Universitá degli Studi di Padova

Mattia Marcello Longato

Support an S-duct optimization design study using state-of-the-art Machine Learning techniques

Dipartimento di Ingegneria Industriale

MSc in Aerospace Engineering

MSc Thesis

Academic Year: 2018-2019

Supervisors:

Prof. Ernesto Benini

Dr. Timoleon Kipouros

April 2020

MSc in Aerospace Engineering

MSc Thesis

Academic Year: 2018-2019

Mattia Marcello Longato

Support an S-duct optimization design study using state-of-the-art Machine Learning techniques

Supervisors:

Prof. Ernesto Benini

Dr. Timoleon Kipouros

April 2020

UNIVERSITÁ DEGLI STUDI DI PADOVA

Abstract

Dipartimento di Ingegneria Industriale

MSc in Aerospace Engineering

Support an S-duct optimization design study using state-of-the-art Machine Learning techniques

by Mattia Marcello Longato

With the advent of CFD tools, the computer-aided flow analysis has become a trustful and cheap alternative to wind tunnel experiments, however, much time is required by the code to be run even the increasing computational power available in our days.
Alternately to this calculation method, different state-of-the-art machine learning techniques are explained and implemented in the following work. The purpose is to build confidential ML models learning from given input data about a specific problem allowing the computer to predict solutions with the smallest error. Deep learning methods are applied like the Extreme Gradient Boosting (XGBoost) to make predictions about new aero-engine intakes S-duct geometries.
Several optimization problems have been executed using a Multi-Objective optimization algorithm for the design space investigation followed by the specific trained machine learning model. The results obtained running the ML optimizations are shown with the corresponding parallel coordinate geometry plot.
The best results are later evaluated using some CFD simulations to confirm and validate the machine learning models built before. The CFD evaluations will be done only for the pareto front points using firstly Ansys Fluent, which is a RANS model and secondly implementing the Lattice Boltzmann Method, which is a LES simulation.

UNIVERSITÁ DEGLI STUDI DI PADOVA

Sommario

Dipartimento di Ingegneria Industriale

MSc in Aerospace Engineering

Support an S-duct optimization design study using state-of-the-art Machine Learning techniques

by Mattia Marcello Longato

Con l'avvento delle nuove tecniche di simulazione CFD oggigiorno si è in grado di risparmiare una considerevole quantità di tempo e denaro rispetto alle vecchie tecniche di simulazione basate esclusivamente su esperimenti in galleria del vento. Il progredire della tecnologia consente di avere a disposizione strumenti sempre più performanti, come nel caso delle tecniche di machine learning implementate in questo elaborato di tesi. Questi codici innovativi consentono alle macchine di apprendere e classificare seguendo specifici criteri l'informazione in elaborazione. Lo scopo di questa tesi è quello di utilizzare lo stato dell'arte dei sopracitati codici di machine learning con lo scopo di analizzare le caratteristiche aerodinamiche di un condotto di prediffusione applicato in aeronautica. Il condotto in questione è un S-duct introdotta nell'esperimento di Wellborn nel 1993 dove le variabili da vagliare corrispondono al coefficiente di recupero di pressione (Cp) fra ingesso ed uscita assieme al coefficiente di distorsione del flusso denominato angolo di swirl. Diverse ottimizzazioni di questo condotto sono state messe a punto sfruttando l'algoritmo chiamato Multi-Objective Tabu Surch per la ricerca del design space seguito dall'implementazione della specifica tecnica di machine learning. I risultati cosi ottenuti sono stati verificati attraverso delle analisi CFD per convalidare i codici di ML implementati precedentemente. Le classiche tecniche di Spalart-Allmaras, conosciute meglio come RANS sono state condotte a termine in un primo momento per poi proseguire la trattazione sfruttando un modello LES il quale implementa l'innovativo metodo CFD delle Lattice Boltzmann.

Acknowledgements

I can not express enough thanks to my committee for their continued support and encouragement: Dr Timoleon Kipouros and Professor Ernesto Benini. I offer my sincere appreciation for the learning opportunities provided by my committee, and I want to thank again everybody who made this fantastic experience in Cranfield possible.

The completion of this project could not have been accomplished without the support of my classmates, Andrea, Luca, Annalisa, and Alberto. Thank you all for helping me with my research. Thanks also to my parents, Enzo and Rosanna. A big hug also goes to my brother Simone and my cousin Laura including my uncles Fabio and Antonella.

Contents

List of Figures

List of Tables

Abbreviations

CDF	**C**umulative **D**istribution **F**unction
DP	**D**esign **P**oint
MOTS	**M**ulti-**O**bjective **T**abu **S**earch
ODE	**O**rdinary **D**ifferential **E**quation
PDF	**P**robability **D**ensity **F**unction
RANS	**R**eynolds-**A**veraged **N**avier-**S**tokes equations
LES	**L**arge **E**ddy **S**imulation
ML	**M**achine **L**earning
Cp	**P**ressure **R**ecovery
Sw	**S**wirl **A**ngle
AI	**A**rtificial **I**ntelligence

Symbols

Cp	Profile pressure coefficient
Sw	Swirl number
D	Drag
L	Lift
M_∞	Free stream Mach number
P	Pressure
R	Universal gas constant
Re	Reynolds number
V_∞	Free-stream velocity
X	General second order random process
A	Area
D	Diameter
dt	Time step
R	Radius
AIP	Aerodynamic Interface Plane
CFD	Computation Fluid Dynamics
$Dc(\theta)$	Distortion coefficient for a sector of θ deg
LBM	Lattice Boltzmann Method
LES	Large Eddy Simulation
$RANS$	Reynolds Averaged Navier Stokes
$MOTS$	Multi objective Tabu search
ML	Machine learning
f_i	function of x

γ	ratio
δ_{ij}	delta step
μ	mean value
ρ	density
σ	standard deviation
ξ	gaussian standard variable

In this MSc thesis, a machine learning code is developed and presented to investigate an aerodynamic S-duct intake problem. Five-month were needed to develop the work, which was followed by Dr. Timoleon Kipouros at Cranfield University (UK). The topics studied are all focusing on the unsteady flow distortion inside an S-duct aero-engine intake, which is the duct that connects the external ambient flow to the fan inlet (AIP). The previous studies were done by D'Ambros [1], Delot [2], and Wellborn [3]; they had the goal to find out a duct with an **S** shape that allowed to increase the static pressure for the fan. Further studies developed by Tridello [4], Rigobello [5] and DalMagro [6] were centred on optimizing the duct shape to improve the performance, minimizing the loss.

The aim of this work consists of collecting the results obtained from the above studies and generate a model using state-of-the-art machine learning techniques to calculate and compare the aerodynamic coefficients we want to study. Until now, the S-duct shape has always been analyzed using Computational Fluid Dynamics programs, which are the only way that engineers have to solve fluid dynamics models. As we know, the CFD approach takes a long time to be run because we need to generate a new mesh every time and reinitialize the solver on the new geometry implemented. Machine learning techniques allow computers to learn from experience and understand the influence of complex variables. We can use the S-Duct problem we want to analyze as an example; we should now think in terms of a hierarchy of concepts, where each idea is defined through some relations to more straightforward concepts.

Explaining this idea, we can say that: by gathering knowledge from experience (previous studies), a machine learning approach avoids the need for human operators to specify all the knowledge that computer needs formally. The hierarchy of concepts enables the computer to learn complicated concepts by building them out from simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to artificial intelligence deep learning.

In the first chapter, the state-of-the-art about the S-duct will be explained together with previous studies about the geometry shape that has been drawn in the last works. Going through the chapter, the aerodynamics variables like the pressure recovery and the swirl angle will be illustrated together with the Gauss distribution introduced from a statistic point of view. The further explanations are regarding the CFD results coming out from the previous approach that has all been analyzed and classified to build the machine learning models as robust as possible. Different types of machine learning configurations will be used and developed in the further chapters through the thesis. The purpose is to find out which is the model that fits in the best way possible the data. This allows us to predict very fast and confidentially, the coefficients we want to study instead of using the CFD approach.

In chapters two and three, the machine learning technique will be explained exhaustively going through some algorithms. The introduction of some algorithms examples like the logistic regression algorithm can help to understand the way the computer used to think about problems given in input. Other parameters explained in these two chapters are also regarding another machine learning algorithm called Naive Bayes that can, for example, separate legitimate e-mail from spam e-mail. The main algorithm parameters will be illustrated with some problems afforded by the mathematical developers of the machine learning software. In chapter number three, the machine learning algorithm used in this S-duct optimization called XGBoost will be explained in-depth, showing which is the way the code interprets the data and organizes them by building different decision trees.

Going through chapter four, we have the explication about all the machine learning XGBoost models built, going through how each one works, and how the setup was made. Five different ML models are generated, each of which with a different configuration. The models learning and testing processes explained in this chapter are all done using the same CFD database **.txt** containing the S-duct simulations results given by DalMagro's previous work. [6] Comparisons between how each model fits the data will be made looking for the best one that has to minimize the mean squared error between the data learned and the predicted number.

Chapter number five will show the optimization loop results done with each model built before. An introduction about the Multi-objective Tabu search is done to explain how this optimizer works to find out the optimal Pareto front, moving throw the S-duct design space specified. Two optimization loops are made with every machine learning model to minimize the pressure recovery coefficient and minimize the swirl angle of the duct at the AIP. Every optimization has 500 iterations, so we have 1000 results for every ML method implemented, and in each case, the Pareto front will be plotted. The design space analyzed by the Multi-objective Tabu search (MOTS) gives us the S-duct geometries which are evaluated by the ML in the optimization loop, predicting the Cp and the swirl values. The next step shown in chapter five consists of assessing only the ML Pareto front geometries, with a CFD simulation and plot the results on the same graphic; this allows us to validate our ML predicted points.

Chapter number six goes through a different CFD approach called **Lattice Boltzmann method** (LBM) which is a **Large Eddy Simulation** (LES). This new method consists of build-up fictive particles and such particles perform consecutive propagation and collision processes over a discrete lattice mesh. The difference between the traditional CFD method consists in the numerical solution method because the traditional CFD code integrates the conservation equations of the macroscopic fluid properties numerically.
The idea is to study with **LBM** the Pareto front geometries that had the best results between the Cp and the swirl. Because time, only the first Wellborn [3] S-duct configuration will be analyzed. This configuration will be used as an example of lattice mesh implementation, showing the first 4.2285e-3s of the LES solution reported in chapter six. Furthers works should pick up this point and continue with the Lattice Boltzmann simulations on the rest of the S-duct geometries.

Chapter 1

S-duct

1.1 S-duct state-of-the-art

The aero intake duct is a turbomachinery component that allows the static pressure to increase before reaching the fan inlet plane called AIP. In nowadays, many types of ducts intakes are built and study, like the one we are interested in, the S-duct shape. Starting from a review by the Wellborn experiment [3] in 1993 and going through the further CFD analysis conducted by Delot [2], which both have been the milestones for this topic, we will introduce the S-duct shape bellow.

1.1.1 Wellborn experiment

The experimental investigation performed by Wellborn had the goal of providing a comprehensive benchmark dataset for the compressible flow through a representative diffusing S-duct. The details of the flow separation region and the mechanisms which drive this complicated flow phenomenon were both deeply investigated and reported in [3]. The facility used by Wellborn is shown in figure 1.1. Three main parts characterize the experiment: the settling chamber, the test section and the exhaust region. In the settling chamber, we can find a perforated spreader cone that mixes the flow coming in.

Going through the duct, the flow finds a coarse mesh conditioning screen that reduces the non-uniformities. At this point the flow finds a honeycomb-screen, that removes the large scale turbulence fluctuations, a reduction of the area follows that.

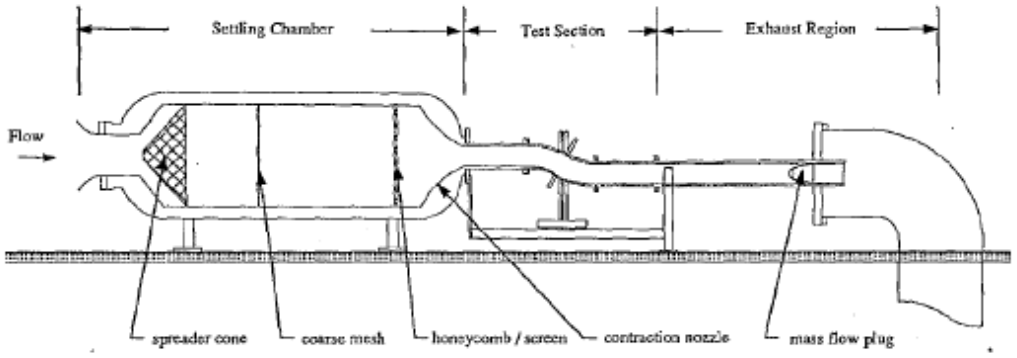

FIGURE 1.1: Wellborn facility [3].

The second part is the test section. It can be divided into three parts: the S-duct, and two additional components with a constant area, that are located before and after the S-shape.
The last section called the exhaust region is composed of a circular pipe, a mass flow plug and a sub-atmospheric plenum. The purpose of this component is to delete the influences of the exhaust plenum on the test section.

Figure 1.2 illustrates the S-duct geometry, which has circulars cross-sections; everyone is normal to the common centerline. This centerline is built by a function of two planar circles with have the same geometrical shape, the duct parameters are written in Table 1.1.

TABLE 1.1: S-duct domain dimensions

parameters	**values**
R	102.1 [cm]
$\frac{\theta_{max}}{2}$	30 [°]
Inlet radius (r1)	10.21 [cm]
Exit radius (r2)	12.57 [cm]
$\frac{A_2}{A_1}$	1.52

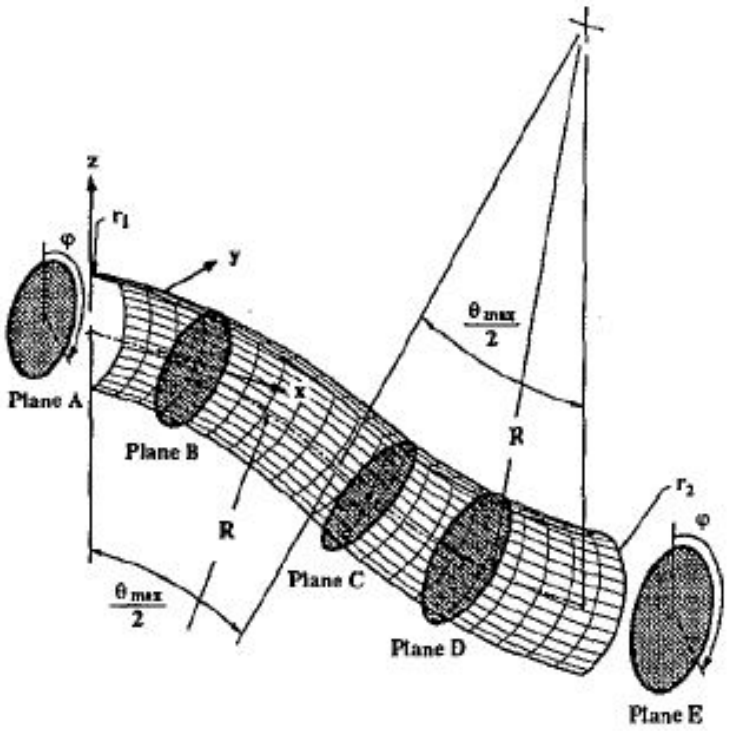

FIGURE 1.2: Wellborn S-duct geometry [3]

All the data were computed on five planes, using 220 static pressure taps spread on the duct surface. This experiment has discovered several behaviours of the flow; the presence of a separated region in correspondence to the first band, that develop in vortexes in the symmetry plane and two counter-rotating vortices at the AIP surface. These counter-rotating vortices in the flow create several total pressure losses. In Figure 1.3, it's evident how the boundary layer detaches from the duct walls. Very clear is shown in Plane E, the two vortices converge the fluid with the low momentum towards the duct centre. From these results, it is easy to conclude, highlighting the drop in pressure and, as a consequence, the velocity magnitude changes.

1.1.2 CFD analysis

In the following section, the CFD analysis that deeply examinated the S-duct with Wellborn's geometry is illustrated. There are a series of researches made by: Enrico Manca [7], Marco Barison [8], Aurora Rigobello [5], Riccardo Tridello [4] and Alessio D'Ambros [1]. The most important and recent work is the one performed by Alessio D' Ambros, precisely, his research is based on the optimization of the Wellborn's geometry considering two objective functions: the pressure losses and the swirl angle.
It is important to remind how this author proceeded: the geometry management has been controlled with the Free-Form Deformation (FFD) technique, whereas the analysis of the flow has been performed using the steady-state computational fluid dynamics (CFD).
Furthermore, the design space exploration has been achieved using the heuristic optimization algorithm Tabu Search (MOTS) [**?**]. Davide DalMagro [6] implemented the same technique, where using the same Multi-Objective Tabu Search algorithm he went through new design spaces founding new geometries and new results for the pressure recovery and the swirl angle.

The data coming from DalMagro [6] CFD optimization are the ones that will be used in chapter number four to set up the ML models and through all the train and test process.

1.2 Objectives functions Cp, Swirl

The CFD simulations done using the optimization code by DalMagro [6] investigates the physics inside the S-Duct giving as output from the loop the pressure recovery coefficient and the swirl angle of the specific geometry studied. The model inputs in the optimization loop were thirty-six numbers corresponding to the geometry control points coordinates given by the Free-Form Deformation (FFD) technique used to control the S-duct shape.

1.2.1 Input ML optimization function

- **Model input** Thirty-six numerical parameters are representing the geometry control points from the Free-Form Deformation algorithm. These parameters will be used to know the duct shape we are working with. The machine learning model built should be able to recognize and interpret these 36 geometry variables and predict the pressure recovery together with the swirl angle.

1.2.2 Outputs ML optimization function

- **Model output** As we explained before, in the S-duct aero intake, we have total pressure losses caused by the S bending, which creates some flow separations regions.

  These losses are numerically associated with the total pressure value through the duct. The total pressure parameter in a generic flow region can be defined as equation 1.1.

$$P_{tot} = P_{static} + \frac{1}{2} * \rho * v^2 \tag{1.1}$$

  Using the pressure recovery (PR) parameter, defined by equation 1.2, it is easy to know the losses amount in the duct.

$$PR = \frac{P_{Tot,Out}}{P_{Tot,In}} \tag{1.2}$$

  In figure 1.3, the flow separation region is visible after the first bend of the duct shape. Where this event happens, the total pressure value decreases.

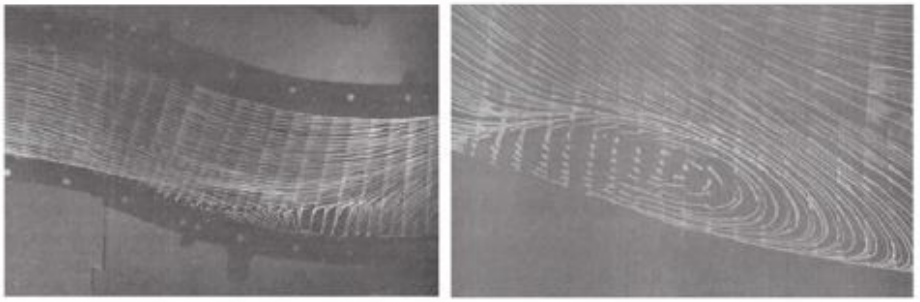

FIGURE 1.3: Vortex in the S-duct through the bending S shape.

- **Model output** The second parameter used to investigate the duct is the Swirl number. The swirl determines the flow distortion inside the duct. As we explained before, considering the geometry with cylindrical coordinates, it's defined by equation 1.3.

$$\alpha = \arctan \frac{V_{\theta,AIP}}{V_{z,AIP}} \tag{1.3}$$

In figure 1.4 are reported different Swirl types considering the flow distortion in the duct.

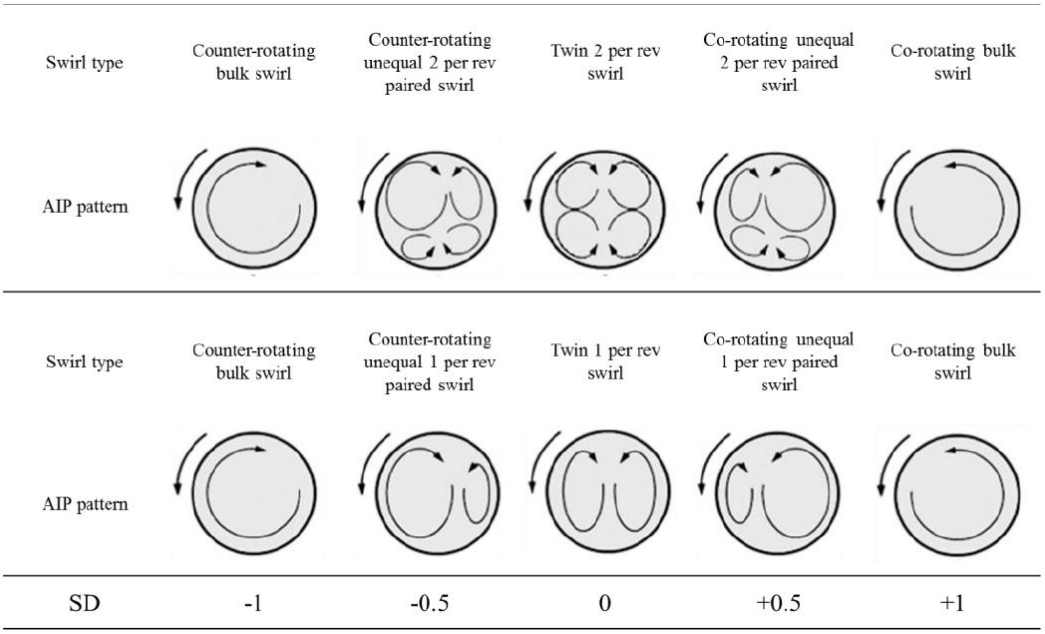

FIGURE 1.4: Swirl directivity (SD) spectrum for multiple pre-revolution swirl distortion.

1.3 Statistic Gauss distribution

In this section, some statistics concepts will be introduced because the Gaussian distribution of the two variables investigated by DalMagro [6] in the duct is calculated. The variables investigated are the Cp and the swirl, both analyzed by the Gaussian distribution, which is given like: $P(x^{(i)}) = N(x^{(i)}; \mu, \sigma^2)$, where $i \in \{1, ..., m\}$.

Recall that Gaussian probability density function is given by:

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(x^{(i)} - \mu)^2}{2\sigma^2}) \tag{1.4}$$

The typical estimator of the Gaussian mean parameter is known as the **sample mean**:

$$\mu_m = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \tag{1.5}$$

The other parameter that characterizes our Gaussian distribution is the **standard deviation** from the mean value:

$$\sigma_m^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu_m)^2 \tag{1.6}$$

At the end of this implementation, we will have two different Gaussian distribution of DalMagro's CFD data, one distribution for each parameter (Cp, Sw). This work is done to understand later how many machine learning predicted values, coming from the model's predictions, are inside the following ranges:

$$\begin{gathered} \mu - \sigma \;\&\; \mu + \sigma \\ \mu - 2\sigma \;\&\; \mu + 2\sigma \\ \mu - 3\sigma \;\&\; \mu + 3\sigma \end{gathered} \tag{1.7}$$

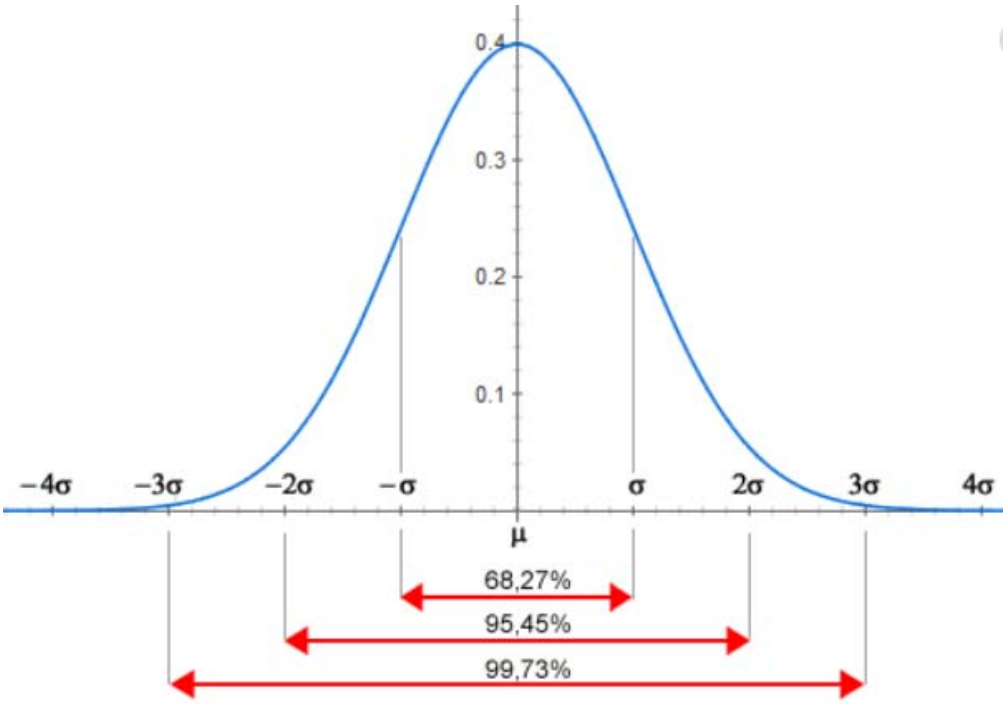

FIGURE 1.5: Example of a Gaussian distrubution.

The machine learning models done and explained with their performance in chapter four, have to be as stronger as possible learning from a certain amount of CFD data called training dataset given by DalMagro's optimization loop [6]. When the ML model is built, we have to check it, making predictions using the rest of geometries already studied by the CFD, remaining in the dataset file; this is called: data test.

This learning and testing process will be iterated and rerun to build similar models every time until we get the smallest error between the real value and the predicted one of the same variable. Another check can be done using the Gaussian distribution, and it consists in: When the most of the new predicted data from the remaining examples in the database not used during the training process, fall inside the $\mu-\sigma \,\&\, \mu+\sigma$ space under the Gaussian curve means the model is accurate. In this way, we can check how many predicted values are close to the centre of the gaussian curve, and it means that the value predicted is next to the average.

Let's start to discover how the Gaussian distribution looks like for the two variables we are studying in the DalMagro's CFD dataset used to train the ML models.

- **CP Gaussian distribution:**

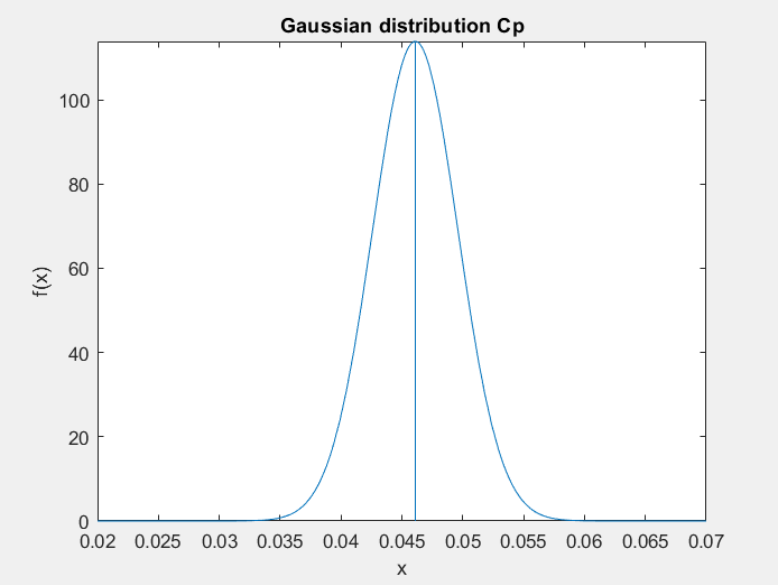

FIGURE 1.6: DalMagro's Gaussian distrubution for the CP.

Calculating the distribution we find that:

$$Cp\,mean = 0.0461$$

$$Cp\,standard\,deviation = 0.0035$$

- **Swirl Gaussian distribution:**

Calculating the distribution, we find that:

$$Swirl\,mean = 3.2600$$

$$Swirl\,standard\,deviation = 0.6099$$

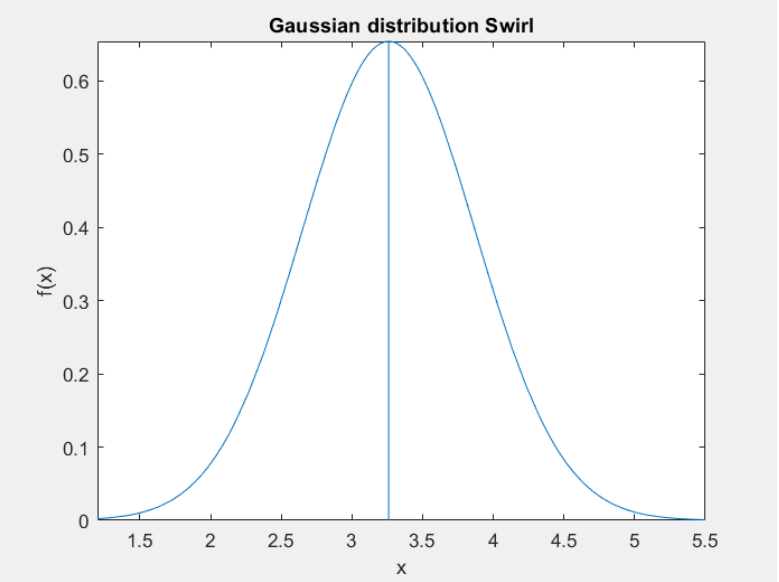

FIGURE 1.7: DalMagro's swirl Gaussian distrubution

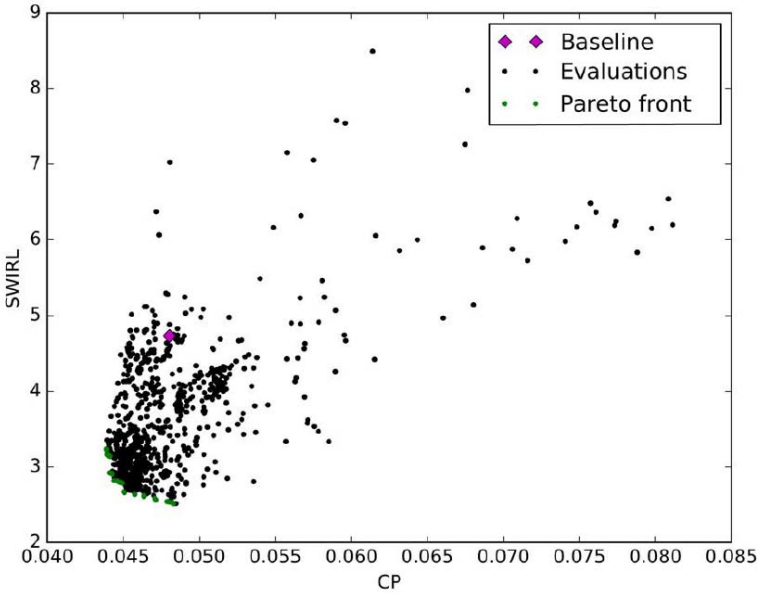

FIGURE 1.8: DalMagro's Pareto front optimization.

1.4 The input format database

The following section will explain how the variables dataset should be written to allow the model to learn on the fastest and the best way possible. The correct order and position of the data given to the training model is crucial. To do that, precise rules must be followed because the model is susceptible to it.
The database has to be structured neatly to learn from it. An example will be shown to understand better how computers can learn from the correct data given. For example, a person in everyday life has an immense amount of knowledge about the world. Much of this knowledge is subjective and intuitive and therefore difficult to articulate formally. Computers need to capture this same knowledge to behave intelligently.

One of the critical challenges in artificial intelligence is how to get this informal knowledge into a computer; that is why we need to focus on the database input. The difficulties faced by systems relying on hard-coded knowledge suggest that artificial intelligence (AI) systems need the ability to acquire their knowledge by extracting patterns from raw data. This capability, as we said, is known as machine learning. The introduction of machine learning enabled computers to tackle problems involving knowledge of the real world and make decisions that appear subjective.

For training the machine learning model, the code takes an instance file with the format shown below:

$$
\begin{aligned}
&1\ 5 : 0.235\ 6 : 0.784\ 7 : 0.248\\
&0\ 5 : 0.257\ 6 : 0.697\ 7 : 0.478\\
&0\ 5 : 0.781\ 6 : 0.617\ 7 : 0.819\\
&1\ 5 : 0.753\ 6 : 0.961\ 7 : 0.759\\
&0\ 5 : 1.025\ 6 : 0.473\ 7 : 0.429
\end{aligned}
\tag{1.8}
$$

Each line represents a single instance. The first line is the first instance label; $5, 6, 7$ are the feature indices of the first instance; $0.235, 0.784, 0.248$ are the feature values. In the binary classification case, 1 is used to indicate positive samples, and 0 is used to indicate negative samples. ML model also supports probability values in $[0, 1]$ as a label, to indicate the probability of the instance being positive.
On the instances in the training data, we may assign some weights to differentiate the relative importance of the variables. For example, if we provide an instance weight file for the training dataset like the example below:

$$\begin{matrix} 1 \\ 0.5 \\ 0.5 \\ 1 \\ 0.6 \end{matrix} \tag{1.9}$$

It means that the code will emphasize more on the first and fourth instances while training. If this file exists, the instance weights will be extracted and used at the time of training the features. The following section will explain how the dataset of the variables should be built to allow the model to learn in the fastest way possible, also because XGBoost allows us to have many ways to give input data.

Chapter 2

Machine Learning basics

2.1 Introduction

When programmable computers were first conceived, people wondered whether such machines might become intelligent. Today, artificial intelligence (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labour, understand speech or images, make diagnoses in medicine and support basic scientific research.

In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers; that solve problems that can be described by a list of formal, mathematical rules. The real challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally, problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. This capability is known as machine learning.

2.2 Artificial Intelligence with Machine Learning

The performance of a simple machine learning algorithm depends heavily on the representation of the data they are given. Many artificial intelligence tasks, for example, can be solved by designing the right set of features and then providing these features to a simple machine learning algorithm. For example, a useful feature for speaker identification from the sound is to estimate the size of the speaker's vocal tract. This feature gives a strong clue as to whether the speaker is a man, woman, or child. [9]

For many tasks, it is difficult to know what features should be extracted. For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might want to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape, but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.

One solution to this problem is to use machine learning to discover not only the mapping from representation but also the representation itself. This approach is known as representation learning. Learned representations often result in much better performance than can be obtained with hand-designed representations. They also enable AI systems to adapt to new tasks, with minimal human intervention rapidly. A representation learning algorithm can discover a good set of features for a simple task in minutes, or a complex task in hours to months. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers. Figure 2.1 illustrates the relationship between different artificial intelligence disciplines.

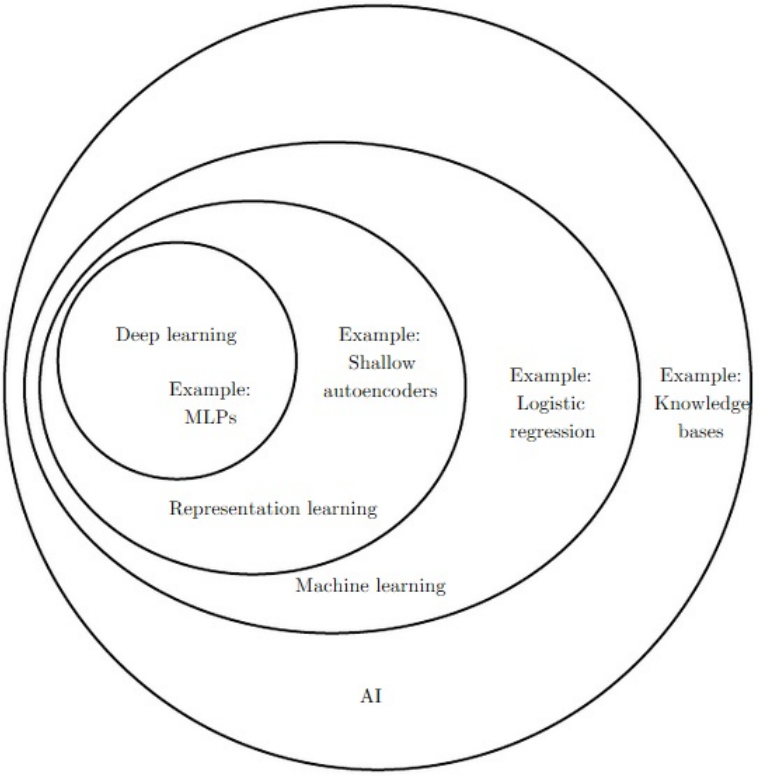

FIGURE 2.1: The diagram is showing how deep learning is a kind of representation learning, which is, in turn, a kind of machine learning. Each section of the diagram includes an example of artificial intelligence (AI) technology. Source: [9]

When designing features or algorithms for learning features, our goal is usually to separate the factors of variation that explain the observed data. Such factors are often not quantities that are directly observed. Instead, they may exist as either unobserved objects or unobserved forces in the physical world that affect observable quantities. They may also exist as constructs in the human mind that provide useful simplifying explanations or inferred causes of the observed data.

They can be thought of as concepts or abstractions that help us make sense of the rich variability in the data. A significant source of difficulty in many real-world artificial intelligence applications is that many of the factors of variation influence every single piece of data we can observe. Most applications require us to disentangle the factors of variation and discard the ones that we do not care about. Of course, it can be very difficult to extract such high-level, abstract features from raw data.

Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, more straightforward representations. Deep learning enables the computer to build complex concepts out of more straightforward ideas. Networks with greater depth can execute more instructions in sequence.

Sequential instructions offer high power because the following instructions can refer back to the results of earlier instructions. There are two main ways of measuring the depth of a model.

- The first view is based on the number of sequential instructions that must be executed to evaluate the architecture.
- The depth of a model as being not the depth of the computational graph but the depth of the graph describing how concepts are related to each other.

In the second case, the depth of the flowchart of the computations needed to compute the representation of each concept may be much deeper than the graph of the concepts themselves. This is because the system's understanding of the simpler concepts can be refined given information about the more complex concepts.
Because it's not always clear which of these two views: the depth of the computational graph or the depth of the probabilistic modelling graph is most relevant, and because different people choose different sets of smallest elements from which to construct their graphs, there is no single correct value for the depth of architecture, just as there is no single correct value for the length of a computer program. Nor is there a consensus about how much depth a model requires to qualify as "deep." However, deep learning can be safely regarded as the study of models that involve a more considerable amount of composition of either learned functions or learned concepts than traditional machine learning does.

2.3 Deep Learning

Deep learning dates back to the 1940s. Deep learning only appears to be new, because it was relatively unpopular for several years preceding its current popularity, and because it has gone through many different names, only recently being called "deep learning." The field has been rebranded many times, reflecting the influence of different researchers and different perspectives. Some essential context is useful for understanding deep learning. There have been three waves of development: deep learning known as cybernetics in the 1940s–1960s, deep learning known as connectionism in the 1980s-1990s, and the current resurgence under the name deep learning begins in 2006.

2.3.1 Historical Trends in Deep Learning

It is easiest to understand deep learning with some historical context. Rather than providing a detailed history of deep learning, we identify a few key trends:

- Deep learning has had a long and rich history, but has gone by many names, reflecting different philosophical viewpoints, and has waxed and waned in popularity.
- Deep learning has become more useful as the amount of available training data has increased.
- Deep learning models have grown in size over time as computer infrastructure (both hardware and software) for deep learning has improved.
- Deep learning has solved increasingly complicated applications with increasing accuracy over time.

2.3.2 Fundamentals of Deep Learning

Some of the earliest learning algorithms we recognize today were intended to be computational models for biological learning. Such models help us to understand how learning happens or could happen in our brain. As a result, one of the names that deep learning has gone by is artificial neural networks (ANNs). The corresponding perspective on deep learning models is that they are engineered systems inspired by the biological brain (whether the human mind or the brain of another animal). The neural perspective on deep learning is motivated by two main ideas. One idea is that the brain provides a proof by example that intelligent behaviour is possible, and a conceptually straightforward path to building intelligence is to reverse engineer the computational principles behind the brain and duplicate its functionality.

Next figure 2.2 will show a simple Flowchart in which we can observe the different ways to build a Machine Learning code starting from a simple hand-designed program. From the past, all the codes written by humans were built to solve singles math problems where we are forced to do iterations to solve the problem. To proceed with the iterations inside the loops, peoples were building features that working together were proceeding straight to the solution. Classic Machine Learning methods were born just making all these features communicating together to get the outputs. All these features inside the code are not static but are always growing up to interpret in every iteration better the data we give in the input. This kind of strategy allows the computer to learn the problem in every iteration regenerating new features starting from the previous one.

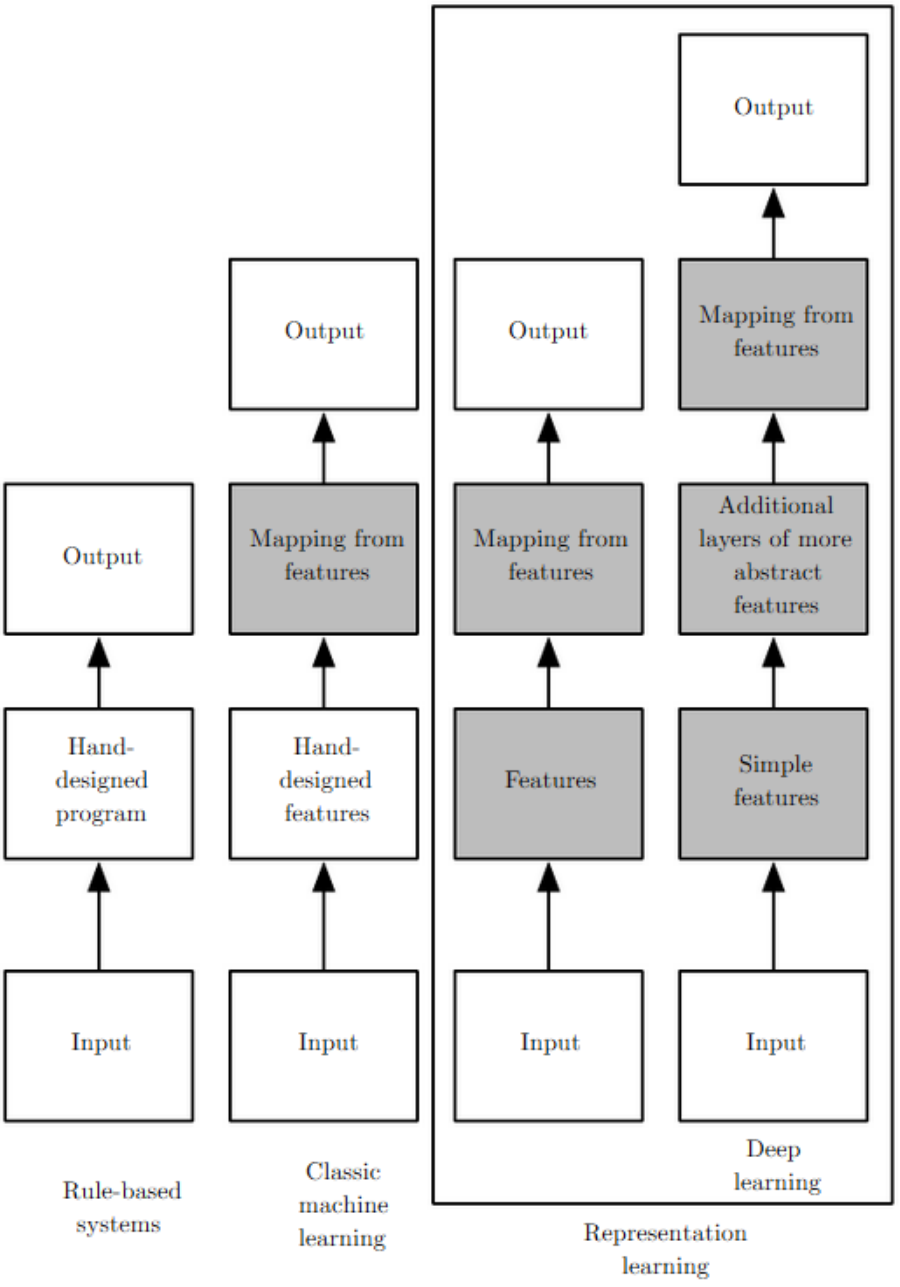

FIGURE 2.2: Flowcharts showing how the different parts of an AI system relate to each other within different artificial intelligence disciplines. Source: [9]

The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective. These models were designed to take a set of n input values $x_1,...,x_n$ and associate them with an output y. These models would learn a set of weights $\omega_1,...,\omega_n$ and compute their output $f(x, \omega) = x_1$ $\omega_1,...,x_n$ ω_n. This first wave of neural networks research was known as cybernetics.

The McCulloch-Pitts neuron, for example, (McCulloch and Pitts, 1943) was an early model of brain function. This linear model could recognize two different categories of inputs by testing whether $f(x, \omega)$ is positive or negative. Of course, for the model to correspond to the desired definition of the categories, the weights needed to be set correctly. The human operator could set these weights. In the 1950s, the perceptron (Rosenblatt, 1958, 1962) became the first model that could learn the weights that defined the categories given examples of inputs from each category. The adaptive linear element (ADALINE), which dates from about the same time, returned the value of $f(x)$ itself to predict a real number and could also learn to predict these numbers from data.

These simple learning algorithms greatly affected the modern landscape of machine learning. The training algorithm used to adapt the weight of the ADALINE was a special case of an algorithm called **stochastic gradient descent**. Slightly modified versions of the stochastic gradient descent algorithm remain the dominant training algorithms for deep learning models today.

Models based on the $f(x, \omega)$ used by the perceptron and ADALINE are called **linear models**. These models remain some of the most widely used machine learning models, though in many cases they are trained in different ways than the original models were trained.

Media accounts often emphasize the similarity of deep learning to the brain. While it is true that deep learning researchers are more likely to cite the brain as an influence than researchers working in other machine learning fields, such as kernel machines or Bayesian statistics, one should not view deep learning as an attempt to simulate the brain.

New deep learning draws inspiration from many fields, especially applied math fundamentals like linear algebra, probability, information theory, and numerical optimization. While some deep learning researchers cite neuroscience as an essential source of inspiration, others are not concerned with neuroscience at all.

2.3.3 Increasing Dataset Sizes

One may wonder why deep learning has only recently become recognized as a crucial technology even though the first experiments with artificial neural networks were conducted in the 1950s. Deep learning has been successfully used in commercial applications since the 1990s but was often regarded as being more of an art than technology and something that only an expert could use, until recently. Some skill is indeed required to get excellent performance from a deep learning algorithm. Fortunately, the amount of skill required reduces as the amount of training data increases. The learning algorithms reaching human performance on complex tasks today are nearly identical to the learning algorithms that struggled to solve toy problems in the 1980s, though the models we train with these algorithms have undergone changes that simplify the training of very deep architectures. The most important new development is that today we can provide these algorithms with the resources they need to succeed. The increasing digitization of society drives this trend. As more and more of our activities take place on computers, more and more of what we do is recorded. As our computers are increasingly networked together, it becomes easier to centralize these records and curate them into a dataset appropriate for machine learning applications. The age of **Big Data** has made machine learning much more useful in our life. At the same time, the machine learning algorithm has to work successfully also with smaller datasets, this is an important research area, focusing in particular on how we can take advantage of small quantities of unlabeled examples, with unsupervised or semi-supervised learning.

2.3.4 Increasing Model Sizes

Another key reason that neural networks are wildly successful today after enjoying comparatively little success since the 1980s is that we have the computational resources to run much larger models today. One of the central insights of connectionism is that animals become intelligent when many of their neurons work together. An individual neuron or small collection of neurons is not particularly useful. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years.

Faster computers drive this growth with more significant memory and by the availability of more massive datasets. More extensive networks can achieve higher accuracy on more complex tasks. This trend looks set to continue for decades. The increase in model size over time, due to the availability of faster CPUs, the advent of general-purpose GPUs, speedier network connectivity and better software infrastructure for distributed computing, is one of the most important trends in the history of deep learning. This trend is generally expected to continue well into the future.

2.3.5 Increasing Accuracy, Complexity and Real-World Impact

Since the 1980s, deep learning has consistently improved in its ability to provide accurate recognition and prediction. Moreover, deep learning has consistently been applied with success to broader and broader sets of applications. Deep networks have also had spectacular successes for pedestrian detection and image segmentation and yielded superhuman performance in traffic sign classification. At the same time that the scale and accuracy of deep networks have increased, so has the complexity of the tasks that they can solve.

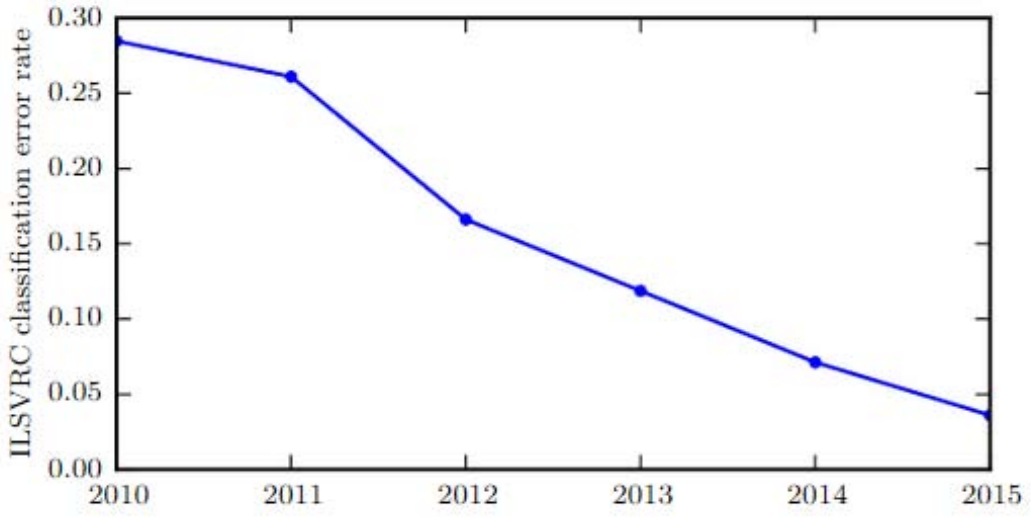

FIGURE 2.3: Decreasing error rate over time. Source: [9]

Another crowning achievement of deep learning is its extension to the domain of reinforcement learning. In the context of reinforcement learning, an autonomous agent must learn to perform a task by trial and error, without any guidance from the human operator. DeepMind demonstrated that a reinforcement learning system based on deep learning is capable of learning to play Atari video games, reaching human-level performance on many tasks. Deep learning has also significantly improved the performance of reinforcement learning for robotics.

Many of these applications of deep learning are highly profitable. Deep learning is now used by many top technology companies, including Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA, and NEC.

Deep learning has also made contributions to other sciences. Deep learning also provides useful tools for processing massive amounts of data and making useful predictions in scientific fields. It has been successfully used to predict how molecules will interact to help pharmaceutical companies design new drugs and to search for subatomic particles. We expect deep learning to appear in more and more scientific fields in the future. In summary, deep learning is an approach to machine learning that has drawn heavily on our knowledge of the human brain, statistics and applied math as it developed over the past several decades. In the next section, we will explain how Deep learning works and which are the fundamentals of how computers can think.

Chapter 3

Machine Learning algorithms, XGBoost

Deep learning is a specific kind of machine learning. To understand deep learning well, one must have a solid understanding of the basic principles of machine learning. This chapter provides a brief course in the most important general principles that are applied throughout the rest of the thesis.

Most machine learning algorithms have settings called hyperparameters, which must be determined outside the learning algorithm itself; we discuss how to set these using additional data. Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased focus on proving confidence intervals around these functions. Most machine learning algorithms can be divided into categories of supervised learning and unsupervised learning. Most deep learning algorithms are based on an optimization algorithm called stochastic gradient descent. We describe how to combine various algorithm components, such as an optimization algorithm, a model, and a dataset, to build a machine learning algorithm. All these particular characteristics will be explained in the following chapter because they are the basis of XGBoost. This type of Machine Learning with some specific parameters set up will be used to predict the aerodynamics variables we need to find for our S-Duct problem, as explained in the first chapter.

3.1 Learning Algorithms

A machine learning algorithm is an algorithm that can learn from data. But what do we mean by learning? Mitchell (1997) provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [9] One can imagine a wide variety of experiences E, tasks T, and performance measures P. In the following sections, we provide intuitive descriptions and examples of the different kinds of tasks, performance measures, and experiences that can be used to construct machine learning algorithms.

3.1.1 The T Task

From a scientific and philosophical point of view, machine learning is interesting because developing our understanding of it entails developing our understanding of the principles that underlie intelligence. Machine learning tasks are usually described in terms of how the machine learning system should process an **example**. An example is a collection of features that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $\boldsymbol{x} \in \Re^n$ where each entry x_i of the vector is another feature. For example, the features of an image are usually the values of the pixels in the image. Many kinds of tasks can be solved with machine learning. Some of the most common machine learning tasks are the following:

- **Classification:** The computer program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \Re^n \rightarrow 1, ..., k$. When $y = f(x)$, the model assigns an input described by vector x to a category identified by numeric code y.

An example of this classification task is like in our case where different numbers inside the data identify one geometry for our S-Duct and the output from the classification process inside the program is a numeric code identifying the specific geometry we are taking into consideration to analyze.

- **Classification with missing inputs:** To solve the classification task, the learning algorithm only has to define a *single* function mapping from a vector input to a categorical output. When some of the inputs may be missing, rather than providing a single classification function, the learning algorithm must learn a *set of functions*. One way to efficiently define such a broad set of functions is to learn a probability distribution over all the relevant variables, then solve the classification task by marginalizing out the missing variables. With n input variables, we can now obtain all 2^n different classification functions needed for each possible set of missing inputs, but the computer program needs to learn only a single function describing the joint probability distribution.

- **Regression:** In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \Re^n \rightarrow \Re$. This type of task is similar to classification, except that the format of the output is different.

- **Transcription:** In this task, the machine learning system is asked to observe a relatively unstructured representation of some data and transcribe the information into a discrete textual form. Google Street View, for example, uses deep learning to process address numbers in this way.

- **Machine translation:** In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a series of symbols in another language. This is commonly applied to natural languages, such as translating from English to French. Deep learning has recently begun to have a substantial impact on this kind of task.

- **Structured output:** Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with essential relationships between the different elements. This is a broad category and subsumes the transcription and translation tasks described above, as well as many other tasks. This task is used frequently in the XGBoost iterations because the program must output several values that are all tightly interrelated together.

- **Anomaly detection:** In this type of task, the computer program sifts through a set of events or objects and flags some of them as being unusual or atypical. An example can be done with the XGBoost Machine Learning method we are using to predict our aerodynamic variables when the program has to automatically detect the input data that has no sense to build the model with.

- **Synthesis and sampling:** In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. In some cases, we want the sampling or synthesis procedure to create a specific kind of output given the input. In our case, the numbers inside the data set that has no sense for the model must be deleted automatically by this Machine Learning task.

- **Imputation of missing values:** In this type of task, the machine learning algorithm is given a new example $\boldsymbol{x} \in \Re^n$, but with some entries x_i of X missing. The algorithm must provide a prediction of the values of the missing entries. In XGBoost, when some part of the training data seems to be wrong, is deleted automatically by the code, then the missing values have to be replaced with a new number. The capacity of this task is that: interpreting the rest of the data, Machine Learning can provide a prediction of the values that are missing.

- **Denoising:** In this type of task, the machine learning algorithm is given as input a *corrupted example* $\boldsymbol{a} \in \Re^n$ obtained by an unknown corruption process from a *clean example* $\boldsymbol{A} \in \Re^n$ The learner must predict the clean example **A** from its corrupted version **a**. This kind of task is used in the XGBoost Deep Learning code when we have one predicted value for a variable we are calculating thas has been discarded because far away from the considerable range. When a predicted value is deleted because it is not considered reliable, the Machine Learning code sees the value as a corrupted one, and its purpose will be to replace it with a new one.

- **Density estimation or probability mass function estimation:** In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{model} : \Re^n \rightarrow \Re$, where $p_{model}(x)$ can be interpreted as a probability density function (if x is continuous) or a probability mass function (if x is discrete) on the space that the examples were drawn from.

  To implement the task well, the algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur. Most of the tasks described above require the learning algorithm to at least implicitly capture the structure of the probability distribution. Density estimation enables us to capture that distribution explicitly. In principle, we can then perform computations on that distribution to solve the other tasks.

Of course, many other tasks and types of tasks are possible. The types of functions we list here are intended to provide examples of what is running inside my Machine Learning code built up with XGBoost.

3.1.2 The Performance Measure, P

To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually, this performance measure **P** is specific to the task T being carried out by the system. For tasks such as classification with missing inputs, and transcription, we often measure the accuracy of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the error rate, the proportion of examples for which the model produces incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular instance is 0 if it is correctly classified and 1 if it is not.

Usually, we are interested in how well the machine learning algorithm performs on data that the code has never seen before since this determines how well it will work when deployed in the real world. We, therefore, evaluate these performance measures using a test set of data that is separate from the data used for training the machine learning system. The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behaviour of the system. In some cases it is difficult to decide what should be measured. When performing a regression task, for example, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes huge mistakes?

These kinds of design choices depend on the application. In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. In our case, we use a part of the data set to validate the Machine Learning model to verify how far away were the predicted values from the real values almost existing inside the data set.

3.1.3 The Experience, E

Machine learning algorithms can be broadly categorized as **unsupervised** or **supervised** by what kind of experience they are allowed to have during the learning process. Most of the learning algorithms, like the one used to predict our aerodynamic variables, can be understood as being allowed to experience an entire **dataset**. A dataset is a collection of many examples, and sometimes we call the examples **data points**.

- **Unsupervised learning algorithms:** Experience a dataset containing many features, then learn useful properties of the structure of the dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly, as in density estimation, or implicitly, for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform different roles, like clustering, which consists of dividing the dataset into clusters of similar examples.
- **Supervised learning algorithms:** Experience a dataset containing features, but each example is also associated with a **label** or **target**. This kind of operation way is the one XGBoost has implemented inside the code to build the model to predict the values of the variables we are asking for.

Unsupervised learning involves observing several examples of a random vector $\mathbf{x}$ and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution; while supervised learning involves observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, then learning to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating $p(y \mid x)$. The term **supervised learning** originates from the view of the target $\mathbf{y}$ being provided by an instructor who shows the machine learning system what to do.

In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide. Unsupervised learning and supervised learning are not formally defined terms.

The lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $\boldsymbol{x} \in \Re^n$, the joint distribution can be decomposed as:

$$p(x) = \prod_{i=1}^{n} p(x_i \mid x_1, ..., x_{i-1}). \tag{3.1}$$

This decomposition means that we can solve the unsupervised problem of modelling $p(x)$ by splitting it into n supervised learning problems. Alternatively, we can solve the supervised learning problem of learning $p(y \mid x)$ by using traditional unsupervised learning technologies to learn the joint distribution $p(x, y)$, then inferring:

$$p(y \mid x) = \frac{p(x, y)}{\sum_{y'} p(x, y')} \tag{3.2}$$

Though unsupervised learning and supervised learning are not wholly formal or distinct concepts, they do help roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning.

Some machine learning algorithms do not just experience a fixed dataset. For example, **reinforcement learning** algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences, but this is something we will not go through because is not used in the XGBoost code.

Most machine learning algorithms experience dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples, which are, in turn, collections of features. For us, these features will be some numerical parameters identify the S-Duct geometry we want to study with the machine learning model in XGBoost.

One common way of describing a dataset is with a design matrix. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For example, in our case, we have thirty-six different numerical variables that identify one S-Duct geometry.

Every column of the matrix that XGBoost builds inside the code means a different variable, and in our case, every variable means a different control point coming out from an early parametrization of the S-Duct geometry. Every row of the same matrix inside XGBoost means a different parametrization of the Duct geometry. Very important is to order all the parameters that every variable needs to stay in the correct row and column, if not the machine learning will confuse everything and build a model completely wrong. We will talk about it later when XGBoost is explained in more details.

To continue the explanation on the dataset we need, we can look at the data file we have below in Figure 3.1. It contains almost 900 S-Duct geometries examples with thirty-six features for each sample. This means we can represent the dataset with a design matrix $\mathbf{X} \in \Re^{900x36}$, where $X_{i,1}$ is the first variable of the i Duct geometry parametrization, $X_{i,2}$ is the second parameter of the same Duct, etc. Of course, to describe a dataset as a design matrix, it must be possible to describe each example as a vector, and each of these vectors must be the same size. This is not always possible.

2.45216	0.081872	2.74001	-0.07758	3.01436	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.72764	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.01436	-0.2028	0.136162	-0.02588	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.01436	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.01436	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.01436	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.01436	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.72764	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.01436	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.72764	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	-0.03308	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	-0.02588	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.38176
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.115775	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.68326	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.38176
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.38176
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.38176
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.38176
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.07035	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	-0.07758	3.1177	-0.2028	0.136162	0.312611	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.13729
2.45216	0.081872	2.82164	0.166881	3.1177	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.23159	0.285019	2.64602	-0.13729
2.45216	0.326335	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.130857	3.07692	-0.55928	0.145939	2.33807	-0.72052	0.285019	2.72764	-0.13729
2.45216	0.326335	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.130857	3.07692	-0.55928	0.315183	2.33807	-0.72052	0.285019	2.64602	-0.13729
2.45216	0.326335	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.130857	3.07692	-0.55928	0.315183	2.33807	-0.72052	0.285019	2.72764	-0.13729
2.45216	0.326335	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.130857	3.07692	-0.31482	0.315183	2.33807	-0.72052	0.285019	2.72764	-0.13729
2.45216	0.326335	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.72052	0.285019	2.72764	-0.13729
2.45216	0.326335	2.82164	-0.07758	2.91102	-0.2028	0.136162	0.143367	0.150572	2.41529	-0.01942	0.285019	2.76488	-0.17588	0.300101	3.07692	-0.31482	0.315183	2.33807	-0.47605	0.285019	2.72764	-0.13729

Figure 3.1: S-duct database geometry

3.1.4 Linear Regression

All the above features and tasks explained are the heart of the Deep Learning algorithms working in our days, especially inside XGBoost, the code used to solve the S-Duct problem in this work.

To make all these concepts more concrete, we present an example of a machine learning algorithm implemented in the XGBoost code specified as: **linear regression**. We will return to this example repeatedly as we introduce more machine learning concepts that help to understand the algorithm's behaviour.

As the name implies, linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $\mathbf{x} \in \Re^n$ as input and predict the value of a scalar $y \in \Re$ as its output. The output of linear regression is a linear function of the input. Let $\widehat{y}$ be the value that our model predicts y should take on. We define the output to be:

$$\widehat{y} = \omega^{\intercal} x \tag{3.3}$$

where $\omega \in \Re^n$ is a vector of **parameters**.

Parameters are values that control the behaviour of the system. In this case, ω_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features. We can think of ω as a set of **weights** that determine how each element affects the prediction. If a feature x_i receives a positive weight of ω_i, then increasing the value of that feature increases the value of our prediction $\widehat{y}$. If a feature gets a negative weight, then increasing the value of that feature decreases the value of our prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction.

We thus have a definition of our task T : to predict y from $\mathbf{x}$ by outputting $\widehat{y} = \omega^{\intercal} x$. Next, we need a definition of our performance measure, P. Suppose that we have a design matrix of m example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of y for each of these examples. Because this dataset will only be used for evaluation, we call it the test set. We refer to the design matrix of inputs as $X^{(test)}$ and the vector of regression targets as $y^{(test)}$.

One way of measuring the performance of the model is to compute the **mean squared error** of the model on the test set. If $\widehat{y}^{(test)}$ gives the predictions of the model on the test set, then the mean squared error is given by:

$$MSE_{test} = \frac{1}{m} \sum_{i} (\widehat{y}^{(test)} - y^{(test)})_i^2. \tag{3.4}$$

Intuitively, one can see that this error measure decreases to 0 when $\widehat{y}^{(test)} = y^{(test)}$. We can also see that:

$$MSE_{test} = \frac{1}{m} \parallel \widehat{y}^{(test)} - y^{(test)} \parallel_2^2, \tag{3.5}$$

so the error increases whenever the Euclidean distance between the predictions and the targets increases.

To make a machine learning algorithm, we need to design an algorithm that will improve the weights ω in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(X^{(train)}, y^{(train)})$. One intuitive way of doing this is to minimize the mean squared error on the training set, MSE_{train}.

To minimize MSE_{train}, we can simply solve for where its gradient is 0:

$$\nabla_\omega MSE_{train} = 0 \tag{3.6}$$

$$\Rightarrow \nabla_\omega \frac{1}{m} \parallel \widehat{y}^{(train)} - y^{(train)} \parallel_2^2 = 0 \tag{3.7}$$

$$\Rightarrow \frac{1}{m} \nabla_\omega \parallel X^{(train)}\omega - y^{(train)} \parallel_2^2 = 0 \tag{3.8}$$

$$\Rightarrow \nabla_\omega (X^{(train)}\omega - y^{(train)})^\intercal (X^{(train)}\omega - y^{(train)}) = 0 \tag{3.9}$$

$$\Rightarrow \nabla_\omega (\omega^\intercal X^{(train)\intercal} X^{(train)}\omega - 2\omega^\intercal X^{(train)\intercal} y^{(train)} + y^{(train)\intercal} y^{(train)}) = 0 \tag{3.10}$$

$$\Rightarrow 2X^{(train)\intercal} X^{(train)}\omega - 2X^{(train)\intercal} y^{(train)} = 0 \tag{3.11}$$

$$\omega = (X^{(train)\intercal} X^{(train)})^{-1} X^{(train)\intercal} y^{(train)} \tag{3.12}$$

The system of equations whose solution is given by equation **3.12** is known as the **normal equations**. Evaluating equation **3.12** constitutes a simple learning algorithm. For an example of the linear regression learning algorithm in action, see figure 3.2.

It is worth noting that the term **linear regression** is often used to refer to a slightly more sophisticated model with one additional parameter, an intercept term b. In this model:

$$\widehat{y} = \omega^\intercal x + b \tag{3.13}$$

the mapping from parameters to predictions is still a linear function, but the mapping from features to predictions is now an affine function.

Below we can see an example of a linear regression way to proceed.

A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector ω includes only a single parameter to learn, ω_1. *(Left)* Observe that linear regression learns to set ω_1 such that the line $y = \omega_1 x$ comes as close as possible to pass through all the training points. *(Right)* The plotted point indicates the value of ω_1 found by the standard equations, which we can see minimizes the mean squared error on the training set.

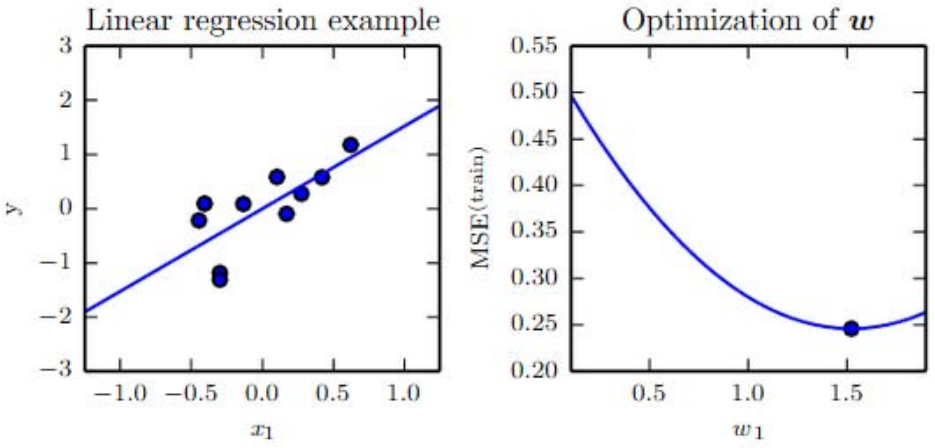

FIGURE 3.2: The linear regression problem

This extension to affine functions means that the plot of the model's predictions still looks like a line, but it need not pass through the origin. Instead of adding the bias parameter b, one can continue to use the model with only weights but augment $\mathbf{x}$ with an extra entry that is always set to 1. The weight corresponding to the extra 1 entry plays the role of the bias parameter.

The intercept term b is often called the **bias** parameter of the affine transformation. Linear regression is, of course, a straightforward and limited learning algorithm, but it provides an example of how a learning algorithm can work.

3.2 Overfitting and Underfitting

The central challenge in machine learning is that our algorithm must perform well on *new, previously unseen* inputs, not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization**.

Typically, when training a machine learning model, we have access to a training set; we can compute some error measure on the training set, called the **training error**, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the **generalization error**, also called the **test error**, to be low as well. The generalization error is defined as the expected value of the error on new input.

We typically estimate the generalization error of a machine learning model by measuring its performance on a **test set** of examples that were collected separately from the training set. In our linear regression example, we trained the model by minimizing the training error:

$$\frac{1}{m^{train}} \parallel X^{(train)}\omega - y^{(train)} \parallel_2^2, \tag{3.14}$$

but we actually care about the test error,

$$\frac{1}{m^{test}} \parallel X^{(test)}\omega - y^{(test)} \parallel_2^2, \tag{3.15}$$

How can we affect performance on the test set when we can observe only the training set? The field of **statistical learning theory** provides some answers. If we are allowed to make some assumptions about how the training and test set are collected, then we can make some progress. The training and test data are generated by a probability distribution over datasets called the **data-generating process**.

We typically make a set of assumptions known collectively as the independent and identically distributed variables (i.i.d.) assumptions. These assumptions are that the examples in each dataset are **independent** from each other, and that the training set and test set are **identically distributed**, drawn from the same probability distribution as each other. This assumption enables us to describe the data-generating process with a probability distribution over a single example. The same distribution is then used to generate every training example and every test example. We call that **data-generating distribution**, denoted p_{data}. This probabilistic framework and the Independent and identically distributed assumptions enables us to study the relationship between training error and test error mathematically.

Of course, when we use a machine learning algorithm, we do not fix the parameters ahead of time, then sample both datasets. We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform its ability to:

1. Make the training error small.

2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: **underfitting** and **overfitting**.

Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large. These phenomena will be shown in Figure 3.3. We can control whether a model is more likely to overfit or underfit by altering its **capacity**.

Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

Machine learning algorithms will generally perform best when their capacity is appropriate for the real complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit.

Figure 3.3 shows this principle in action. We compare a linear, quadratic and degree-9 predictor attempting to fit a problem where the right underlying function is quadratic.

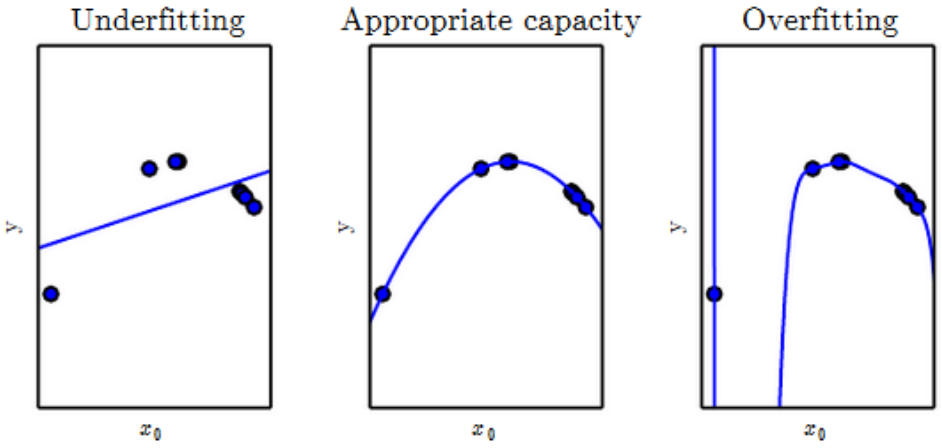

FIGURE 3.3: Underfitting and overfitting

Explaining Figure 3.3, we can say the following.

We fit three models to this example training set. The training data was generated synthetically by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. *Left* A linear function fit to the data suffers from underfitting. It cannot capture the curvature that is present in the data. *Center* A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. *Right* A polynomial of degree 9 fit to the data suffers from overfitting.

The solution passes through all the training points correctly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley between two training points that do not appear in the real underlying function. It also increases sharply on the left side of the data, while the actual function decreases in this area.

The linear function is unable to capture the curvature in the real underlying problem, so it underfits. The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely many other functions that pass precisely through the training points, and it is not valid. In this example, the quadratic model is perfectly matched to the actual structure of the task, so it generalizes well to new data.

Another essential thing to say is that there are many ways to change a model's capacity. Capacity is not determined only by the choice of model. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective. This is called the **representational capacit**y of the model. In many cases, finding the best function within this family is a difficult optimization problem. In practice, the learning algorithm does not find the best function, but merely one that significantly reduces the training error.

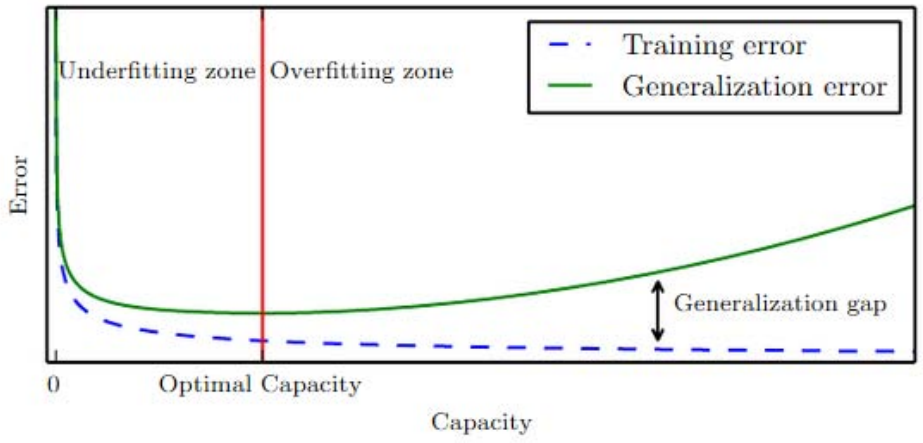

FIGURE 3.4: Capacity and error relationship

Typically, generalization error has a U-shaped curve as a function of model capacity. This is illustrated in figure 3.4, where training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the **overfitting** regime, where capacity is too large, above the optimal capacity.

The ideal model is an oracle that knows the true probability distribution that generates the data. Even such a model will still incur some error on many problems, because there may still be some noise in the distribution. In the case of supervised learning, the mapping from $\mathbf{x}$ to y may be inherently stochastic, or y may be a deterministic function that involves other variables besides those included in $\mathbf{x}$. The error incurred by an oracle making predictions from the true distribution $p(\mathbf{x}, y)$ is called the **Bayes error**. Training and generalization errors vary as the size of the training set varies. Expected generalization error can never increase as the number of training examples increases. Note that it is possible for the model to have the optimal capacity and yet still have a large gap between training and generalization errors. In this situation, we may be able to reduce this gap by gathering more training examples.

3.2.1 Regularization

The idea behind everything is that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems that we ask the algorithm to solve, it performs better.
So far, the only method of modifying a learning algorithm that we have discussed concretely is to increase or decrease the model's representational capacity by adding or removing functions from the hypothesis space of solutions the learning algorithm can choose from.

We can also give a learning algorithm preference for one solution over another in its hypothesis space. This means that both functions are eligible, but one is preferred. The unpreferred solution will be chosen only if it fits the training data significantly better than the preferred solution.
For example, we can modify the training criterion for linear regression to include **weight decay**. To perform linear regression with weight decay, we minimize a sum $J(\omega)$ comprising both the mean squared error on the training and a criterion that expresses a preference for the weights to have smaller squared L^2 norm. Especially, where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights. When $\lambda = 0$, we impose no preference, and larger λ forces the weights to become smaller.

$$J(\omega) = MSE_{train} + \lambda\omega^{\intercal}\omega \tag{3.16}$$

Minimizing $J(\omega)$ results in a choice of weights that make a tradeoff fitting the training data. This gives us solutions that have a smaller slope, or that put weight on fewer of the features. As an example of how we can control a model's tendency to overfit or underfit via weight decay, we can train a high-degree polynomial regression model with different values of λ.
More generally, we can regularize a model that learns a function $f(x;\theta)$ by adding a penalty called a **regularizer** to the cost function. In the case of weight decay, the regularizer is $\Omega(\omega) = \omega^{\intercal}\omega$.

There are many other ways of expressing preferences for different solutions, both implicitly and explicitly. Together, these different approaches are known as **regularization**. *regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.* Regularization is one of the central concerns of the field of machine learning, rivalled in its importance only by optimization.

3.3 Hyperparameters and Validation Sets

Most machine learning algorithms have hyperparameters, settings that we can use to control the algorithm's behaviour. The learning algorithm itself does not adapt to the values of hyperparameters. Though we can design a nested learning procedure in which one learning algorithm learns the best hyperparameters for another learning algorithm. The polynomial regression example in figure 3.5 has a single hyperparameter: the degree of the polynomial, which acts as a **capacity** hyperparameter. The λ value used to control the strength of weight decay is another example of a hyperparameter.

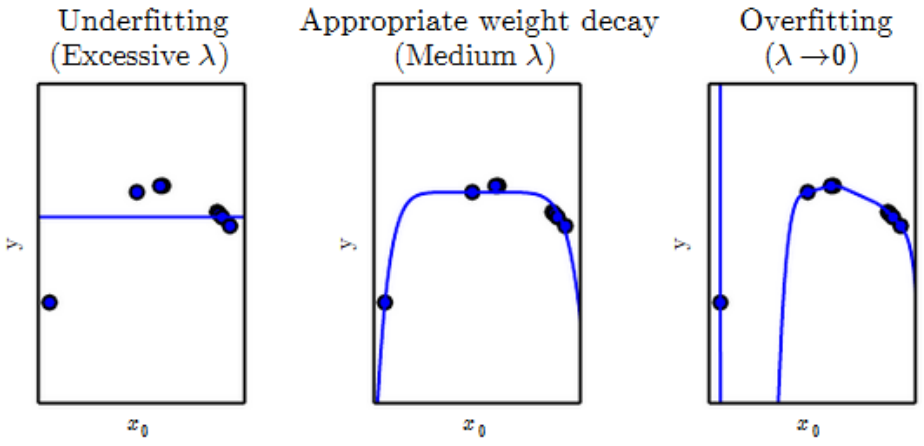

FIGURE 3.5: The hyperparameter effect

We fit a high-degree polynomial regression model from figure 3.5. The real function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting.

Left With very large λ, we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. *Center* With an average value of λ, the learning algorithm recovers a curve with the right general shape. *Right* With weight decay approaching zero, the degree-9 polynomial overfits significantly, as we saw in figure 3.3.

For example, we can always fit the training set better with a higher-degree polynomial and a weight decay setting of $\lambda = 0$ than we could with a lower-degree polynomial and a specific weight decay setting, in fact, To solve this problem, we need a **validation set** of examples that the training algorithm does not observe.

3.3.1 Cross-Validation

Dividing the dataset into a fixed training set and a fixed test set can be problematic if it results in the test set being small. A small test set implies statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task.
When the dataset has hundreds of thousands of examples or more, this is not a serious issue. When the dataset is too small, alternative procedures enable one to use all the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the training and testing computation on different randomly chosen subsets or splits of the original dataset.

3.4 Supervised Learning Algorithms

Supervised learning algorithms are, roughly speaking, learning algorithms that learn to associate some input with some output, given a training set of examples of inputs $\mathbf{x}$ and outputs $\mathbf{y}$. In many cases, the outputs $\mathbf{y}$ may be difficult to collect automatically and must be provided by a human "supervisor," but the term still applies even when the training set targets were collected automatically.

3.4.1 Probabilistic Supervised Learning

Most supervised learning algorithms used are based on estimating a probability distribution $p(y \mid x)$. We can do this simply by using maximum likelihood estimation to find the best parameter vector θ for a parametric family of distributions $p(y \mid x; \theta)$.

We have already seen that linear regression corresponds to the family:

$$p(y \mid x; \theta) = N(y; \theta^{\intercal} x, I). \tag{3.17}$$

We can generalize linear regression to the classification scenario by defining a different family of probability distributions. If we have two classes, class 0 and class 1, then we need only specify the probability of one of these classes. The probability of class 1 determines the probability of class 0 because these two values must add up to 1.
The normal distribution over real-valued numbers that we used for linear regression is parametrized in terms of a mean. Any value we supply for this means valid. A distribution over a binary variable is slightly more complicated because its mean must always be between 0 and 1. One way to solve this problem is to use the logistic sigmoid function to squash the output of the linear function into the interval $(0, 1)$ and interpret that value as a probability:

$$p(y = 1 \mid x; \theta) = \sigma(\theta^{\intercal} x). \tag{3.18}$$

This approach is known as logistic regression.
In the case of linear regression, we were able to find the optimal weights by solving the standard equations. Logistic regression is somewhat more difficult. There is no closed-form solution for its optimal weights. Instead, we must search for them by maximizing the log-likelihood.

This same strategy can be applied to essentially any supervised learning problem, by writing down a parametric family of conditional probability distributions over the right kind of input and output variables.

3.4.2 Other Simple Supervised Learning Algorithms

Another type of learning algorithm that also breaks the input space into regions and has separate parameters for each region is the **decision tree** and its many variants. As shown in figure 3.6, each node of the decision tree is associated with a region in the input space, and internal nodes break that region into one subregion for each child of the node; typically using an axis-aligned cut.
Space is thus subdivided into nonoverlapping regions, with a one-to-one correspondence between leaf nodes and input regions. Each leaf node usually maps every point in its input region to the same output. Decision trees are usually trained with specialized algorithms. The learning algorithm can be considered nonparametric if it is allowed to learn a tree of arbitrary size, though decision trees are usually regularized with size constraints that turn them into parametric models in practice.

Diagrams are describing how a decision tree works. *(Top)*Each node of the tree chooses to send the input example to the child node on the left (0) or to the child node on the right (1). Internal nodes are drawn as circles and leaf nodes as squares. Each node is displayed with a binary string identifier corresponding to its position in the tree, obtained by appending a bit to its parent identifier ($0 = chooseleftortop, 1 = chooserightorbottom$). *(Bottom)* The tree divides space into regions. The $2 - D$ plane shows how a decision tree might divide $\Re^2$. The nodes of the tree are plotted in this plane, with each internal node drawn along the dividing line it uses to categorize examples, and leaf nodes drawn in the centre of the region of examples they receive. The result is a piecewise-constant function, with one piece per leaf. Each leaf requires at least one training example to define, so the decision tree can't learn a function that has more local maxima than the number of training examples.

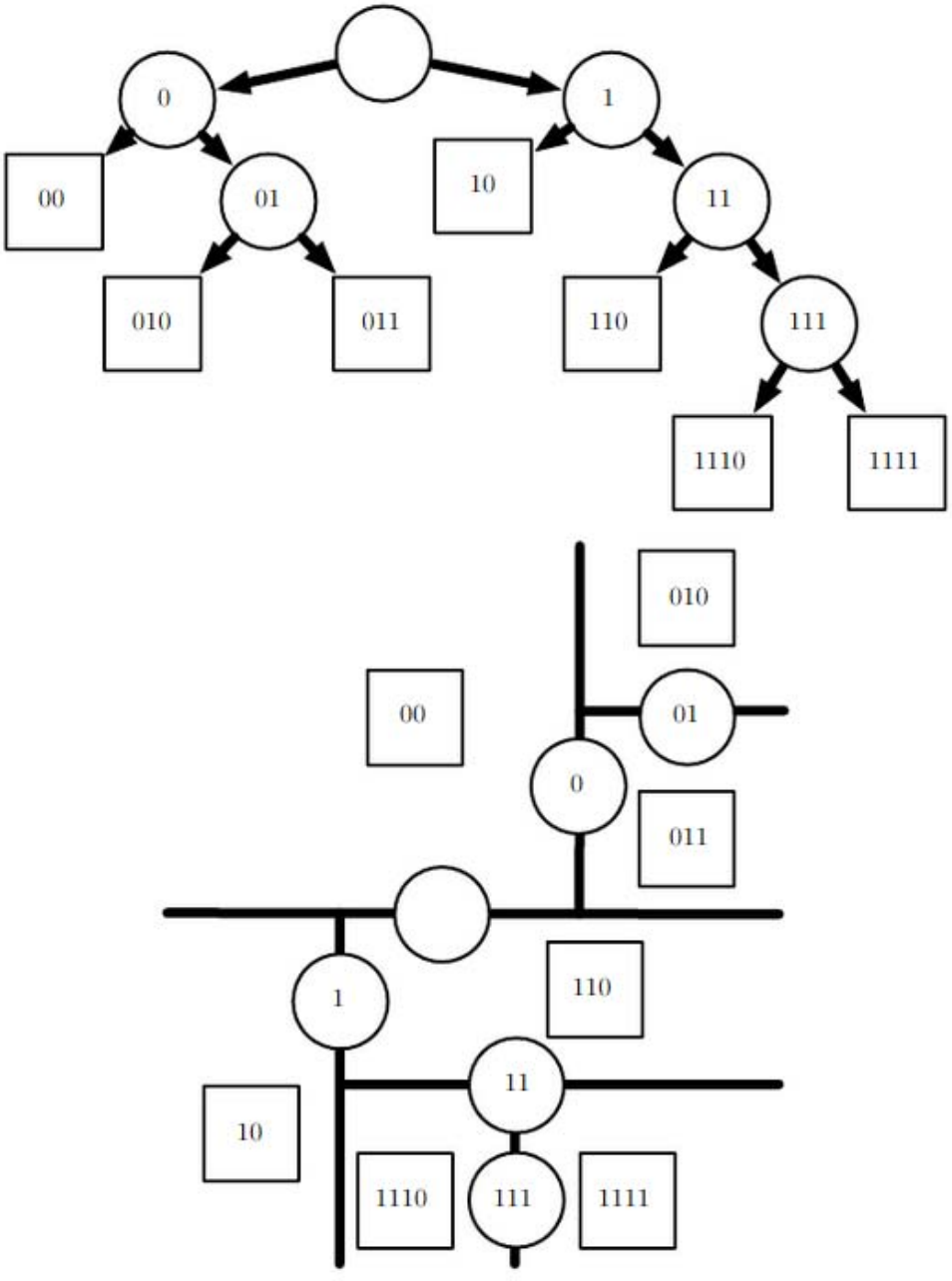

FIGURE 3.6: Decision tree Diagrams

3.5 Unsupervised Learning Algorithms

Unsupervised algorithms are those that experience only "features" but not a supervision signal. The distinction between supervised and unsupervised algorithms is not formally and rigidly defined because there is no objective test for distinguishing whether a value is a feature or a target provided by a supervisor. Informally, unsupervised learning refers to most attempts to extract information from a distribution that do not require human labour to annotate examples.
A classic unsupervised learning task is to find the "best" representation of the data. By "best" we can mean different things, but generally speaking, we are looking for a representation that preserves as much information about $\mathbf{x}$ as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than $\mathbf{x}$ itself.

There are multiple ways of defining a more straightforward representation. Three of the most common include lower-dimensional representations, sparse representations, and independent representations.

- Low-dimensional representations attempt to compress as much information about $\mathbf{x}$ as possible in a smaller representation.
- Sparse representations embed the dataset into a representation whose entries are mostly zeros for most inputs. The use of sparse representations typically requires increasing the dimensionality of the representation, so that the representation becoming mostly zeros does not discard too much information. This results in an overall structure of the representation that tends to distribute data along the axes of the representation space.
- Independent representations attempt to *disentangle* the sources of variation underlying the data distribution such that the dimensions of the representation are statistically independent.

Of course, these three criteria are certainly not mutually exclusive. Low- dimensional representations often yield elements that have fewer or weaker dependencies than the original high-dimensional data. This is because one way to reduce the size of a representation is to find and remove redundancies.

Identifying and removing more redundancy enables the dimensionality reduction algorithm to achieve more compression while discarding less information. The notion of representation and classification is one of the central themes of deep learning and, therefore, one of the central themes in my thesis. In this section, we develop examples of representation learning algorithms that XGBoost will use inside the predicting code. The remaining chapters introduce and explain the principles features and functions of the XGBoost deep learning method I use in my thesis.

3.6 XGBoost

3.6.1 Introduction to Boosted Trees

XGBoost stands for "Extreme Gradient Boosting". The gradient boosted trees have been around for a while, and there are many materials on the topic. This chapter will explain boosted trees in a self-contained and principled way using the elements of supervised learning explained before.

3.6.2 Elements of Supervised Learning

XGBoost is used for supervised learning problems, where we use the training data (with multiple features) x_i to predict a target variable y_i. Before we learn about trees specifically, so just start to recall some basic elements in supervised learning.

3.6.3 Model and Parameters

The **model** in supervised learning usually refers to the mathematical structure of by which the prediction y_i is made from the input x_i. A typical example is a *linear model*, where the prediction is given as a linear combination of weighted input features.

$$\widehat{y}_i = \sum_j \theta_j x_{ij} \tag{3.19}$$

The prediction value can have different interpretations, depending on the task, i.e., regression or classification. For example, it can be logistically transformed to get the probability of positive class in logistic regression, and it can also be used as a ranking score when we want to rank the outputs.
The parameters are the undetermined part that we need to learn from data. In linear regression problems, the parameters are the coefficients θ. Usually, we will use θ to denote the parameters, but there are many other parameters inside a model.

3.6.4 Objective Function: Training Loss + Regularization

With judicious choices for y_i, we may express a variety of tasks, such as regression, classification, and ranking. The task of **training** the model amounts to finding the best parameters θ that best fit the training data x_i and labels y_i. To train the model, we need to define the **objective function** to measure how well the model fits the training data. A salient characteristic of objective functions is that they consist two parts: **training loss** and **regularization term**:

$$obj(\theta) = L(\theta) + \Omega(\theta) \tag{3.20}$$

where **L** is the training loss function, and Ω is the regularization term.

The training loss measures how *predictive* our model is with respect to the training data. A common choice of **L** is the *mean squared error* , which is given by

$$L(\theta) = \sum_i (y_i - \widehat{y}_i)^2 \tag{3.21}$$

Another commonly used loss function is logistic loss, to be used for logistic regression:

$$L(\theta) = \sum_i [y_i \ln(1 + e^{-\widehat{y}_i}) + (1 - y_i) \ln(1 + e^{\widehat{y}_i})] \tag{3.22}$$

The **regularization term** controls the complexity of the model, which helps us to avoid overfitting. This sounds a bit abstract, so I will consider the following problem in the following figure 3.7. Try to *fit* visually a step function given the input data points on the upper left corner of the image. Which solution among the three is the best fit?

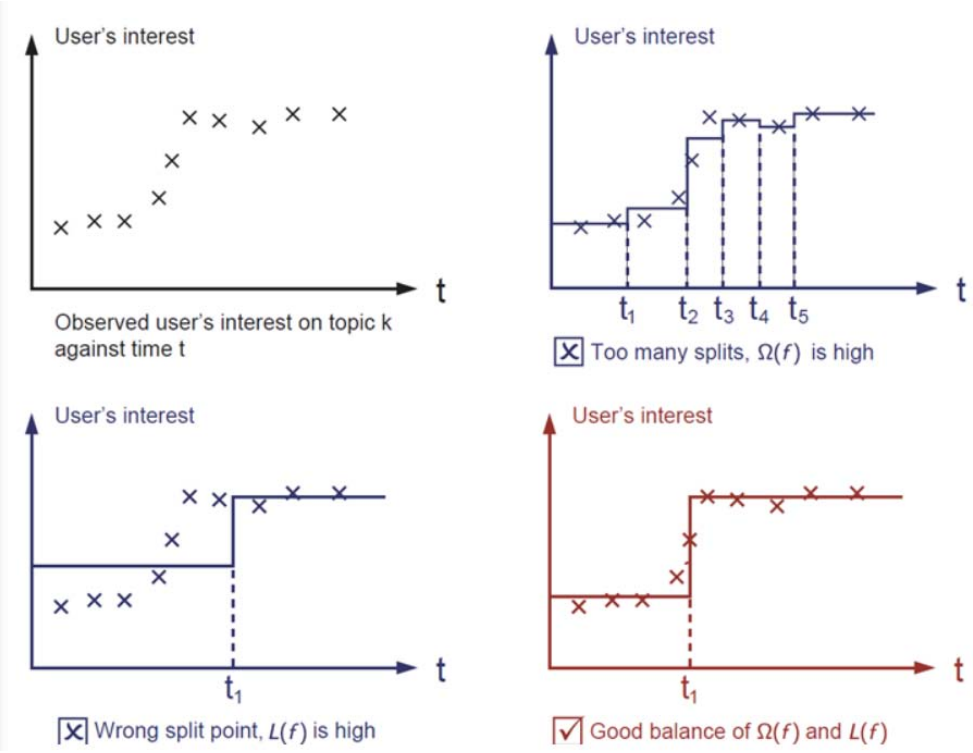

FIGURE 3.7: Example of a step function

The correct answer is marked in red. The general principle is we want both a *simple* and *predictive* model. The tradeoff between the two is also referred to as **bias-variance tradeoff** in machine learning.

3.6.5 Decision Tree Ensembles

Now that we have introduced the elements of supervised learning, let's start with real trees. To begin, it is necessary to learn about the model choice of XGBoost: **decision tree ensembles**. The tree ensemble model consists of a set of classification and regression trees (CART). Below in Figure 3.8, there is a simple example of a CART.

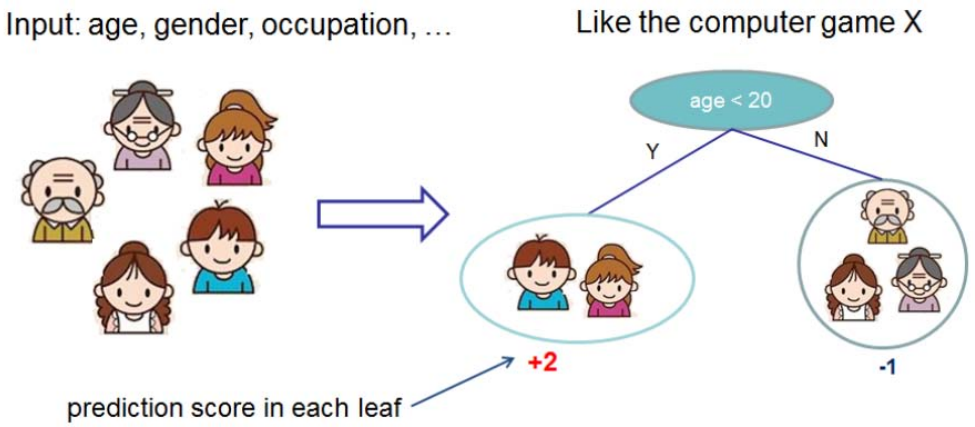

FIGURE 3.8: classification example

We classify the members of a family into different leaves and assign them the score on the corresponding leaf. A CART is a bit different from decision trees, in which the leaf only contains decision values. In CART, a real score is associated with each of the leaves, which gives us richer interpretations that go beyond classification. This also allows for a principled, unified approach to optimization, as we will see in a later part of this chapter.

Usually, a single tree is not strong enough to be used in practice. What is actually used is the ensemble model, which sums the prediction of multiple trees together.

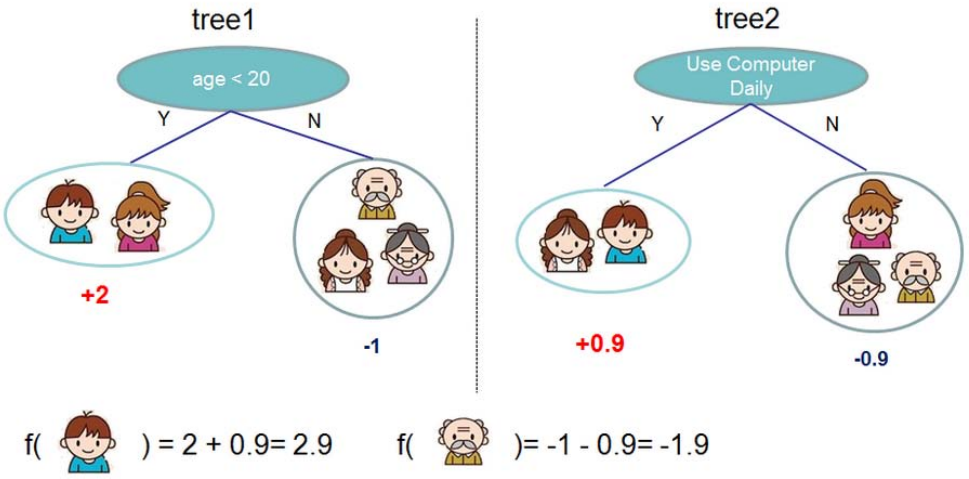

FIGURE 3.9: Multiple decision trees

In figure 3.9, there is an example of a tree ensemble of two trees. The prediction scores of each individual tree are summed up to get the final score. If you look at the example, an important fact is that the two trees try to *complement* each other. Mathematically, we can write our model in the form:

$$\widehat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in F \tag{3.23}$$

where **K** is the number of trees, **f** is a function in the functional space **F**, and **F** is the set of all possible CARTs. The objective function to be optimized is given by:

$$obj(\theta) = \sum_{i}^{n} l(y_i, \widehat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{3.24}$$

Now we can understand the type of model used in XGBoost when we refer to **random forests**. Tree ensembles methods are used. So we understand that random forests and boosted trees are the same models. The difference arises from how we train them. This means that, if you write a predictive service for tree ensembles, you only need to write one, and it should work for both random forests and gradient boosted trees.

3.7 Tree Boosting

Now that the models are introduced, the next step is to train them. The way to proceed is, as always for all supervised learning models: *we should define an objective function and optimize it!*

The objective function will be defined as the following one. The function has to contain the training loss and the regularization term:

$$obj = \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) \tag{3.25}$$

3.7.1 Additive Training

The first issue we need to fix is the **parameters** of the trees. It's easy to understand that what we need to learn are those functions f_i, each containing the structure of the tree and the leaf scores. Learning tree structure is much harder than a traditional optimization problem where you can take the gradient. It is intractable to learn all the trees at once. Instead, we use an additive strategy: fix what we have learned, and add one new tree at a time. We write the prediction value at step t as $\widehat{y}_i^{(t)}$.

Then we have:

$$\widehat{y}_i^{(0)} = 0 \tag{3.26}$$

$$\widehat{y}_i^{(1)} = f_1(x_i) = \widehat{y}_i^{(0)} + f_1(x_i) \tag{3.27}$$

$$\widehat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \widehat{y}_i^{(1)} + f_2(x_i) \tag{3.28}$$

$$... \tag{3.29}$$

$$\widehat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \widehat{y}_i^{t-1} + f_t(x_i) \tag{3.30}$$

During this iteration, we have to understand which tree do we want at each step. The natural thing is to add the one that optimizes our objective function.

$$obj^{(t)} = \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) \tag{3.31}$$

$$= \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant \tag{3.32}$$

If we consider using mean squared error (MSE) as our loss function, the objective becomes:

$$obj = \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant \tag{3.33}$$

The form of MSE is friendly, with a first-order term (usually called the residual) and a quadratic term. For other losses of interest (for example, logistic loss), it is not so easy to get such a gentle form. So in the general case, we take the *Taylor expansion* of the loss function up to the second-order:

$$obj^{(t)} = \sum_{i=1}^{n} [l(y_i, \widehat{y}_i^{(t-1)}) + g_i f_t(x_i)) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + constant \tag{3.34}$$

where the g_i and h_i are defined as:

$$g_i = \partial_{\widehat{y}_i^{(t-1)}} l(y_i, \widehat{y}_i^{(t-1)}) \tag{3.35}$$

$$h_i = \partial^2_{\widehat{y}_i^{(t-1)}} l(y_i, \widehat{y}_i^{(t-1)}) \tag{3.36}$$

After we remove all the constants, the specific objective at step **t** becomes:

$$\sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{3.37}$$

This becomes our optimization goal for the new tree. One important advantage of this definition is that the value of the objective function only depends on g_i and h_i. This is how XGBoost supports custom loss functions. We can optimize every loss function, including logistic regression and pairwise ranking, using the same solver that takes g_i and h_i as input.

3.7.2 Model Complexity

We have introduced the training step, but there is one crucial thing, the regularization term. We need to define the complexity of the tree $\Omega(f)$. To do so, let us first refine the definition of the tree $f(x)$ as:

$$f_t(x) = \omega_{q(x)}, \omega \in \Re^T, q : \Re^d \to \{1, 2, ..., T\} \tag{3.38}$$

Here ω is the vector of scores on leaves, **q** is a function assigning each data point to the corresponding leaf, and **T** is the number of leaves. In XGBoost, we define the complexity as:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2 \tag{3.39}$$

Of course, there is more than one way to define the complexity, but this one works well in practice. The regularization is one part that usually people are less carefully, or ignore. This is because the traditional treatment of tree learning only emphasized improving impurity, while the complexity control was left to heuristics. By defining it formally, we can get a better idea of what we are learning and obtain models that perform well in the wild.

3.7.3 The Structure Score

Here is the central part of the derivation. After re-formulating the tree model, we can write the objective function as:

$$obj^{(t)} \approx \sum_{i=1}^{n} [g_i \omega_{q(xi)} + \frac{1}{2} h_i \omega_{q(xi)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2 \tag{3.40}$$

$$= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i)\omega_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda)\omega_j^2] + \gamma T \tag{3.41}$$

Where $I_j = \{i | q(x_i) = j\}$ is the set of indices of data points assigned to the j_{th} leaf. Notice that in the second line we have changed the index of the summation because all the data points on the same leaf get the same score. We could further compress the expression by defining $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$:

$$obj^{(t)} = \sum_{j=1}^{T} [G_j \omega_j + \frac{1}{2}(H_j + \lambda)\omega_j^2] + \gamma T \tag{3.42}$$

In this equation, ω_j are independent with respect to each other, the form $G_j\omega_j + \frac{1}{2}(H_j + \lambda)\omega_j^2$ is quadratic and the best ω_j for a given structure $q(x)$ and the best objective reduction we can get is:

$$\omega_j^* = -\frac{G_j}{H_j + \lambda} \tag{3.43}$$

$$obj^* = -\frac{1}{2}\sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{3.44}$$

The last equation measures *how good* a tree structure $q(x)$ is.

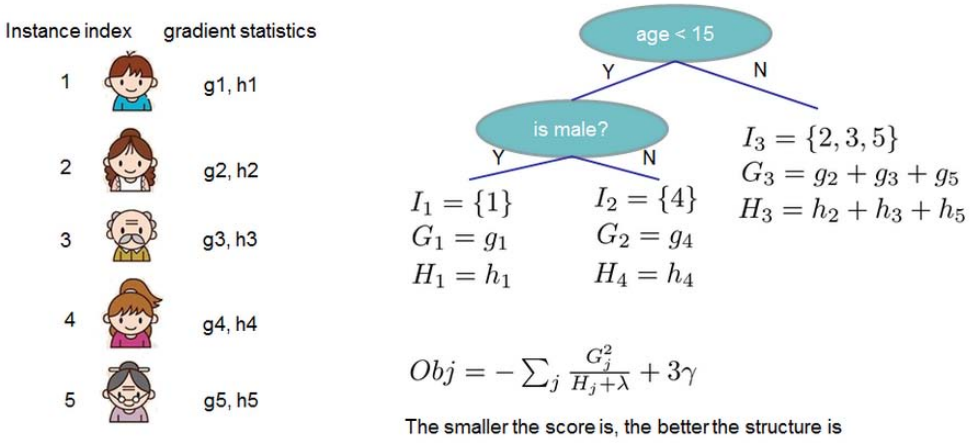

FIGURE 3.10: XGBtree tree classification score.

If all this sounds a bit complicated, let's take a look at the picture and see how the scores can be calculated. Basically, for a given tree structure, we push the statistics g_i and h_i to the leaves they belong to, sum the statistics together, and use the formula to calculate how good the tree is. This score is like the impurity measure in a decision tree, except that it also takes the model complexity into account.

3.7.4 Learn the tree structure

Now that we have a way to measure how good a tree is, ideally we would enumerate all possible trees and pick the best one. In practice, this is intractable, so we will try to optimize one level of the tree at a time. Specifically, we try to split a leaf into two leaves, and the score it gains is:

$$Gain = \frac{1}{2}[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{(H_L + H_R + \lambda}] - \gamma \tag{3.45}$$

This formula can be decomposed as:

1. The score on the new left leaf
2. The score on the new right leaf
3. The score on the original leaf
4. The regularization on the additional leaf

We can see an important fact here: if the gain is smaller than y, we would do better not to add that branch. This is precisely the **pruning** technique in tree-based models! By using the principles of supervised learning, we can naturally come up with the reason these techniques work.

For real-valued data, we usually want to search for an optimal split. To efficiently do so, we place all the instances in sorted order, like the following picture.

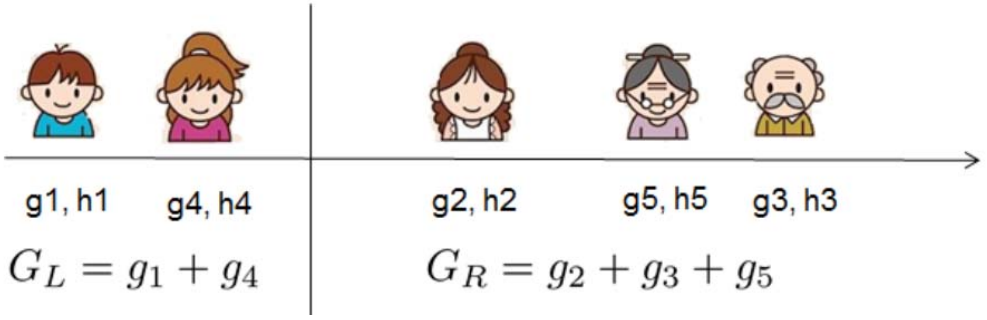

FIGURE 3.11: structure score of possible split solutions.

3.7.5 Limitation of additive tree learning

Since it is intractable to enumerate all possible tree structures, we add one split at a time. This approach works well most of the time, but some edge cases fail due to this approach. For those edge cases, training results in a degenerate model because we consider only one feature dimension at a time.

In conclusion, XGBoost is precisely a tool motivated by the formal principle introduced before. More importantly, it is developed with both deep learning consideration in terms of systems optimization and principles for machine learning. The goal of this discussion is to understand and improve until the extreme of the computation limits of machines to provide a scalable, portable and accurate full ML code.

3.8 DART booster

XGBoost mostly combines a huge number of regression trees, as we saw with a small learning rate. In this situation, trees added early are significant, and trees added late are unimportant. Vinayak and Gilad-Bachrach proposed a new method to add dropout techniques to boosted trees and reported better results in some situations.

This new method implemented inside XGBoost is an instruction of a new tree booster called **Dart Booster**.

3.8.1 Features

- Drop trees are built to solve the over-fitting problem.
- Trivial trees, to correct trivial errors, may be prevented.

Because of the randomness introduced in the training model, we expect the following few differences while the code is running:

- training can be slower than **gbtree** because the random dropout prevents the usage of the prediction buffer.
- The early stop might not be stable due to the randomness considered data for the building model.

3.8.2 How it works

In m_{th0} training round, suppose k trees are selected to be dropped.

Let $D = \sum_{i \in K} F_i$ be the leaf scores of dropped trees and $F_m = \eta \tilde{F_m}$ be the leaf scores of a new tree.

The objective function is as follows:

$$Obj = \sum_{j=1}^{n} L(y_j, \hat{y}_j^{m-1} - D_j + \tilde{F_M}) + \Omega(\tilde{F_m}) \tag{3.46}$$

D and F_m are overshooting, so using a scale factor like:

$$\hat{y}_j^m = \sum_{i \in \kappa} F_i + a(\sum_{i \in \kappa} F_i + bF_m). \tag{3.47}$$

3.8.3 Parameters

The booster *dart* inherits *gbtree* booster, so it supports all parameters that gbtree does, such as *eta*, *gamma*, *max depth* that we will explain in the next chapter. For example, some additional parameters for the concrete use for Dart Booster are noted below:

- **sample type:** type of sampling algorithm.

  uniform: dropped trees are selected uniformly.

  weighted: dropped trees are selected in proportion to weight.

- **normalize type:** type of normalization algorithm.

  tree: New trees have the same weight of each of dropped trees.

$$\begin{aligned} a(\sum_{i\in\kappa} F_i + \frac{1}{k}F_m) &= a(\sum_{i\in\kappa} F_i + \frac{\eta}{k}\tilde{F_m}) \\ &\sim a(1+\frac{\eta}{k})D \\ &= a\frac{k+\eta}{k}D = D, \\ a &= \frac{k}{k+\eta} \end{aligned} \tag{3.48}$$

  forest: New trees have the same weight of sum of dropped trees (forest).

$$\begin{aligned} a(\sum_{i\in\kappa} F_i + F_m) &= a(\sum_{i\in\kappa} F_i + \eta\tilde{F_m}) \\ &\sim a(1+\eta)D \\ &= a(1+\eta)D = D, \\ a &= \frac{1}{1+\eta}. \end{aligned} \tag{3.49}$$

- **rate drop:** dropout rate, range: [0.0 , 1.0].

- **skip drop:** probability of skipping dropout if a dropout is skipped, new trees are added in the same manner as gbtree. Parameter Range: [0.0 , 1.0].

Chapter 4

XGBoost models

4.1 Introduction

In the previous chapters, we learned about machine learning techniques used in our days to solve daily problems in our life. As introduced in the first chapter, we will investigate the aerodynamic inside an S-Duct with the machine learning method. During the previous studies done by DalMagro [6], many CFD simulations were made on the geometry we already have shown before. These CFD simulations are considered inside the optimization loop as a high fidelity model. The variable's values coming out from the CFD optimization loop will be all taken as the most reliable one.

As we know, a complex CFD optimization loop will take months to be run; this is why we want to set up and run machine learning models built from CFD data. As we can imagine, the results coming out from these models has less reliability than the CFD ones. The ML (Machine Learning) techniques will be used to learn from the CFD data we give in input to predict results, always reducing the model Mean Squared Error.

4.2 Machine Learning models

Different machine learning models have been built inside the code to predict the most reliable as possible the variables inside our S-Duct; the Cp and the swirl. To do this, XGBoost allows us to set many parameters inside the code to fit better the data we are given. We will discuss through the chapter about why we choose some parameters instead of others. Many parameters have values correlated; this allows us to control how the model will be built. Different configurations of the same model also have been run during the code test to find the best model for our dataset with the smallest Mean Squared Error.

Many different variables will be taken into account, building the models because every single one has to fit the variable we want to learn in the best way possible. The models will be trained with a certain amount of random data from the CFD dataset and then tested on the remaining one. A precise number of data will be used to train the models because we want the smallest error possible on the prediction process. This part will be explained further in the chapter going through the conclusions of which model is the best to run the optimization loops with. Once the model has been built, we have to know how good it is in terms of how well the specific configuration fits the data we give in the input. This validation process has been done looking at the smallest MSE but also from a statistic point of view. Assuming the dataset from the CFD simulation the most reliable, we calculated the average and the standard deviation for the two variables we are studying (Cp, Swirl). These numbers calculated are used to build a gaussian distribution of our training data. When the model is done, we have to test it predicting the two variables (Cp, Swirl), giving in input the geometries we already have in the same training dataset file. The point now is to verify how close are the predicted values from the one we already have inside the test dataset file. The test dataset is made splitting the original CFD data; some random geometries are used to build the model and the rest to test it. After this step, we can change the number of random training geometries used to build the model to decrease the MSE.

To implement this idea, firstly, we split the original CDF dataset with 911 geometries examples using 451 random geometries to train the Cp and the Swirl model. The remaining geometries from the CDF dataset (460) are used to test the above models. This splitting number came out doing different iterations, always reducing the MSE increasing the prediction precision.

4.3 First Machine Learning Configuration (reg:tweedie)

The first model built is the simplest one; only five parameters will be set to control it. We will explain these parameters to be able to compare the other models built below.

4.3.1 Model parameters

- **parameters:** reg:tweedie, tweedie variance power, gbtree, num parallel tree, max depth

  The objective regression function use in the first model is called **reg:tweedie**. This parameter specifies the learning task and the corresponding learning objective of the model.

  The second parameter used is called **Tweedie variance power**. What it does is to control the data variance var(y) with the tweedie distribution statistic variance in the model as follow:

$$var(y) \sim E(y)^{tweedie\ variance\ power} \tag{4.1}$$

  The next configuration parameter used to set the booster code inside the model is the **gbtree**. This parameter, as we spoke in the previous chapters, allows the model to classify the data as a tree, growing it up and splitting the data classification into branches through different ways, also giving weights to the leaves.

The **number of parallel trees** parameter is another function very useful; it allows the model to be built with more than one classification tree. Every classification tree is built in similar ways, but everyone will be different due to the random choice of training geometries from the CFD database.

The last parameter used in this model is the **max depth** number. Increasing this value will make the model more complex but also more likely to overfit. It can also be set to 0, but it indicates we have no limit on the depth of the model. Beware that XGBoost aggressively consumes memory when training a deep tree. This parameter can be set from 1 $\rightarrow$ inf.

4.3.2 Improving trained model

When the configuration is done, we need to check the model built. In other words, we need to test the model and see how reliable it is. To do this, in Figure 4.1 is shown in red, the S-duct pressure recovery coefficient calculated during the optimization process done by DalMagro.[6]. On the same plot, in black, we can find the values predicted with this machine learning method (reg:tweedie). We can observe that the black line can interpolate the Cp geometry's values smoothly. The CFD Cp value change in a range between 0.04 and 0.065, and because of the geometries random choice, it can vary from one example to the next one. The ML interpolates the Cp cutting out the peaks trying to stay stable in a central range. On the second plot show in Figure 4.2, we can figure out how many Cp values predicted with the ML method are close to the real ones calculated in the optimization process with the CFD. In this model, the test is essential to remember that the geometries used to predict the Cp with the ML are the ones recorded inside the CDF dataset not used for the training model process. This evaluation comparison is made using formula 4.2, taking the same geometry from the dataset and evaluate the difference between the CFD result and the ML prediction in percentage (prediction accuracy).

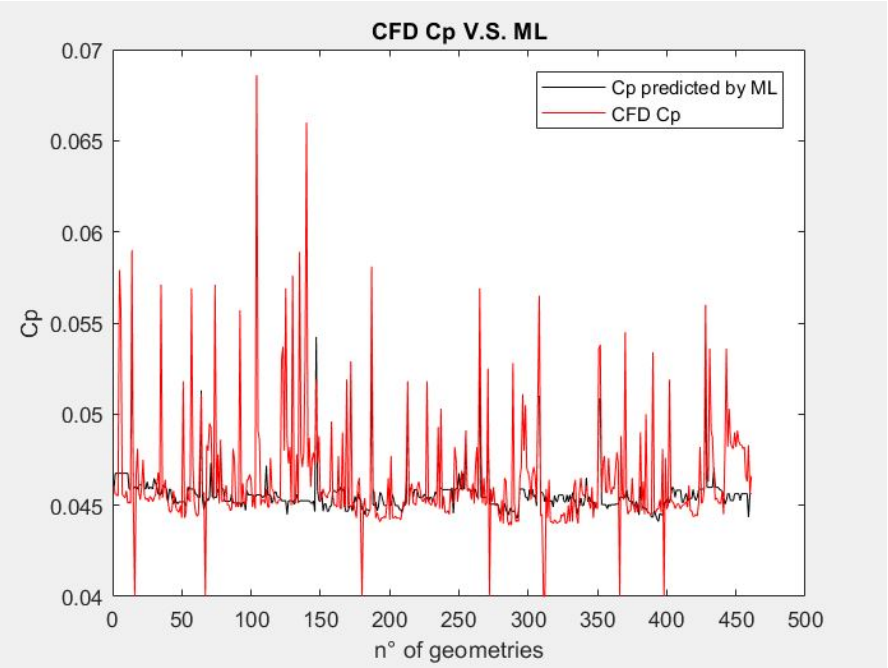

FIGURE 4.1: In red is shown the pressure recovery coefficient predicted by the CFD simulations for 460 geometries. In black is shown the pressure recovery coefficient for the same 460 geometries predicted by the ML model.

To calculate how close are the values in percentage form each other we use:

$$accuracy(\%) = \frac{Cp_{CFD} - Cp_{ML}}{Cp_{CFD}} * 100 \tag{4.2}$$

All the values calculated with formula 4.2 are plotted in Figure 4.2. The consideration we can do is that more than 250 of 460 testing geometries have the ML predicted value very close to the real one (CFD) with an accuracy close to 100%. Only 100 geometries have a ML Cp prediction with an accuracy between 96% and 98%.

The second variable we want to test the model with is the Swirl angle. Following in Figure 4.3, we can observe how this model fits this variable. Like in the last case, we have in red from the test dataset the CFD Swirl angle and in black, the ML predicted one on the same geometry.

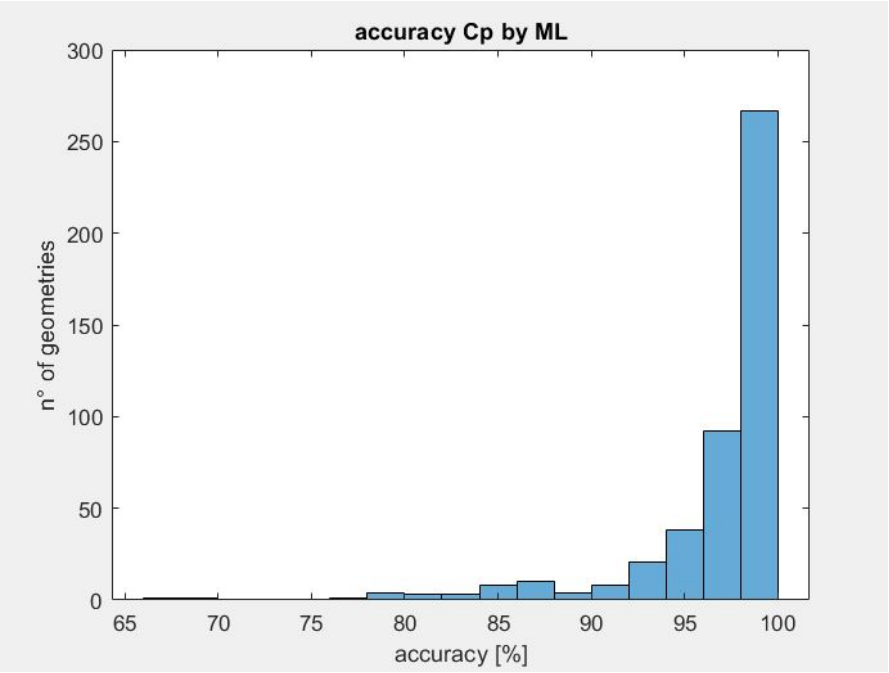

FIGURE 4.2: The model accuracy checking the differece between the Cp values ML predicted and the CFD one.

We can observe that the black line interpolates the Swirl geometries values. The CFD Swirl value change in a range between 2.5 and 5. Because of the random geometry choice, the Swirl number can vary from one example to the next one. Now the same model fits differently this new variable producing a different range of accuracy for the testing geometries we are analyzing. In figure 4.4, we can appreciate this result.

As done before for the pressure recovery coefficient, in Figure 4.4 is plotted the model accuracy that allows us to see how many geometries are fitted reaching the highest accuracy possible.
Observing figure 4.4, we can say that fewer geometries are fitted due to the less precision of the model fitting the CFD Swirl angle. As we saw, this model can fit very well the Cp but in a less precise way the swirl. In this case, the error value between the ML predicted value and the CFD one is higher. Only 100 geometries rich the 100% of accuracy and 150 geometries are fitted for the Swirl angle by the 90%. The rest of the testing geometries are fitted with an accuracy less than 90%.

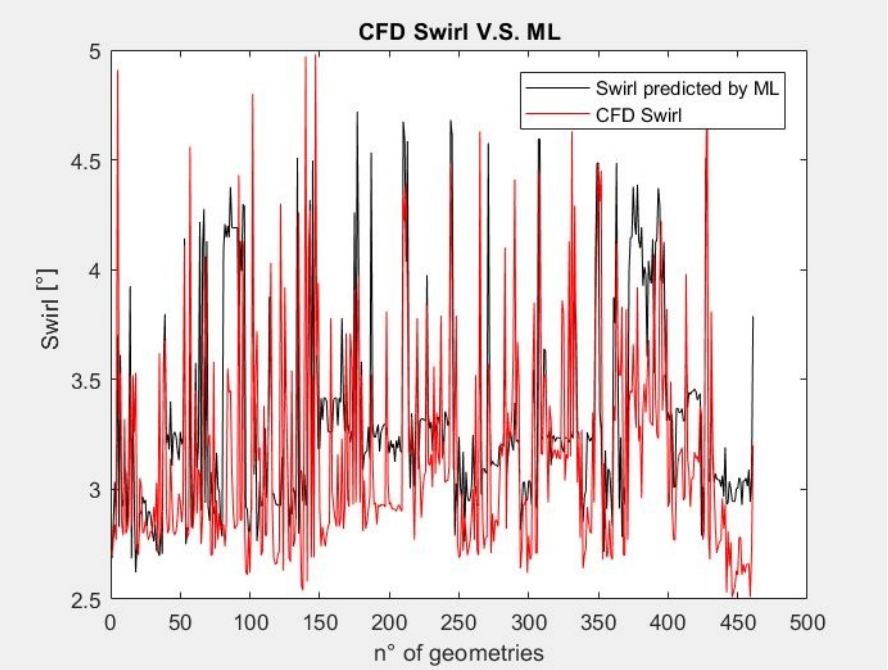

FIGURE 4.3: In red is shown the Swirl angle calculated by the CFD simulations for 460 geometries. In black is shown the Swirl angle for the same 460 geometries predicted by the ML model.

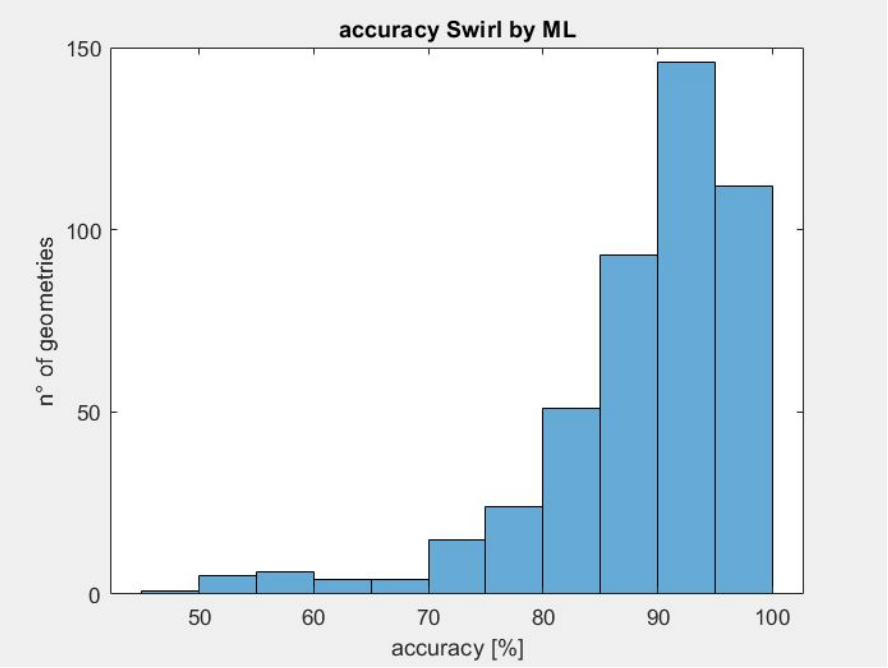

FIGURE 4.4: In this Figure, we can see the model accuracy checking the Swirl values precision between the ML predictions and the CFD ones.

4.3.3 Model test

Analyzing the model precision we built, we can check the values of the different errors introduced in the previous chapters. The training loss function measures how *predictive* our model is with respect to the training data. A common choice for the error estimation is the *mean squared error*, or another commonly used loss function is the *logistic loss*.

- Mean Squared Error (MSE) calculated for the two variables we are predicting:

$$L(\theta) = \sum_i (y_i - \widehat{y}_i)^2 \tag{4.3}$$

$$\begin{aligned} Pressure\,recovery\,MSE &= 0.0041 \\ Swirl\,MSE &= 39.259 \end{aligned} \tag{4.4}$$

- logistic loss (Logloss) calculated for the two variables we are predicting:

$$L(\theta) = \sum_i [y_i \ln(1 + e^{-\widehat{y}_i}) + (1 - y_i) \ln(1 + e^{\widehat{y}_i})] \tag{4.5}$$

$$\begin{aligned} Pressure\,recovery\,Logloss &= 256.1596 \\ Swirl\,Logloss &= 329.1983 \end{aligned} \tag{4.6}$$

This error is a dimensionless number that must be minimized as much as possible because we want the model as strong as possible. The *mean squared error* tells us the sum of the distances between all the data test and the predicted one. This calculation allows us to know how close are the predicted ML data from the CFD one during the test.
As we can see in List 4.4, we have the minimal MSE for the Cp and the Swirl model we implemented now. Comparisons will be made with the subsequent models. For now, we can check that from this error numbers this model will be used to predict the Cp and Swirl for new geometries in our Machine Learning optimization process explained later in chapter five.

Another study through the prediction results made from the model is to check which kind of Gaussian distribution this testing result has. The goal is to see the distance between the CFD gaussian distribution curves calculated from the CFD data and from the ML predictions. The reason for this calculation is to discover where is the average from the CFD data and the ML predictions. In other words, we will discover how many predicted ML data would be inside the 3σ, 2σ, and σ range of the above Gaussian distribution built from the CFD data test.
In the next two figures, 4.5 and 4.6, we can discover the ML distribution.

$$\begin{aligned} \sigma &\rightarrow \mu - \sigma \ \& \ \mu + \sigma \\ 2\,\sigma &\rightarrow \mu - 2\sigma \ \& \ \mu + 2\sigma \\ 3\,\sigma &\rightarrow \mu - 3\sigma \ \& \ \mu + 2\sigma \end{aligned} \tag{4.7}$$

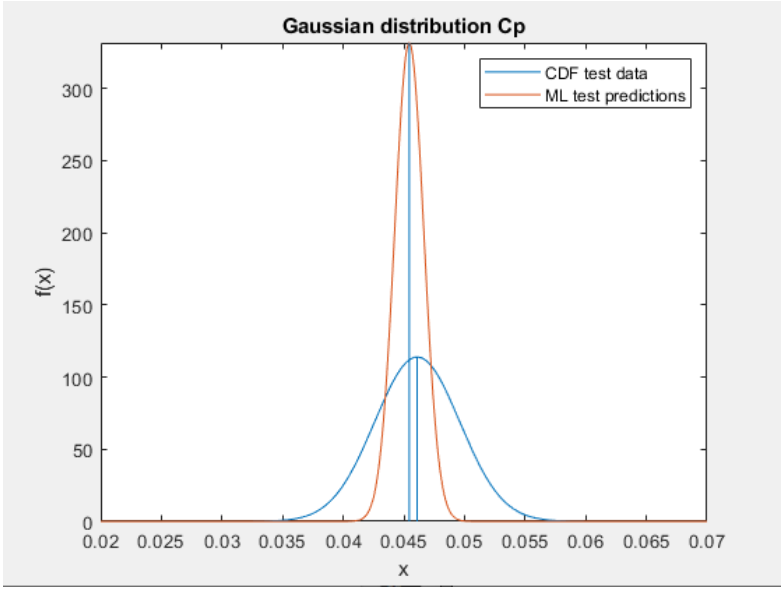

FIGURE 4.5: Observing in blue the pressure recovery gaussian distribution from the CFD data . In red, we can observe the Cp ML prediction distribution.

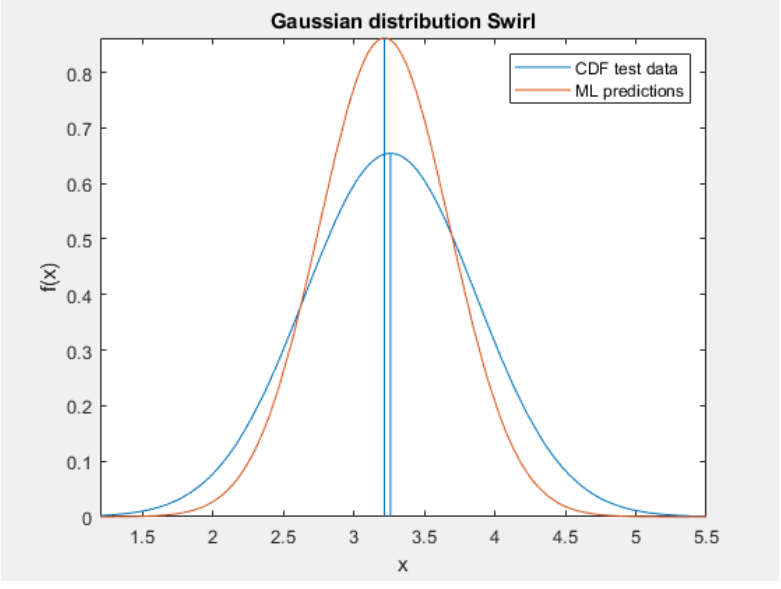

FIGURE 4.6: In blue, we have the Swirl gaussian distribution from the CFD data. In red, we can observe the Swirl angle ML prediction distribution.

Following the discussion is interesting to see how the effective numbers of geometries are fitted under the CFD gaussian curves. As we explained before, the test dataset is made by 460 geometries for the Cp and Swirl.

In table 4.1 is shown in percentage the amount of ML predicted geometries that are fitted under the CFD gaussian curve. For example, on the first line, we can see that all the 100% Cp values predicted from the ML are inside the CFD range $(\mu - 3\sigma \,\&\, \mu + 3\sigma)$, where μ is the average value, and σ is the standard deviation. Going through the Cp predictions, we can observe that in the CDF $(\mu - 2\sigma \& \mu + 2\sigma)$ range we have the 95.349% of the 460 geometries predicted with the ML are inside. The last range $(\mu - \sigma \,\&\, \mu + \sigma)$ under the CFD curve, just the 94.398% of the ML predictions are fitted. As we analyze the Cp case, we can do the same thing with the swirl. Looking the table 4.1 below is easy to observe that this model fits very well the CFD Swirl number and also the Cp one. The 93.732% of the test geometries are fitted under the CFD gaussian distribution inside the $(\mu - \sigma \,\&\, \mu + \sigma)$ range for the Swirl case.

Table 4.1: ML geometries percentage fitted under the CFD gaussian distribution.

Variable	3σ	2σ	σ
Cp	100%	95.349%	94.398%
Swirl	100%	98.095%	93.732%

4.4 Second Machine Learning Configuration booster:dart

The second model built runs a new method to split the data organization trees implemented by Vinayak and Gilad-Bachrach. As we introduce it in chapter three, the big difference now is that trees added earlier during the model building process are more significant than trees added later that becomes unimportant. This new method implemented inside XGBoost is an instruction of a new tree booster called *booster:dart*.

4.4.1 Model parameters

- **parameters:** reg:gamma, normalize type, rate drop, skip drop, dart, max depth.

  The objective regression function use in this second model is called **reg:gamma**. This parameter, like the one in the first model, specifies the learning task and the corresponding learning objective of this new model.

  This second parameter **normalize type** means the type of normalization algorithm we have in the model building process. If this parameter is set to "Tree", it means that the new trees built through the iterations have the same weight for each of the dropped trees during the drop out process.

  This parameter called **rate drop** can manage the drop out process observing the learning rate during the model learning process. This parameter is set at the beginning to 0, but it can be changed in a range between $0 \rightarrow 1$.

  This parameter called **skip drop** can change skipping the dropout probability procedure during a boosting iteration. If a dropout is skipped, new trees are added in the same manner as gbtree.

  The **Dart booster** parameter we already show and explained in the last chapter.

The parameter **Max depth** means the maximum depth of a tree during the iteration process. Increasing this value will make the model more complex and more likely to overfit.

4.4.2 Improving trained model

With this second configuration done, we need again to check how reliable the model is. To do this, we have to test it with the remaining data in the dataset file. The number of examples used to build, train and test the model is always the same explained in this chapter introduction. The choice for the dataset splitting is made doing different iterations between the same model in the training process. This iteration is done until testing the model; we get to a good prediction accuracy minimizing the MSE.

Following the same treatment did for the first model, in Figure 4.7 is shown in red the pressure recovery coefficient for the S-duct geometry found during the CFD optimization process done by DalMagro.[6] On the same plot, we can find in black the values calculated with this machine learning method (Dart booster).
On the second plot shown in Figure 4.8, always about the S-duct Cp, we can figure out how many Cp predicted values calculated with ML are close to the real one calculated with the optimization process using CFD. In this model, the test is essential to remember again that the geometries used to predict the Cp with the ML are the ones recorded inside the CDF dataset not used for the training model process. This evaluation comparison is made always using formula 4.8.

To calculate how close are in percentage the values form each other we use:

$$accuracy(\%) = \frac{Cp_{CFD} - Cp_{ML}}{Cp_{CFD}} * 100 \tag{4.8}$$

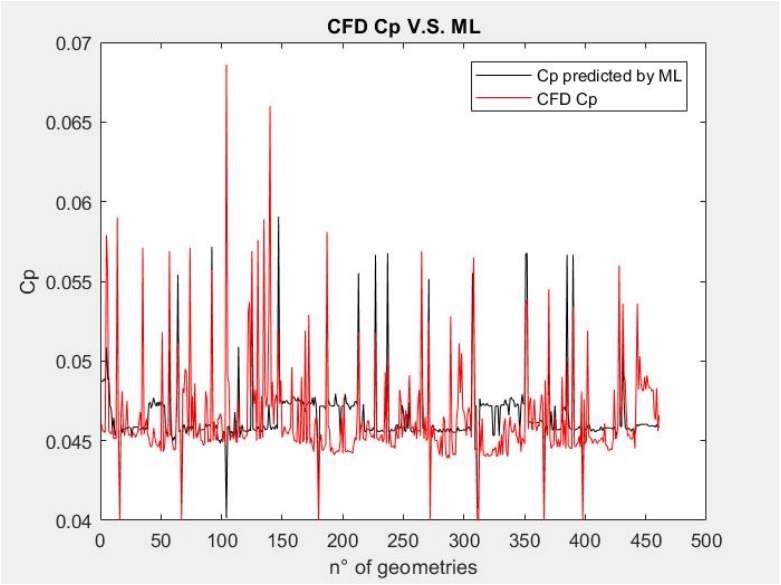

FIGURE 4.7: Drawn in red is shown the pressure recovery coefficient predicted by the CFD simulations for 460 geometries. In black is shown the pressure recovery coefficient for the same 460 geometries predicted by the ML model.

All the values calculated with formula 4.8 are plotted in Figure 4.8. The consideration we do is that almost 200/460 testing geometries have the ML predicted value very close to the CFD, one with an accuracy close to 100%. Only 100 geometries have a Cp accuracy prediction from the ML between 94 and 98%.

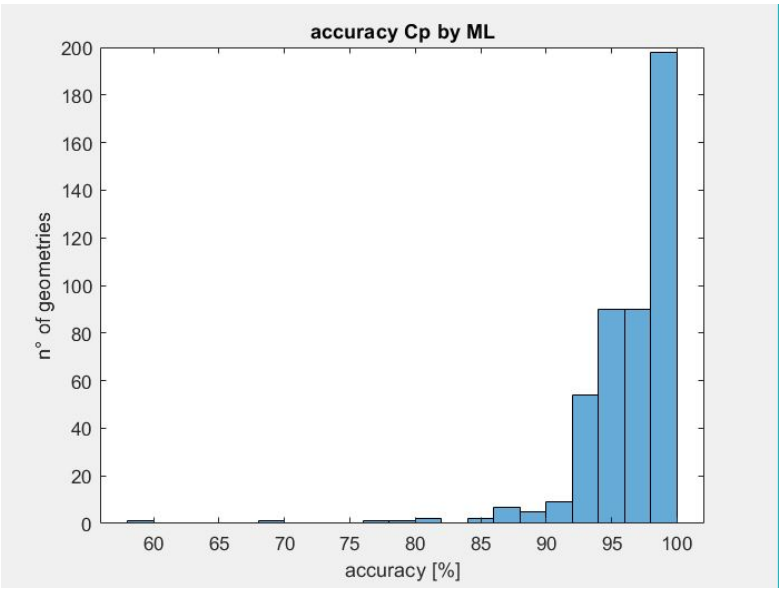

FIGURE 4.8: In this Figure, we can see the accuracy of the model checking the Cp values precision between the predicted one (ML) and the real one (CFD).

The next two Figures, 4.9 ana 4.10, will show the Swirl features calculated and observed from the machine learning model during the testing process. In Figure 4.9 is shown in red the Swirl angle from the CDF predictions, and in black is shown the ML predictions for the same geometries.

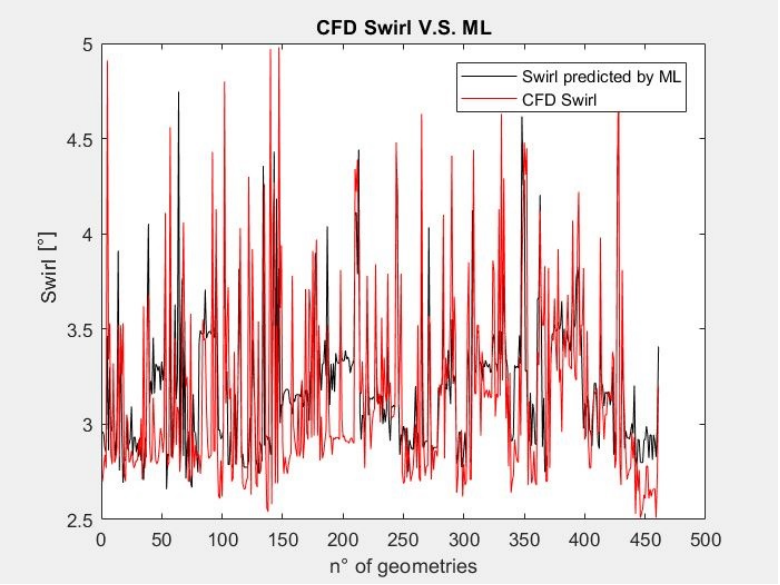

FIGURE 4.9: In red is shown the Swirl angle estimated by the CFD simulations for 460 geometries. In black is shown the Swirl angle for the same 460 geometries predicted by the ML model.

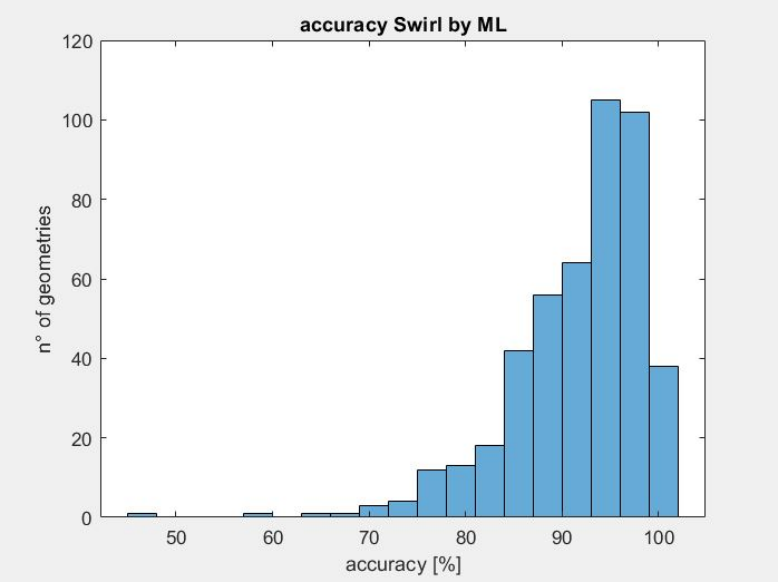

FIGURE 4.10: In this Figure, we can see the accuracy of the model checking the Swirl values precision between the predicted ones (ML) and the real ones (CFD).

As done for the pressure recovery coefficient before, we will plot in Figure 4.10 the model accuracy to see how many geometries are fitted in the correct way reaching the highest accuracy possible.

Observing figure 4.10, we observe that fewer geometries have the highest accuracy. This model can fit the Cp but also the Swirl angle. In this case, the error between the ML predicted value and the CFD one is bigger for the swirl. We have only 40 geometries riching the 100% of accuracy and 100 geometries are fitted for the swirl angle by the 90% $\rightarrow$ 95%. The rest of the testing geometries are fitted with an accuracy value less than 90%.

4.4.3 Model test

Analyzing the model precision, we can check the values for the **Mean Squared Error** and the **logistic loss** introduced in the last section. The training loss error measures again how *predictive* our model concerns the training data. The numbers from Formula 4.9 tell us the sum of the distances between all the data test and the predicted one, so we can know how close are the one from the others during the test.

- Mean Squared Error (MSE) calculated for the three variables we are predicting:

$$L(\theta) = \sum_i (y_i - \widehat{y}_i)^2 \tag{4.9}$$

$$\begin{aligned} Pressure\,recovery\,MSE &= 0.0038 \\ Swirl\,MSE &= 65.2727 \end{aligned} \tag{4.10}$$

This error is a dimensionless number that must be minimized as much as possible because we want the model as strong as possible. Different models were trained with different random input geometries, and as we can see in List 4.10, we have the minimal MSE for the Cp and the swirl we implemented now.

We can check that this model is more precise than the first one for the Cp predictions because it has the smallest MSE error; we have 0.0038 instead of 0.0041. Looking at the Swirl MSE number, we have 39.25 in the first model and 65.27 in the second one. Because of this, we can say that this second model is less precise on the predictions than the first one. Both models will be used to predict both values during the ML optimization process. Obviously, we will have less precise predictions for the swirl in the ML optimization but higher precision estimating the Cp.

The last study, through the prediction results made from the model, is to check which kind of Gaussian distribution these results have. The goal is to see the distance between the CFD gaussian distribution curves calculated from the CFD database and from the ML predictions. The reason for this calculation is to discover where are the CFD average values and the ML predictions ones. In other words, we will find how many predicted ML data would be inside the range of 3σ, 2σ, and 1 σ of the CFD gaussian distribution built from the CFD database. In the next two figures, 4.11 and 4.12, we can discover the ML distribution.

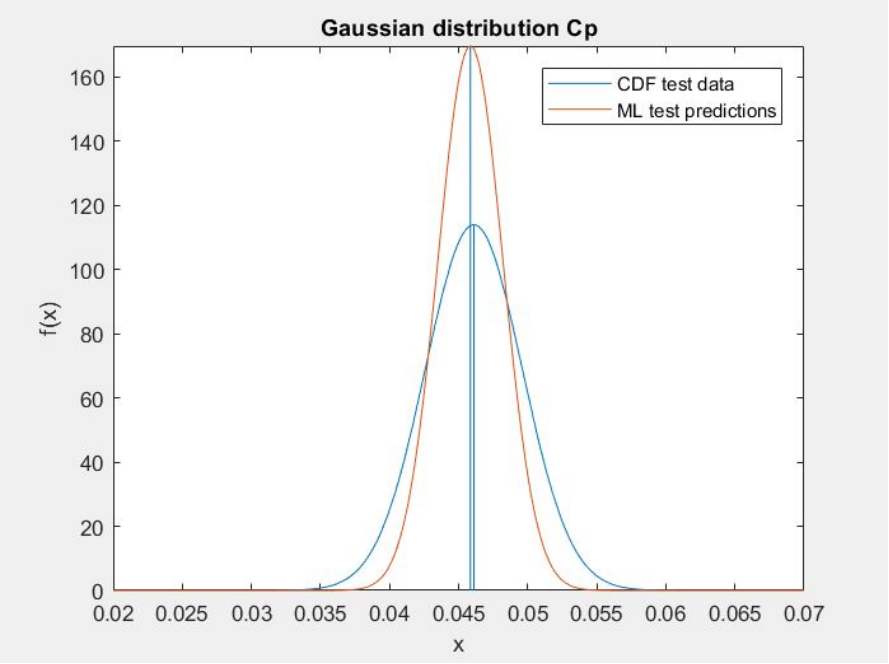

FIGURE 4.11: Observing in blue the gaussian distribution of the pressure recovery from the CFD data. In red, we can observe the Cp ML prediction distribution.

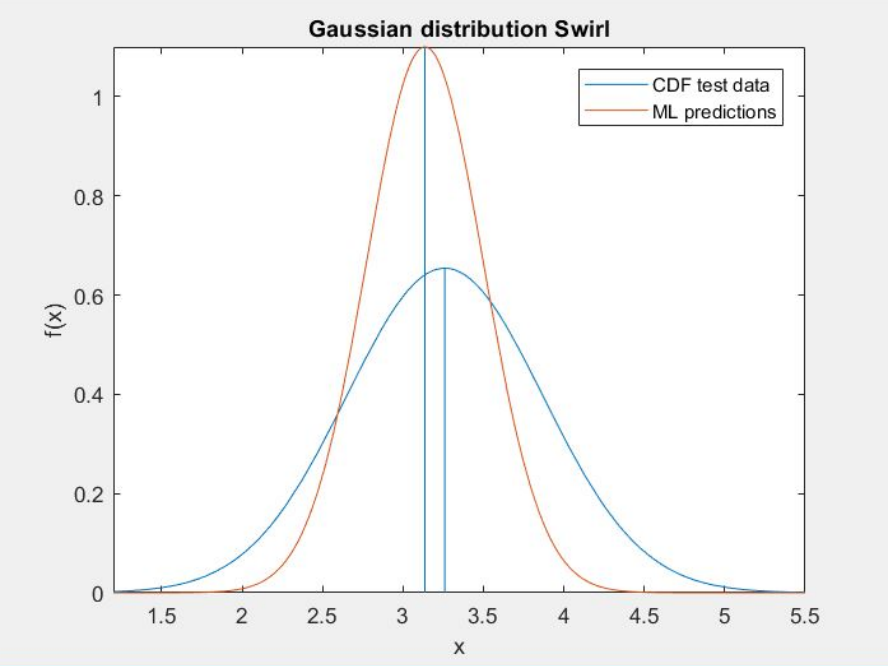

FIGURE 4.12: In blue, we have the gaussian distribution of the Swirl from the CFD data. In red, we can observe the Swirl angle ML prediction distribution.

In table 4.2 is shown in percentage the amount of ML predicted geometries fitted under the CFD gaussian curve. For example, looking to the first line, we can see that all the 100% of Cp predicted from the ML are inside the CFD range ($\mu - 3\sigma \,\&\, \mu + 3\sigma$), where μ is the average value, and σ is the standard deviation. Going through the Cp predictions, we can observe that in the CDF ($\mu - 2\sigma \& \mu + 2\sigma$) range just the 96.525% of the 460 geometries predicted with the ML are inside. The last range tighter ($\mu - \sigma \,\&\, \mu + \sigma$) under the CFD curve, just the 95.059% of the ML predictions are in this range. Concluding we can say that: comparing the MSE, this model is more precise on the Cp predictions than the first one, but on the other hand, we can say that this second model is less precise with the Swirl interpolations than the first one.

TABLE 4.2: ML geometries percentage fitted under the CFD gaussian distribution.

Variable	3σ	2σ	σ
Cp	100%	96.525%	95.059%
Swirl	100%	94.349%	80.368%

4.5 Third Machine Learning Configuration booster:gbtree

The third model implemented runs a classic method to split the data organized into different trees to be able to classify the S-duct geometries. As we introduce and explained the tree's structure in chapter three, I will go straight through the model's parameters. This method implemented inside XGBoost runs the tree booster instruction called *booster:gbtree*. Many parameters have already been explained before, and some of them will be just mentioned.

4.5.1 Model parameters

- **parameters:** reg:tweedie, tweedie variance power, gbtree, max depth.

  The objective regression function used in this third model is again **reg:tweedie**. This parameter specifies the learning task and the corresponding learning objective of the ML model.

  The second parameter used is called **Tweedie variance power**. It controls the statistic variance of the Tweedie data distribution in the model. It goes from $1 \rightarrow 2$ and by default is set equal to $1,5$.

  The other configuration parameter used to set the booster code inside the model is the **gbtree**. This parameter, as we spoke in the above paragraph, allows the model to organize the data as a tree during the classification process giving weights to the tree leaves.

  The parameter **Max depth** means the maximum depth of a tree during the iteration process. Increasing this value will make the model more complex and heavy. The range of this parameter goes from $0 \rightarrow \inf$.

4.5.2 Improving trained model

With this third configuration done, we need again to check how reliable the model will be. To do this, we have to test it as always with the remaining CFD data we already have in the CFD database file.

Following the same treatment did for the last two models before, now in the next Figure 4.13 will be shown respectively in red and black the S-duct pressure recovery coefficient found during the optimization process done by DalMagro [6] and the machine learning values calculated with this new configuration method (gbtree). In the second plot shown in Figure 4.14, we can figure out how many Cp ML predicted values are close to the real ones calculated with the optimization process using CFD.

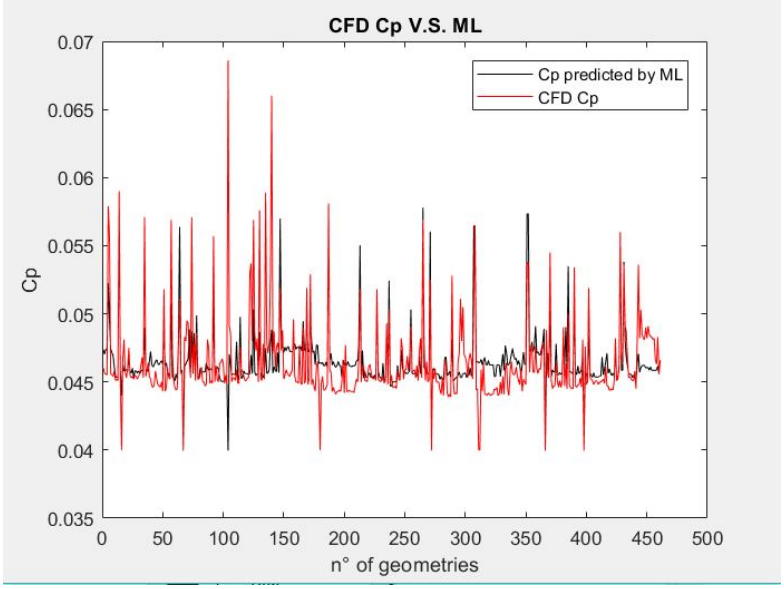

FIGURE 4.13: Drawn in red is shown the pressure recovery coefficient predicted by the CFD simulations for 460 geometries. In black is shown the pressure recovery coefficient for the same 460 geometries predicted by the ML model.

To calculate how close are the values in percentage form each other, we use the same mathematical equation explained in the last two models.

The consideration we can do now is that almost 200/460 testing geometries have the ML predicted values with an accuracy close to 100%. Only 120 and 100 geometries have an accuracy Cp prediction from the ML between 94% and 98%.

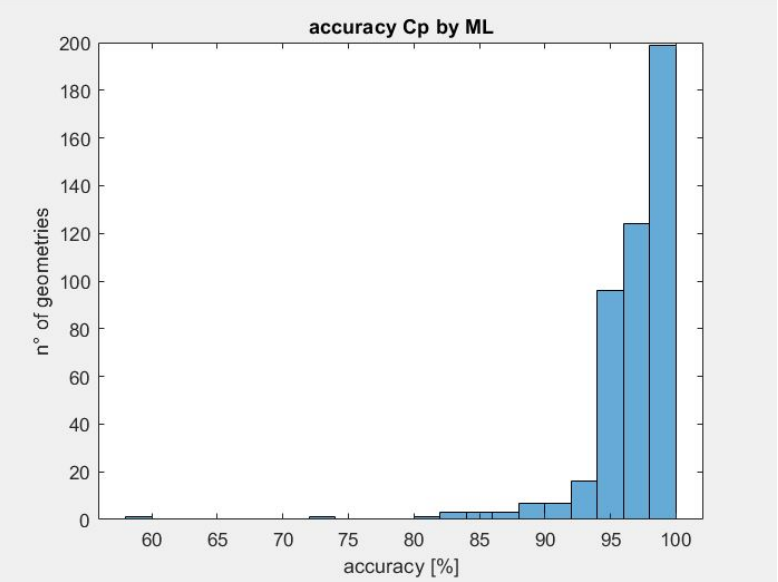

FIGURE 4.14: In this Figure, we can see the accuracy of the model checking the Cp values precision between the predicted one (ML) and the real one (CFD).

The next two Figures, 4.15 and 4.16, will show the Swirl features during the testing process. In Figure 4.15 is shown in red the Swirl angle from the CDF predictions and in black, the ML predictions for the same testing geometries.

We will plot in Figure 4.16 the Swirl model accuracy to see how many geometries are fitted in the correct way reaching the highest accuracy.

This model can fit the Swirl angle with a good approximation. In this case, only 35 geometries rich the 100% accuracy. After that, we have 100 and 90 geometries fitted by 93% $\rightarrow$ 98%. The rest of the testing geometries are fitted with an accuracy of less than 93% as shown below in Figure 4.16.

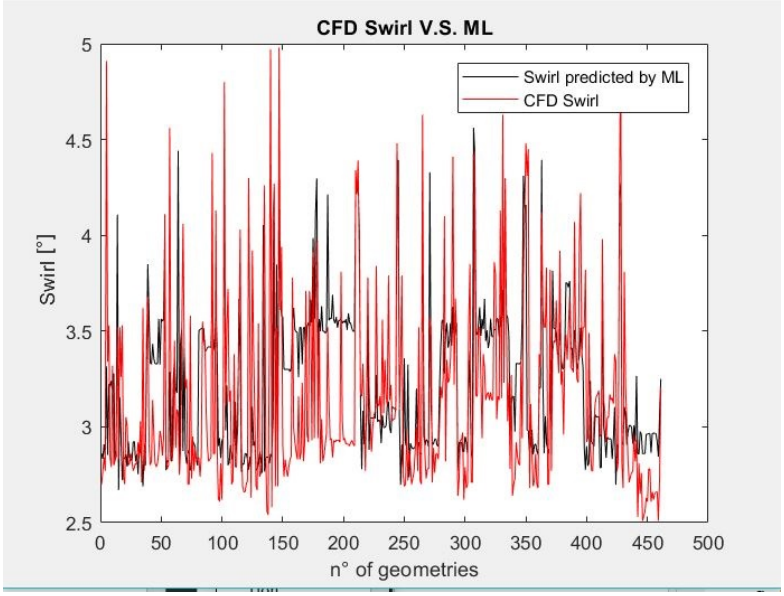

FIGURE 4.15: Drawn in red is shown the Swirl angle estimated by the CFD simulations for 460 geometries. In black is shown the Swirl angle for the same 460 geometries predicted by the ML model.

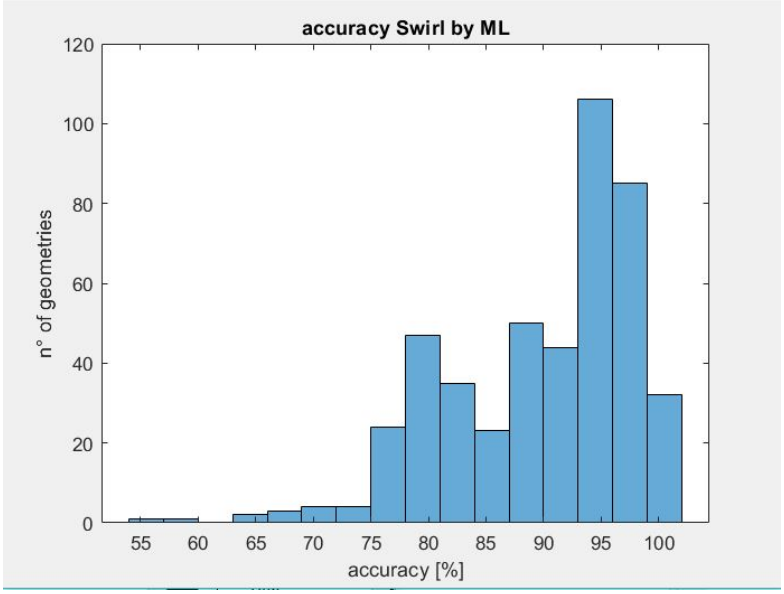

FIGURE 4.16: In this Figure, we can see the accuracy of the model checking the Swirl values precision between the ML predicted ones and the CFD ones.

4.5.3 Model test

Analyzing the model precision again, we can check the values for *mean squared error* introduced in the previous chapters. This error tells us the sum of the distances between all the data test and the predicted one after that will make a comparison between the three models done until now.

- Mean Squared Error (MSE) calculated for the two variables we are predicting Cp and Swirl:

$$\begin{aligned} Pressure\,recovery\,MSE &= 0.0030 \\ Swirl\,MSE &= 86.9376 \end{aligned} \tag{4.11}$$

This error is a dimensionless number that must be minimized. This model can provide a precise fitting of the two variables, and it will be used again in the ML optimization process in the next chapter number six.

Let's go through the last study checking which kind of Gaussian distribution this new result has. We want to find the average again from the CFD data and from the ML predictions values. In other words, we will discover how many predicted ML data would be inside the range of 3σ, 2σ, and σ of the CFD gaussian distribution built with the original database file.
In the next two figures, 4.17 and 4.18, we can discover the ML gaussian distribution.

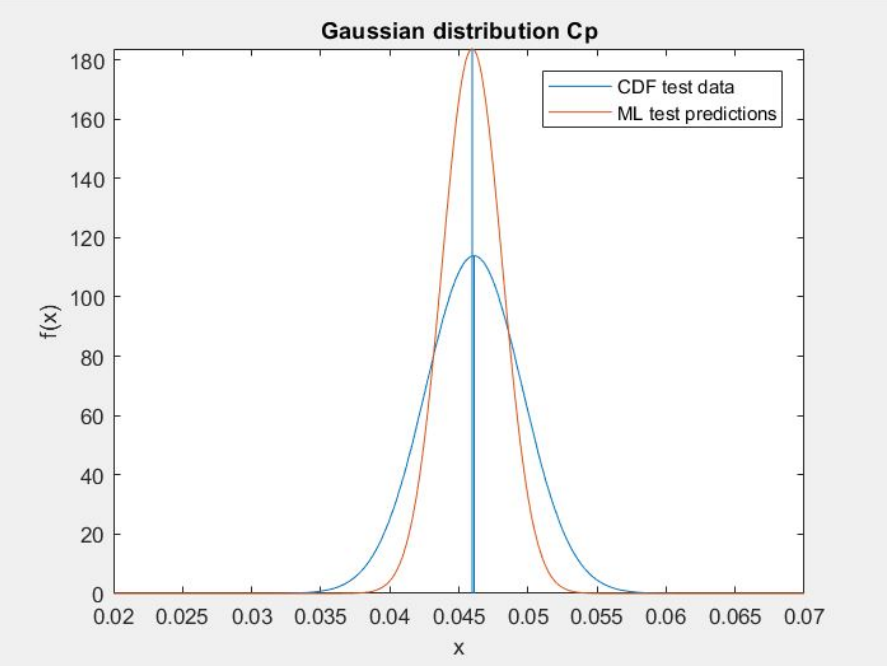

FIGURE 4.17: Observing in blue the gaussian distribution of the pressure recovery from the CFD data. In red, we can observe the Cp ML prediction distribution.

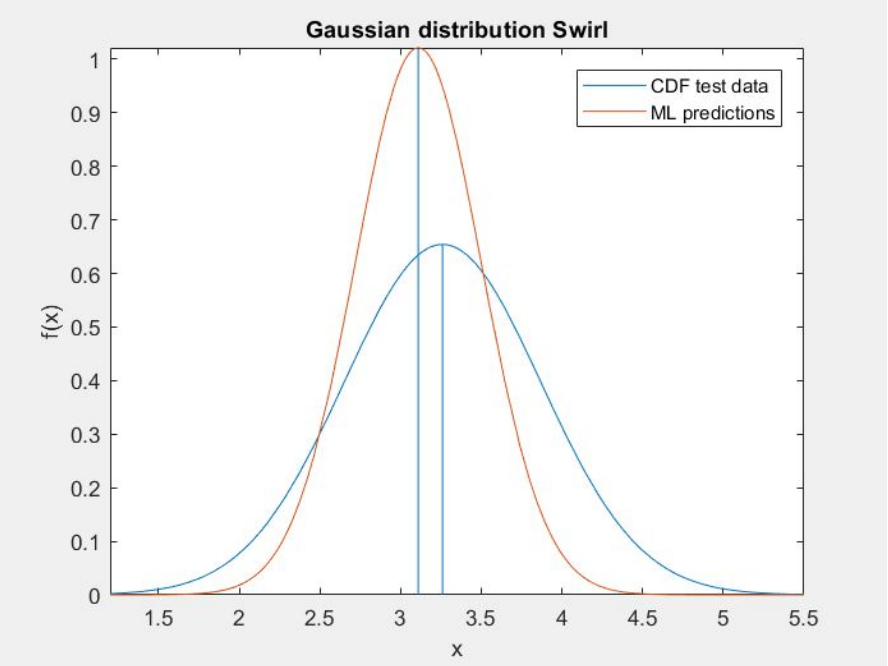

FIGURE 4.18: In blue, we have the gaussian distribution of the Swirl from the CFD data. In red, we can observe the Swirl angle ML prediction distribution.

On the last step is interesting to see the effective numbers of geometries fitted under the CFD curve.

In table 4.3 is shown in percentage the amount of ML predicted geometries fitted under the CFD gaussian curve. Looking to the first line, we can see that all the 100% of Cp predicted from the ML are inside the CFD range ($\mu - 3\sigma \,\&\, \mu + 3\sigma$), where μ is the average value, and σ is the standard deviation. Going through the Cp predictions, we can observe that in the CDF ($\mu - 2\sigma \,\&\, \mu + 2\sigma$) range the 99.602% of the 460 geometries predicted with the ML are inside. The last range tighter ($\mu - \sigma \,\&\, \mu + \sigma$) under the CFD gaussian curve, the 98.456 % of the ML predictions are in this range. As we analyze the Cp case, we can do the same thing with the swirl. Looking at table 4.3 and to the MSE error before it is easy to observe that this model fits very well with the CFD Cp from the database. We can also observe that it has the smallest MSE error for the Cp predictions but also the biggest MSE value until now for the Swirl predictions. The conclusion is that this is the best model until now to predict the Cp, but its also the less precise one for the Swirl predictions.

TABLE 4.3: ML geometries percentage fitted under the CFD gaussian distribution.

Variable	3σ	2σ	σ
Cp	100%	99.602%	98.456%
Swirl	100%	92.658%	78.854%

4.6 Fourth Machine Learning Configuration tree method:hist

The fourth model implemented runs a very similar classification method for the data organization. The CFD geometries are all split inside the different trees ramifications, and the model parameters will be shown below. This method implemented inside XGBoost also runs the tree booster instruction introduced before called *booster*:*gbtree* but with some other additionals parameters.

4.6.1 Model parameters

- **parameters:** reg:tweedie, eta, gbtree, max delta step, tree method, max bin, num paralel tree, max depth.

  The objective regression function used in this fourth model is again **reg:tweedie**. This parameter specifies the learning task and the corresponding learning objective of the ML model.

  The second parameter used is called **eta**. This parameter controls the step size shrinkage used in the model update loop to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative. It goes from $0 \rightarrow 1$ and by default is set equal to 0.3.

  The other parameter used for the booster configuration code inside the model is the **gbtree**. This parameter, as we spoke in the last chapter, allows the model to organize the data as a tree during the classification process giving weights to the tree leaves.

  The parameter **max delta step** is used for the maximum delta step; we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help to make the update step more conservative.

Usually, this parameter is not needed in the model configuration, but if it is set between $1 \rightarrow 10$ might help to control the update loop inside the model iterations.

The parameter **tree method** is the tree construction algorithm used inside XGBoost.

The parameter **max bin** is only used if the tree method is running. It is the maximum number of discrete bins to bucket continuous features. Increasing this number improves the optimality of splits at the cost of higher computation time.

The **number of parallel trees** parameter is another function explained in the first model very useful. It allows the model to be built with more than one classification tree.

The parameter **max depth** means the maximum depth of a tree during the iteration process. Increasing this value will make the model more complex and more likely to overfit. The range of this parameter goes from $0 \rightarrow \text{inf}$.

4.6.2 Improving trained model

With this fourth configuration done, we need again to check how reliable the model will be. To do this, we have to test it as always with the remaining CFD data we already have in the CFD database file.

Following the same treatment did for the last two models before, in Figure 4.19 will be shown respectively in red and black the S-duct pressure recovery coefficient found during the optimization process done by DalMagro [6] and the machine learning values calculated with this new ML tree configuration method called hist. On the second plot shown in Figure 4.20, we can figure out how many Cp ML predicted values are close to the real ones calculated with the optimization process using CFD.

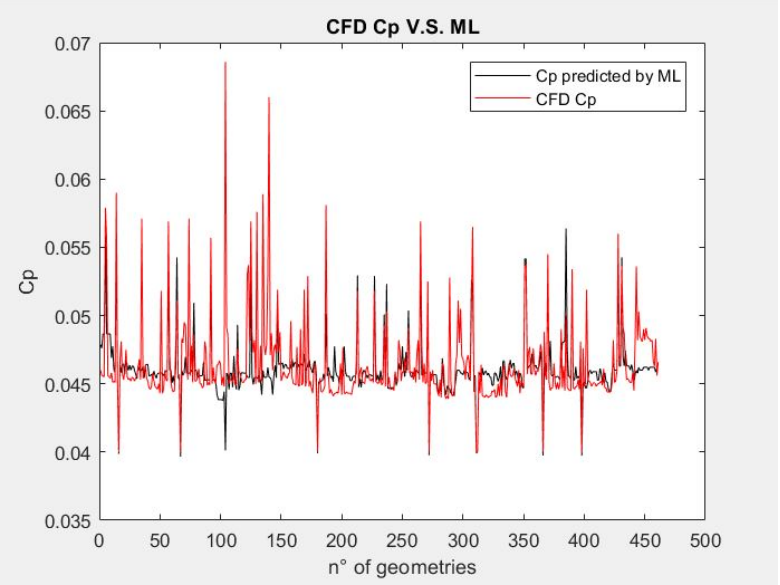

FIGURE 4.19: Drawn in red is shown the pressure recovery coefficient predicted by the CFD simulations for 460 geometries. In black is shown the pressure recovery coefficient for the same 460 geometries predicted by the ML model.

To calculate how close the values are in percentage form each other, we use the same mathematical equation explained in the last two models.

The consideration we can do now is that almost 250/460 testing geometries have the ML predicted values with an accuracy close to 100%. Only 100 and almost 50 geometries have an accuracy Cp prediction from the ML between 94% and 98%.

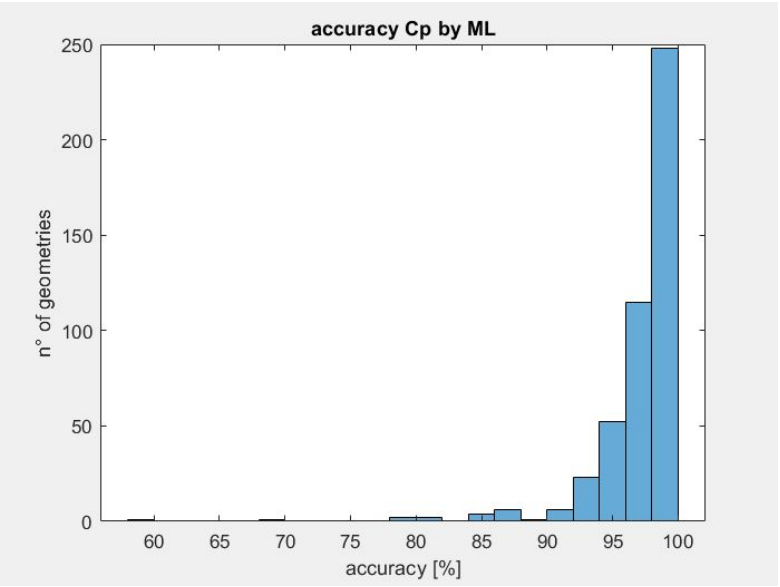

FIGURE 4.20: In this Figure, we can see the accuracy of the model checking the Cp values precision between the predicted one (ML) and the real one (CFD).

The next two Figures, 4.21 and 4.22, will show the Swirl features during the testing process. In Figure 4.21 is shown in red the Swirl angle from the CFD predictions and in black, the ML predictions for the same testing geometries.

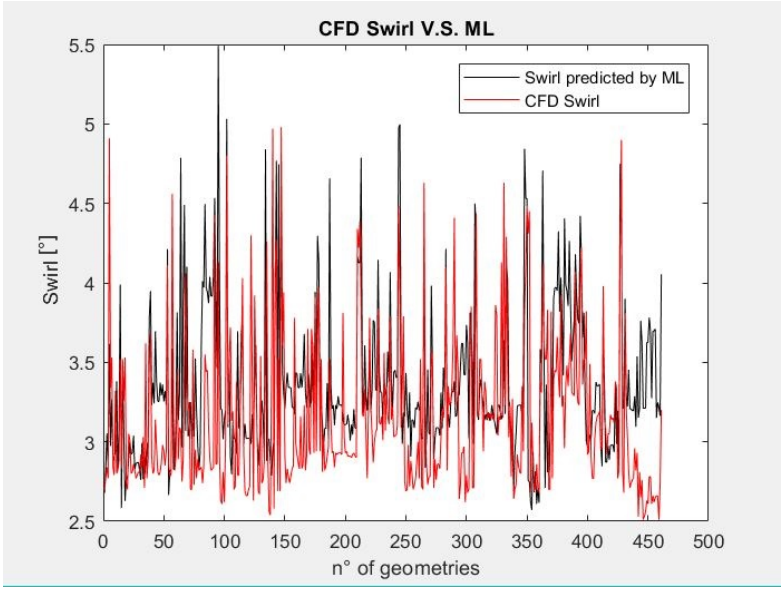

FIGURE 4.21: Drawn in red is shown the Swirl angle estimated by the CFD simulations for 460 geometries. In black is shown the Swirl angle for the same 460 geometries predicted by the ML model.

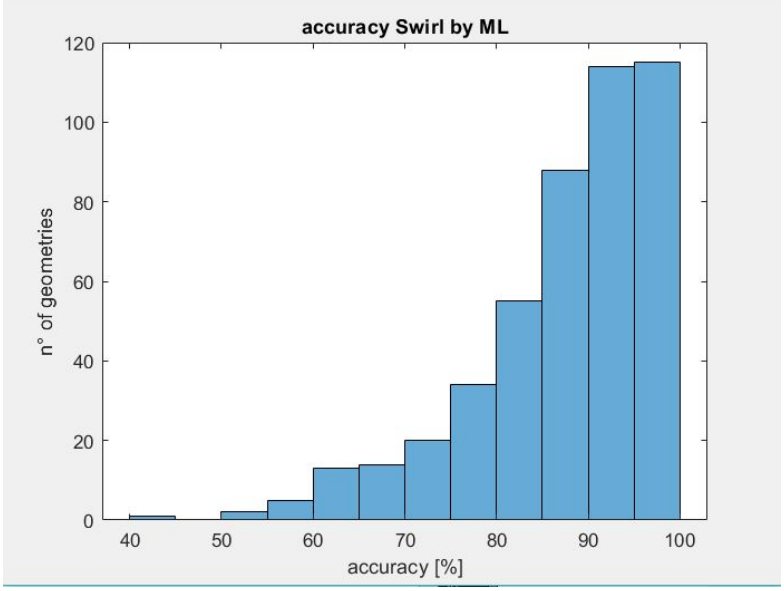

FIGURE 4.22: In this Figure, we can see the accuracy of the model checking the Swirl values precision between the ML predicted ones and the CFD ones.

We will plot in Figure 4.22 the Swirl model accuracy to see how many geometries are fitted in the correct way reaching the highest accuracy.

This model can fit the Swirl angle with a considerable approximation. In this case, almost 120 and 100 geometries rich the 90% $\rightarrow$ 100% range of accuracy. After that, we decrease the accuracy having from 80 and 60 geometries fitted by the 80% $\rightarrow$ 90% accuracy range. The rest of the testing geometries are fitted with an accuracy less than 80% as shown in Figure 4.22.

4.6.3 Model test

Analyzing the model precision again, we can check the values for *mean squared error* introduced in the previous chapters. This error tells us the sum of the distances between all the data test and the predicted one after that will make a comparison between the three models done until now.

- Mean Squared Error (MSE) calculated for the two variables we are predicting Cp and Swirl:

$$\begin{aligned} Pressure\,recovery\,MSE &= 0.0033 \\ Swirl\,MSE &= 117.2373 \end{aligned} \tag{4.12}$$

This error is a dimensionless number that must be minimized. This model will be used in the ML optimization process in the next chapter number five.
Let's go throw the last study checking which kind of Gaussian distribution this new ML has. We want to find the average again from the CFD data and the ML predictions values. In other words, we will discover how many predicted ML data would be inside the range of 3σ, 2σ, and σ of the CFD gaussian distribution built with the original database file. In the next two figures, 4.23 and 4.24, we can discover the ML gaussian distribution.

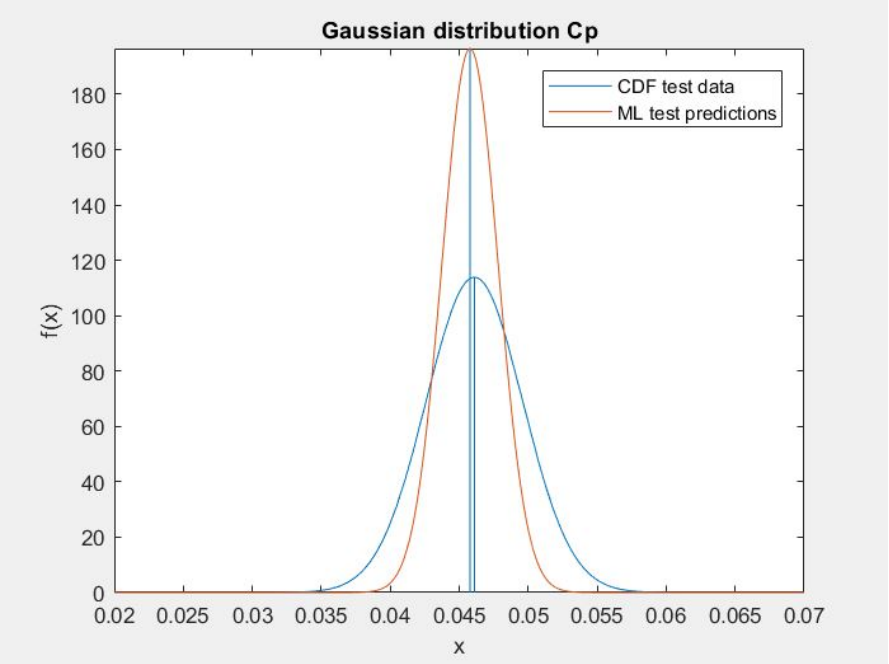

FIGURE 4.23: Observing in blue the gaussian distribution of the pressure recovery from the CFD data. In red, we can observe the Cp ML prediction distribution.

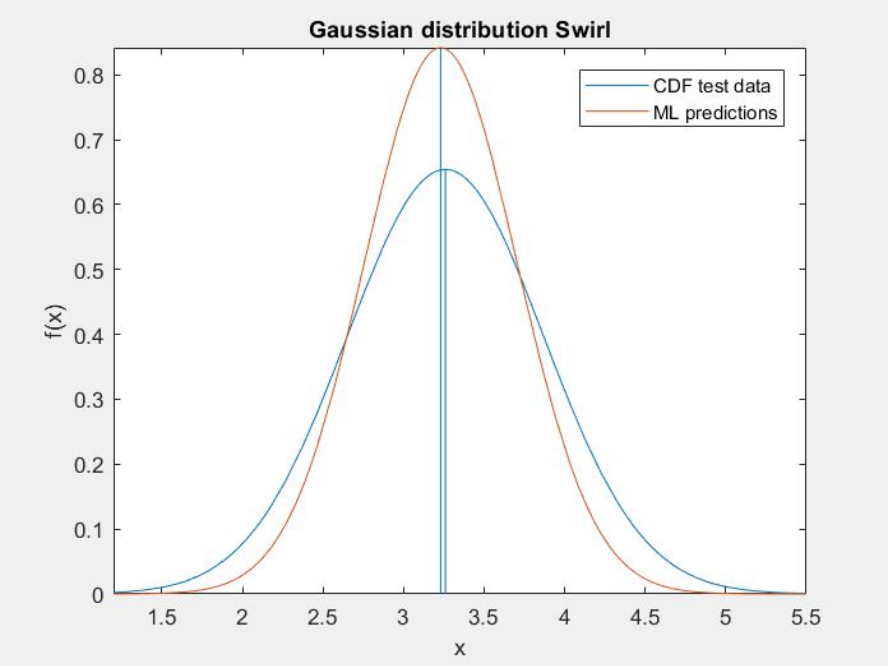

FIGURE 4.24: In blue, we have the gaussian distribution of the Swirl from the CFD data. In red, we can observe the Swirl angle ML prediction distribution.

On the last step is interesting to see the effective numbers of geometries fitted under the CFD curve.

In table 4.4 is shown in percentage the amount of ML predicted geometries fitted under the CFD gaussian curve. Looking to the first line, we can see that all the 100% of Cp predicted from the ML are inside the CFD range ($\mu - 3\sigma \,\&\, \mu + 3\sigma$), where μ is the average value, and σ is the standard deviation. Going through the Cp predictions, we can observe that in the CDF ($\mu - 2\sigma \,\&\, \mu + 2\sigma$) range the 97.831% of the 460 geometries predicted with the ML are inside. The last range tighter ($\mu - \sigma \,\&\, \mu + \sigma$) under the CFD gaussian curve, the 97.125 % of the ML predictions are in this range. As we analyze the Cp case, we can do the same thing with the swirl. Looking at table 4.4 and to the MSE error before it is easy to observe that this model can fit both variables. With this fourth model, we have the Cp MSE error bigger than the third model, but it is less than the first and the second one. Speaking about the Swirl angle, we can observe that the model error is much bigger now than the last three models. After this conclusion, we can say that this model will also be used inside the optimization loop to make comparisons between the other models looking at the type of Pareto front coming out from all of them on the optimization ML process explained in chapter five.

TABLE 4.4: ML geometries percentage fitted under the CFD gaussian distribution.

Variable	3σ	2σ	σ
Cp	100%	97.831%	97.125%
Swirl	100%	90.445%	74.165%

4.7 Fifth Machine Learning Configuration deep method

The fifth model implemented runs a deep method for the data organization. The CFD geometries are all split inside the ramifications of different deep trees, and the model parameters will be shown below. This method implemented inside XGBoost runs the deepest tree the booster can build introduced in the last section.

4.7.1 Model parameters

- **parameters:** reg:tweedie, dart, max depth.

  The objective regression function used in this fourth model is again **reg:tweedie**. This parameter specifies the learning task and the corresponding learning objective of the ML model.

  The **Dart booster** parameter we already showed and explained in chapter number 3, but we can say that this booster object type will perform the dropouts trees. During the evaluations, we have that only some of the trees will be taken into account. To obtain correct results, the parameter *number tree limit* must be set on a nonzero value.

  The parameter **max depth** means the maximum depth of a tree during the iteration process. Increasing this value will make the model more complex and more likely to overfit. The range of this parameter goes from $0 \rightarrow \text{inf}$. In this case, it will be set to inf.

4.7.2 Improving trained model

With this fourth configuration done, we need again to check how reliable the model will be. To do this, we have to test it as always with the remaining CFD data we already have in the CFD database file. Following the same treatment did for the last two models before, in the next Figure 4.25 will be shown respectively in red and in black the S-duct pressure recovery coefficient found during the optimization process done by DalMagro [6] and the machine learning values calculated with this new tree configuration method.

In the second plot shown in Figure 4.26, we can figure out how many Cp ML predicted values are close to the real ones calculated with the optimization process using CFD.

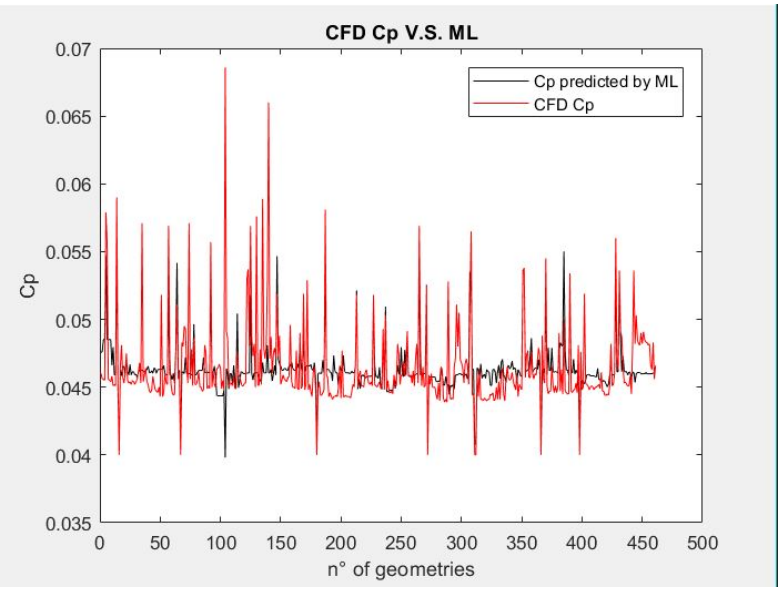

FIGURE 4.25: Drawn in red is shown the pressure recovery coefficient predicted by the CFD simulations for 460 geometries. In black is shown the pressure recovery coefficient for the same 460 geometries predicted by the ML model.

To calculate how close the values are in percentage form each other, we use the same mathematical equation explained in the first model.

The consideration we can do now is that almost 200/460 testing geometries have the ML predicted values with an accuracy close to 100%. After that, only 150 and almost 100 geometries have an accuracy Cp prediction from the ML between 94 and 98%. The next two Figures 4.27 and 4.28 will show the Swirl features during the testing process. In Figure 4.27 is shown in red the Swirl angle from the CDF predictions and in black, the ML predictions for the same testing geometries. We will plot in Figure 4.28 the Swirl model accuracy to see how many geometries are fitted in the correct way reaching the highest accuracy possible for the model.

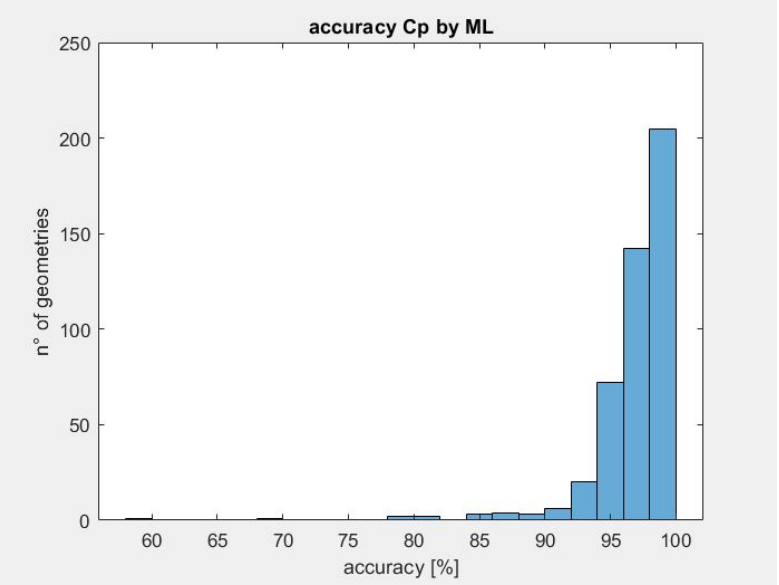

FIGURE 4.26: In this Figure, we can see the accuracy of the model checking the Cp values precision between the predicted one (ML) and the real one (CFD).

This model can fit the Swirl angle with a considerable approximation. In this case, 40 geometries rich the 100% range of accuracy. After that, we decrease the accuracy having from 100 and almost 80 geometries fitted by 90% $\rightarrow$ 98% accuracy range. The rest of the testing geometries are fitted with an accuracy less than 90% as shown in Figure 4.28.

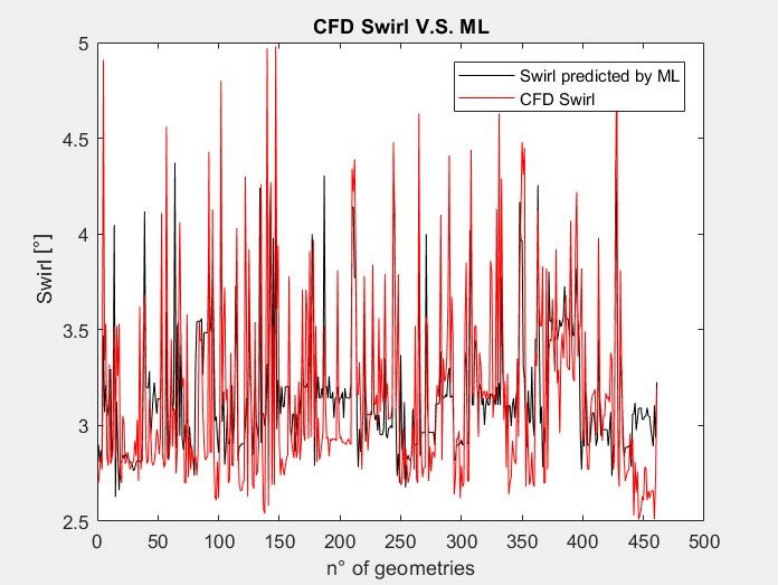

FIGURE 4.27: Drawn in red is shown the Swirl angle estimated by the CFD simulations for 460 geometries. In black is shown the Swirl angle for the same 460 geometries predicted by the ML model.

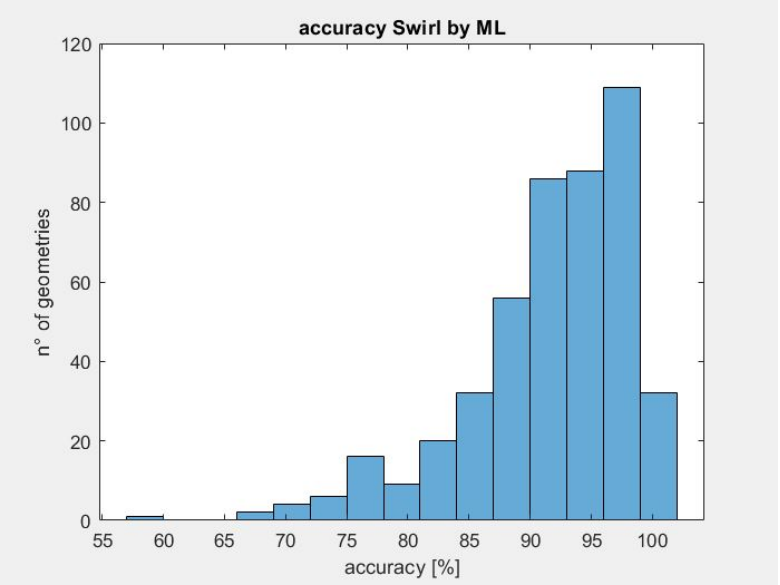

FIGURE 4.28: In this Figure, we can see the accuracy of the model checking the Swirl values precision between the ML predicted ones and the CFD ones.

4.7.3 Model test

Analyzing the model precision again, we can check the values for *mean squared error* introduced in the previous chapters. This error tells us the sum of the distances between all the data test and the predicted one.

- Mean Squared Error (MSE) calculated for the two variables we are predicting Cp and Swirl:

$$\begin{aligned} Pressure\,recovery\,MSE &= 0.0032 \\ Swirl\,MSE &= 62.8713 \end{aligned} \tag{4.13}$$

This error is a dimensionless number that must be minimized. This model can provide precise fitting for the two variables, and it will be used again in the ML optimization process showed in chapter five.

Let's go through the last study checking which kind of Gaussian distribution this new ML has. We want to find the average again from the CFD data and the ML predictions values. In other words, we will discover how many predicted ML data would be inside the range of 3σ, 2σ, and σ of the CFD gaussian distribution built with the original database file.

In the next two figures, 4.29 and 4.30, we can discover the ML gaussian distribution.

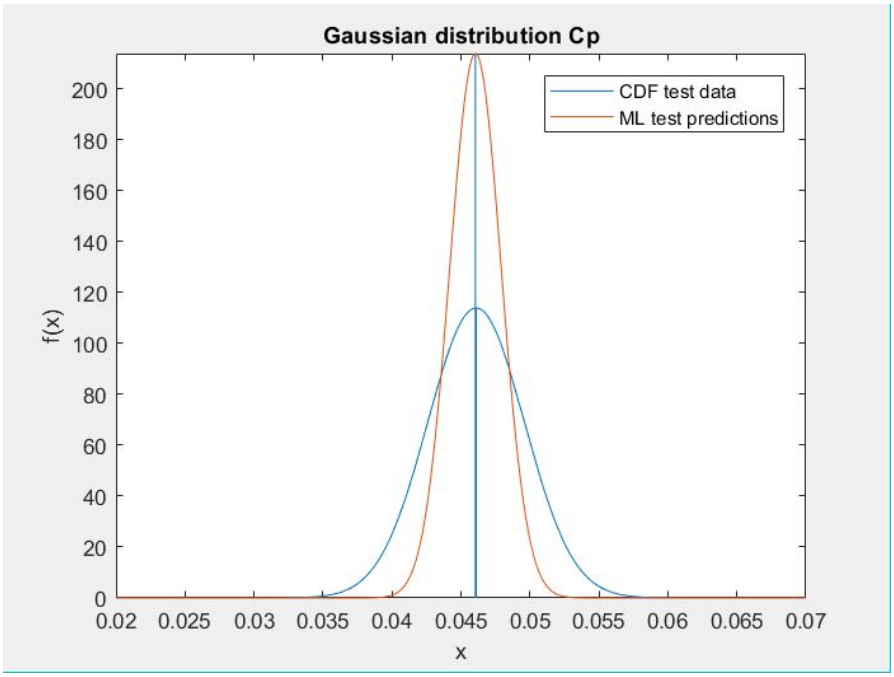

FIGURE 4.29: Observing in blue the gaussian distribution of the pressure recovery from the CFD data. In red, we can observe the Cp ML prediction distribution.

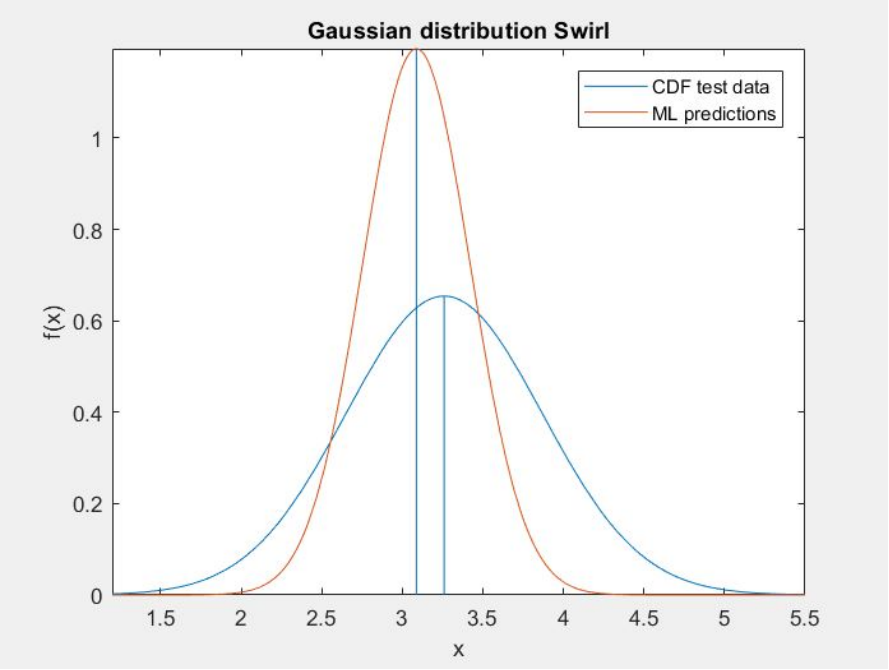

FIGURE 4.30: In blue, we have the gaussian distribution of the Swirl from the CFD data. In red, we can observe the Swirl angle ML prediction distribution.

In table 4.5 is shown in percentage the amount of ML predicted geometries fitted under the CFD gaussian curve. Looking to the first line, we can see that all the 100% of Cp predicted from the ML are inside the CFD range ($\mu - 3\sigma \,\&\, \mu + 3\sigma$), where μ is the average value, and σ is the standard deviation. Going through the Cp predictions, we can observe that in the CFD ($\mu - 2\sigma \,\&\, \mu + 2\sigma$) range the 98.937% of the 460 geometries predicted with the ML are inside. The last range tighter ($\mu - \sigma \,\&\, \mu + \sigma$) under the CFD gaussian curve, the 97.542 % of the ML predictions are in this range. As we analyze the Cp case, we can do the same thing with the Swirl angle. Looking at table 4.5 and to the MSE error before it is easy to observe that this model fits very well the CFD Cp and also the swirl angle from the database. We can observe that it has the second smallest MSE error between all the models introduced for the Cp. This model predictions are less precise than the third one, but it is more accurate than all the other three models for the Cp. Looking at the swirl, we can see that this model is less accurate than the first one but more precise on the predictions than the other three. The conclusion is that this model is not the best one between the other, but it is the second looking at the precision of the predictions interpolating the test data in the CFD database we have for the Cp but also for the swirl.

TABLE 4.5: ML geometries percentage fitted under the CFD gaussian distribution.

Variable	$3\,\sigma$	$2\,\sigma$	σ
Cp	100%	98.937%	97.542%
Swirl	100%	94.496%	82.538%

4.8 Conclusions

After introducing all the models, bellow is written the table containing the machine learning *Mean Squared Error*, to be able to compare them and see which one is the more precise and the one with less accuracy.

Table 4.6: Machine learning models *Mean Squared Error*.

Models	**Cp**	**Swirl**
Reg:Tweedie	0.0041	39.259
booster:dart	0.0038	65.2727
booster:gbtree	0.0030	86.9376
tree method:hist	0.0033	117.2373
method:deep	0.0032	62.8713

Looking at Table 4.6, we can notice the model with the smallest error for the Cp predictions and also for the Swirl number. Model number three has the smallest MSE score for the Cp, so we aspect from it the best results on the optimization S-duct loop. Model number four and five have a similar Cp MSE value to number three, so the three models predictions will be equivalent. With these motivations, we aspect of having similar values ranges between model three four and five. Concluding the Cp case, we can say that models one and two have the MSE error higher than the others so we will aspect less precise results, and we will get numbers that are outside from the more precise model's range.

Analyzing the Swirl angle, we can notice that the smallest MSE error number belongs to the first machine learning configuration. We aspect excellent results from the optimization loop when this model will run. The error increases in model numbers two and five; this is because the model configuration is changed. The prediction range will be similar for both models. The last consideration Table number 4.6 allows us to do is about model three and four for the swirl. The predictions will not be as precise as the other models, but we will use these models as a comparison tool to verify the outputs also from the other models.

Chapter 5

Multi-Objective Optimization using Machine learning

5.1 Introduction

The following chapter will face up to the Multi-Objective optimization problem for the S-duct with the machine learning codes introduced before. This introduction will start to explain what an optimization process is.

Optimization is a procedure employed in many different fields of application, for example, in engineering. The fundamental idea behind this is to increase the performance of the object we are analyzing. To do that, the first step is to model the subject of the analysis as a mathematical function. The second step you have to do is to maximize or minimize it, always having constraints to respect. However, in many situations, we can also have in the software the opportunity to optimize minimizing the function of different objective functions at the same time. It may usually happen that there are two o three objectives functions to minimize and many constraints around.

5.2 Multi-Objective Optimization

As we can imagine in the optimization process can happen that a solution might be optimal for one objective function but not for another. In the S-duct machine learning optimization, we will take into account as objective function, the pressure recovery coefficient and the Swirl angle. The ML models are all trained and tested on the same aerodynamics variables because the goal of all the work is to optimize the S-duct ranging a big design space without using CFD.
As introduced in DalMagro's work, [6], we can see in Figure 5.1 on the x and y-axis the two objective functions, respectively. The lines called *side constraints, behaviour constraints* are the constraints of the objective function, absolutely fundamental since they impose the boundaries conditions for each project parameter.

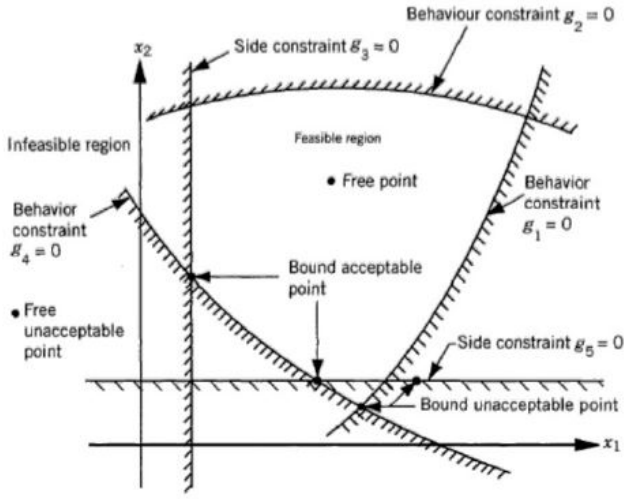

FIGURE 5.1: Objective functions optimization range with the side and the behavior constraints.

It is essential now to introduce the dominance concept. The definition of dominance, as reported in [10], is the following: in a minimization problem, the solution **A** dominates a solution **B**. These statements have to bee verified:

- Solution **A** is not worst than solution **B** in each objective function. This means that $f_j(\mathbf{A}) \leq f_j(\mathbf{B}) \; \forall \; j = 1, ..., m$ where m is the number of the objective functions.

- The solution **A** is strictly better then the solution **B**, in at least one objective function. As a consequence, this means that $f_k(\mathbf{A}) < f_k(\mathbf{B})$ for at least one k in $1, ..., m$.

If solution **A** dominates solution **B**, this means that both solutions are non-dominated. Figure 5.2 helps to clarify the comprehension of what just said. f_1 and f_2 are the objective functions. The points chosen are non dominated and considered optimal; in fact, all these points have the peculiarity that if one objective function improves, the other has to become worst.

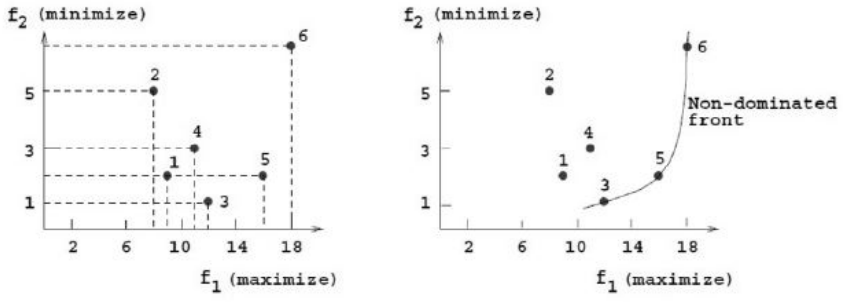

FIGURE 5.2: Pareto front example.

There are other two fundamental parameters deeply linked to the dominant concept explained:

- **Pareto Optimal Set:** is the decision variable subspace, it is the set of non-dominated solutions.
- **Pareto Front:** is the Pareto optimal set image; it contains all the non-dominated solutions.

5.2.1 Tabu Search

The Multi-objective Tabu Search was born in 1989 thanks to the work of Glover [11]. This strategy is implemented to explore the design space given between the boundaries conditions using three types of memories as it is represented in Figure 5.3. Moreover, this method has already been tested in previous works such as [1] and [5] with accurate results, and at the same time, the method was efficient. Many Tabu search variants are available, but in this work, the alternative proposed by Dr. Kipouros [12] with the software he developed called Multi-objective Tabu search (MOTS).

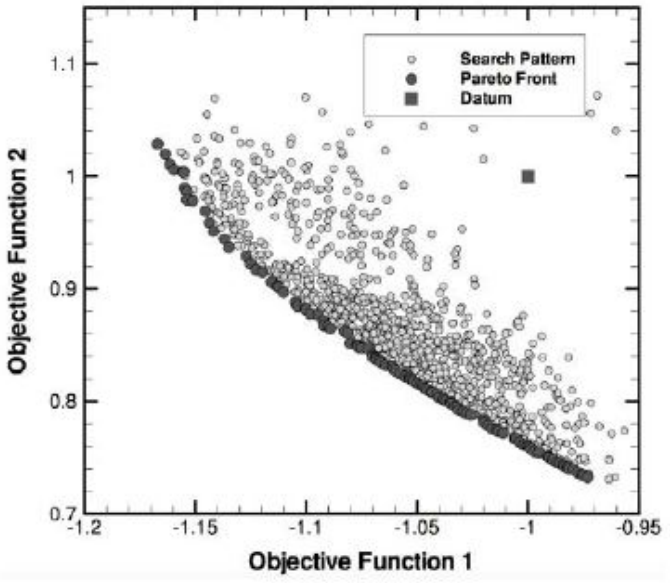

FIGURE 5.3: Pareto optimal set.

On the optimization loop, every time that an iteration is realized, $2n_{var}$ new points are created systematically by the optimizer with *nvar* as design variables. To originate new points, the MOTS optimizer uses a step that is decided by the user, to increase $(x_i + \delta_i)$ or decrease $(x_i - \delta_i)$ the variable value. Once the new point inside the design space is made, the code proceeds through the evaluations of all the objective functions. The point x_{i+1} that results better than the others, it will be the new starting point for the following step.

5.3 Machine learning using MOTS optimization

5.3.1 Introduction

This section will go through the optimization results given by all the five different machine learning methods created before using the Multi-objective Tabu search. When a new point is created in the design space by Tabu search, the current ML model implemented will predict the pressure recovery and the Swirl angle values for the new geometry built by the optimizer.

This loop is made twice with 500 iterations each, so every model will run 1000 iterations. The model results will be analyzed, looking for the dominated and non dominated points on the Pareto front.

With the Pareto front built from every ML model optimization, we will look for the optimized geometry that has the best results between the Cp and Swirl angle.

Going through my work, I decided to analyze only the ML Pareto front points with the CFD fluent verifying and checking the values the machine learning has predicted. The corresponding parallel coordinates with only the Pareto front points will be plot for every ML model that shows which design space range we studied in the optimization.

The ML Pareto points evaluations are done using the Ansys fluent code, but the further idea is to analyze them using Lattice Boltzmann's approach to predict the unsteady characteristics of the flow in the S-duct. This approach will be introduced better in chapter number six.

5.4 First ML optimization (reg:tweedie)

As explained in the introduction, two optimizations loops are made with this first machine learning method. Two different starting points are given as different starting inputs to the Multi-objective Tabu search (MOTS), one for each optimization. A problem that happens when the code runs is that: if the MOTS does a little change on the geometry control points values in the design space, the same ML prediction will be calculated for the variables by the model. To solve this problem, we have to improve the model forcing the ML algorithm to split up deeper and bigger trees at the expense of the model heaviness.
In Figure 5.4 is shown the first machine learning Pareto front build from the MOTS iterations. With the blue points is shown, the fist Pareto front calculated with the first ML optimization loop. With the red points is showed the second machine learning Pareto front containing the geometries predictions. We can observe we have one geometry with the best Swirl number in each iteration and also other geometries with the best pressure recovery value. Between these two extreme values, we have the trade-off geometries for each iteration.
Implementing the last CFD evaluation step, we get the green points. The geometries from the first optimization Pareto front (blue) are reevaluated with the CFD fluent code to check and confirm the ML results. We can observe that the green points are much close to the blue ones predicting the swirl number. This fact is confirmed because this first model is the most precise predicting the Swirl values. After all, it has the smallest MSE error but is not the best for the Cp evaluations.

To continue on the machine learning analysis, in figure 5.5, the original CFD data from where the models are built (the original CFD training and testing data from DalMagro's work. [6]) are plotted in black. Thanks to that, on the same figure, we can observe the comparison between the CFD obtained in this Thesis Pareto front points plotted in green evaluated from the ML optimization and again DalMagro's [6] Pareto front data in black.

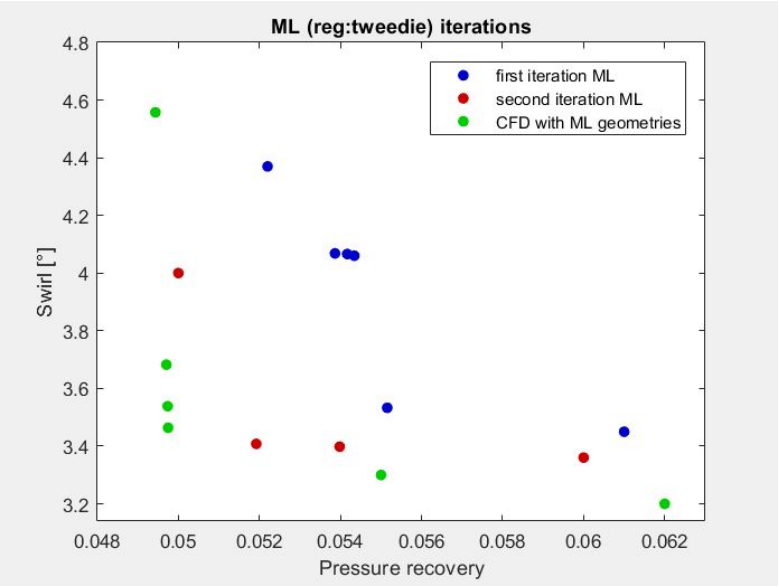

FIGURE 5.4: Machine learning optimization Pareto front. First machine learning optimization in blue. Second machine learning optimization in red. The CFD fluent evaluation in green.

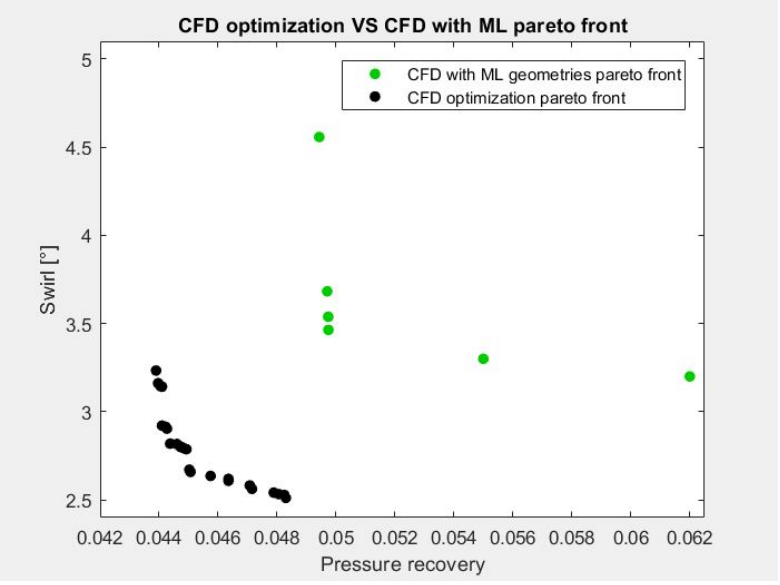

FIGURE 5.5: In green, we observe the machine learning Pareto front evaluated with CFD. In black is shown the CFD Pareto front done by DalMagro. [6]

Only the Pareto front points evaluated with the machine learning are shown on the following parallel coordinates plot in Figure 5.6 because this plot is handy to understand the design space analyzed. This interpretation helps to understand the design space range studied with the first optimization loop. In this way, it is possible to modify the starting point for the second ML optimization loop. Changing the geometries range, we force the optimizer to move through other areas where the Tabu search has to slip to study new geometries configurations. But as is shown in figure 5.4, even if the optimizer begins the cycle from a second starting point, it will always move through the same area because it is the only region where we have the minimum values for the Cp and the swirl.

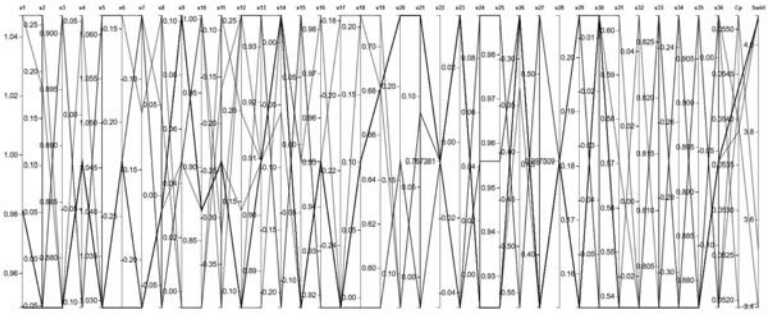

FIGURE 5.6: Paralel coordinates first optimization model

5.5 Second ML optimization (booster:dart)

As explained before, two optimizations loops are made again with this second machine learning method. Two different starting points are given as different inputs to the Multi-objective Tabu search (MOTS), one for each optimization. The different starting points in the iterations again allows the Tabu search to move through other geometries that have not been studied yet.

In Figure 5.7 is shown the two iterations loop Pareto front with this second machine learning model. In blu is shown the Pareto front points calculated with in the first ML optimization loop. With the red points is showed the second machine learning optimization loop Pareto front containing the geometries predictions.

We can observe again we have geometries with the best Swirl number in each iteration and also other geometries with the best pressure recovery value. Between the two extreme position's values, we have the trade-off geometries for each iteration. Implementing the CFD fluent evaluation step we had the green points.
The geometries from the first optimization loop (in blue) are reevaluated with the CFD fluent code to check and confirm the ML results. We can observe that in this case, the green points are close again to the red and to the blue ones like in the first model. The difference in the distances between the green and the blue points is due to the less precision the ML model has compared to the CFD results. As we said in chapter number four, the second model has a higher Mean Squared Error predicting the Swirl angle but a lower error predicting the Cp values. So we aspect of having the predicted blu points closer to the CFD ones looking at the pressure recovery values but not for the swirl. This is confirmed by Figure number 5.7.

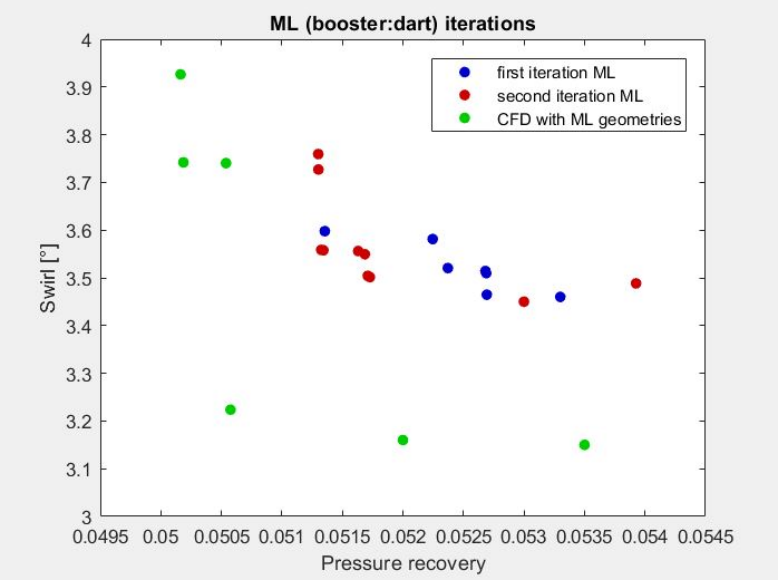

FIGURE 5.7: Machine learning optimization Pareto front. First machine learning optimization in blue. Second machine learning optimization in red. The CFD fluent evaluation in green.

In figure 5.8, the comparison is provided between CFD obtained in this Thesis Pareto front points in green, evaluating the ML predictions, and in black the DalMagro's [6] Pareto front data (the CFD data file from where I built and trained the ML models).

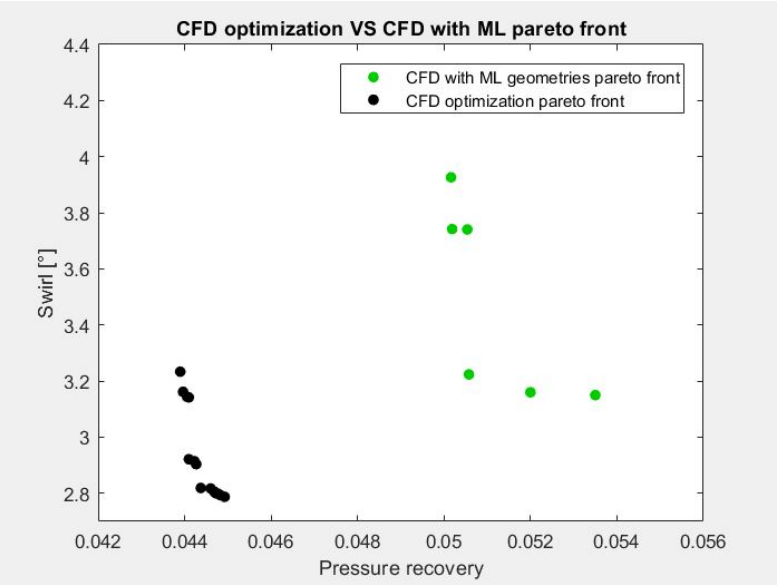

FIGURE 5.8: In green we observe the machine learning Pareto front evaluated with CFD. In black is shown the CFD Pareto front done by DalMagro. [6]

Just like with the first model before, only the Pareto front points evaluated with the machine learning are plotted in the following parallel coordinates in Figure 5.9. This plot helps to understand the design space range studied with the first optimization loop. In this way, it is easy to change the investigated range varying the starting point for the second ML optimization loop. Swapping the geometries range, we force the optimizer to move through other areas where the Tabu search has to slip to study new geometries configurations.

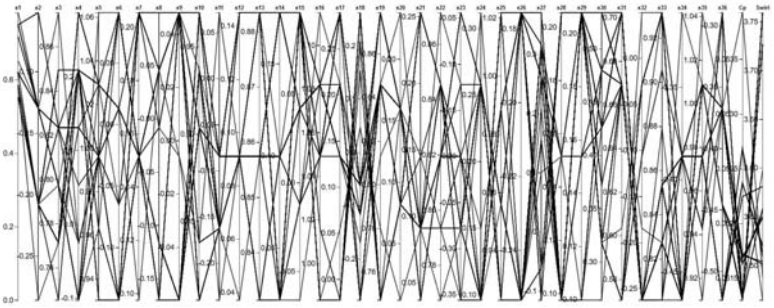

FIGURE 5.9: Paralel coordinates second optimization model.

5.6 Third ML optimization (booster:gbtree)

Two other optimizations loops are made with this third machine learning method. Two different starting points are always given as different input to the Multi-objective Tabu search (MOTS), one for each optimization.

In Figure 5.10 is shown the two iterations loop Pareto front with this third machine learning model. In blu, as always, is shown the Pareto front points calculated with the first ML loop. With the red points is showed instead of the second machine learning optimization loop Pareto front containing the geometries predictions. We can observe again we have geometries with the best Swirl number in each iteration penalizing the Cp and also other geometries with the best pressure recovery value penalizing the swirl instead. Between the two extreme positions values, we have the trade-off geometries for each iteration.
The geometries from the second Pareto front (red) are reevaluated with the CFD fluent code to check and confirm the ML results; the green points represent the CFD results. We can observe this case in Figure 5.10, where it is easy to notice that green points are again close to the blue ones but also to the red ones. The differences between the two Pareto fronts are due firstly because we have two different starting points for both the loops and secondly because the model precision we have, as we spoke in chapter number four, is different from the other models. This third model is the most precise predicting the Cp values because it has the smallest MSE error, but is not the best one for the Swirl predictions. Because that, we aspect to have the green points very close to the red ones looking for the Cp.

In figure 5.11 the comparison is provided between CFD obtained in this Thesis Pareto front points plotted in green evaluating the ML predictions, also shown on Figure 5.10, and in black the DalMagro [6] Pareto front data (the CFD data file from where the ML models are built and trained).

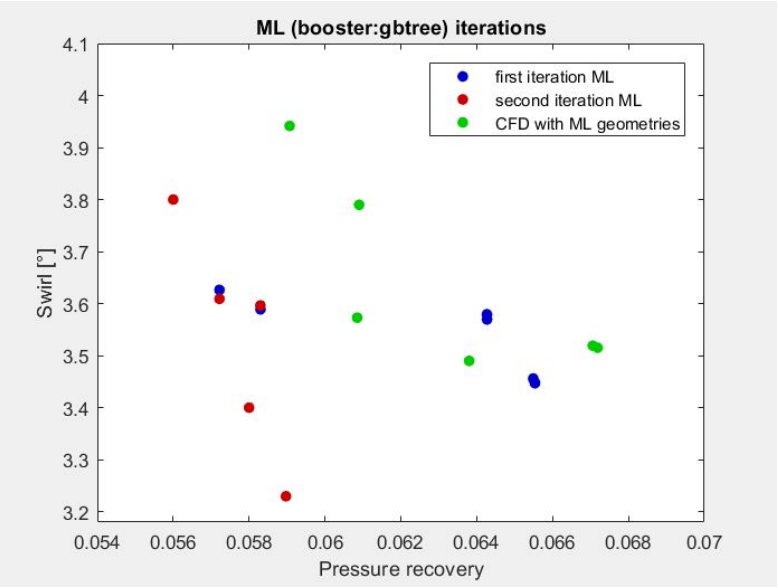

FIGURE 5.10: Machine learning optimization Pareto front. First machine learning optimization in blue. Second machine learning optimization in red. The CFD fluent evaluation in green.

As we can observe, we have the green Pareto front, which is the machine learning CFD evaluations, and in black, we can observe the CFD optimization original Pareto front.

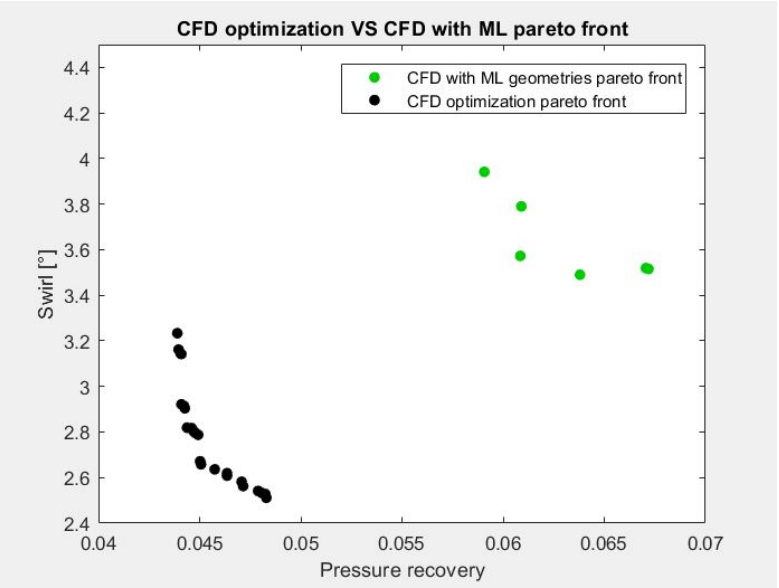

FIGURE 5.11: In green we observe the machine learning Pareto front evaluated with CFD. In black is shown the CFD Pareto front done by DalMagro. [6]

Just like with the first two models before, only the Pareto front points evaluated with the machine learning are plotted in the following parallel coordinates in Figure 5.12.

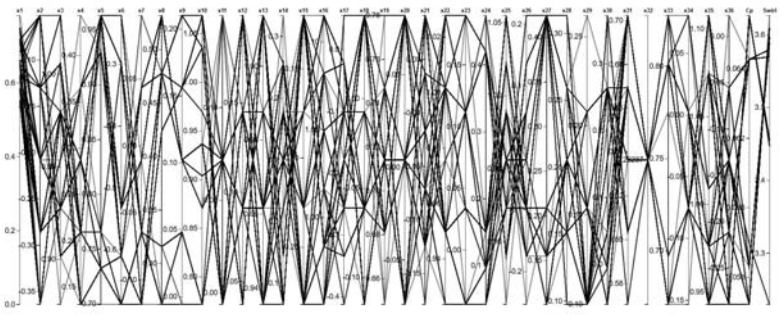

FIGURE 5.12: Paralel coordinates third optimization model.

5.7 Fourth ML optimization (tree method:hist)

Two other optimizations loops are made with this fourth machine learning method. Two different starting points are always given as input to the Multi-objective Tabu search (MOTS), one for each optimization. In Figure 5.13 is shown the two iterations loop Pareto front with this fourth machine learning model. In blu, as always, is shown the Pareto front points calculated with the first ML loop. With the red points is showed the second machine learning optimization loop Pareto front instead. We can observe again like in the last models we have geometries with best Swirl number in each iteration penalizing the Cp and also other geometries with the best pressure recovery value penalizing the swirl instead. Between the two extreme position's values, we have the trade-off geometries for each iteration. The geometries from both Pareto front are reevaluated with the CFD fluent code to check and confirm the ML results; the green points plot the CFD results. We can observe this case in Figure 5.13, where it is easy to notice that green points are now closer to the red ones and also very close to the blue ones. This fourth model seems to be very precise predicting both values because all points are close to the green CFD ones even if this ML model has not the smallest MSE error, as shown in chapter four.

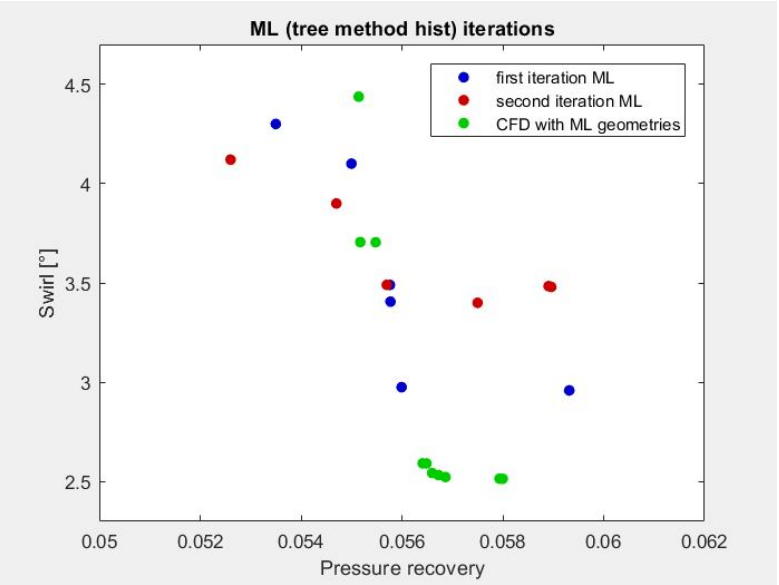

FIGURE 5.13: Machine learning optimization Pareto front. First machine learning optimization in blue. Second machine learning optimization in red. The CFD fluent evaluation in green.

In figure 5.14 the comparison is provided between CFD obtained in this Thesis Pareto front points plotted in green evaluating the ML predictions and in black the DalMagro's [6] Pareto front data (the CFD data file from where the ML models are built and trained).

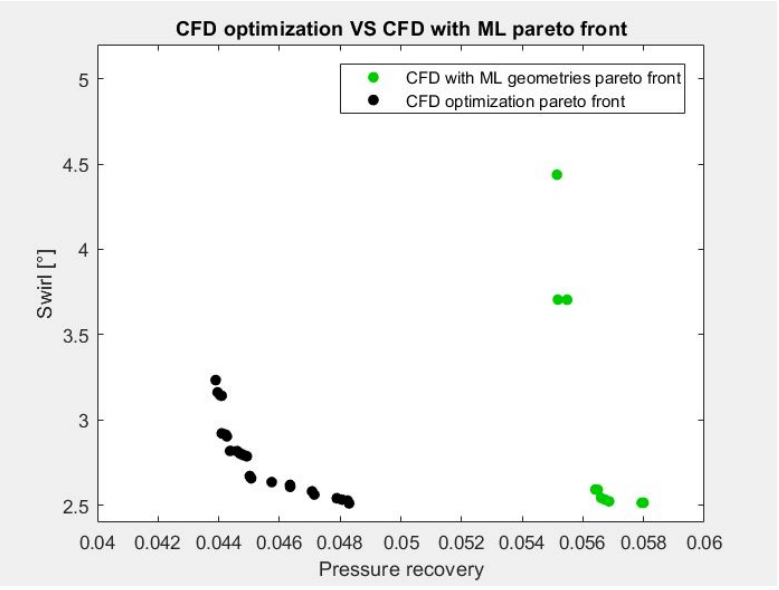

FIGURE 5.14: In green we observe the machine learning Pareto front evaluated with CFD. In black is shown the CFD Pareto front done by DalMagro. [6]

Just like with the other models before, only the Pareto front points evaluated with the machine learning are plotted in the following parallel coordinates in Figure 5.15.

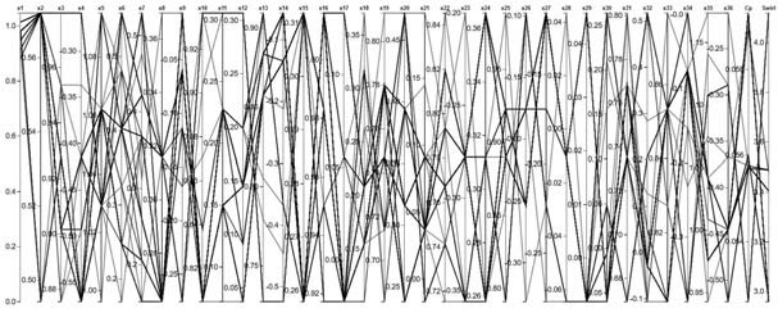

FIGURE 5.15: Paralel coordinates fourth optimization model.

5.8 Fifth ML optimization (deep method)

Two optimizations loops are also made with this fifth machine learning method. Two different starting points are always given as input to the Multi-objective Tabu search (MOTS), one for each optimization. In Figure 5.16 is shown the two iterations loop Pareto front with this fifth machine learning model. In blu, as always, is shown the Pareto front points calculated with the first ML loop. With red points is shown the second machine learning optimization loop Pareto front instead. We can observe again like in the last cases we have geometries with best Swirl number and also other geometries with the best pressure recovery value. Between the two extreme position's values, we have the trade-off geometries for each iteration. The geometries from the fist optimization Pareto front are reevaluated with the CFD fluent code to check and confirm the ML results; the green points plot the CFD results. We can observe this case in Figure 5.16, where it is easy to notice that green points are now very close to the blue ones and also very close to the red ones. This fifth model, like the fourth one, seems to be very powerful, predicting Cp and Swirl values because all points (ML) are close to the green ones (CFD). As we show in chapter four, this ML model has the second smallest MSE error for both variables. With this clarification, we can explain the proximity of the blue and red points to the green ones.

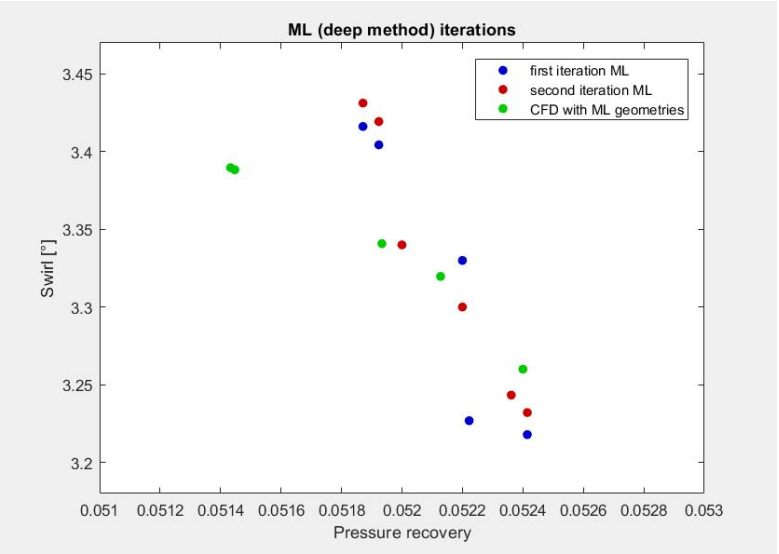

FIGURE 5.16: Machine learning optimization Pareto front. First machine learning optimization in blue. Second machine learning optimization in red. The CFD fluent evaluation in green.

In figure 5.17 the comparison is provided between CFD obtained in this Thesis Pareto front points in green evaluating the ML predictions and in black the DalMagro's [6] Pareto front data (the CFD data file from where the ML models are built and trained).

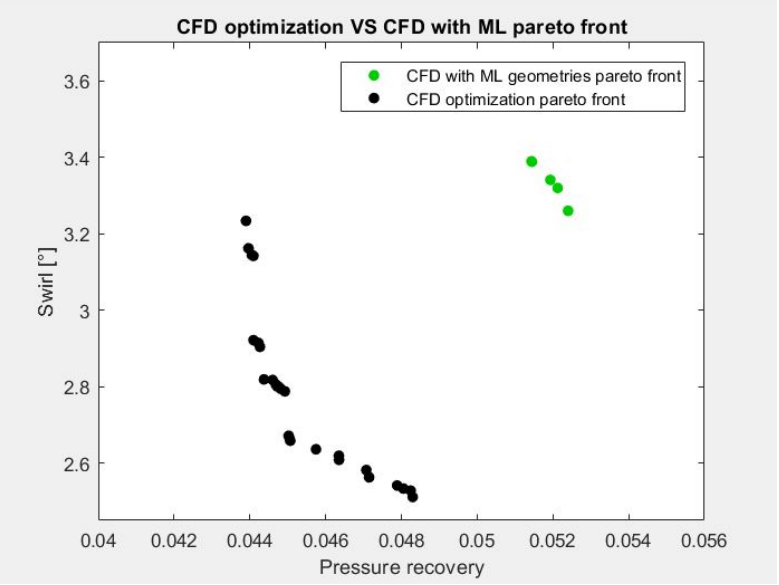

FIGURE 5.17: In green we observe the machine learning Pareto front evaluated with CFD. In black is shown the CFD Pareto front done by DalMagro. [6]

Just like with the other models before, only the Pareto front points evaluated with the machine learning are plotted in the following parallel coordinates in Figure 5.18. This plot helps to understand again the design space range studied with the optimization loops.

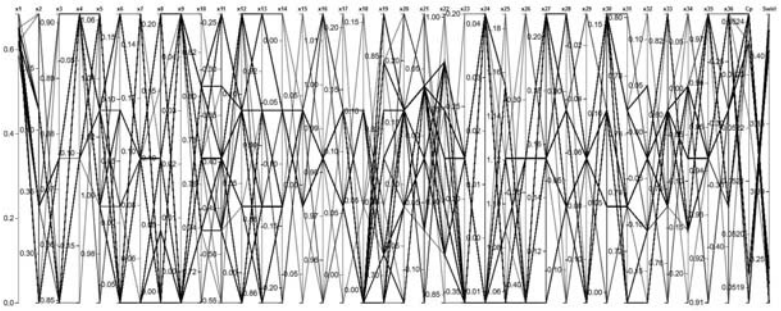

FIGURE 5.18: Paralel coordinates fifth optimization model.

Chapter 6

S-duct study using Lattice Boltzmann method

In chapter number five, the Pareto front points were all evaluated with the CFD fluent approach, which is a powerful tool to analyze the aerodynamic problems. The following idea is to analyze the same Pareto geometries with a second approach called the Lattice Boltzmann method.

The principal numerical methodologies investigated so far with the CFD fluent were the classical finite volume and finite element methods, based on the Navier-Stokes equation. However, even though those methods have been widely investigated, they still face essential drawbacks limiting their application to real industrial problems. Alternative particle-based methods are now emerging and consist of a real alternative to overcome these drawbacks. Among them, the lattice-Boltzmann method (LBM) is becoming one of the most interesting alternatives, and the development of new commercial LBM codes such as XFlow provides new alternatives for companies to solve complex engineering problems.

This chapter aims to introduce and investigate the capabilities of a modern Lattice Boltzmann method (LBM) implementation to characterize the unsteady features of the distorted flow field within S-duct aero-engine intakes.

The main features that potentially make such an approach attractive for unsteady flow predictions are the low computational cost and the minimum meshing requirements compared to conventional LES (large eddy simulation) or DES (Detached eddy simulation) approaches.

6.1 Turbulence modeling

The turbulence is a complex mechanism of fluid dynamics, and its modelling has always been a real challenge in CFD. In practice, almost all engineering applications are turbulent nowadays, and it is, therefore, a vital issue to accurately model the turbulence.

The Navier-Stokes equations have no analytic solution unless you are considering substantial simplifications and assumptions and their direct resolution (Direct Numerical Simulation, known as DNS) is therefore almost impossible on real engineering cases due to computational limitations and is limited to academic applications only. The turbulence modelling first started with Joseph Boussinesq, who proposed to introduce the concept of turbulent viscosity to relate turbulence stresses to mean flow. Many turbulence models have been developed based on this hypothesis to solve the Reynolds-Averaged Navier-Stokes equations and are known as RANS turbulence models. The most common ones are the Spalart-Allmaras, k - ϵ, k - ω. The RANS models are nowadays widely applied in the industry, mostly because of their low computational cost and their reasonable accuracy. However, they are empirical models and thus face the drawback of relying on a large number of constants that one must carefully calibrate to model the turbulence correctly. Each of those models may include different sets of constants and may predict a considerably different solution as separation prediction depends strongly on the turbulence. Also, the RANS turbulence models are modelling all eddy scales without distinction and filter all instantaneous variations induced by the turbulence since based on time-averaged equations. They are mostly limited to steady-state analysis.

An intermediate class of turbulence models was first proposed by Joseph Smagorinsky in 1964 to generalize the use of the turbulent viscosity applied on subgrid scales that are unresolved. This turbulence modelling approach is named the Large Eddy Simulation (LES) and proposes an excellent intermediate solution between the RANS and DNS since most of the turbulence is directly resolved. However, smaller scales are modelled to reach accessible computation times, as illustrated in Figure 6.1.

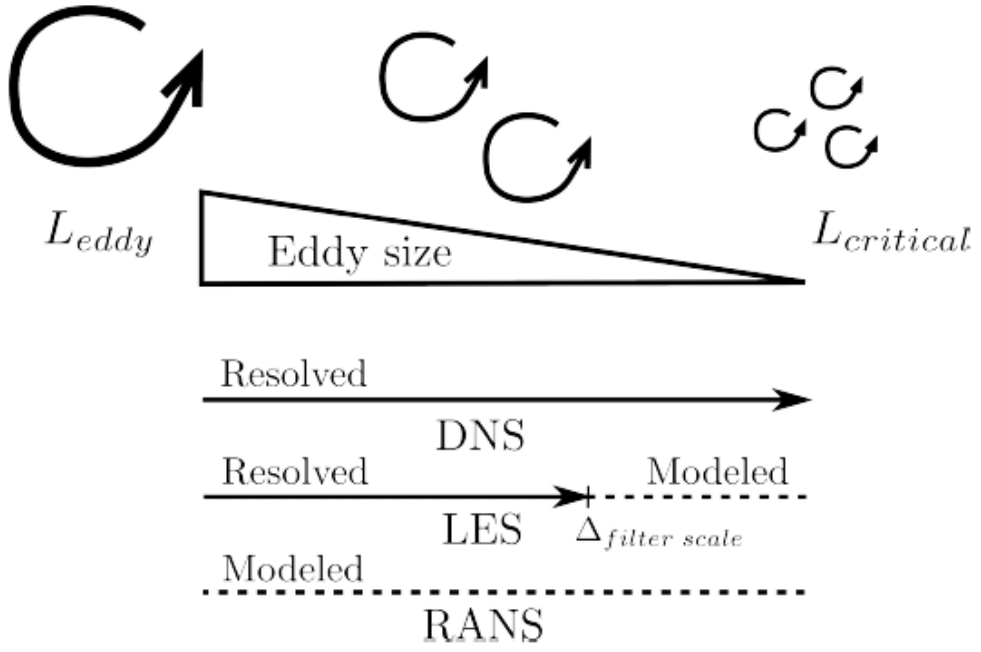

FIGURE 6.1: Turbulence modeling approaches comparison.

The LES turbulence models are not widely employed because the computation cost is relatively high compared to RANS models with the Navier-Stokes solvers. They are used for high-end applications or in industries that require transient simulations with highly turbulent and separated flows, where the RANS models tend to have weak predictions.

6.2 Particle-Based Method

Particle-based methods have been developed for several decades now and start to be emerging recently. Among them, the Lattice-Boltzmann Method (LBM) can solve many of the drawbacks presented by the traditional CFD methods. The meshing process is removed as the simulation relies on an automatically generated lattice, which is organized in an Octree structure, the LBM scheme allows the use of Large Eddy Simulation (LES) turbulence model at a low computational cost.

6.2.1 Lattice scheme

Some of the particle-based methods like LBM works on a spatial discretization named lattice, consisting of a Cartesian distribution of discrete points with a discrete set of velocity directions. The discrete set of velocities has a common terminology referred to the dimension of the problem and the number of velocity vectors DnQm, where n represents the dimension of a problem and m refers to the number of velocities direction.

For a three dimensional model, the most scheme is the D3Q19 model illustrated in Figure 6.2 and involves 19 velocity vectors defined by:

Other LBM implementations use the scheme D3Q27, which provides a higher number of velocity directions, and this provides a higher-order scheme despite a slightly higher computational cost. Indeed, the more velocity directions and the more equations are to solve.

6.2.2 Octree lattice structure mesh

The pre-processor generates the initial octree lattice structure based on the input geometries, the user-specified lattice resolution for each geometry, as well as the farfield resolution.

$$\vec{e}_i = \begin{cases} (0,0,0) & \text{i} = 0 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1) & \text{i} = 1,\ 2,\ ...,\ 5,\ 6 \\ (\pm 1,\pm 1,0),(\pm 1,0,\pm 1),(0,\pm 1,\pm 1) & \text{i} = 7,\ 8,\ ...,\ 17,\ 18 \end{cases}$$

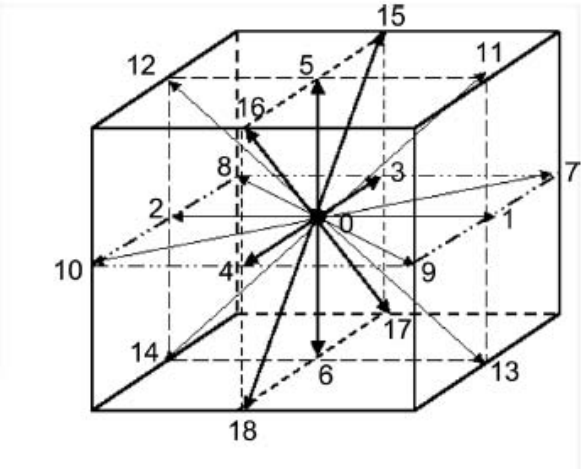

FIGURE 6.2: D3Q19 Lattice model of velocities discretization.

User-defined regions (sphere, box, cylinder, etc.) can also be created to refine arbitrary regions at the specified lattice resolution. The different spatial scales employed are hierarchically arranged. Each level solves spatial and temporal scales twice smaller than the previous level, thus forming the octree mentioned above structure Figure 6.3. Finite volume and elements usually use a global time step, which is inefficient for the coarser mesh cells, whereas the local time step approach in XFlow allows having always an adapted time step for every lattice size of your fluid domain.

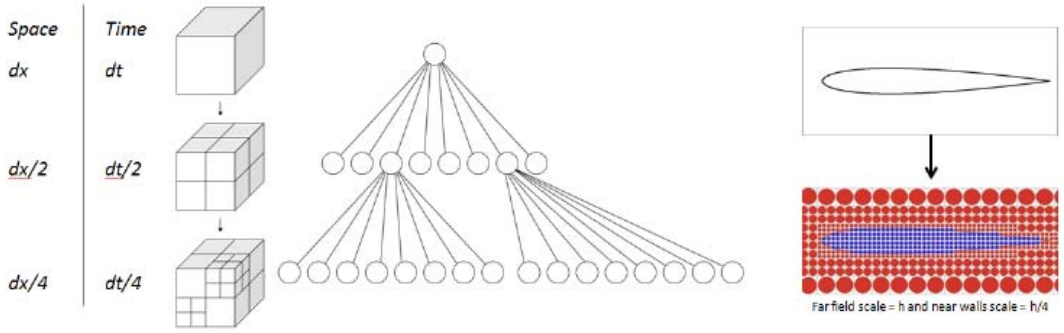

FIGURE 6.3: Octree lattice structure with different lattice resolution.

This initial lattice structure can be modified during the simulations by the XFlow solver based on several criteria. First, if the computational domain changes due to the presence of moving geometries, the lattice can dynamically be refined to follow the new position of the geometry every time step. Other adaptive refinement criteria to adapt the flow physics are also available. A refinement algorithm based on the level of vorticity is effective to dynamically refine the wake region Figure 6.4, characterized by high vorticity. Also, the free surface and multiphase flows can be refined dynamically at the free surface or interface, and therefore have an efficient tracking of the interface, including droplets and splashing.

FIGURE 6.4: Example of lattice structure with adaptive wake refinement.

6.3 Lattice-Boltzmann CFD modelling

For the CFD simulations, the commercial software XFlow was employed based on the Lattice Boltzmann method (LBM). The CFD software XFlow developed by Next Limit Technologies provides a fully Lagrangian particle-based kinetic approach, based on the lattice-Boltzmann method. The turbulence modelling and near-wall treatment are based on a state-of-the-art Wall-Modeled Large Eddy Simulation (WMLES) that offers advanced turbulence prediction.

The common characteristic to all the LBM models is their time-stepping model, based on a propagate-collide scheme, on top of a lattice discretization. The propagation step performed on a lattice enforces a constant timestep dt and a discrete set of velocities ($e_i i = 1, ..., b$) that ensure that the positions of the motion of the simulated particles are restricted to lattice sites. The set of velocities thus generates the lattice and for each lattice site, b probability distribution functions (PDFs) $f_i(\mathbf{r}, t)$ are stored. Where $f_i(\mathbf{r}, t)$ are the particle distribution function at time $\mathbf{t}$ and position $\mathbf{r}$.

The three-dimensional lattice structure used by XFlow includes 27 velocity directions, providing a higher-order spatial discretization scheme than the traditional LBM codes. In the continuous space (with discrete velocities), the Boltzmann transport equation can be written as follows:

$$\frac{\partial f_i}{\partial t} + e_i \cdot \nabla f_i = \Omega_i,\ i = 1, ..., b \tag{6.1}$$

Where Ω_i is the collision operator that computes a post-collision state conserving mass and linear momentum. This equation is discretized on the lattice as:

$$f_i(\mathbf{r} + e_i, t + dt) = f_i(\mathbf{r}, t) + \Omega_i(f_1, ..., f_b), i = 1, ..., b \tag{6.2}$$

The stream-and-collide scheme of the LBM can be interpreted as a discrete approximation of the continuous Boltzmann equation. The propagation step models allow the motion of the particle distribution functions along with discrete directions. At the same time, most of the physical phenomena are modelled by the collision operator, which also has a substantial impact on the numerical stability of the scheme.

6.4 Wellborn's S-duct LBM simulation

Due to the short time left only the Wellborn [3] S-duct shape simulation will be shown and explain below.

For the LBM simulations, the computational domain is created from Wellborn's work reported in Figure 6.5. The geometry analyzed with Xflow using the Lattice Boltzmann method has the following dimensions and parameters:

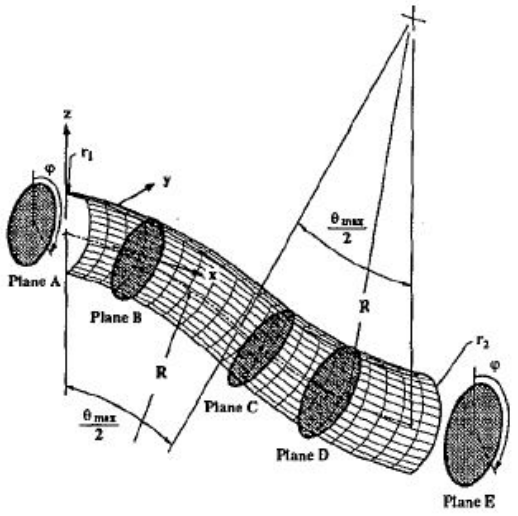

FIGURE 6.5: Definition of the S-duct shape.

The velocity inlet boundary condition was imposed in the inlet plane of the S-duct to prescribe a velocity of $V = 250m/s$. That is combined with the pressure outlet by the exit plane of the S-duct equal to 0. This configuration of the duct shape is shown in Figure 6.7.

Parameter	Value
Offset	2R(1-cos($\theta_{max}/2$)
L_{S-duct}	R
L_{inlet}	$8r_1$
L_{outlet}	$6r_2$
$L_{AIP} = L_{inlet} + L_{S-duct} + r_1$	$9r_1$+R
$L_{TOT} = L_{inlet} + L_{S-duct} + L_{outlet}$	$14r_1$+R

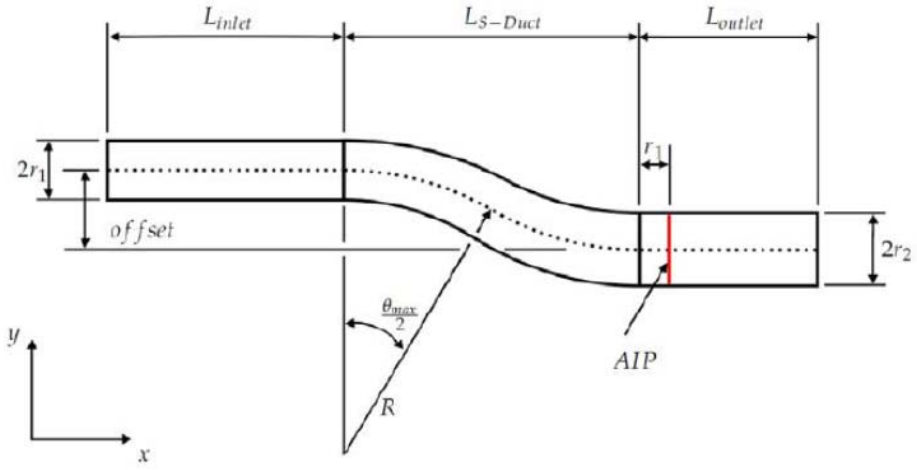

FIGURE 6.6: S-duct symmetry plane.

TABLE 6.1: S-duct domain dimensions

parameters	**values**
R	2.0811 [m]
$\frac{\theta_{max}}{2}$	30 [°]
Inlet radius (r1)	0.2081 [m]
Exit radius (r2)	0.2566 [m]
S-duct total length	5.2857 [m]
$\frac{A_2}{A_1}$	1.52

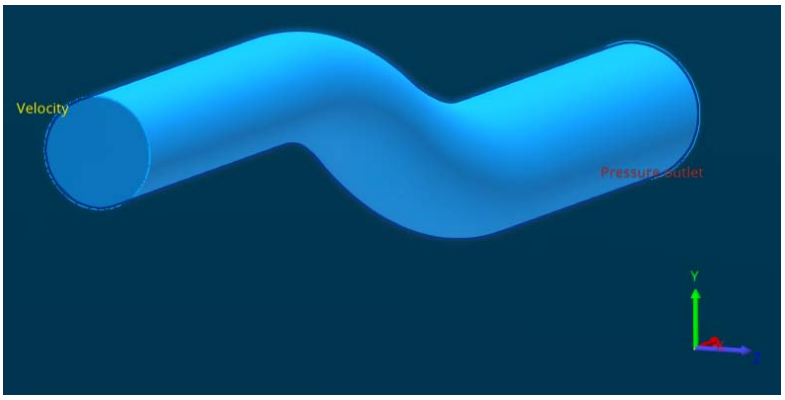

FIGURE 6.7: S-duct shape with boundaries inlet (velocity) and outlet conditions (pressure).

6.4.1 Lattice mesh

In Figure 6.8, a non-uniform lattice resolution mesh is shown as it is implemented with a local refinement near the walls.

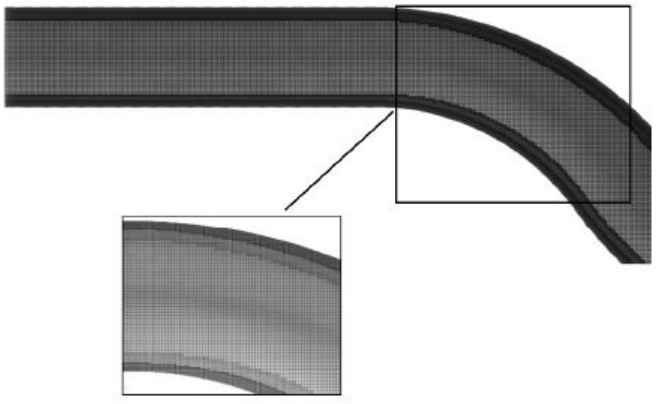

FIGURE 6.8: Lattice mesh structure.

The lattice mesh structure through a cutting plane is shown in figure 6.9. Three types of refinement are appreciable. The duct centre volume has meshed with a precision of 0.0025m, the second refinement going near the wall is meshed with a cell length of 0.000625m, and next to the wall, a precision of 0.0003125m is used. The last refinement next to the wall has to be smaller than the wall roughness set to 0 in this case.

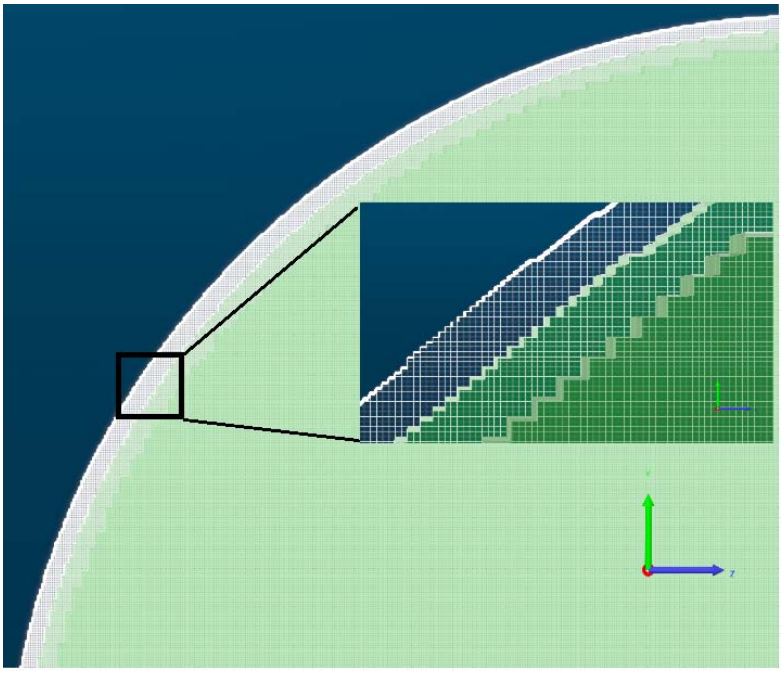

FIGURE 6.9: Lattice refinement.

```
2020-03-12 15:30:38 [ INFO  ] ## DOMAIN GENERATION ##
2020-03-12 15:30:38 [ INFO  ] XFlow Build 104.00
2020-03-12 15:30:38 [ INFO  ] Execution line: generateDomain3d ./sduct.xfp -mpi=8 -maxcpu=16
2020-03-12 15:30:38 [ INFO  ] Distributed computation enabled: 8 nodes.
2020-03-12 15:30:38 [ INFO  ] Computation limited to: 16 cores.
2020-03-12 15:30:38 [ INFO  ] License validation OK
2020-03-12 15:30:38 [ INFO  ] Num cpus detected: 16
2020-03-12 15:30:39 [ INFO  ] Seed: 0.5,0,0
2020-03-12 15:30:39 [ INFO  ] Generating Octree with 3 levels.
2020-03-12 15:30:39 [ INFO  ] Level 0
2020-03-12 15:30:45 [ INFO  ] Level 1
2020-03-12 15:36:16 [ INFO  ] Level 2
2020-03-12 16:19:42 [ INFO  ] Determining lattice/geometry intersection...
2020-03-12 16:33:51 [ INFO  ] Determining regions with fluid
2020-03-12 16:51:39 [ INFO  ] 0 discarded regions
2020-03-12 16:51:39 [ INFO  ] 1 identified regions
2020-03-12 16:51:39 [ INFO  ] Generating node map.
2020-03-12 16:58:47 [ INFO  ] Num active elements at level  0: 52739796 / 180978096
2020-03-12 16:58:47 [ INFO  ] Num active elements at level  1: 23114896 / 1447824768
2020-03-12 16:58:47 [ INFO  ] Num active elements at level  2: 188758937 / 11582598144
2020-03-12 16:58:47 [ INFO  ] Total number of elements: 264613629
2020-03-12 16:59:25 [ INFO  ] Writing domain
2020-03-12 17:10:04 [ INFO  ] Num boundary elements: 27173585
2020-03-12 17:11:59 [ INFO  ] Overall broken links: 190553578
2020-03-12 17:13:36 [ INFO  ] Domain successfully generated.
```

FIGURE 6.10: File output mesh size.

6.4.2 Lattice Boltzman simulation

To establish the full unsteady solution, a total simulation time of 0.0845712 s was set, which corresponds to approximately 4 through-flow times calculated based on the mean axial velocity and the length of the S-duct mean line. A time step dt, of 8.45712e-07 s, is needed to maintain a low instability parameter.

An amount of 100000 iterations will be calculated with a total simulation time of 21 days using 128 cores. We should consider only the last cycle of iterations for the post-processing to avoid the initial transient part of the simulation from the steady to the established unsteady solution.

Due to the long time the full simulation takes, only the first 4.2285e-6s will be shown in the next six figures. In this period, the fluid only has run 1.0571m through the duct. The velocity magnitude will be plotted in the figures below.

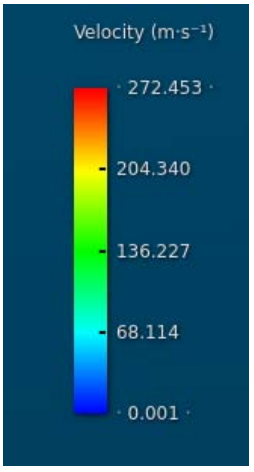

FIGURE 6.11: Velocity scale.

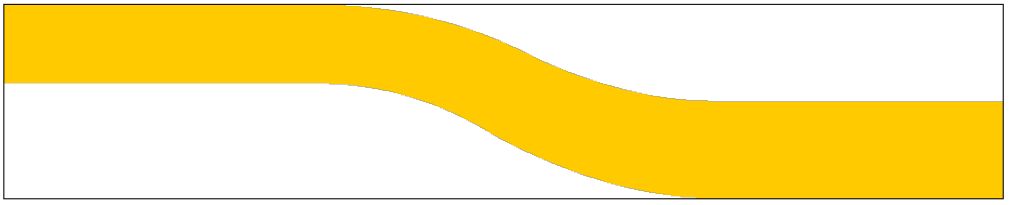

FIGURE 6.12: S-duct at initial time 0s (0m).

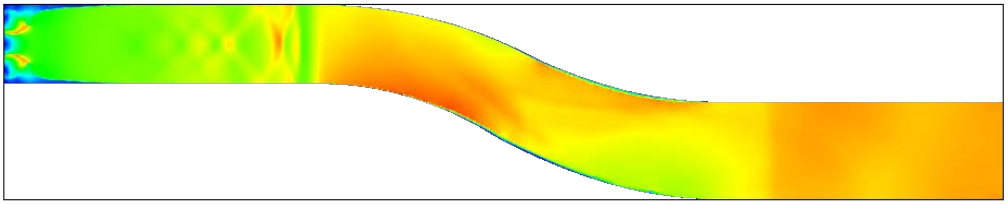

FIGURE 6.13: S-duct at time 8.4571e-4s (0.2114m).

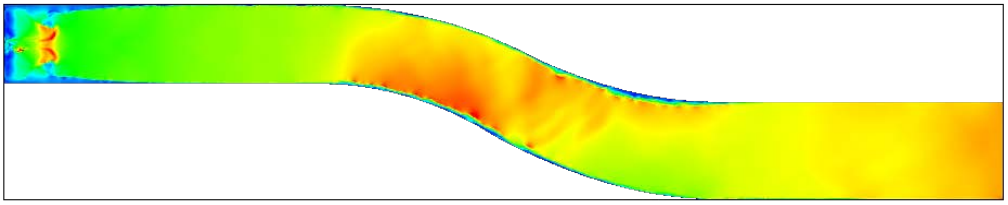

FIGURE 6.14: S-duct at time 1.6914e-3s (0.4228m).

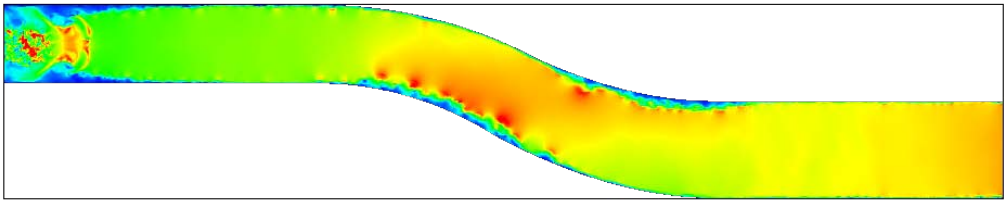

FIGURE 6.15: S-duct at time 2.5371e-3s (0.6342m).

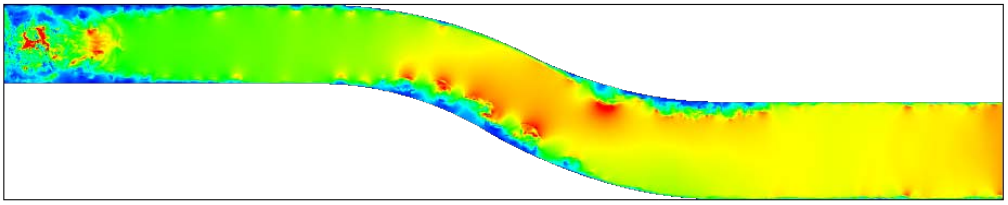

FIGURE 6.16: S-duct at time 3.3828e-3s (0.8457m).

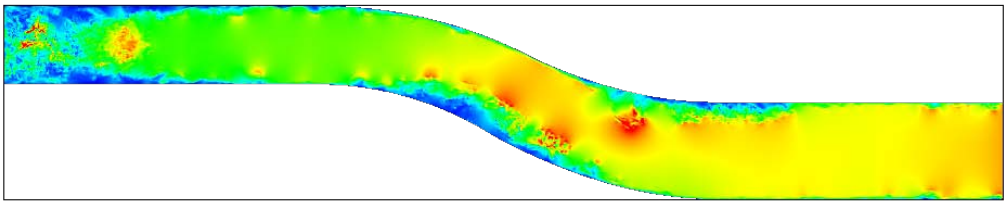

FIGURE 6.17: S-duct at time 4.2285e-3s (1.0571m).

Chapter 7

Conclusions & future works

This work aims to set up and run an optimization loop using state-of-the-art machine learning techniques. As shown in chapter four, five different configurations of machine learning codes were written using the python libraries of XGBoost. The purpose of these models is to predict the aerodynamics coefficients (pressure recovery and swirl angle) of an S-duct aero-engine intake.

Many analysis using CFD techniques were implemented in the last works on the same intake geometry, discovering the physical characteristics of this duct. The results coming from the above studies were all collected in a single file from where the machine learning techniques implemented are being built and tested. During the test process, for every model prediction, the sum of the mean squared error between the real value of the variable and the ML prediction is calculated. In this way, we can have an estimation of the most precise and reliable model. The scores we had in these calculations are reported in chapter four, in table 4.6. We can discover that the best two models for the pressure recovery predictions are models number three and number five with an MSE value of 0.0030 and 0.0032, respectively. Looking at the swirl number, we can find the lowest MSE reached by model number one and model number five.

Two differents optimizations were run with each model built before using two different starting points for the Tabu search optimizer. From each optimization loop, two different Pareto fronts were plotted and showed in chapter five with the red and the blue points.

To test the model's reliability again, only the Pareto front points were re-analyzed with the conventional CFD techniques. The result of this study is that every model worked precisely on the Cp and swirl predictions because all the points from the CFD fluent are very close to the ML ones. This point should be picked up in the furthers works where the CFD analysis can be applied with the LBM method investigating deeply the unsteady features of the distorted flow field going through the S-duct of an aero-engine intake.

We found that every ML model has a smaller MSE, and as a consequence, the highest reliability as much randomly we choose the data during the learning and the testing process. Starting from here, furthers work should use the codes runned and improve them, adding more constraints parameters available in the python packages. A more significant learning database should be made for the ML models; this will help on the random geometry choice form where the code learns.

Bibliography

[1] A. D'Ambros, T. Kipouros, P. Zachos, M. Savill, and E. Benini. Computational design optimization for s-ducts. *Design MDPI, 2(4):36*, 2018.

[2] Anne-Laure Delot and Richard Scharnhorst. A comparison of several cfd codes with experimental data in a diffusing s-duct. In *49th AIAA/ASME/SAE/ASEE Joint PropulsionConference*, page 3796. 2013.

[3] Wellborn Steven R., Reichert Bruce A., and Theodore H. Okiishi. Study of the compressible flow in a diffusing s-duct. *Journal of Propulsion and Power, Volume 10, Number 5*, September 1994.

[4] R. Tridello. Comparison of genetic and tabu search alghoritms in aerodynamic design of s-ducts. Master's thesis, Universitá degli Studi di Padova, 2017.

[5] A. Rigobello. A multi-objective shape optimization of an s-duct intake through nsga-ii genetic algorithm. Master's thesis, Universitá degli Studi di Padova, 2015.

[6] D. Dal Magro. Implementation of uncertainty management techniques in the design of s-ducts intakes. Master's thesis, Universitá degli Studi di Padova, 2018.

[7] Enrico Manca. Unsteady aerodynamic investigation of the flow within an optimized s-duct intake. Master's thesis, Cranfield University, 2016.

[8] Marco Barison. Shape optimization of highly convoluted intakes using genetic algorithms and metamodels. Master's thesis, Cranfield University, 2016.

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

[10] H. W. Coleman and W. G. Steele. *Experimentation, Validation and Uncertainty Analysis for Engineers*. John Wiley & Sons, Hoboken, NJ, USA, 3rd edition, 2001.

[11] Fred Glover. Tabu search: Part i. *ORSA Journal on Computing*, 1(3), 1989.

[12] Timoleon Kipouros, Daniel M., Jaeggi, William N., Geoffrey T. Dawes, Parks A., Mark Savill, and P. John Clarkson. Biobjective design optimization for axial compressors using tabu search. *AIAA Journal*, 46(3):701-711(DOI: 10.2514/1.32794), March 2008.