning hyperparameters have to be set in constructing SVMs for regression. The three most important ones are the insensitivity zone ε, the penalty parameter C (that determines the trade-off between the training error and simplicity of the obtained approximation hyperplane), and the shape parameter of the kernel function (i.e., variance of a gaussian kernel or degree of the polynomial one). The user should set all three hyperparameters and, to this end, the most popular method is the cross-validation.

4.2.4 Solver Algorithms for SVM Regression Optimization Problem

The primal objective minimization problem can be expressed in standard quadratic programming (QP) form and solved using common techniques available for QP problems. However, it can be expensive from the computational point of view to use QP algorithms, as well as the elements of the kernel matrix can be too many to be stored in memory.

Using a decomposition method instead allows to speed up the calculations and do not run out of memory. Decomposition methods are also called chunking methods or working set methods because they essentially separate all observations into two different sets: the working set and the remaining one. In each iteration these methods modify only the elements in the working set and thus, only few elements of the kernel matrix are needed at each iteration, reducing the amount of storage needed.

Sequential minimal optimization (SMO) is the most popular algorithm for solving SVM problems. SMO performs iteratively a series of two-point optimizations. In each iteration, two points are chosen as working set based on selection rules, then the Lagrange multipliers for this active set are solved analytically. In SVM regression, after each iteration the gradient vector ∇L for the working set is updated and the decomposed equation for the gradient vector is

$$(\nabla L)_j \begin{cases} \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) K(\boldsymbol{x}_i, \boldsymbol{x}_j) + \varepsilon - y_j \,, j \leq N \\ -\sum_{i=1}^{N} (\alpha_i - \alpha_i^*) K(\boldsymbol{x}_i, \boldsymbol{x}_j) + \varepsilon + y_j \,, j > N \end{cases}$$

Iterative single data algorithm (ISDA) is another decomposition method and works updating one Lagrange multiplier with each iteration. ISDA is often executed without the bias term b by adding a small positive constant to the kernel function. Dropping the bias term cancels out also the sum constraint in the dual equation: this allows to update one Lagrange multiplier in each iteration, making it easier than SMO to remove outliers. To choose the working set to update, ISDA selects the worst KKT violator among all the α_i and α_i^* values.

Convergence Criteria

Each of the above-mentioned solver algorithms runs iteratively until the specified convergence condition is met. There are several options for convergence criteria:

- Feasibility gap - It is expressed as

$$\Delta = \frac{J(\boldsymbol{w}) + L_d(\boldsymbol{\alpha})}{J(\boldsymbol{w}) + 1}$$

  where $J(\boldsymbol{w})$ is the primal variables objective function and $L_d(\boldsymbol{\alpha})$ is the dual variables Lagrangian. The algorithm evaluates the feasibility gap after each iteration and if it is less than the value specified, then the algorithm met the convergence criterion and the software returns a solution.
- Gradient difference - The gradient vector ∇L is evaluated after each iteration and if the difference in its values between two consecutive iterations is less than the value specified, then the algorithm met the convergence criterion and the software returns a solution.
- Largest KKT violation - After each iteration, the algorithm evaluates the KKT violation for all the α_i and α_i^* values. If the largest violation is less than the value specified, then the algorithm met the convergence criterion and the software returns a solution.

SVMs are powerful tools, but their computational and storage requirements rapidly rise with the number of training observations. As previously stated, the core of an SVM is a QP problem, separating support vectors from the rest of the training data. The time required by the QP solver generally scales between $O\left(N_{features}N_{samples}^2\right)$ and $O\left(N_{features}N_{samples}^3\right)$ depending on dataset size.

Finally, SVM algorithms are not scale invariant, so it is highly recommended to scale your data for better results. For example, each feature on the input observations can be normalized or standardized. Note that the same scaling must be applied to the test vector to obtain meaningful results.

4.3 Gradient Tree Boosting

Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT) is a non-parametric supervised learning technique, one of the most powerful for building predictive models. Boosting is a general approach that can be applied to many statistical learning methods, but here we restrict the description of boosting to the context of decision trees. GBDT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems in a wide range of areas. Before embarking upon this topic, a brief overview of tree-based methods and their characteristics will be beneficial for the reader.

4.3.1 Decision Trees

The simplest tree-based method for classification and regression is the decision tree algorithm. It involves the segmentation (or stratification) of the multidimensional predictors (i.e., features) space into a number of different simple regions. Each one of them is characterized by a value which is the mean or the mode of the labels of the training observations which belong to that region. When making a prediction for a new unseen observation the value corresponding to the region to which this new sample belongs is output. This approach is known as decision tree because the set of splitting rules used to segment the feature space can be represented by a tree.

The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple problems (i.e., it is infeasible from the computational point of view to consider every possible partition of the predictors space into a number of regions). Consequently, practical decision-tree learning algorithms are based on heuristic approaches like that named *recursive binary splitting*. This is a 'top-down' technique because it starts with all observations in a unique region and then splits the feature space successively. It is also a 'greedy' method in the sense that at each step (i.e., node) of the building tree the best split is chosen based on locally optimal decisions, not looking ahead to search for the split that at the end of tree construction will lead to the optimal feature space subdivision. Commonly adopted criteria to minimize for determining features and thresholds for future splits are Mean Squared Error (MSE) as well as Mean Absolute Error (MAE).

Decision trees are simple to understand (they can be visualized) and to interpret (they use a white box model) and they require very little data preparation. However, they suffer from high variance and can be unstable (even small variations in the data may generate a completely different tree), and they are not good at extrapolation (because are piecewise constant approximations), hence generally are not competitive with the best supervised learning approaches in terms of prediction performance. The process described before may produce very

good predictions on the training set, but leading to poor test performance because the resulting tree might be over-complex and is likely to not generalize the data well (in other words, it overfits). To overcome these issues a valid strategy may be to apply a pruning mechanism: that means, to grow a very large tree, and then prune it back to obtain a subtree that should intuitively lead to the lowest test error rate. Given a subtree, this can be evaluated using cross-validation or the validation set approach. However, due to the extremely large number of possible subtrees, estimating the cross-validation error for all of them would be too cumbersome. Cost-complexity pruning (also called weakest link pruning) gives us a way to select a small set of subtrees, rather than considering every possible subtree, through a nonnegative tuning parameter α. It controls the trade-off between a subtree's complexity and its fit to the training data: as α increases, the price to pay for having a tree with many terminal nodes increases, and so the loss associated with α will tend to be minimized for a smaller subtree. A proper value of α can be selected, as usual, using a validation set or using cross-validation.

4.3.2 Bagging Methods and Random Forests

To cope with the above-mentioned weaknesses of decision trees, ensemble methods are introduced. In general, ensemble learning is concerned with approaches that combine predictions from two or more models, based on the same algorithm or not. We can identify a model as an ensemble learning technique if it comprises two or more models and if predictions are combined.
The main goal of ensemble methods is to combine the predictions of many base estimators built with a given learning algorithm in order to improve robustness (i.e., stability) and generalization (prediction performances) capabilities over a single estimator. Ensemble methods are usually subdivided into two main families: averaging methods and boosting ones. In the former case, the driving principle is to build several estimators independently and then to average their predictions. The combined estimator is usually better than any of the single base estimator because its variance is reduced. In the latter case, base estimators are built sequentially and the last one created tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble with enhanced performance.
In our case, by training and aggregating multiple decision trees, using methods like bagging and random forests, an ensemble learner is built that has substantially improved predictive performance: we use trees as building blocks to construct more powerful models.
In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. Bagging methods get different

names depending on the way they draw random subsets of the training set: pasting when random subsets of the samples are drawn without replacement from the original dataset, bagging when samples are drawn with replacement. In many cases, bagging methods constitute a very simple way to improve with respect to a single model, without making it necessary to adapt the underlying base algorithm. Now we briefly explain this method. Since, in general, averaging a set of observations reduces variance, an intuitive way to reduce the variance and hence increase the prediction accuracy of a statistical learning method is to draw many training sets from the population, use each of them to build a different prediction model, and average the resulting predictions. Repeated samples from the available training data set are usually taken because in most cases we do not have access to multiple training sets. So, bagging (or bootstrap aggregation) is a general-purpose procedure for reducing the variance of a statistical learning method and is frequently used in dealing with decision trees because particularly useful. To apply bagging to regression trees, a number of them is constructed using a randomly different bootstrapped training set for each of them, and then the resulting predictions are averaged. Note that these trees are grown deep and are not pruned in order to have trees with high variance, but low bias, since the following averaging reduces the variance. In fact, as they are able to reduce overfitting, bagging methods work best with complex models (i.e., deep and fully developed decision trees), in contrast with boosting methods which usually performs best with weak models (i.e., shallow trees or stumps). Bagging has shown impressive improvements in prediction performance by combining even thousands of trees into a single procedure.

Another averaging algorithm is Random Forest that is a *perturb-and-combine* technique specifically designed for trees. This means a diverse set of estimators is created by introducing randomness in their construction. Random forest algorithms provide an improvement over bagged trees through a small tweak that decorrelates the trees. Firstly, we start building, as in bagging, a number of decision trees on bootstrapped training samples (i.e., draw samples with replacement from the original dataset). Furthermore, each time a split in a tree is considered, a random subsample of predictors (i.e., features) is chosen as split candidates from the full set of the available predictors. Hence, the split is allowed to use only one of those preselected predictors and at each split a fresh new subsample of the features is taken. The number of predictors considered at each split is typically chosen equal to the square root of their total number, that means, in building a random forest, the algorithm is only allowed to consider a minority of the available features at each split in the tree. This is the keystone of this approach, and here we explain why. Suppose having in the dataset a predictor much stronger than the others, then in the ensemble of bagged trees built during training, most or even all of the trees will use this strong predictor in the first split. Consequently, all these trees will be quite similar

to each other and thus their predictions will be highly correlated. Averaging many highly correlated quantities unfortunately does not lead to a large reduction in variance as averaging many uncorrelated quantities do. This setting means that even bagging will not result in a substantial reduction in variance over a single tree. Random forests overcome this problem by forcing each split to consider only a random subset of the predictors. Therefore, on average most of the splits will not even consider the stronger feature, and so other predictors will have more chances to be used for splits. We can think of this process as decorrelating the trees, hence making the average of the trees output less variable and thus more reliable. In other words, the injected randomness in forest ensembles yield decision trees with "decoupled" prediction errors and, by taking an average of those predictions, some errors can cancel out. To summarize, the main difference between bagging and random forests is the choice of a random feature subset when splitting each node and the second method achieves a reduced variance by combining diverse trees (that individually, as said, exhibit high variance and an overfitting tendency), usually at the cost of a little increase in bias, which is anyway widely countervailed by the former variance reduction. Using a small number of predictors in building a random forest will generally help when we have many correlated predictors.

4.3.3 Boosting

Boosting is another approach for improving decision tree predictions performance. Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification, but here we refer to boosting application to the context of regression decision trees. The idea of boosting came out of the one of whether a weak learner can be modified to become better. A weak hypothesis or weak learner is defined as one whose performance is at least slightly better than random guessing, for example a small decision tree. Boosting a hypothesis was the idea of filtering observations, leaving those ones that the weak learner can easily handle and focusing on developing new weak learners to handle the remaining difficult observations. The first popular boosting algorithm was Adaptive Boosting or AdaBoost for short. The core principle of AdaBoost is to fit a sequence of weak learners on recursively modified versions of the data and the predictions from all of them are then combined through a weighted sum to produce the final prediction. The weak learners in AdaBoost are decision trees with a single split, called decision stumps. The data modifications at each 'boosting' iteration consist of applying weights to each training sample: initially, those weights are all set equal (the first learning step simply trains a weak learner on the original dataset), while for each successive iteration the sample weights are individually modified and the learning algorithm is reapplied to

the reweighted data. Therefore, at every step, training examples incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. Thus, examples that are very difficult to predict receive ever-increasing influence as learning process proceeds. New weak learners are thereby added sequentially and each one is forced to concentrate its training on patterns that are missed by the previous ones in the sequence. After AdaBoost, boosting was recast in a statistical framework (later called gradient tree boosting) as a numerical optimization problem where the objective is to minimize the loss of the model by adding weak learners using a procedure like gradient descent. These algorithms were described as a stage-wise additive model because at each iteration one new weak learner is added and the existing ones in the model are frozen and left unchanged. This generalization allowed arbitrary differentiable loss functions to be used. In fact, a benefit of the gradient boosting framework is that it is generic enough that any loss function can be used, that is a new boosting algorithm does not have to be derived for each loss function chosen. The loss function used depends on the type of problem being solved, but we emphasize that it must be differentiable.

A gradient descent procedure is used to minimize the loss when adding trees. Traditionally, gradient descent is used to minimize a set of parameters, but here, instead of them, we have weak learners, more specifically decision trees. To perform the gradient descent procedure, we must compute the loss and then add a tree to the model that reduces it. We do this by parameterizing the tree and modifying its parameters in order to move in the direction that reduce the residual error. This is the so called functional gradient descent approach, even known as gradient descent with functions. Following this procedure, a fixed number of trees are added or training is stopped once the loss reaches an acceptable level or no longer improves on an external validation dataset.

Boosting and bagging work in a similar way, both involve combining many decision trees, but the first difference can be already noted. Recall that the former involves fitting multiple decision trees to different subsamples of the original training dataset, and then combining all of the trees in order to create a single predictive model. Therefore, each tree built is independent of the others, while in boosting trees are grown sequentially and each one is grown using information from previously grown trees (i.e., the construction of each tree depends strongly on the trees that have already been grown).

As described before, decision trees are used as the weak learner in gradient boosting. In our case, regression trees are used that output real values for splits and those ones can be added together, allowing subsequently added tree outputs to correct the residuals in the predictions. Recall that trees are constructed in a greedy manner, choosing the best split points to minimize the loss. It is

common to constrain the weak learners in specific ways (that we will show later) to ensure that the learners remain weak, but can still be constructed in a greedy manner.

Instead of fitting a single complex decision tree to the data, which means fitting the data hard and potentially overfitting, the boosting approach learns slowly. It means that, given the model at the current iteration, we fit a decision tree to the residuals from the model: in other words, we fit a tree using the current residuals as the response, rather than the output value. Then, we add this new decision tree into the fitted function to update the residuals. Each of these trees can rather be simple and small, with just a few terminal nodes. By fitting them to the residuals, we slowly improve the model in predictions where it makes bigger errors. The reason for this proceeding is that, in general, statistical learning approaches that learn slowly tend to perform well.

It is worth underline one key difference between boosting and random forests: in the former method, because the growth of each tree takes into account the trees that have already been grown, smaller trees are typically sufficient. Moreover, using smaller trees can aid in interpretability as well.

We have seen that gradient boosting is a greedy algorithm and can quickly overfit our training dataset. Hence, it can benefit from regularization methods that penalize in many different ways the algorithm and generally improve its performance by reducing overfitting. A first implementable enhancement is adding tree constraints because it is important that the weak learners have skill but remain weak. A good general heuristic is that the more constrained tree creation is, the more trees will be required in the model, and vice versa. There are many constraints that can be imposed on the construction of decision trees: increasing the number of trees (adding more trees to the model generally can slow the overfitting occurrence); limiting tree depth (deeper trees are more complex and tend to overfit) or the number of nodes (it constrains the size of the tree too); imposing a minimum number of samples per split to constrain the amount of observations at a training node before a split can be considered; setting a minimum improvement to loss that any split added to a tree need to generate.

As previously mentioned, the predictions of each tree are added together sequentially. So, another improvement to the algorithm can be made by weighting the contribution of each tree to this sum to slow down even further the learning of the model. This weighting is called a shrinkage or a learning rate. Slowing down the learning requires in turn more trees to be added to the model, and this implies in turn longer time to train, providing a configuration trade-off between the number of trees and the learning rate. It is common to select small values for this hyperparameter.

We can think of one more possible regularization method. Allowing trees to be greedily created (like with bagging ensembles and random forests) from subsamples of the training dataset can be

used to reduce the correlation between the trees in the sequence in gradient boosting models. This variation of boosting is called stochastic gradient boosting. A few variants of stochastic boosting are available: subsample rows before creating each tree; subsample columns before creating each tree; subsample columns before considering each split. Those approaches have been described in the previous paragraphs. Generally, aggressive sub-sampling has shown to be beneficial.

Additional constraints can be imposed on the parameterized trees in addition to their structure. For example, the leaf weight values of the trees can be regularized using regularization functions, such as the widespread L_1 and L_2 weights regularization.

Mathematically, for gradient boosting regression trees the prediction $\hat{y}_i$ for a given input x_i is of the following form:

$$\hat{y}_i = F_M(x_i) = \sum_{m=1}^{M} h_m(x_i) ,$$

where the h_m are the weak learner output in the context of boosting and the parameter M represents the number of estimators.

The greedy fashion in which these trees are built can be represented as

$$F_m(x) = F_{m-1}(x) + h_m(x) ,$$

where the newly added tree h_m is fitted to minimize a sum of losses L_m, given the previous ensemble F_{m-1}:

$$h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{n} l\big(y_i, F_{m-1}(x_i) + h(x_i)\big) ,$$

where $l\big(y_i, F(x_i)\big)$ represents the loss function used. By default, the initial model F_0 is chosen as the constant that minimizes the loss. Using a first-order Taylor approximation, the value of l can be approximated as follows:

$$l\big(y_i, F_{m-1}(x_i) + h_m(x_i)\big) \approx l\big(y_i, F_{m-1}(x_i)\big) + h_m(x_i) \left[\frac{\partial l\big(y_i, F(x_i)\big)}{\partial F(x_i)}\right]_{F=F_{m-1}} .$$

The quantity $\left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}$ is the derivative of the loss with respect to its second parameter, evaluated at $F_{m-1}(x)$. It is easy to compute for any given $F_{m-1}(x_i)$ in a closed form since the loss is differentiable. We will denote it by g_i. Removing the constant terms, we have:

$$h_m \approx \arg\min_h \sum_{i=1}^{n} h(x_i)\, g_i \,.$$

This is minimized if $h(x_i)$ is fitted to predict a value that is proportional to the negative gradient $-g_i$. Thus, the estimator h_m is fitted to predict the negative gradients of the samples at each iteration and the gradients are updated at each iteration. This can be thought of as gradient descent applied in a functional space.

Gradient boosting has three main tuning parameters:

- The number of trees (estimators). Boosting can overfit if the number of weak learners is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select this hyperparameter and later we will explain this technique.
- The shrinkage parameter or learning rate. It is a small positive number that controls the rate at which boosting learns, as explained above. The right choice can depend on the problem, anyway very small value can require a very large number of estimators to be used in order to achieve good performance.
- The number of splits in each tree. It controls the complexity of the boosted ensemble. Generally, it represents the interaction depth, and controls the interaction order of the boosted model (the number of splits indicates variables at most involved).

4.4 Deep Feedforward Fully Connected Neural Networks

The deep feedforward fully connected neural networks, also called multilayer perceptrons (MLPs), are algorithms widely used in the field of deep learning. In the case of application to regression problems, these algorithms aim to find an approximation function f, function of parameters θ, which outputs a value given certain features x.

They are called feedforward because the information flows forward: from the input data, through the intermediate stages of processing to the output of function f. They are called fully connected because each neuron of a given layer provides its output (suitably weighted) to all the neurons of the next layer, as well as receives all the outputs (again suitably weighted) of the neurons of the previous layer.

The function to be optimized, representing the neural network, will consist of several functions nested one inside the other, each corresponding to a layer. The longer this chain is, the deeper the model. The final layer is called the output layer (to which the label is associated), the first is the input layer (to which the features are associated) and the intermediate ones are called hidden

layers. During the training phase we try to approximate through $f(x)$ the target value y associated with a set of features x, provided by the examples used.

The main and simplest unit of a Deep Feedforward Neural Network is the perceptron also called unit or node. This communicates with adjacent nodes through links generating the network itself. The perceptron is characterized by three fundamental elements:

- the connections, each of which characterized by a parameter w (weight)
- an adder, capable of summing the inputs coming from the various connections (appropriately weighted) with the previous layer, producing a linear combination of the input data
- the activation function, which is used to limit the width of the output and make nonlinear the function f.

A neuron can be characterized by the following equation:

$$y_k = \varphi\left(\sum_{i=1}^{m} w_{k,i}x_i + b_k\right),$$

where:

- $w_{k,i}$ are the parameters (weights) assigned to the inputs of neuron k
- x_i is the input value received by the neuron
- b_k is the bias
- φ is the activation function.

Regarding the activation function, generally three types are used in neural networks:

- The step function: it is a discontinuous function that returns 0 for all negative values and 1 for positive values.

$$\varphi(x) = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x \geq 0 \end{cases}$$

- Sigmoid function: it is strictly increasing and develops a balance between linear and non-linear trend.

$$\varphi(w) = \frac{1}{1 + e^{-aw}},$$

  where a allows to vary the slope of the function. This activation function can generate problems related to the back-propagation of the gradient, generating the vanishing gradient phenomenon.

- Rectified Linear Unit (ReLU) function: solves the vanishing gradient problem, and for this reason they are among the most used.

$$\varphi(z) = \begin{cases} 0 & if\ z < 0 \\ z & if\ z \geq 0 \end{cases}$$

4.4.1 Neural Networks Training

As already said, the main goal of a deep learning algorithm (as well as ML) is to be able to achieve high performance when using unknown inputs. To do this it is necessary to be able to optimize the network training phase to the maximum.

The first step to take is to define a function that measures the error between the model outputs and the target values (labels). The function in question is called a cost or error function (loss function). The optimization of a network is developed through a function whose goal is to modify the parameters of the network (weights, bias, etc.) to obtain a loss function that is as low as possible. The modification of weights and bias takes place through a learning algorithm that calculates a gradient vector which, for each weight, indicates how much the error increases or decreases if the weights are varied by an infinitesimal amount.

The most used approach in order to minimize the error is developed by an algorithm called Stochastic Gradient Descent (SGD). It consists in calculating the output of a network, the associated error and the average gradient by varying the weights. This procedure is iterated for smaller sets of samples (batches) taken from the training set until the mean of the error function stops decreasing.

Loss functions are usually convex functions having local minimums and a global minimum. In both cases the gradient of the function is zero, this can be a problem as if it converges on a local minimum it will no longer be possible to obtain convergence on the global minimum.

It is supposed to analyse the cost function optimization process starting from the output of a neuron:

$$y_{ANN} = \sum_{j=1}^{n_b} f\left(\sum_{i=1}^{n_{var}} w_{ij}^{(1)} x_i\right) w_{ji}^{(2)}$$

where:

- $w_{ij}^{(1)}$ is the weight between the input neuron i and the neuron j of the hidden layer
- $w_{ji}^{(2)}$ is the weight between the neuron j of the hidden layer and the neuron leaving the network
- n_{var} is the number of neurons at the entrance
- n_b neurons in the hidden layer.

The network is trained with a sample containing N examples:

$$x_m = \left(x_1, \dots, x_{n_{var}}\right)$$

For each training event m, the output function y_{ANN} is calculated and compared with the expected value $\hat{y}_m$, defining the error function E:

$$E(x_1, \dots, x_N|\boldsymbol{w}) = \sum_{m=1}^{N} E_m(x_m|\boldsymbol{w}) = \sum_{m=1}^{N} \frac{1}{2}\left(y_{ANN,m} - \hat{y}\right)^2$$

where w represents the set of weights of the neural network, the best is the one that minimizes the loss function. The set of weights that optimize the network is calculated through the SGD starting from a set of causal weights $w^{(\rho)}$ which are varied by small displacements in space in the directions $\nabla_w E$:

$$w^{(\rho+1)} = w^{(\rho)} - \eta \nabla_w E \,,$$

where η indicates a positive number between 0 and 1, it represents the learning rate. ρ instead indicates the layer of connections to which the networks refer.

$$\Delta w_{ij}^{(2)} = -\eta \sum_{m=1}^{N} \frac{\partial E_m}{\partial w_{ij}^{(2)}} = -\eta \sum_{m=1}^{N} \left(y_{ANN,m} - \hat{y}\right) y_{j,m}^{(2)}$$

The phase in which the weights are updated in this way is called back-propagation: the weights are in fact modified starting from the last layer and proceeding backwards. A cycle of presentation to the network of all the observations belonging to the training set is called an epoch.
Following this procedure, what is commonly called the test phase is developed to evaluate the model's ability to generalize.

4.4.2 Overfitting in Neural Networks

As already described, one of the main problems related to the use of artificial intelligence algorithms and in particular deep learning ones is overfitting. To avoid or reduce this phenomenon, there are several techniques that can be applied directly. Among these are mainly distinguished:

- L_2 Regularization – It is the most common one and it consists of adding a term to the loss function called the regularization term. Also known as *weight decay*, it is a way to force

the network to learn weights of small value, which reduces the complexity of the model and therefore its ability to pick up the noise of the data.

- L_1 Regularization – It is analogous to the previous one, the only change concern the function used to deploy the weight decay.
- Dropout – Unlike the previous techniques, the dropout does not modify the loss function, but modifies the network itself. The dropout technique consists in randomly ignoring a certain set of neurons in the hidden layers during the training phase, that is, they are not considered in the calculation of a certain epoch. At every training round, each neuron is maintained with a certain probability p or neglected by the network with probability $1 - p$. This forces the network to reproduce the output without the contribution of a random subset of parameters. Therefore, the dropout procedure is equivalent to averaging the results of a large number of different networks: each network will tend to have overfitting in a different way and the overall effect will be a global reduction of overfitting.

Chapter 5

Analysis and Results

In this chapter, we enter in the core part of this thesis work. The dataset relating to the diesel engine subject of this study and used for the application and validation of the ML and DL algorithms described in the previous chapter will be presented. A necessary pre-processing stage will be carried out before proceeding with the model simulations. The last part of this chapter will also involve the work of a colleague of mine whose thesis related the use of a neural network model for the same data.
Let us start, therefore, defining which are the quantities with which the models will work to determine the final prediction.

5.1 Working Datasets

For the models building and development, experimental data provided by AVL Italia were used, collected in various tests carried out in quasi-stationary conditions with the engine mounted on the vehicle and this positioned on a roller bench. Each test consists of load and EGR sweeps at a constant rotation speed, which was then varied between tests in this way: 1250 rpm, 1500 rpm, 1750 rpm, 2000 rpm, 2250 rpm. Hence, we have at disposal 6 datasets and each of them is characterized by the number of samples shown in the Table 2. The reference engine is a 2.2 L compression ignition engine whose specifications have not been given.

Engine Speed	***Dataset Size***
1250 rpm	16834 samples x 24 parameters
1500 rpm	22363 samples x 24 parameters
1750 rpm	11192 samples x 24 parameters
2000 rpm	12318 samples x 24 parameters
2250 rpm	10988 samples x 24 parameters
2500 rpm	11486 samples x 24 parameters

Table 2. Dataset size and composition

For the acquisition of soot emissions values, AVL Micro Soot Sensors were used, both in engine-out and in tail pipe positions. These sensors are of the photoacoustic type (PASS, Photoacoustic Soot Sensors) and their operation principle is schematically represented in Figure 18 and here described: sampled raw or diluted exhaust gases are conveyed through a measuring chamber, then a modulated laser beam thermally animate soot (black absorbing) particles in the gas stream, consequently periodic pressure pulsations are produced by this modulated heating of

the particles. Finally, a microphone detects the acoustic waves produced by this process and the signal is properly amplified and filtered before the record.

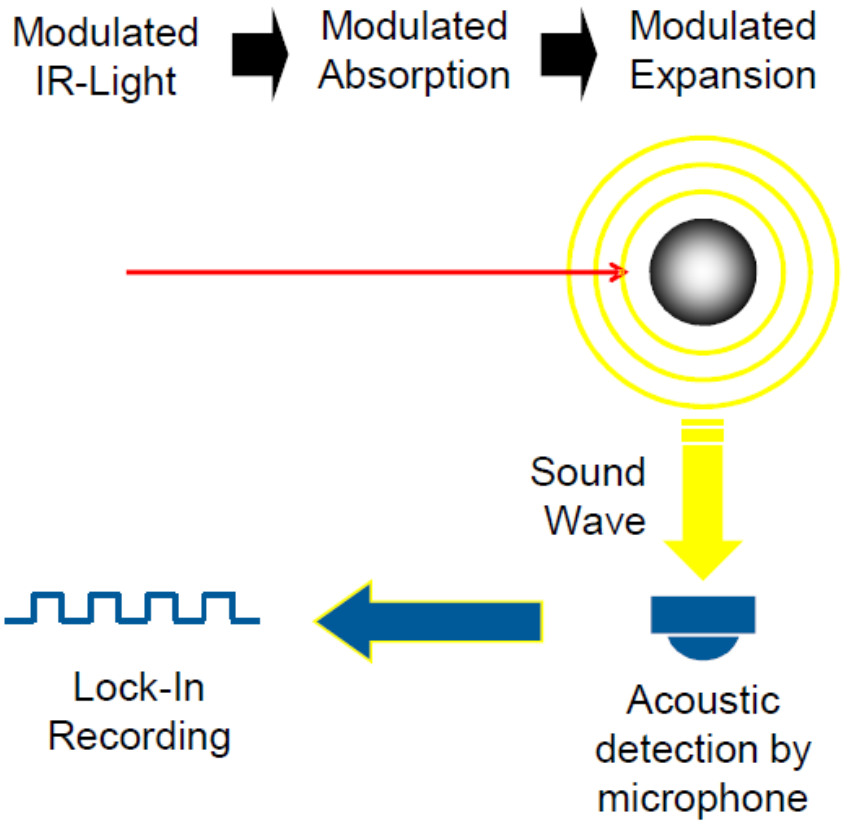

Figure 18. PASS operating principle (source: AVL Micro Soot Sensor Applications, AVL Tech Days 2012, AVL)

It has been demonstrated that this sensor signal is directly proportional to the soot concentration in the exhaust gases, with excellent intrinsic linearity. Furthermore, experimental evidence has also shown that soot emissions calculated from the photoacoustic measurement correlate excellently with the non-volatile gravimetric PM emissions.
In Table 3 and Table 4 we have divided the engine and environment parameters present in the supplied dataset into features and labels. Since this work focuses on engine-out soot emissions, available variables related to the after-treatment system and tail-pipe emissions has been removed.

5.2 Data Pre-processing

An important phase, which has already been mentioned, is dataset preparation. It consists of some pre-processing steps to prepare the data to be used as input of the ML and DL models. To this end, the following operations are carried out:

- changing of data sampling frequency

- cleaning of the dataset (outlier elimination)
- feature scaling.

Variable (Feature) Name	*Unit of Measure*	*Description*
Intake Temperature	°C	Air temperature at engine intake
Air Temperature	°C	Ambient air temperature
Rail Press	bar	Fuel injection pressure
Air Mass	mg/str	Air mass measured with HWA
Air Set point	mg/str	Expected air mass
Air Mass Flow	kg/h	Measured air mass flow rate
Coolant Temp	°C	Engine coolant temperature
Amb press	mbar	Ambient air pressure
Abs boost Pressure	mbar	Air pressure in the intake manifold
Intake Pressure	mbar	Air pressure at engine intake
Injected Quantity	mm^3/cyl	Fuel injected quantity
Lambda	-	Ratio between actual air-fuel ratio and stoichiometric air-fuel ratio
RPM	rpm	Engine speed
Speed	km/h	Vehicle speed
Exhaust gas Flow rate	m^3/s	Exhaust gas flow rate
Exhaust gas Temperature	°C	Exhaust gas temperature
EGR Position	%	EGR valve position
EGR Target Position	%	Expected EGR valve position

Table 3. Dataset variables (features)

Label Name	*Unit of Measure*	*Description*
MSS_EO_CONC	mg/m^3	Engine-out soot concentration

Table 4. Dataset label

5.2.1 Data Sampling Frequency

The data was acquired with a sampling frequency of 20 Hz, which is very high considering the quasi-stationary conditions. This could easily lead to a sharp overfitting of the models we will use due to the high number of similar samples. To solve this issue, it is necessary to reduce the sampling frequency of the data from 20 Hz to 2 Hz. The dataset sizes shown in the Table 5 are thus obtained. Combining the newly sampled datasets we get as result 8521 samples.

Engine Speed	*Dataset Size*
1250 rpm	1683 samples x 24 parameters
1500 rpm	2236 samples x 24 parameters
1750 rpm	1119 samples x 24 parameters
2000 rpm	1231 samples x 24 parameters
2250 rpm	1098 samples x 24 parameters
2500 rpm	1148 samples x 24 parameters

Table 5. Dataset size and composition after resampling

5.2.2 Dataset Cleaning

The dataset cleaning operation consists in eliminating all outliers (null values, measurement errors, and so on). In particular, we have found points characterized by an anomaly in the sampling system and for this reason they should be discarded from the dataset in order to have optimal learning, even if these points are only a few with respect to the total number of available samples.

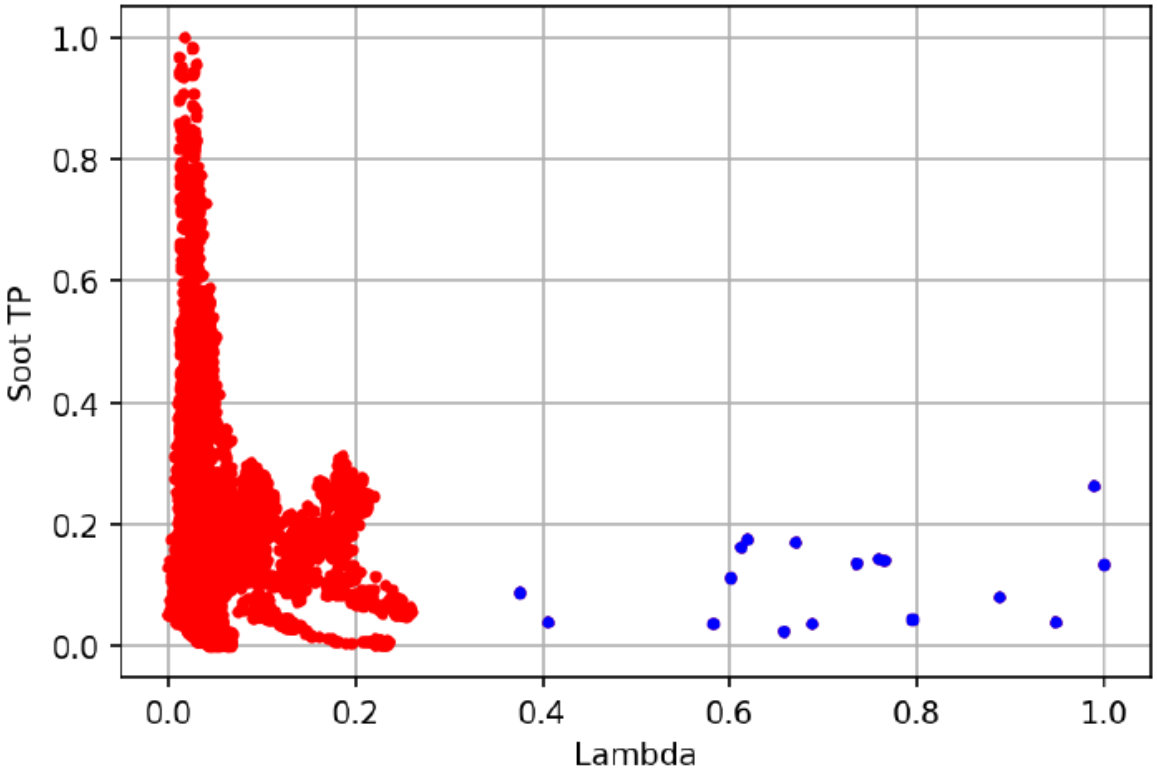

Figure 19. 'Lambda' outlier points (highlighted in blue) present in the dataset [17]

The determination of these points is of paramount importance because most of the algorithms used can be sensitive to outliers in the sense that they can make training less stable. Moreover, they can enhance overfitting tendency, thus leading to drops in performance and in

generalization capabilities of the models, especially for those able to adapt particularly well to the training data.

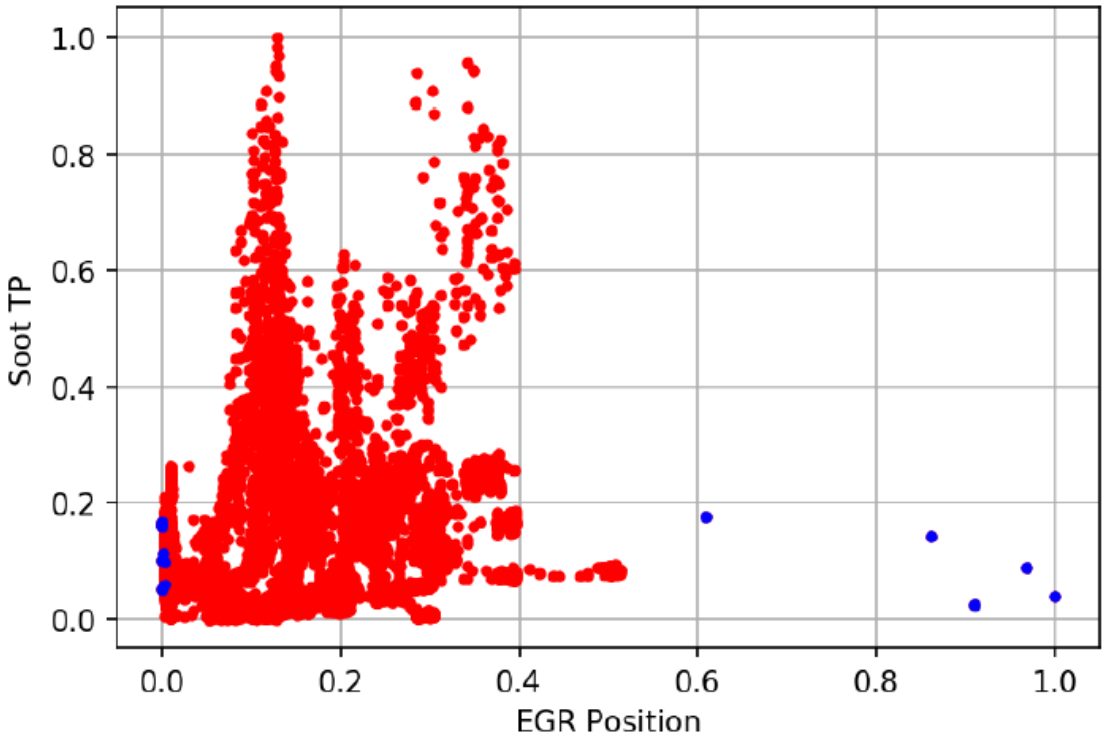

Figure 20. 'EGR Position' outlier points (highlighted in blue) present in the dataset [17]

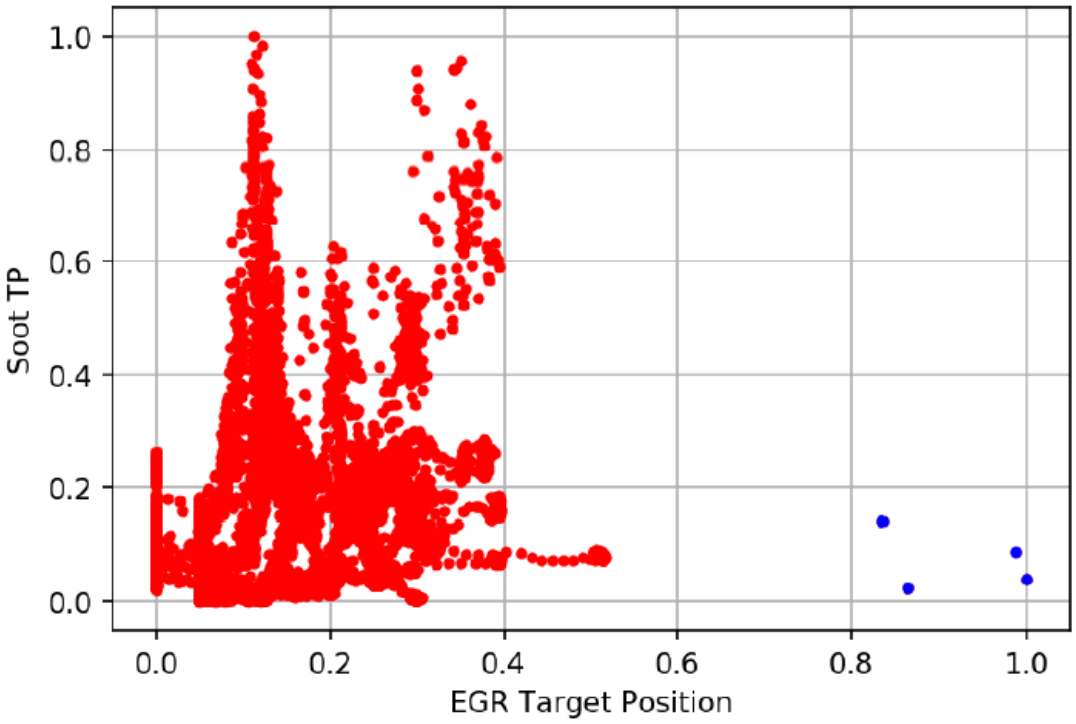

Figure 21. 'EGR Target Position' outlier points (highlighted in blue) present in the dataset [17]

Outliers were found for the ‘Lambda’ (17 points with a value greater than 3), ‘EGR Target Position’ (4 points with a value higher than 80 %) and ‘EGR Position’ (4 points with a value higher than 60 % and 8 points with a negative value) variables, as is shown in the Figures 19-20-21.

5.2.3 Feature Scaling

Due to the high variability of the samples present in the datasets provided, input features might have different units of measure that, consequently, may imply the variables have very different scales. These differences between features scales can increase the learning process difficulty for the model. For example, large input values can result in a model that learns large parameter values. A model with large weight values is quite unstable, meaning poor performance during learning and higher generalization error due to great sensitivity to input values. On the other hand, a target variable (label) with a wide spread of values might result in large error gradient values causing values of model parameters to change dramatically and making the learning process unstable. The solution is represented by feature (and label) rescaling. Feature scaling, or data normalization, is a method used to normalize the range of features (variables) and labels of our dataset. As just said, we use this technique to avoid algorithm objective functions will not work properly. Another reason why feature scaling is applied is that gradient-descent-based optimization algorithms will converge much faster. It is also important to apply feature scaling if regularization is used as part of the loss function used, so that penalties are appropriately given to its parameters.

The two most common data scaling methods are normalization and standardization. Normalization consists of rescaling each real-valued numeric feature into the range which is usually between zero and one. It is useful to scale the input attributes for an algorithm that relies on the magnitude of their values, such as distance measures used in k-nearest neighbors. A first flavour, and also the simplest, of this method is called min-max normalization, also known as unity-based normalization, and consists in rescaling the values in the range between 0 and 1. Its general formula is given as:

$$z = \frac{x - \min(x)}{\max(x) - \min(x)},$$

where x is the original value and z the normalized one.

Another version of this method is the mean normalization:

$$z = \frac{x - \text{mean}(x)}{\max(x) - \min(x)},$$

where x is the original value, z the normalized one.

Standardization refers to shifting the distribution of each feature to have zero-mean and a standard deviation of one (unit-variance). It is useful to standardize attributes for a model that relies on the distribution of attributes.

$$z = \frac{x - \text{mean}(x)}{\sigma},$$

where x is the original feature vector, z the normalized one and σ is its standard deviation.

For the construction of the models used in this work, we have applied the min-max normalization described above which is implemented by the `MinMaxScaler` function in the Scikit-learn library for Python. Its unique important parameter is `feature_range`: a tuple must be passed to it with the indication of the minimum and maximum values of the desired range of transformed data.

SVM algorithms are not scale invariant, so it is necessary to scale data to obtain meaningful results. The same goes for Nearest Neighbors algorithms as they have to compute distances between two points in multidimensional feature space and if one of the attributes has a broad range of values, the distance will be governed by this particular feature. This consideration still holds true for neural networks: in fact, the high variability, due to the different units of measurement, can give rise to anomalies within the network, causing some features to take precedence over others. Instead, in the case of gradient boosting for regression it is not strictly necessary to perform feature scaling. So, we have followed these guidelines in building the models used for the simulations performed.

5.3 Experimental Setting

After having performed the pre-processing stages described so far, the next step consists in the splitting of the dataset in two parts: a training set dedicated to the learning process and a test set for evaluating model performance. As said in the previous chapters, this is a common procedure carried out to avoid a methodological mistake, that is let the model to learn the parameters of its approximation function and testing it on the same data. The model would have a perfect score just repeating the labels of the samples it has already seen but would fail to make predictions on yet-unseen data (i.e., overfitting).

In order to perform this split, the function `train_test_split` from the Scikit-Learn Python library is used. The most important parameters of this function are reported below:

- `test_size` – If set to a float number, it should range between 0 and 1 and it represents the proportion of dataset that will be included in the test. If it is set to an integer number, it stands for the number of samples that will end up in the test split. Obviously, in both cases the complement indicates what is assigned to the training set.
- `random_state` - Controls the randomness of the shuffling applied to the data before applying the split. Passing to it an integer number it is possible to obtain reproducible outputs across multiple function calls.
- `shuffle` – It is a Boolean value that state whether to shuffle the data before splitting. If set to `True` it will randomly mix samples in the dataset before generating the training and test sets, otherwise the dataset will be split without mixing the samples.

In the construction of the models described below, we have chosen a test set size of 20 % of the original dataset, in order to balance the samples in the two splits having enough of them to train properly the models and also to evaluate their performance. We have decided to shuffle the data before splitting to avoid the learning process to be influenced by the ordering in which samples were collected. To allow the reproducibility of results and the comparison between different algorithms, a *seed* is fixed to control the randomness of the shuffling applied. In this way, models performance can be compared because obtained using the same data. The code snippet used to set the seed, when building the models, is the following:

```
import numpy as np
import random as python_random
seed = 23
np.random.seed(seed)
python_random.seed(seed)
```

Since every machine learning algorithm has its own hyperparameters that characterize its operative configuration and govern its learning process, another step to take is choosing their values in order to obtain a model optimized for the available data and ready to be trained. At this point is worth to highlight a key point: the difference between parameters and hyperparameters. A model parameter is a configuration variable that is part of the model, and its value is estimated or learned from data. It is internal to the model and thus cannot be set manually by the user. It defines the model skills on the learning problem to which it is applied. On the other hand, a model hyperparameter, also called tuning parameter, is a configuration variable that is external to the model and whose value cannot be estimated from data. It is specified by the user, usually using heuristics, and it is tuned for a particular predicting problem. Coming back to our

experimental setting, we have to set model hyperparameters, also called *tuning*, before starting the learning pipeline. To select their optimal values, a possible procedure is to train a series of models with different settings (hyperparameter values) on the training set and hence evaluate them on the test set, keeping the model that best performed. However, there is a high risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally and, doing so, knowledge about test set samples can "leak" into the model and thus evaluation metrics are no longer useful to rate model generalization performance. To solve this problem, another part of the original dataset can be held out, the so-called validation set: learning proceeds on the training set, after which evaluation is done on the validation set, and when the optimal hyperparameter combination is found, final performance estimation can be carried out on the test set. However, by partitioning the available data in this way, the number of samples which can be used for learning the model is drastically reduced, and the results can heavily depend on the specific choice for the training and validation sets.

As already mentioned, the cross-validation (CV) procedure can be used to overcome this issue. The test set should still be held out for final evaluation, but the validation set is no longer needed when applying this technique. In the basic approach, called k-fold cross-validation, the training set is split into k smaller "folds" and the following procedure is recursively applied:

- the model is trained using $k - 1$ folds as training set
- the resulting model is validated on the remaining k^{th} fold (i.e., it is used as a test set to evaluate the model performance).

The performance metric output by k-fold cross-validation is then the average of the values computed in each iteration (they will be k in total) of the loop. This approach allows to avoid wasting too much data (as in the case of fixing a validation set) at the price of an increase in the required computational time. Anyway, it represents a major advantage in cases where the number of samples is very small. Attention must be paid to the choice of k in order to allow the size of each test partition to be large enough to be a valid sample of the problem, at the same time enabling enough iterations of the model evaluation to provide a fair estimate of its performance on unseen data.

We have said that before starting the learning process we need to set the model hyperparameters to their optimal values for the current problem. Note that models based on different algorithms will require the setting of diverse hyperparameters. Moreover, models based on the same algorithm but applied on different problems (that is, given different datasets to learn from) will show different optimal values of the tuning parameters from each other. Note that it is common that a small subset of those parameters can have a large impact on the predictive or computation performance of the model while others can be left to their default values. To tune the

hyperparameters of a model, one way could be to manually try different values, changing them each time until the best combination is found. Obviously, this will result in a long and tedious searching work. Fortunately, the Scikit-learn library for Machine Learning provides the function `GridSearchCV` that automates this task. In fact, it is the implementation of the approach that exhaustively generates all parameter combinations from a list containing ranges of values for each hyperparameter considered. Through cross-validation the best combination of values is selected as the hyperparameters setting with the best value of a chosen scoring parameter. To summarize, this method of grid searching optimal model hyperparameters consists in an estimator (i.e., our model), a parameter space (containing the ranges of candidate values for each hyperparameter), a cross-validation scheme and a scoring function. In fact, the most important parameters of the `GridSearchCV` function are:

- `estimator` – It represents the model based on the algorithm we want to apply.
- `param_grid` – It consists of the hyperparameters names and the lists of parameter settings to try as values.
- `scoring` – It is the strategy to evaluate the performance of the cross-validated model on the test set (i.e., the k^{th} fold), that means it controls what metric is applied to the estimator evaluated. All metrics follow the convention that higher return values are better than lower return values. Thus, score parameters which represent a measure to be minimized are changed in sign.
- `cv` – It determines the cross-validation splitting strategy to be used. The most common input for this parameter is an integer that specifies the relative number of folds.

For the hyperparameter optimization routines run in the following sections to find the best values for the models used, we have selected the negative mean squared error as the scoring metric to evaluate the different combinations of hyperparameter values. Moreover, we have chosen a value of 3 for the number of the cross-validation folds. For modest sized datasets, as in our case, recommended values for this parameter range from 3 to 10 and the most common are 3, 5 and 10. The choice made here is motivated by the limited number of samples available in our dataset and the hardware at our disposal to carry on the computations required: these constraints prevent us from selecting a higher value of folds. The estimator and the list of hyperparameters (with their corresponding ranges of values explored) change between different algorithms, as already mentioned. For this reason, they will be specified later in each of the following paragraphs, where the different models will be introduced.

5.4 KNN Regression Model

In this part of the chapter, the model based on the Nearest Neighbors regression algorithm is built. Moreover, the results of the model training and hyperparameter tuning processes are reported together with model's behaviour analyses about learning and parameter variations. The chosen implementation is represented by the `KNeighborsRegressor` function provided by the Scikit-learn library. It is the formulation based on the k-nearest neighbors concept. Recall that the target of each query point is predicted by local interpolation of the labels associated to the k nearest neighbors of that point in the training set. The main parameters to be set for this function are the following:

- `n_neighbors` – It specifies the number of neighbors to consider by default for each query point.
- `weights` – It determines the weight function used when making predictions. The user can define and use a custom function or choose two available options: `uniform` and `distance`. The former assigns uniform weights to all points in each neighborhood, that means they are equally weighted. The latter, instead, weights nearest points proportionally to the inverse of the distance from the query point: in this way, closer neighbors will have a greater influence than those further away.
- `algorithm` – It is the algorithm used to compute the nearest neighbors and those available are brute-force search (`brute`), ball tree (`ball_tree`) and KD tree (`kd_tree`) types. There is an additional option, 'auto', which tries to decide the most appropriate algorithm based on the values passed to the function.
- `leaf_size` – It represents a parameter passed to ball tree or KD tree, in case they are the selected algorithm, and it allows to internally switch to brute force searches within leaf node points. It can affect the memory required to store the tree as well as the speed of its construction and query. The optimal value depends on the type of the problem.
- `metric` – It is the distance metric to use for the tree. The two most common ones are the Chebyshev and Minkowski distances. The latter is characterized by an internal parameter that allows to decline the metric into other two.
- `p` – It represents the above-mentioned power parameter for the Minkowski metric. When $p = 1$, this is equivalent to using Manhattan distance, and Euclidean distance for $p = 2$. For arbitrary p, general Minkowski distance is used with the following expression:

$$D_{Minkowski}(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_i |x_i - y_i|^p\right)^{1/p}$$

For our model built with this `KNeighborsRegressor` function, the Minkowski metric is chosen since it allows including different distance metrics that can be selected changing the p parameter. The choice of its optimal value will be then demanded to the hyperparameter search function used in the next step.

Hyperparameter	***Explored Values***
n_neighbors	1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 15 - 20 - 50 - 100
weights	distance - uniform
algorithm	ball_tree - kd_tree - brute
leaf_size	5 - 15 - 30 - 45 - 90 - 200
p	1 - 2 - 3 - 4

Table 6. Grid-searched hyperparameters values for the KNN model

Hyperparameter	***Best Value***
n_neighbors	5
weights	distance
algorithm	ball_tree
leaf_size	30
p	1

Table 7. Best combination of hyperparameters values for KNN model

To find the optimal values for the above disclosed hyperparameters, a grid search is carried out using the `GridSearchCV` function already described. The estimator parameter is set to `KNeighborsRegressor` and the candidate values explored for each hyperparameter are displayed in Table 6. In Table 7 are reported the results of the procedure, that is the combination of the hyperparameter values that lead to the best cross-validated performance (in terms of mean squared error reduction).

Evaluation Metric	***Value***
Test R^2	0.951
Training R^2	1.000
Test MSE	0.0471
Training MSE	0.0000
Test RMSE	0.2170
Training RMSE	0.0000

Table 8. KNN model performance evaluation metrics

Finally, the model is retrained on the whole training set using the values just found as hyperparameters and then its performances are evaluated on the test set. The findings are disclosed in Table 8, where evaluation metrics are reported for both the training and testing phases. The former ones should not be considered quantitatively since they have been obtained on already seen data but comparing them with testing performance can help getting qualitative insights about model generalization capabilities.

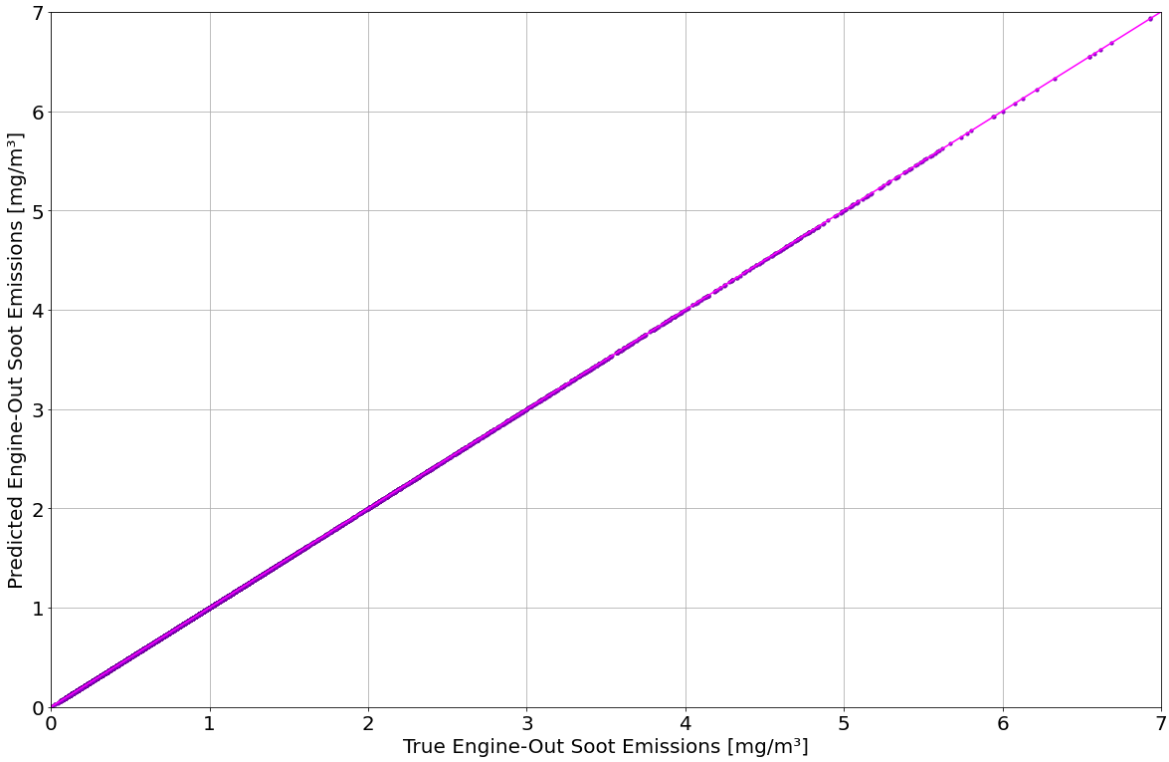

Figure 22. Correlation between actual and predicted emission values on the training set for the KNN model

In Figures 22-23 the correlations between predicted and true (real) values of the label for both the training and testing phases are displayed. As already outlined, the KNN algorithm stores the training observations (features and target) and use them for predicting labels of new samples: this is the reason why predicted and actual values correlate perfectly in the case of training set, as it is shown in Figure 22.

Figure 23 shows us a more meaningful insight on the model predictive capabilities reporting the correlation between observed and predicted label values for the unseen testing data. As can be seen from the plot, the designed model exhibits good performances.

In the plot reported in Figure 24, the model performance along with its fitting and query (prediction) times for different `n_neighbors` parameter values are shown. It can be seen that for very low and very high values of the number of neighbour samples used during prediction the validation MSE is relatively poor, while for values around 5 it is quite good. In the first case, the

reason is that the model's prediction relies only on a single or a couple of stored observations nearest to the query point (overfitting-like situation) and these are the only ones able to affect the model's output. On the other hand, in the second case, the model's prediction can rely on a large number of stored observations nearest to the query point (underfitting-like situation).

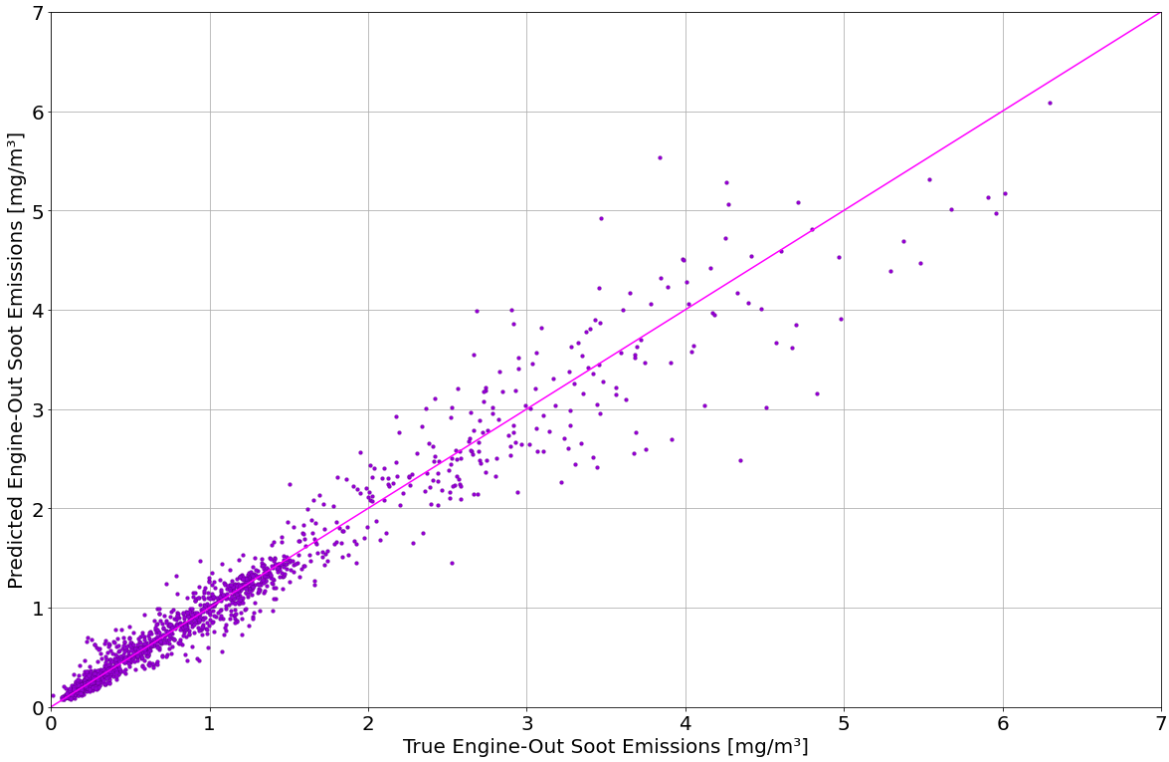

Figure 23. Correlation between actual and predicted emission values on the test set for the KNN model

As can be easily spotted, model prediction time rises as `n_neighbors` (k) increases. This behaviour is typical of ball and KD tree algorithms and it is caused by two concurrent effects: the first is that a larger `n_neighbors` value leads to the need of searching a larger portion of the feature space; secondly, using a parameter value greater than one requires internal queueing of results due to tree traversing. As `n_neighbors` becomes quite large, the ability to prune branches in a tree-based query is fairly compromised and thus brute force queries can result more efficient. On the other hand, fit times are not remarkably affected by varying the value of `n_neighbors`.

Generally, for small dataset sizes a brute force search can be more efficient than a tree-based query, as already discussed. This evidence is accounted for in the ball and KD tree algorithms by the possibility to internally switch to brute force algorithm: the threshold for this switch is stated by the `leaf_size` parameter. Its selected value causes many effects, as highlighted by the Figure 25. Choosing a larger `leaf_size` speeds up the tree construction leading to lower fitting times, because fewer nodes are required to be created during training. On the other hand, concerning the query times the opposite situation takes place: for high values of `leaf_size`

searches become basically brute force and thus prediction times grow up, instead for small values query times tend to decrease.

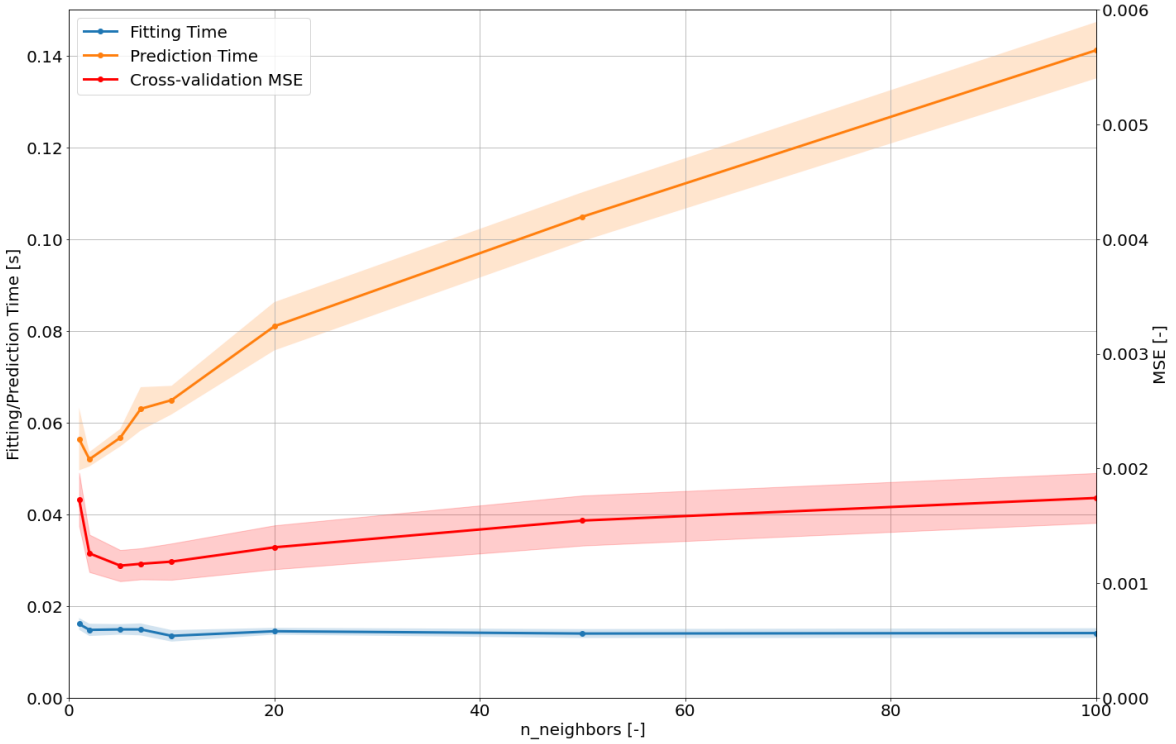

Figure 24. KNN model performance (normalized MSE) and time required for fitting and prediction versus `n_neighbors` *parameter (model settings:* `weights=distance`, `algorithm=ball_tree`, `leaf_size=30`, `p=1`, `metric=minkowski`*) (shaded area represents ± 1σ range)*

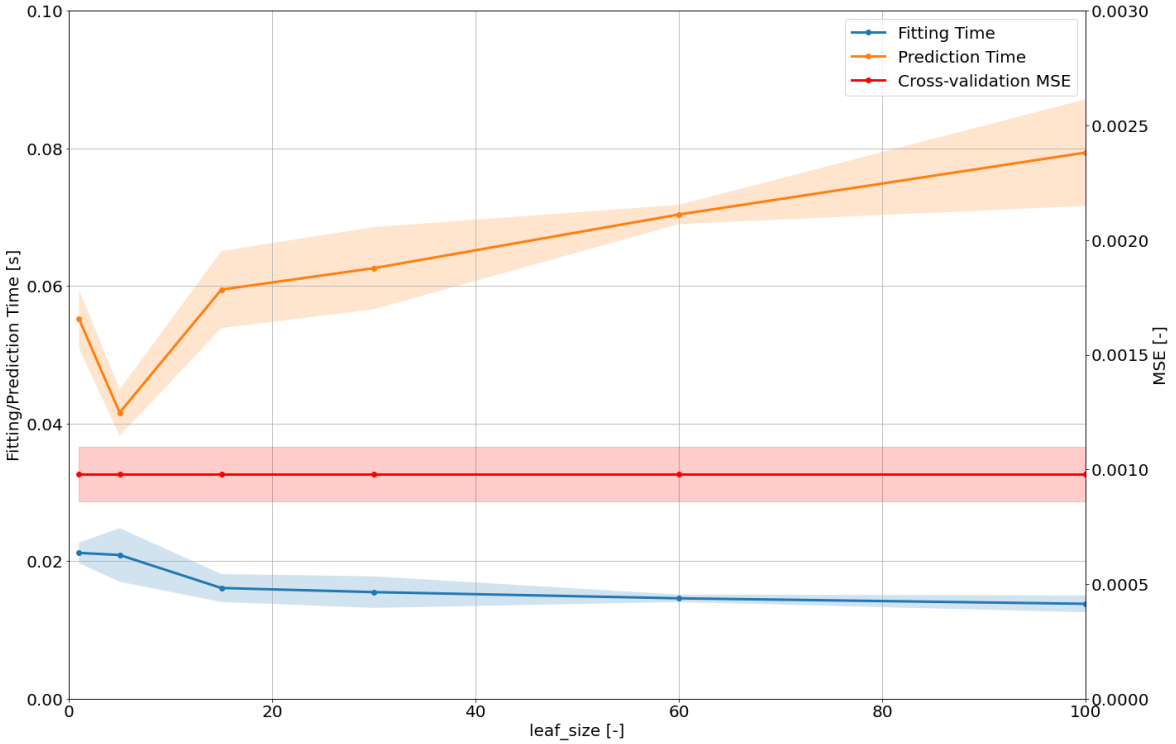

Figure 25. KNN model performance (normalized MSE) and time required for fitting and prediction versus `leaf_size` *parameter (model settings:* `n_neighbors=5`, `weights=distance`, `algorithm=ball_tree`, `p=1`, `metric=minkowski`*) (shaded area represents ±1σ range)*

Approaching `leaf_size` values around one, issues in traversing nodes can lead to high variability in query times. A good trade-off can be achieved choosing `leaf_size` equal to 30 circa, which is also the default parameter value in the function used. As it can be noted from the plot, this parameter does not influence the model's performance at all.

Finally, it is a good practice to check that our model does not suffer from high bias or high variance problems and for this purpose the most common method is based on the analysis of the model learning curves. They show the validation and training score of a model when the number of training samples is varied. They represent a useful tool to find out how much our model can benefit from adding more training data and whether it suffers more from a high variance problem or a high bias one. Every model has its advantages and drawbacks, and its generalization error can be decomposed in terms of bias, variance, and noise. The bias of an estimator is its average error for different training sets. Simplifying assumptions give bias to a model. The more erroneous the assumptions with respect to the true relationship, the higher the bias, and vice versa. Instead, the variance of a model determines how sensitive it is to varying training sets, that is the amount by which it changes as training sets do. Noise is a property of the data and leads to the so-called irreducible error. It can be mathematically demonstrated that both bias and variance can only add to a model's error. Therefore, to obtain a low error, both bias and variance has to be kept at their minimum. However, it is not possible to pursue these two objectives together because they are conflicting: this represents the well-known trade-off between bias and variance.

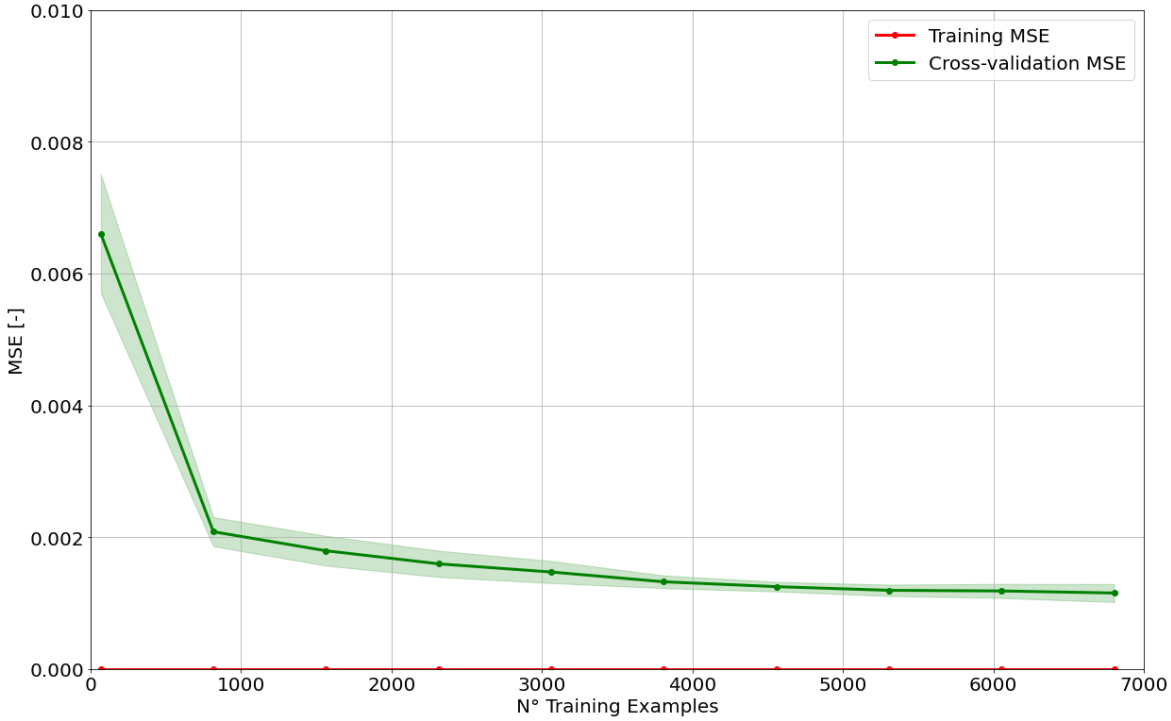

Figure 26. KNN model training and cross-validation learning curves (normalized MSE) by varying the training set size (shaded area represents ±1σ range)

In Figure 26 are reported the learning curves related to our model. As mentioned before, the KNN algorithm stores the training observations and use them for predicting labels of new samples. For this reason, the training error reported in the plot as the number of examples increases is practically zero. The validation error curve has a good shape, that is it drops to a fairly low MSE value as the number of training observations grows. The two previous remarks allow us to state that the model does not suffer from high bias. It seems the curve reaches a plateau and for this reason adding more training instances is not likely to help to reduce the error, at least in a remarkable way. The resulting gap between the two curves can be considered fairly small and hence it suggests our model does not suffer from high variance.

5.5 SVM Regression Model

In this section, the model based on the SVM regression algorithm is built. Furthermore, the results of the model training and hyperparameter tuning processes are reported together with model's behaviour analyses about learning and parameter variations. The selected implementation of Support Vector Regression is represented by the `SVR` function provided again by the Scikit-learn library. It is the formulation based on the epsilon-insensitivity concept. Recall that fit time scales more than quadratically with the number of samples, thus making it hard to scale to datasets bigger than 20000 samples circa. Since our training set is adequately smaller than this threshold, we selected this implementation, which is based on LIBSVM. The main parameters to be set for this function are the following:

- `kernel` – It specifies the type of kernel to be used in the algorithm and those available are linear, sigmoid, polynomial (`poly`) and gaussian (`rbf`) kernel.
- `gamma` – It is the coefficient (i.e., the free parameter) that appears in the kernel formulations, as they have been reported in Table 1. The user can use a floating number or choose two additional options: 'auto' and 'scale'. The former sets this parameter to $1/n_{features}$, while the latter to $1/(n_{features} \cdot X.var())$.
- `C` – It represents the regularization parameter and must be strictly positive. The strength of the regularization obtained is inversely proportional to its value. The penalty is of the squared L_2 type.
- `epsilon` – It specifies the radius of the tube within which no penalty is associated in the training loss function for points predicted within it, that is at a distance smaller or equal to epsilon from the actual value.

For the model built with this `SVR` function, we selected a gaussian kernel. Also known as radial basis function (RBF) kernel, it is a popular choice and widely used in various kernelized learning

algorithms where it has proved to be very effective. In particular, it is very common in support vector machine applications. In the case of RBF kernel, gamma can also be named the shape parameter.

Hyperparameter	***Explored Values***
C	0.001 - 0.01 - 0.1 - 0.5 - 1 - 5 - 10 - 50 - 100 - 500 - 1000
epsilon	0.0001 - 0.0005 - 0.001 - 0.005 - 0.01 - 0.05 - 0.1
gamma	auto - scale - 0.01 - 0.1 - 0.5 - 1 - 5 - 10 - 25 - 50 - 100

Table 9. Grid-searched hyperparameters values for the SVR model

To find the optimal values for the above disclosed hyperparameters, a grid search is carried out using the `GridSearchCV` function already described. The estimator parameter is set to `SVR` and the candidate values explored for each hyperparameter are displayed in Table 9.

Hyperparameter	***Best Value***
C	5
epsilon	0.01
gamma	10

Table 10. Best combination of hyperparameters values for SVR model

In Table 10 are reported the results of the procedure, that is the combination of the hyperparameter values that lead to the best cross-validated performance (in terms of mean squared error reduction).

Finally, the model is retrained on the whole training set using the values just found as hyperparameters and then its performances are evaluated on the test set. The findings are disclosed in Table 11, where evaluation metrics are reported for both the training and testing phases. Here, the same considerations about training metrics exposed in the previous section hold here.

Evaluation Metric	***Value***
Test R^2	0.945
Training R^2	0.964
Test MSE	0.0529
Training MSE	0.0382
Test RMSE	0.2300
Training RMSE	0.1954

Table 11. SVR model performance evaluation metrics

As displayed in Figures 27-28, it is also interesting to plot the correlation between predicted and true (real) values of the label, for both the training and testing phases.

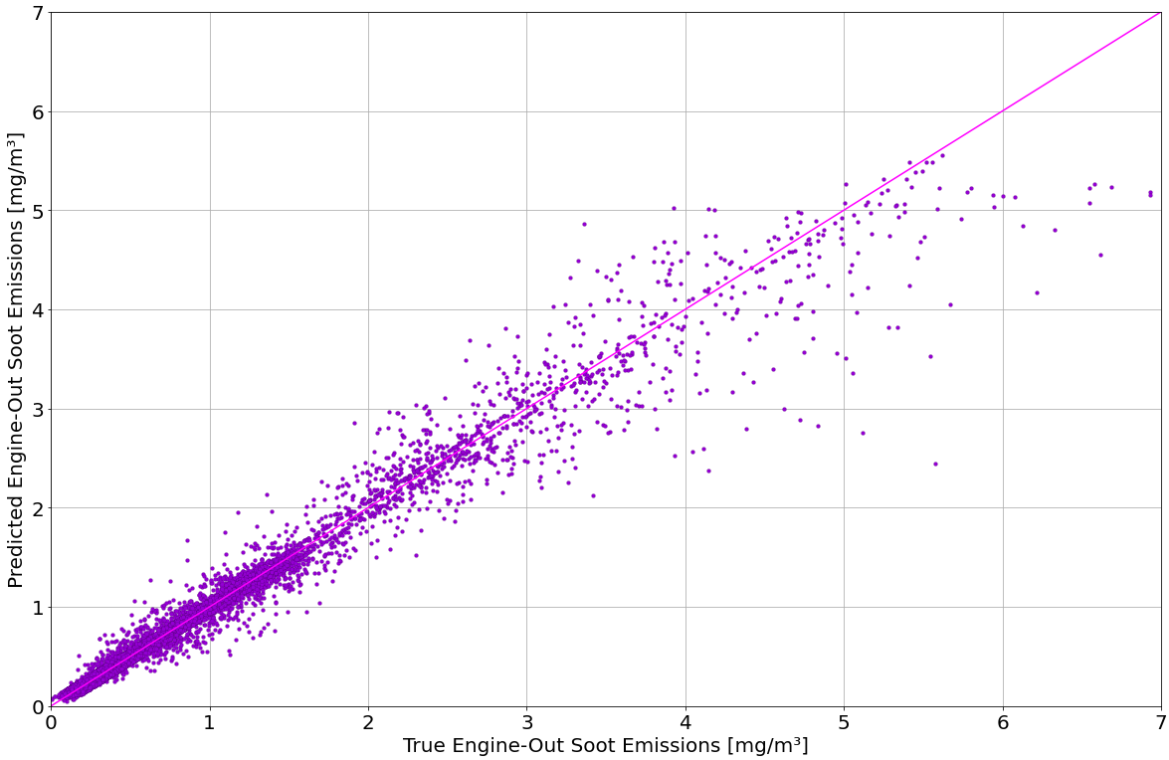

Figure 27. Correlation between actual and predicted emission values on the training set for the SVR model

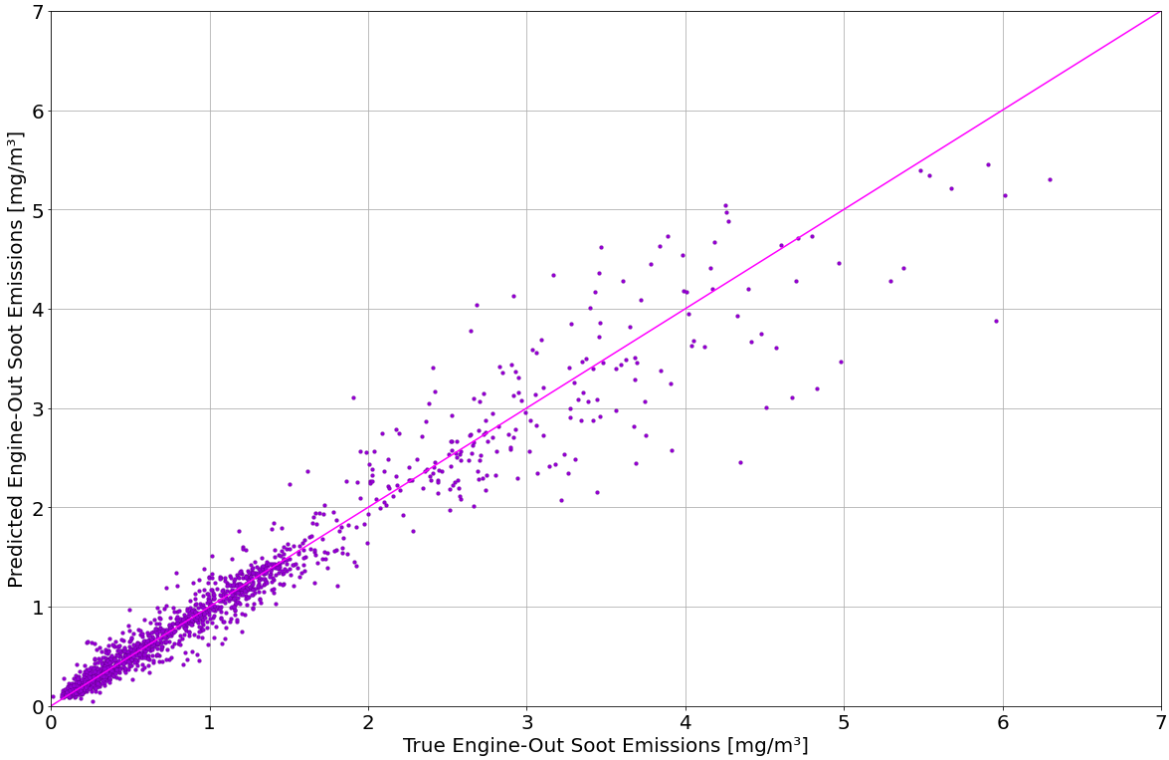

Figure 28. Correlation between actual and predicted emission values on the test set for the SVR model

In Figure 27 the satisfactory degree of correlation achieved by the model on the training set can be noted. It is also possible to spot the effect of the "epsilon tube": there is a very high concentration of points near the perfect-correlation line because the algorithm assigns no penalties to predictions that are no further than epsilon from the observed target.

Figure 28 gives us an interesting insight on the model predictive capabilities reporting the correlation between observed and predicted label values for the unseen testing data. As can be seen from the plot, the designed model confirms the good performance expressed by the evaluation metrics disclosed in the Table 11.

In machine learning applications understanding model's behaviour is of paramount importance. To this end, the effects on our model of the hyperparameters gamma and C can be investigated and the Figures 29-30 illustrate the results.

In Figure 29 training and cross-validation MSE of our model are reported for different values of the kernel parameter gamma. For very small values of gamma, you can see that both the training and the cross-validation MSE are quite high: the model is underfitting the data. Instead, medium values of gamma will result in small values for both errors, that means our model is performing fairly well. If gamma is too high, the model overfit the data, that is the training MSE continues to decrease while the cross-validation one starts increasing again.

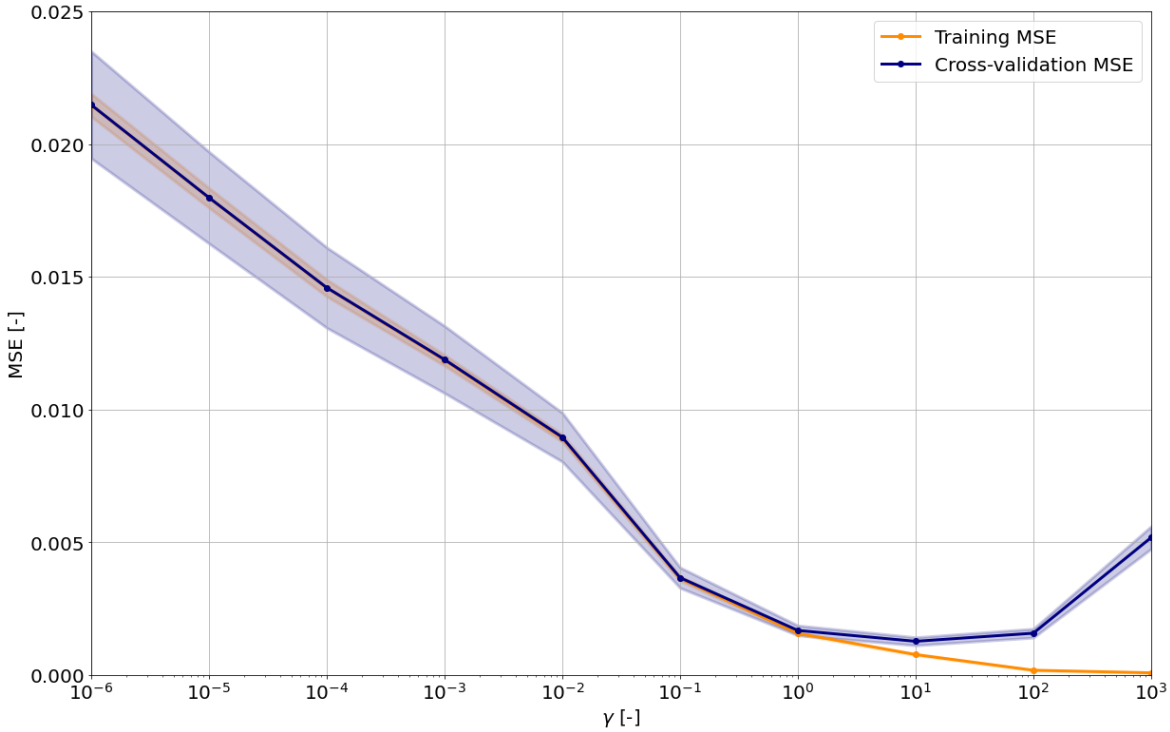

Figure 29. SVR model training and cross-validation performance (normalized MSE) versus gamma parameter (model settings: `C=5, epsilon=0.01`*) (shaded area represents ± 1σ range)*

Training and cross-validation MSE of our model for different values of the C parameter are shown in Figure 30. As for the previous case of gamma, for small values of C, you can see that

both the training and the cross-validation MSE are quite high: the model is underfitting the data. Instead, medium values of C will result in small values for both errors, that means our model is performing well. If C is too high, the model overfit the data, that is the training MSE continues to decrease while the cross-validation one starts increasing again.

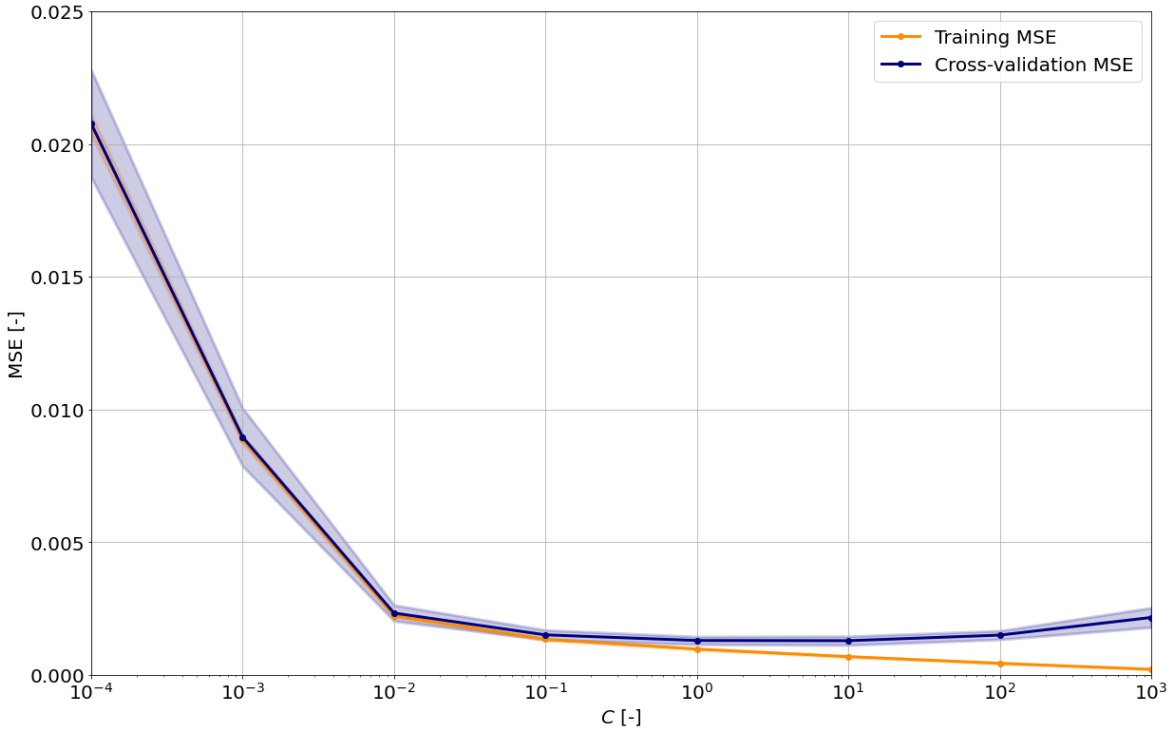

Figure 30. SVR model training and cross-validation performance (normalized MSE) versus C parameter (model settings: `gamma=10, epsilon=0.01`*) (shaded area represents ± 1σ range)*

As already explained, the parameter C, common to all SVM kernels, trades off good approximation of training examples against flatness of the regression hyperplane. A low C makes the latter smooth (i.e., flat) at the cost of higher training error. A high C aims at approximating all training examples correctly, accepting a less regular hyperplane. For this reason, C is said to behave as a regularization parameter in the SVM model. On the other hand, the gamma parameter states how far the influence of each training example reaches, with low values meaning 'far' and high values meaning 'close'. The gamma parameter can be seen as the inverse of the influence radius of samples found by the model as support vectors.

In order to exhaustively describe the influence of the hyperparameters gamma and C, it is necessary also to study the relationship between them and the joint effects they have on the model behaviour. For this purpose, the heatmap of the model's cross-validation MSE as a function of the two hyperparameters is built and reported in Figure 31. As it can be easily spotted, the model's behaviour is very sensitive to the gamma parameter. If it is very large, the radius of the influence area of the support vectors only includes them and no amount of regularization, given by C, will be able to prevent overfitting. On the contrary, when gamma is

too small, the model is too constrained and cannot capture the intrinsic complexity of the data because the influence region of each selected support vector would include the whole training set. For intermediate values, we can see that good model settings can be found on a diagonal of C and gamma.

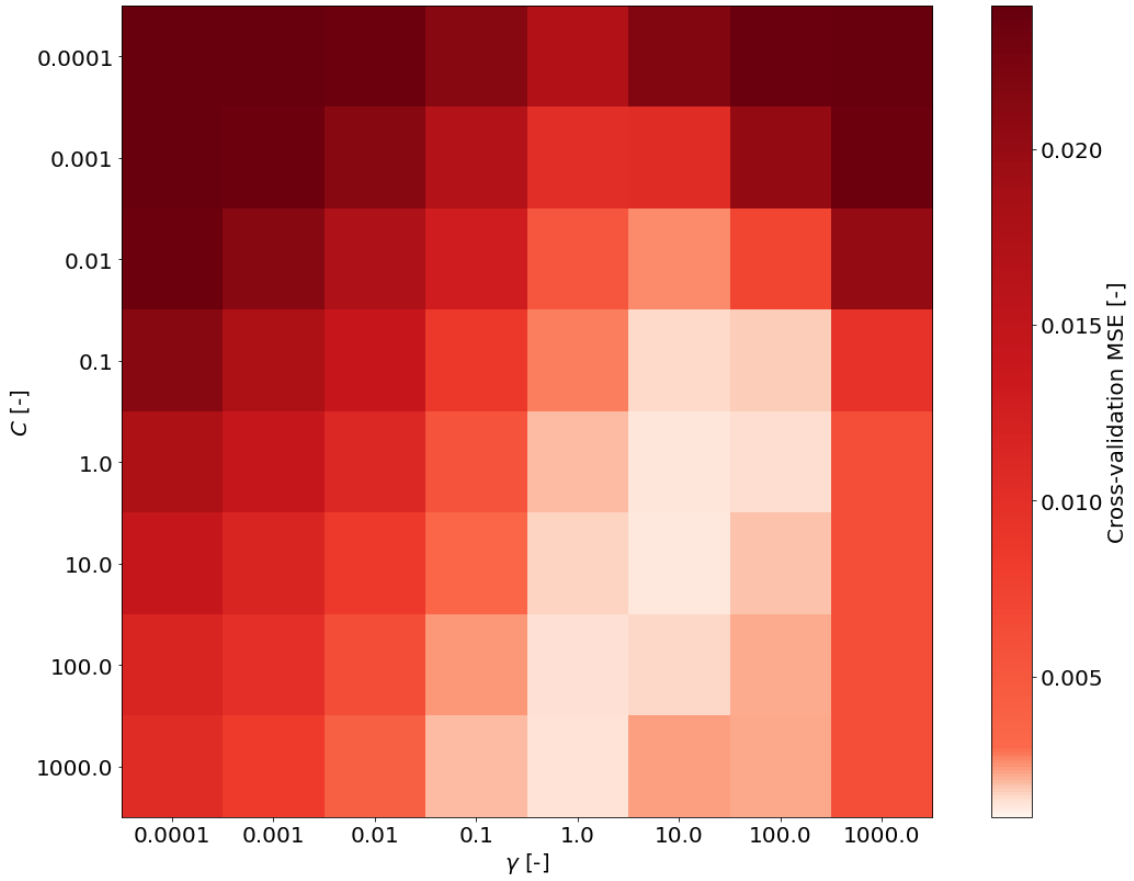

Figure 31. SVR model training and cross-validation performance (normalized MSE) versus gamma and C parameters (model settings: `epsilon=0.01`*)*

If our model is smooth (i.e., with lower gamma values), we can make it more complex by increasing the importance of approximating each sample correctly (i.e., larger C values), hence the represented behaviour is explained. Moreover, it is clearly visible that for intermediate values of gamma we get equally performing model settings at increasing C values: this can suggest that the set of support vectors is not changing anymore. Thus, we can state that gamma alone acts as a good regularization parameter. Scores being equal, it is better to use lower C values, since higher values typically means increased fitting time.

As for the previous algorithm, we check that our model does not suffer from high bias or high variance problems and to this aim the relative learning curves are plotted in Figure 32. The training MSE curve shows the peculiar trend rising from almost zero as the number of training samples grows and reaching a quite low error value. The validation MSE curve has a satisfactory shape, that is it decreases to a quite low error value as the number of training observations

increases. The two previous remarks allow us to state that the model does not suffer from high bias. It seems that both the curves reach a plateau and for this reason adding more training instances is very unlikely to help reducing the error. The resulting gap between the two curves is nearly optimal which means our model does not suffer from high variance.

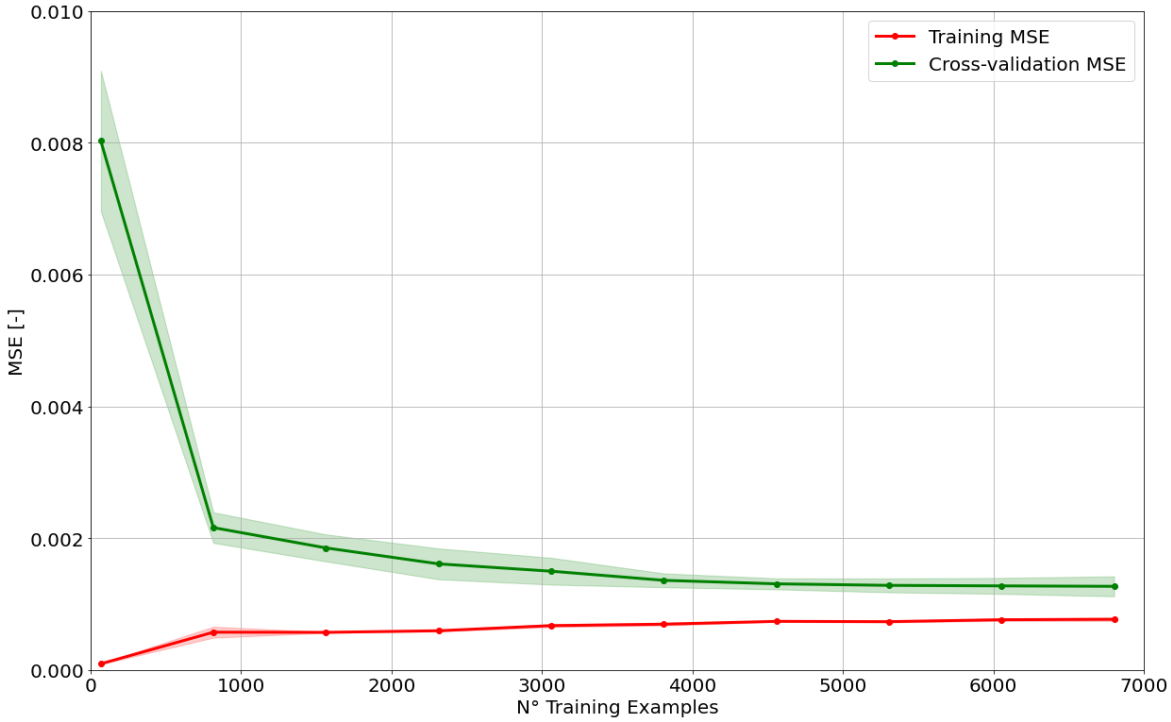

Figure 32. SVR model training and cross-validation learning curves (normalized MSE) by varying the training set size (shaded area represents ± 1σ range)

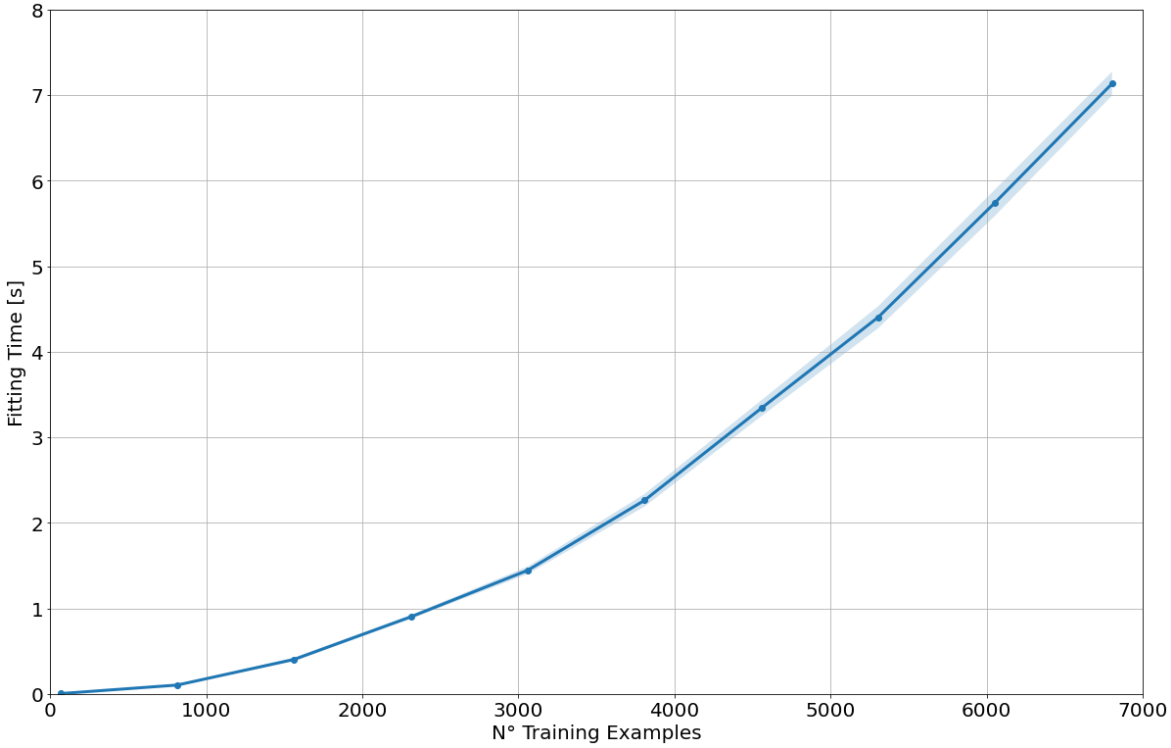

Figure 33. SVR model fitting time by varying the training set size (shaded area represents ± 1σ range)

Finally, in Figure 33 the fit time required to train the model is reported versus the dataset size used for training. As already discussed, the fitting time scales between the square and the cubic power of the number of training samples employed depending on the peculiar dataset used. The reason is that the exploited solver speed depends on how efficiently the implementation library cache is used.

5.6 Gradient Tree Boosting Regression Model

In this section, the model based on the gradient boosting algorithm for regression is built. Furthermore, the results of the model training and hyperparameter tuning processes are reported together with model's behaviour analyses about learning and parameter variations. The selected implementation of Gradient Tree Boosting is represented by the XGBoost algorithm. The name stands for eXtreme Gradient Boosting and it is a library focused on computational speed and model performance, besides offering a number of advanced features. We will use XGBoost through the Scikit-learn wrapper interface `XGBRegressor`: this is done to allow Scikit-learn functions to work with model output and parameters. Recall that gradient boosting builds an additive model with a forward stage-wise approach and in each stage a regression tree is fit on the negative gradient of the given loss function. Remember also that boosting allows for the optimization of arbitrary differentiable loss functions. The main parameters to be set for this function are the following:

- `n_estimators` – It represents the number of gradient boosted trees in the ensemble, and is equivalent to the number of boosting rounds to perform (iterations). It must be an integer number greater than or equal to 1. Gradient boosting is quite robust to overfitting so a large value for this parameter usually results in better performance.
- `max_depth` – It is the maximum tree depth for individual base learners. This parameter limits the number of nodes in the tree and the best value depends on the dataset used. Note that rising this value will consumes more memory during training and will make the model increasingly more complex and more likely to overfit. It must be set to an integer number greater than or equal to 1.
- `learning_rate` – It determines the step size shrinkage of each tree contribution, used in update to prevents overfitting. In other words, after each boosting step, this parameter shrinks the feature weights to make the boosting process more conservative. There is a trade-off between `learning_rate` and `n_estimators`. A low learning rate may reach the best optimum but makes computation slower and requires more rounds to achieve the

same reduction in residual error as a model with a higher learning rate. The value must be between 0 and 1.

- `objective` – It specifies the learning task and the corresponding learning objective. The available options for regression are: `reg:squarederror` (regression with squared loss), `reg:squaredlogerror` (regression with squared log loss), `reg:gamma` (gamma regression with log-link), `reg:pseudohubererror` (regression with Pseudo Huber loss), `reg:tweedie` (Tweedie regression with log-link). A custom objective function can also be specified by the user.
- `min_child_weight` – It represents the minimum sum of instance weight needed in a leaf node. This means if the tree partition step gives rise to a leaf node with the sum of instance weight lower than this value, then the building process will give up further partitioning. The larger this parameter is, the more conservative the model will be. The user should use a positive integer number for its value.
- `gamma` – It determines the minimum loss reduction required to make a further partition on a leaf node of the tree and it is considered a pseudo-regularisation parameter. Increasing its value, the same considerations made for `min_child_weight` hold. Practically, the higher gamma value is, the higher the resulting regularization effect.
- `subsample` – It is the subsample ratio of the training observations to help preventing overfitting. Its value must be between 0 and 1 and indicates the fraction of training data that would be randomly subsampled by the algorithm prior to growing trees. This type of subsampling will occur at each boosting iteration. A lower value prevent overfitting but might lead to underfitting.
- `colsample_bytree` – It belongs to a family of parameters for subsampling of training dataset columns. This parameter represents the subsample ratio of features (i.e., columns) when constructing each tree. Subsampling occurs once for every tree constructed. It must take on values between 0 and 1. It might improve overfitting.

For the model built with the illustrated XGBoost function, we have selected the squared loss regression objective function. It is a popular choice and widely used for regression problems where it has proved to be very effective. In the chosen implementation additional parameters are available for L_1 and L_2 regularization on the weights. Initially, it has been decided not to use regularization and the good performance obtained, and exposed in the following, together with the absence of overfitting demonstrate the correctness of this assumption.

To find the optimal values for the above disclosed hyperparameters, a grid search is carried out using the `GridSearchCV` function already described. The estimator parameter is set to

`XGBRegressor` and the candidate values explored for each hyperparameter are displayed in Table 12.

Hyperparameter	***Explored Values***
n_estimators	100 - 200 - 300 - 400 - 500 - 600 - 700
learning_rate	0.002 - 0.005 - 0.01 - 0.02 - 0.05 - 0.1
max_depth	2 - 4 - 6 - 8 - 10 - 12
min_child_weight	1 - 2 - 3 - 4 - 5
subsample	0.25 - 0.5 - 0.75 - 1
colsample_bytree	0.2 - 0.4 - 0.6 - 0.8 - 1

Table 12. Grid-searched hyperparameters values for the XGB model

In Table 13 are reported the results of the procedure, that is the combination of the hyperparameter values that lead to the best cross-validated performance (in terms of mean squared error reduction).

Hyperparameter	***Best Value***
n_estimators	300
learning_rate	0.02
max_depth	10
min_child_weight	1
subsample	1
colsample_bytree	0.8

Table 13. Best combination of hyperparameters values for the XGB model

Evaluation Metric	***Value***
Test R^2	0.960
Training R^2	0.997
Test MSE	0.0388
Training MSE	0.0033
Test RMSE	0.1971
Training RMSE	0.0572

Table 14. XGB model performance evaluation metrics

Finally, the model is retrained on the whole training set using the values just found as hyperparameters and then its performances are evaluated on the test set. The findings are disclosed in Table 14, where evaluation metrics are reported for both the training and testing

phases. Here, the same considerations about training metrics exposed in the previous section hold here.

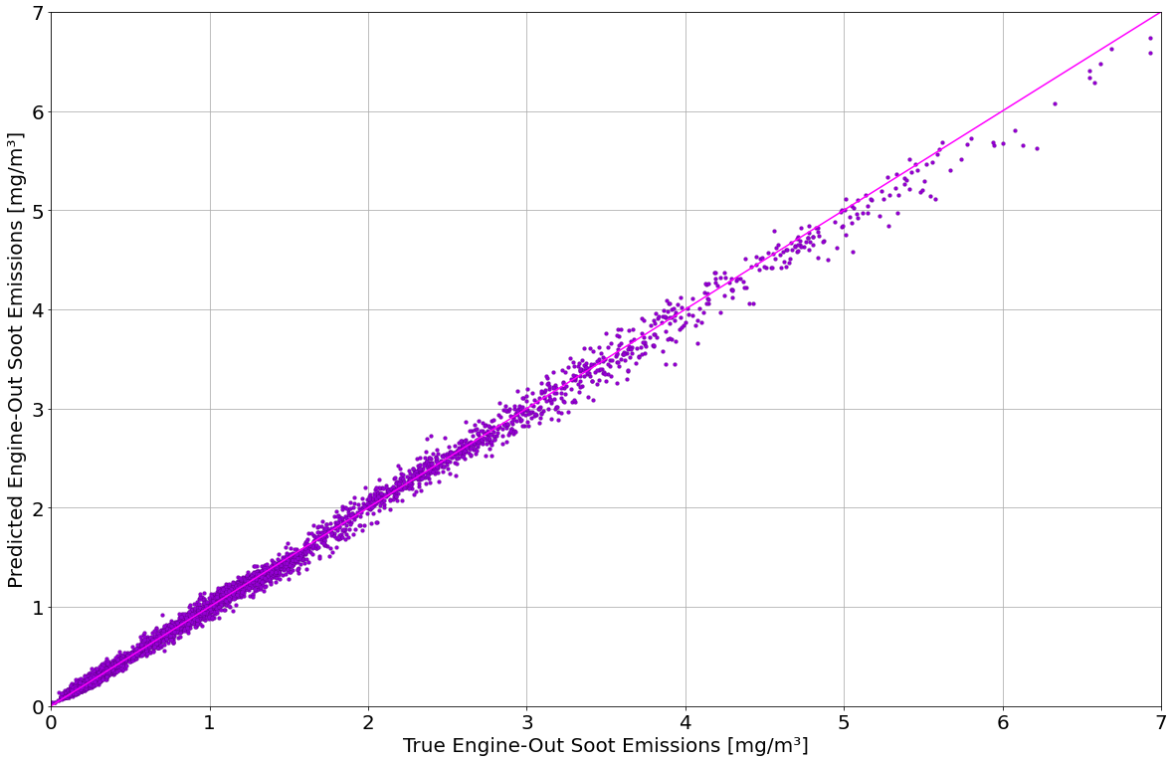

Figure 34. Correlation between actual and predicted emission values on the training set for the XGB model

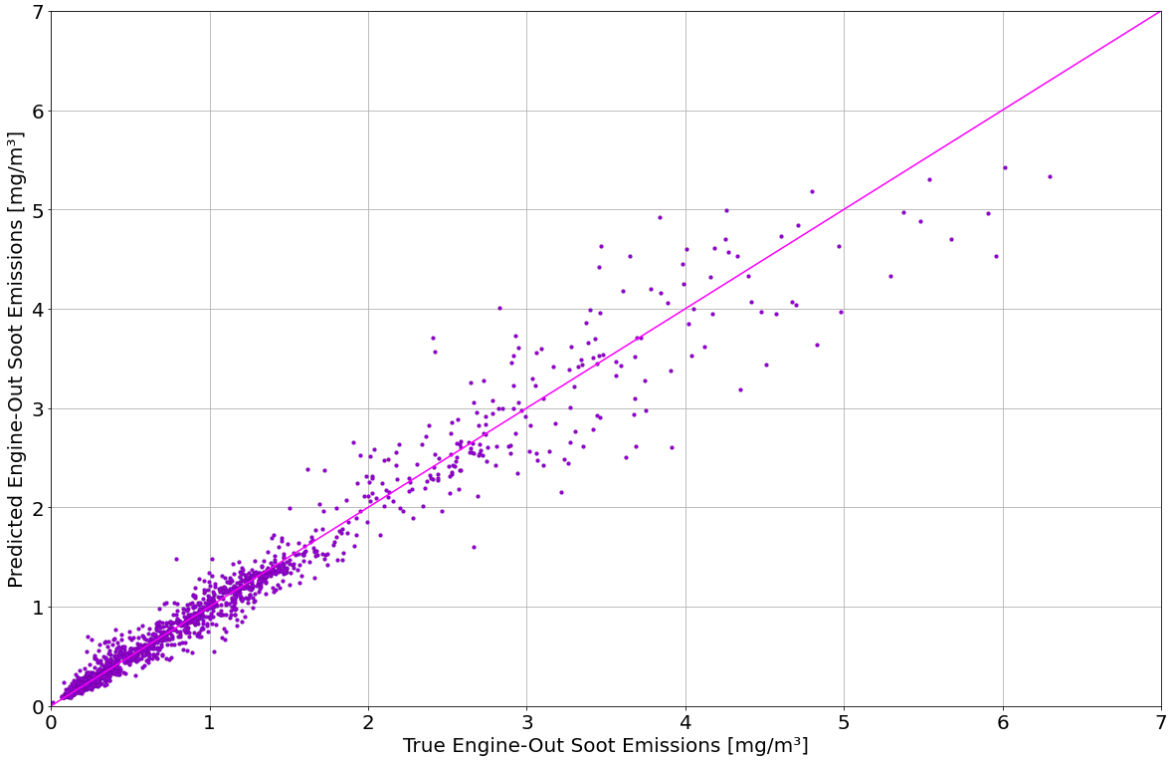

Figure 35. Correlation between actual and predicted emission values on the test set for the XGB model

As displayed in Figures 34-35, it is also interesting to plot the correlation between predicted and true (real) values of the label, for both the training and testing phases. In Figure 34 the very good correlation achieved by the model on the training set can be noted, it is also better than what has been reached by the SVR model.

Figure 35 gives us an interesting insight on the model predictive capabilities reporting the correlation between observed and predicted label values for the unseen testing data. As can be seen from the plot, the designed model confirms the good performance expressed by the evaluation metrics disclosed in the Table 14.

Often features do not contribute equally to predict the target response and in many situations most of the features are in fact irrelevant. A benefit of using gradient boosting is that after the boosted trees ensemble is constructed, it can provide estimates of feature importance from the trained predictive model. Individual decision trees intrinsically perform feature selection by selecting appropriate split points. This information can be used to measure each feature importance which provides a score that indicates how valuable each feature is in the construction of each boosted tree within the model. The more a predictor is used to make key splits in decision trees, the higher its relative importance. This importance is calculated explicitly for each feature in the dataset, allowing them to be ranked and compared to each other. Importance is calculated for a single tree by the amount that each attribute split point improves the performance metric, weighted by the number of observations the node handles. In other words, the relative rank of a feature used in a tree node can be used to assess the relative importance of that feature in predicting the target variable. Features used at the top of the tree contribute to the final prediction decision of a larger fraction of the input samples and thus, this fraction can be used as an estimate of the relative importance of the features. The performance measure may be the least squares error (loss) function used to select the split points. The fraction of samples a feature contributes to is combined with the decrease in error from splitting them to create a normalized estimate of the predictive power of that feature. This notion of importance can be extended to boosted tree ensembles by simply averaging across all the estimators the obtained feature importance values. We will indicate this type as “standard” feature importance.

The standard feature importance approach suffers from two flaws that can lead to misleading conclusions. Firstly, they are computed on statistics derived from the training dataset and therefore do not necessarily give us indications on which features are most important to make precise predictions on held-out dataset. Secondly, they favour high cardinality features (i.e., features with many unique values). Permutation feature importance is an alternative approach that does not suffer from these blunders.

The permutation feature importance can be considered as the decrease in the model score when the values of a single feature are randomly shuffled between different samples. In this way, the relationship between the shuffled feature and the labels is broken, thus the drop in the model score suggests how much the model relies on that feature. This technique can be computed many times with different permutations of the feature. Using a held-out (test) set to calculate permutation importance values, it is possible to spot which features contribute mostly to the generalization capabilities of the model. In other words, permutation importance states how important a particular feature is for the given model.

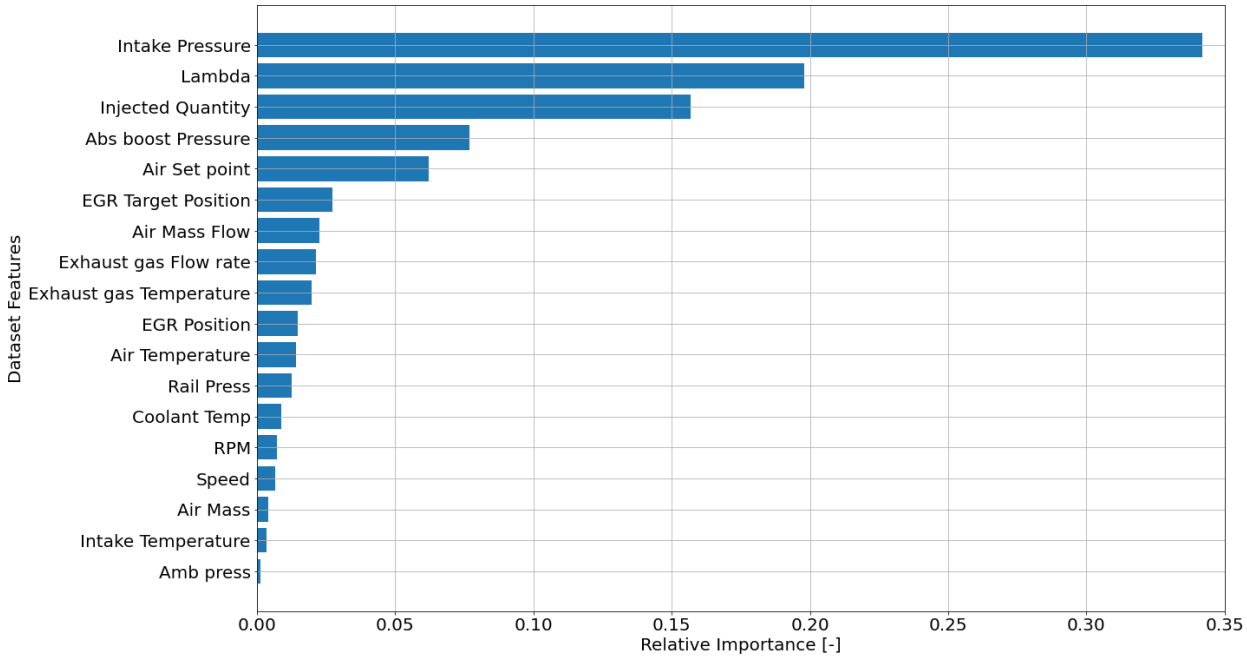

Figure 36. XGB model feature relative importances

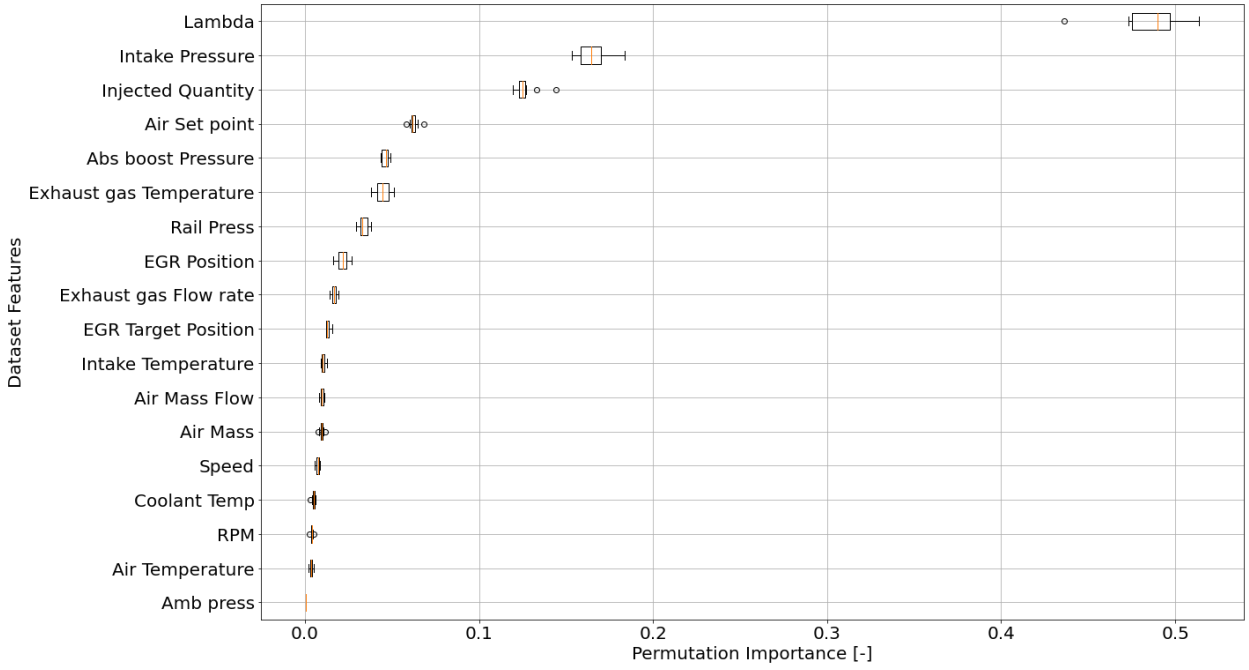

Figure 37. XGB model permutation importance of features

These two methods of obtaining feature importance are explored considering our model and the available dataset. The obtained results are disclosed respectively in Figure 36 and Figure 37. A quite good general matching can be noted from the two plots. However, it can be observed that few features, like 'Intake Pressure', 'EGR Target Position', 'Air Temperature', are considered by the model more important on the training set than on the held-out set and this may cause the model to overfit due to the reasons just mentioned. Despite the just mentioned impeding issue, the good performances of the model disclosed in the Table 14 reflect the previous-stated remark on the positive qualities of the algorithm used.

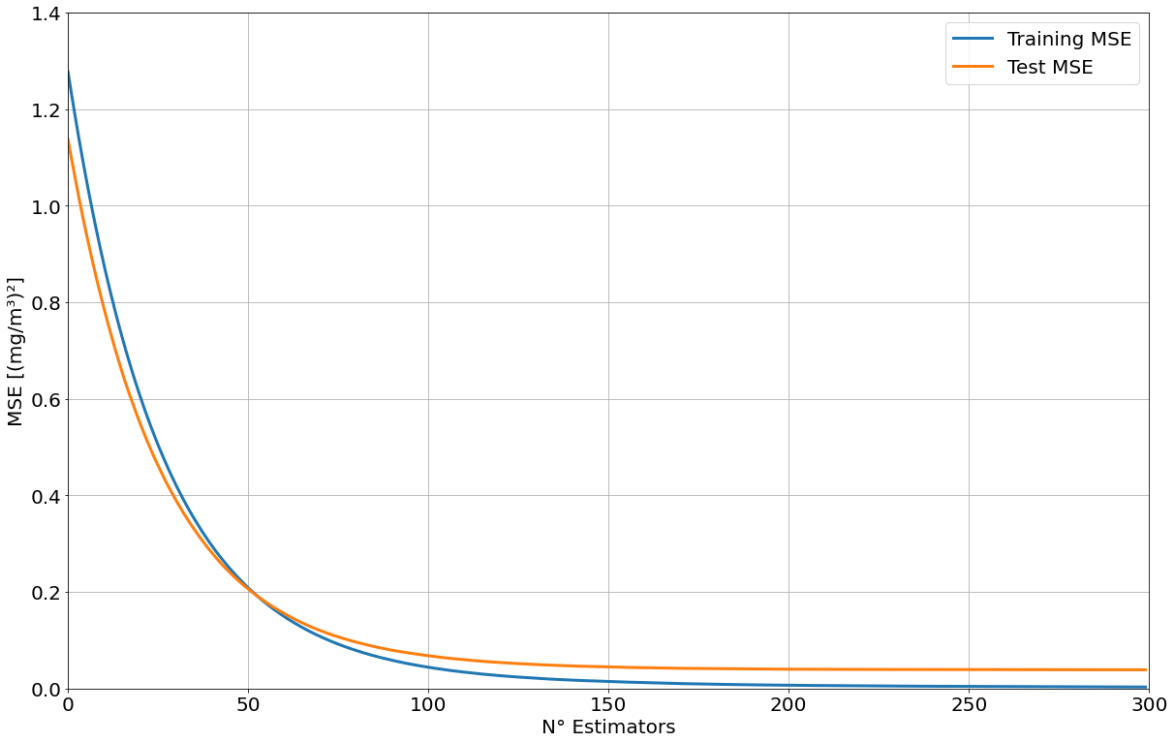

Figure 38. XGB model training and test learning curves by varying the number of estimators

In order to exhaustively describe the behaviour of the model, in Figure 38 the plot of the training and testing MSE curves versus the number of boosted trees (i.e., the number of boosting rounds) composing the ensemble model is reported. This type of plot is referred to as learning curves too, it is typically used for models based on algorithms that contemplate a number of iterations during learning, like gradient tree boosting and neural networks, as we will see in the following section. A quite natural and clean trend can be observed for both the training and test MSE curves meaning the model learning was not affected by overfitting or underfitting phenomena. The remarkably low error value achieved point up that the designed model is a good one and its hyperparameters have been correctly tuned.

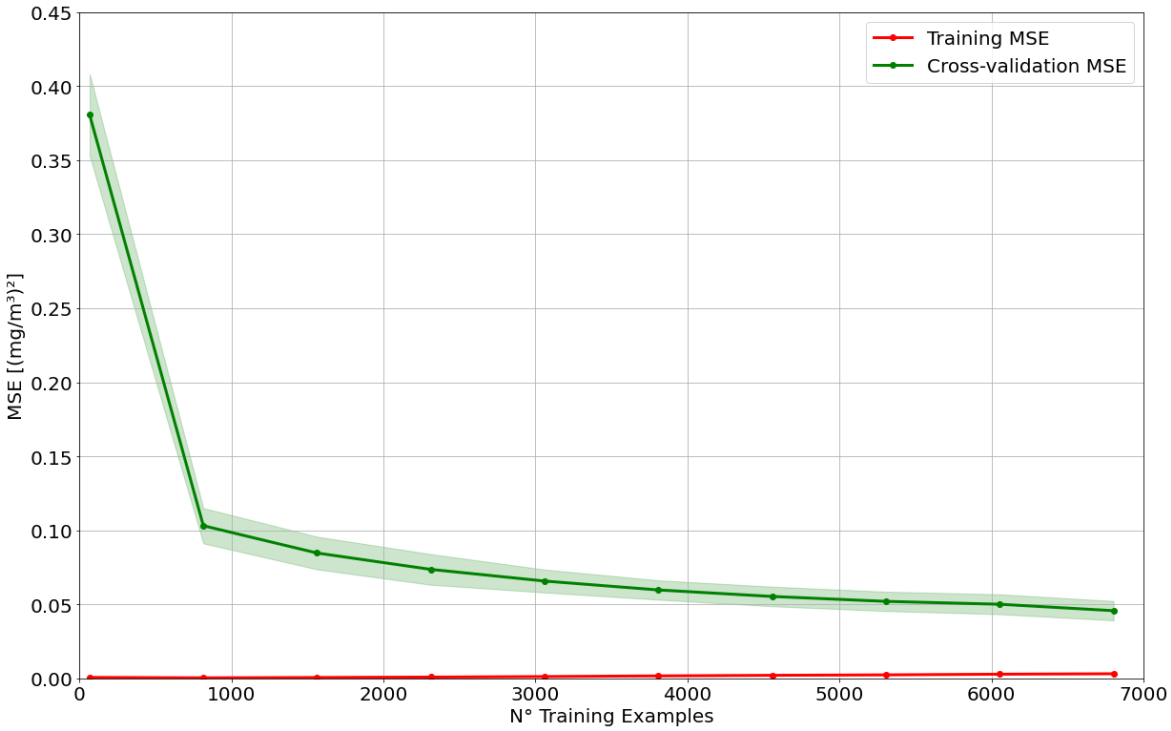

Figure 39. XGB model training and cross-validation learning curves (shaded area represents ± 1σ range) by varying the training set size

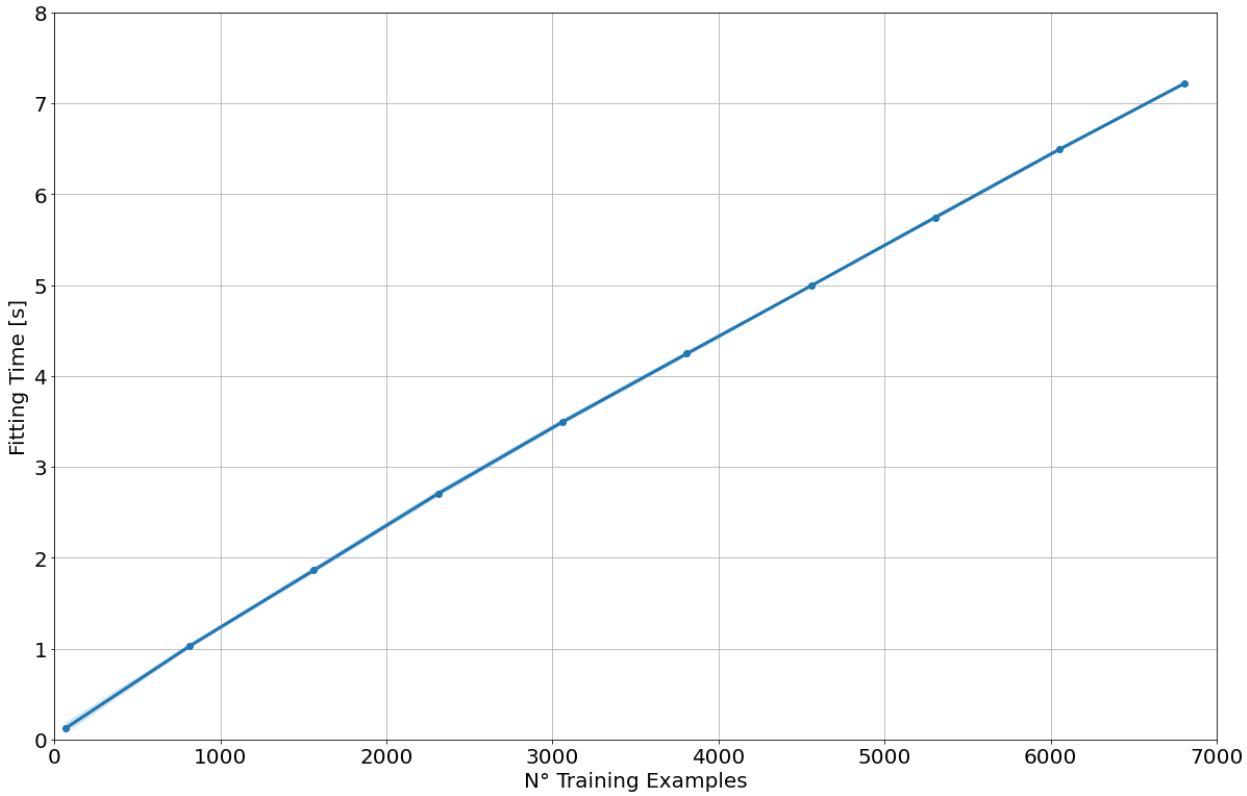

Figure 40. XGB model fitting time by varying the training set size (shaded area represents ± 1σ range)

As for the previous algorithms, we check that our model does not suffer from high bias or high variance problems and to this aim the relative conventional learning curves are plotted in Figure 39. The training MSE curve rises particularly slowly from almost zero as the number of training

samples grows and it reaches a very low error value. The validation MSE curve has a satisfactory shape, that is it decreases to a quite low error value as the number of training observations increases. The two previous remarks allow us to state that the model does not suffer from high bias. It seems that the validation MSE curve can further decrease and for this reason adding more training instances is likely to help to reduce the relative error. The resulting gap between the two curves can be considered fairly small and hence it suggests our model does not suffer from high variance.

Finally, in Figure 40 the fit time required to train the model is reported versus the dataset size used for training. In this case the fitting time scales almost linearly with the number of training samples employed. This behaviour can represent an advantage with respect to the SVR algorithm for larger training dataset sizes.

5.7 Neural Network Regression Model

In this part of the chapter, the model based on the Neural Network regression algorithm is built. Moreover, the results of the model training and hyperparameter tuning processes are reported together with model's behaviour analyses about learning and parameter variations. The chosen implementation is represented by the `Sequential` function provided by the Keras library, included in the TensorFlow package. This function is appropriate for building a model consisting of a plain series of layers where each of them has exactly one input tensor and one output tensor. In fact, it is the formulation based on the fully connected deep feedforward concept. The model built and described in this section is a development of the activities carried out by Gabriele Valvo in his thesis, upon which the work reported in this section is partly based. The main parameters to be set for this function are the following:

- Number of layers – It determines how many layers of neurons to be used. They are usually included in the model through an adding function. For our purpose only layers of the Dense type are used. This parameter is an integer number greater than or equal to one. Increasing the number of layers, the neural network becomes deeper and generally able to deal with more complex problems.
- Number of neurons for each layer – It specifies how many neurons compose each layer and it is a parameter that must be passed to the function used to add a new layer to the model. A different number of neurons can be set for each layer in the network but using the same number of neurons across all the layers is advisable and a good default choice. As for the network layers, increasing the number of neurons makes the network capable

to face more complex problems. It must be set to an integer number greater than or equal to 1.

- `activation` – It is the activation function applied to the outputs of the layer neurons, that are the weighted sums of their inputs. It is used to introduce nonlinearities into the network enabling it to solve nontrivial problems. This parameter can be set into the network model using a dedicated adding function or it can be specified by the function used to add a new layer to the network, as for the definition of the number of neurons. The proper choice of the activation function heavily depends on the type of predictive modeling problem (classification and regression) the network has to face and on the layer to which it is applied (typically, output layer requires a different activation function with respect to the other network layers). The most commonly applied activation functions are the rectified linear unit (`relu`), the softmax (`softmax`), the sigmoid (`sigmoid`) and the hyperbolic tangent (`tanh`).
- `kernel_initializer` – It defines, together with `bias_initializer` (which is less commonly specified, letting the model to use the default value of 1 for the bias terms), the way to set the initial weights associated to the neurons output of each layer: it defines the type of numeric distribution from which initial weights are randomly selected. This parameter can be set into the function used to add a new layer to the network, as in the case of the previous hyperparameters. As for the activation function, also the suitable choice of the weight initializer depends on the framing of the specific predictive modeling problem the network has to tackle.
- `loss` – It represent the loss function used to compute the quantity that the model should seek to minimize during the learning phase. In other word, to repeatedly estimate the current model error the optimization algorithm requires the definition of a loss function, that can be used to estimate the model error so that its internal weights can be updated to reduce the error on the following learning rounds. The chosen loss function must be compatible with the framing of the specific predictive modeling problem. Moreover, the output layer configuration should be suitable for the chosen loss function. For regression tasks, the most used metrics are MSE and MAE.
- `optimizer` – It is the algorithm used for the optimization of the model learning process, that is for updating the network weights. The most commonly used is `SGD` (gradient descent), other ones are `RMSprop`, `Adam`, `Adamax` and `Nadam`.
- `learning_rate` – It determines the amount of change to the model internal parameters during each round of the learning process, that is the step size of the optimization

algorithm. This hyperparameter controls the rate or speed at which the model learns. In other words, it controls the amount of error that is reduced as the model weights are updated. This parameter is set inside the definition of the model optimizer. It is a float and its value often ranges between 0 and 1.

- `metric` – It is the list of metrics to be evaluated by the model during learning on the training and validation datasets. A metric is a function used to assess the model performance. Metric functions are very similar to loss ones, except that the results from evaluating a metric are not used during model training. In fact, any loss function may be used as a metric. For regression tasks, the most used metrics are `mean_squared_error`, `root_mean_squared_error` and `mean_absolute_error`. Custom metrics can be specified by the user.
- `batch_size` – It represents the number of training observations the algorithm has to work through before updating the internal weights of the model. A batch can be thought as a for-loop iterating over one or more observations and making predictions. After that, the predictions are compared to the measured target values and an error is calculated. The latter is then fed into the optimization algorithm that tries to improve the model updating its internal parameters. The dataset to be used for training the network can be divided into one or more batches. In the following, different cases are reported that refer to the gradient descent algorithm: when all the observations in the training dataset are grouped into a single batch, the optimization algorithm is called batch gradient descent; in case each batch contains only one sample, it is referred to as stochastic gradient descent; if the batch size is more than one sample and less than the size of the training dataset, the so called mini-batch gradient descent is used. It must be an integer number greater than or equal to 1.
- `validation_split` – It is the fraction of the training dataset to be used as validation data. It will be set apart by the model which will not train on it. At the end of each epoch the loss and any selected metrics are evaluated by the model on this data to obtain the validation metrics. The validation data is selected from the last samples in the x and y data provided, before shuffling. It must be set to a float between 0 and 1. However, if you want to use a test set as the data on which to evaluate the model performance at the end of each epoch in terms of loss and metrics, the `validation_data` parameter should be set with a tuple reporting the test dataset (feature values and labels).
- `epochs` – It represents the number of times that the algorithm will work through the whole training dataset during the learning process. An epoch means that each available

training sample is used by the algorithm only once to update the model internal weights. An epoch consists of one or more batches depending on the value of the `batch_size` parameter. The number of epochs can be thought as the number of iterations a for-loop containing the learning algorithm executes over the training dataset. It must be an integer number greater than or equal to 1. The number of epochs is generally large (hundreds or thousands) in order to allow the learning process to proceed until the model error has been minimized up to a decent value or until overfitting symptoms start to appear.

- Dropout rate – It defines the frequency rate at which the algorithm sets inputs of layer neurons to zero at each training round. It allows to implement the dropout regularization method which helps prevent overfitting. This parameter is passed to a function that adds a dropout stage to the network. Its value must a float ranging between 0 and 1 and determines the fraction of input to the layer neurons are dropped to zero.

For the model built with the illustrated 'Sequential' function, we have selected the MSE loss function. It is a popular choice and widely used for regression problems and we have used it in the models developed in the previous sections. To obtain reproducible results within neural network models, the following lines of code have to be used in addition to the ones reported in section 5.3:

```
import tensorflow as tf
tf.random.set_seed(seed)
```

The chosen activation function for the hidden layers is the ReLU (rectified linear unit) because it makes models easier to train and allows to achieve better performance. It is commonly used for many types of neural network and it is a piecewise linear function that returns directly the input if it is a positive number, otherwise it outputs zero. Instead, for the output layer, a linear activation function is used to avoid limiting the output (predicted) values range.

Hyperparameter	***Explored Values***
N° hidden layers	1 - 2 - 3 - 4 - 5
N° neurons per layer	50 - 100 - 150 - 200 - 250
kernel_initializer	glorot_uniform - he_uniform - glorot_normal - he_normal
optimizer	Adam - RMSprop - Adamax - Nadam
learning_rate	0.0005 - 0.001 - 0.005 - 0.01
epochs	50 - 100 - 150 - 200 - 300 - 500
batch_size	64 - 125 - 250 - 500
Dropout rate	0 - 0.1 - 0.2 - 0.3 - 0.4 - 0.5 - 0.6
Dropout rate (output layer)	0 - 0.1 - 0.2 - 0.3 - 0.4 - 0.5 - 0.6

Table 15. Grid-searched hyperparameters values for the Neural Network model

To find the optimal values for the above disclosed hyperparameters, a grid search is carried out using the `GridSearchCV` function already described. The estimator parameter is set to the model built with the `Sequential` function and the candidate values explored for each hyperparameter are displayed in Table 15.

In Table 16 are reported the results of the procedure, that is the combination of the hyperparameter values that lead to the best cross-validated performance (in terms of mean squared error reduction).

Hyperparameter	***Best Value***
N° hidden layers	5
N° neurons per layer	200
activation	relu
activation (output layer)	linear
kernel_initializer	he_uniform
optimizer	Adamax
learning_rate	0.005
epochs	200
batch_size	125
Dropout rate	0
Dropout rate (output layer)	0.1

Table 16. Best combination of hyperparameters values for the Neural Network model

Finally, the model is retrained on the whole training set using the values just found as hyperparameters and then its performances are evaluated on the test set. The findings are disclosed in Table 17, where evaluation metrics are reported for both the training and testing phases. Here, the same considerations about training metrics exposed in the previous section hold here.

Evaluation Metric	***Value***
Test R^2	0.939
Training R^2	0.958
Test MSE	0.0583
Training MSE	0.0447
Test RMSE	0.2414
Training RMSE	0.2113

Table 17. Neural Network model performance evaluation metrics

As displayed in Figures 41-42, it is also interesting to plot the correlation between predicted and true (real) values of the label, for both the training and testing phases. In Figure 41 it is possible to spot that the correlation achieved by the model on the training set is fairly good.

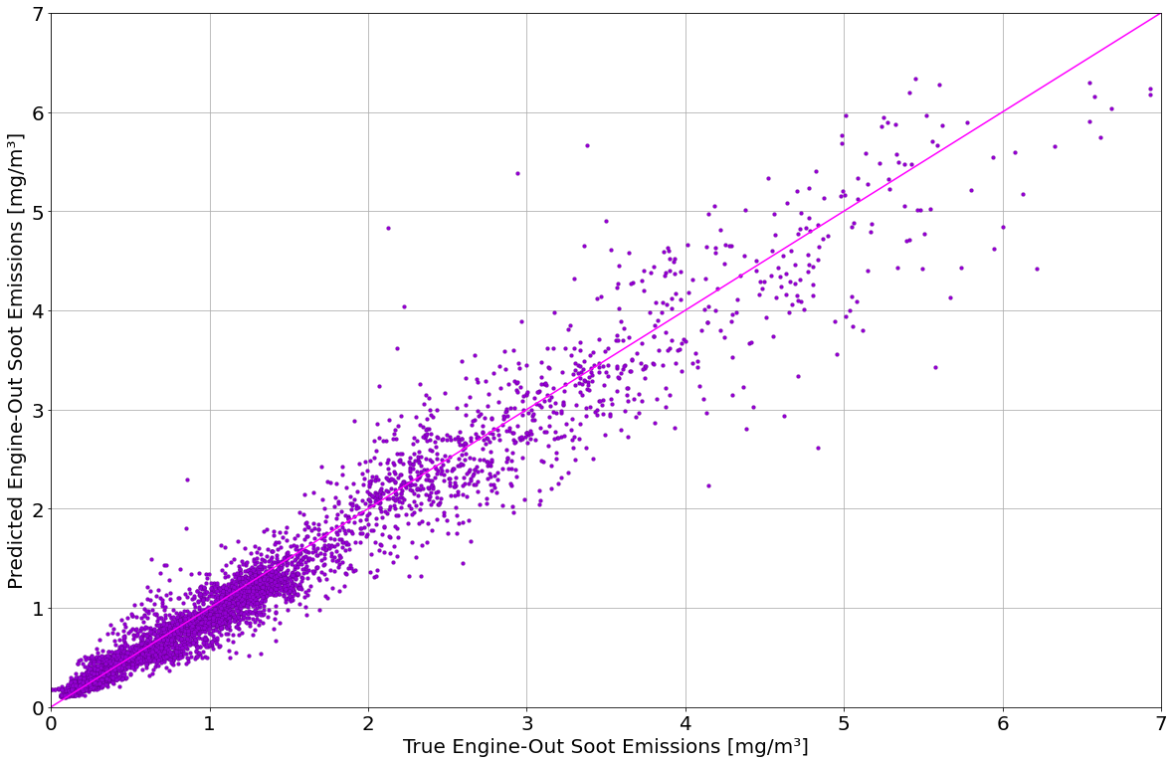

Figure 41. Correlation between actual and predicted emission values on the training set for the Neural Network model

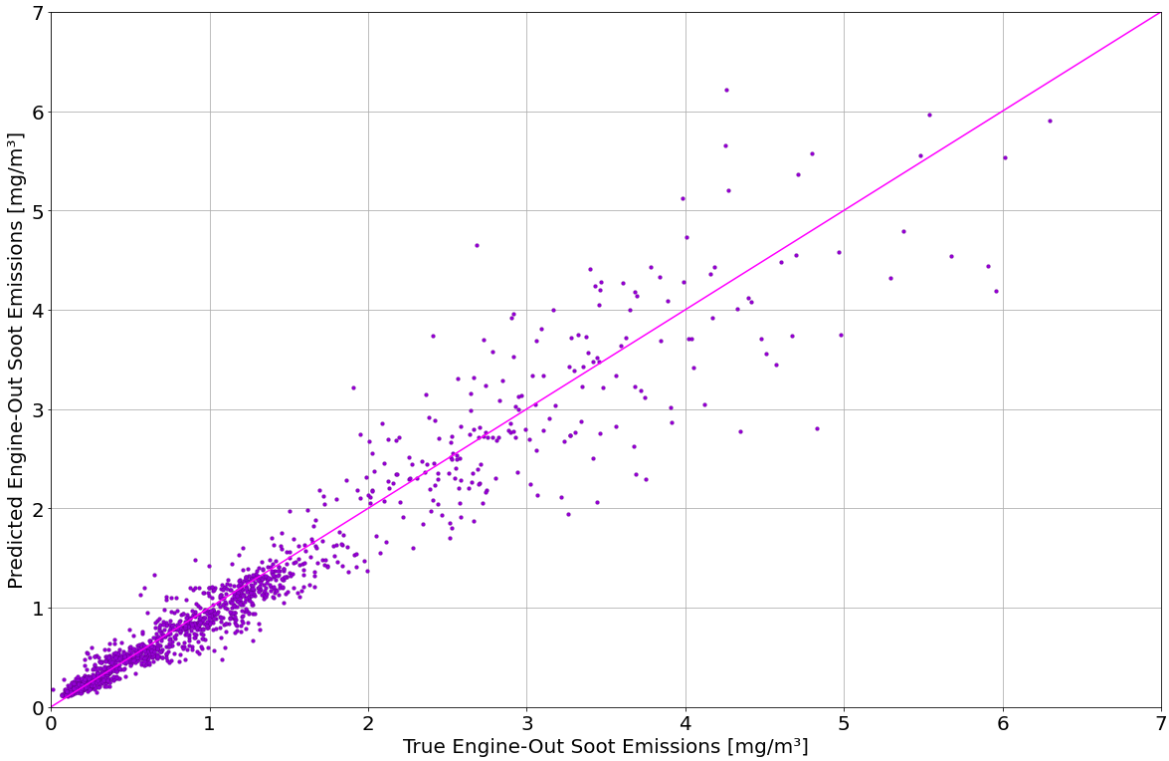

Figure 42. Correlation between actual and predicted emission values on the test set for the Neural Network model

Figure 42 gives us information about the model predictive capabilities reporting the correlation between observed and predicted label values for the unseen testing data. As can be seen, the figure confirms the decent performance expressed on the training set, although it does not reach the optimal performance obtained for some of the models previously disclosed.

In order to exhaustively describe the behaviour of the model, in Figure 43 the plot of the training and testing MSE curves versus the number of epochs is reported. As we have already seen in the previous section, this type of plot is referred to as learning curves and it represents a common tool to check the learning process of neural networks.

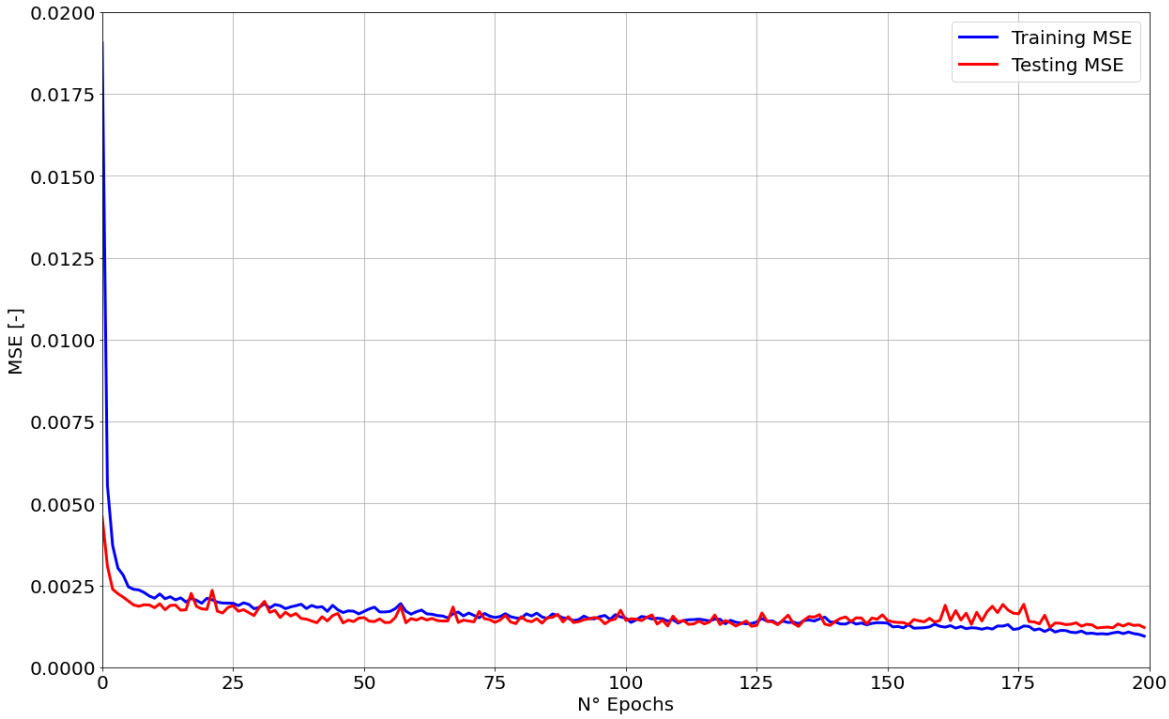

Figure 43. Neural Network model training and test learning curves (normalized MSE) versus the number of epochs

A little bit of noise can be noted, especially in test MSE curves, but the trend of both errors indicates that the model learning was not affected by overfitting or underfitting phenomena. It can be noted that the convergence has been reached, in particular on the test error. The fairly low error value achieved point up that the designed model is a good one and its hyperparameters have been correctly tuned.

Conclusions

This thesis work has dealt with the design, development and validation of models based on Machine Learning and Deep Learning algorithms, for the estimation of the given CI engine soot emissions in quasi-stationary conditions. All the implemented models have achieved very good performances and their robustness, along with their correct learning process, has been verified through the analyses carried out. The XGBoost model turned out to be the most performing one, even if it was not perfectly optimized. In fact, given the large number of available hyperparameters and the relative range of possible values, it was possible to carry out the grid search only on a limited feature space. On the other hand, the optimization process of the models based on the SVM and KNN algorithms has been carried out more deeply, given the small number of hyperparameters to be set, some of which are categorical and, therefore, with a discrete (and small) number of possible values.

For what concerns the neural network model, the same problem has been experienced: the grid search carried out for its optimization could not explore the entire range of possible values for the hyperparameters, given their huge number. Conversely, the neural network performances cannot be considered equal to those shown by the XGBoost model. It sounds quite strange because neural networks algorithm is a very powerful and complex one and from the theoretical point of view it should perform better than most of ML algorithms. However, the latter ones may not be underestimated since they are very popular, due to their simplicity and usability, and have proved capable of solving even complex problems in different fields of application (for example, XGBoost is the most used algorithm among many winners of the most important AI and ML competitions).

In any case, it can be noted that the performances of the various models are very similar in terms of evaluation metrics. Therefore, it can be observed that a sort of limit on the minimum obtainable error (and therefore on the maximum achievable score) has been reached and that the remaining one is the so-called irreducible error, which is due to the noise in the data used to train and validate the models.

It was decided not to apply the feature selection process for the following reasons. Firstly, another algorithm should be used to carry it out, with the consequent need to train and optimize the algorithm: this fact implies the request of additional resources and its performance could affect positively or negatively the model used to carry out the main task. Secondly, the objective of the thesis is to analyse and compare the employed models performances on a dataset containing also not relevant features for the examined problem and to evaluate their consequent behaviour.

The results obtained are certainly a good starting point for the development of a stable and reliable virtual sensor, capable of replacing real sensors present on cars. The future improvements that can lead to an evolution of the studied models are the following:

• the use of larger datasets with additional parameters that can characterize more accurately the problem

• the application of a grid search exploring more combinations of hyperparameters values in order to identify better model configurations.

Finally, other future steps may consist in the use of datasets containing pollutant emissions data, recorded in transient conditions, for the design and development of predictive models to be implemented in mass-production cars.

Bibliography

1. Albon C., *Machine Learning With Python Cookbook: Practical Solutions from Preprocessing to Deep Learning*, O'Reilly Media, 2018

2. Bishop C. M., *Pattern Recognition and Machine Learning*, Springer-Nature, 2006

3. Brownlee J., *XGBoost With Python: Gradient Boosted Trees With XGBoost and scikit-learn*, Machine Learning Mastery, 2018

4. Chollet F., *Deep Learning with Python*, Manning Publications, 2017

5. Dec J. E., *A Conceptual Model of DI Diesel Combustion Based on Laser-Sheet Imaging*, SAE paper 970873, SAE Trans., vol. 106, pp.1319–1348, 1997

6. Falai A., *Applicazione e validazione di un modello Random Forest per la stima della massa di particolato in motori Diesel*, Tesi di Laurea Magistrale, Politecnico di Torino, a.a.2018-19

7. Flynn P. F., Durrett R. P., Hunter G. L., zur Loye A.O., Akinyemi O.C., Dec J.E., Westbrook C. K., *Diesel Combustion: An Integrated View Combining Laser Diagnostics, Chemical Kinetics, and Empirical Validation*, SAE paper 1999-01-0509, SAE Trans., vol. 108, pp. 587−600, 1999

8. Géron A., *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media, 2019

9. Goodfellow I., Bengio Y., Courville A., *Deep Learning*, MIT Press, 2017

10. Heywood J. B., *Internal Combustion Engine Fundamentals*, McGraw-Hill Education, 2018

11. Huang T., Kecman V., Kopriva I., *Kernel Based Algorithms for Mining Huge Data Sets: Supervised, Semi-supervised, and Unsupervised Learning*, Springer, 2010

12. James G., Witten D., Hastie T., Tibshirani R., *An Introduction to Statistical Learning with Applications in R*, Springer Verlag, 2013

13. Kamimoto T., Bae B., *High Combustion Temperature for the Reduction of Particulate in Diesel Engines*, SAE Technical Paper 880423, 1988

14. Mariq M., *Chemical characterization of particulate emissions from diesel engines: a review*, Journal of aerosol science 38, pp. 1079-1118, 2007

15. Millo F., *Propulsori Termici (Appunti del Corso)*, Politecnico di Torino, a.a.2018-19

16. Smola A. J., Schölkopf B., *A tutorial on support vector regression*, Statistics and Computing 14: 199–222, 2004

17. Valvo G., *Sviluppo di un sensore virtuale per la stima della concentrazione di soot allo scarico di motori diesel in ottica OBD*, Tesi di Laurea Magistrale, Politecnico di Torino

Sitography

18. Scikit-learn Python library website: https://scikit-learn.org/stable/index.html

19. Keras Python library website: https://keras.io/api/

20. Machine Learning Mastery blog: https://machinelearningmastery.com/blog/

21. Dataquest blog: https://www.dataquest.io/blog/

22. Neptune blog: https://neptune.ai/blog/

23. Data Science Learner: https://www.datasciencelearner.com/

24. MIT Deep Learning 6.S191 course: http://introtodeeplearning.com/

25. Stanford Coursera Machine Learning course: https://www.coursera.org/learn/machine-learning