POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Understanding plant health status from electrical impedance using Neural Networks

Supervisors

Prof. Maurizio MARTINA

Prof. Danilo DEMARCHI

Ph.D. Umberto GARLANDO

Candidate

Huicai LIU

Academic year 2020-2021

Summary

This thesis is about an exploration in the world of machine learning to find a way to predict the health state of a plant, in particular tobacco plants. The prediction is made possible by using an intrinsic measurement of the plant, which is the electrical impedance, as well as the surrounding environment, such as temperature, air humidity, soil moisture, ambient light, and the time when the measurements are taken.

Nowadays, smart agriculture systems are mainly based on environmental data sensors and the elaboration of visual data. However, visual sensors and cameras are expensive, and with the environmental information, the status of the plant is not directly checked. In order to overcome these limits, some intrinsic value of the plant should be measured by a simple, low power, and low-cost system. Students and professors from Politecnico di Torino and Tel Aviv University have found a new idea by measuring the electric impedance of the stem of a tobacco plant.

Predicting the plant status by knowing stem impedance, module and phase, and the environmental measurements can be seen as a classification problem. Therefore, the most suitable machine learning algorithms are the supervised learning ones. The most common supervised learning algorithms are the k-Nearest neighbor algorithm, Decision Tree, logistic regression, Support Vector Machine, and Neural Networks. Finally, Artificial Neural Networks are chosen for this work for many reasons, such as their ability to learn and model non-linear and complex relationships; they do not impose any restrictions on the input variables.

Preparing the dataset for the training is one of the most challenging parts, not only because the sampled data, the inputs, must make sense, but also the labels, i.e., the outputs, must be correct. The training set is made by the data from 4 different plants from March 24th to May 4th. The plants' impedance and environmental value are saved every hour. In order to label the plants correctly, the pictures of the plants in this period are looked at one by one, and the health status is determined based on the leaves conditions of the tobacco plants.

Once the dataset is ready, the neural network has been implemented in Python by exploiting the open-source library TensorFlow. The model defined is formed by

two hidden layers, each composed of 6 nodes. The performance reached is 78.6% of training accuracy and 80.8% of testing accuracy. Furthermore, this model is used to predict the outputs of the data from May 6th to June 7th, reaching 90% of total accuracy on the four plants. These are not excellent results, but they are definitively something to say that it may be possible to detect the plant status with a neural network.

It is interesting to see if it is possible to predict the plant status with only impedance values. Testing accuracy of 70.6% is reached, but this result is tricky because there maybe be overfitting. Using this model on the future data from May 6th to June 7th, the performance is terrible, below 40% of accuracy.

In order to prove the importance of the impedance, the *"reductio ad absurdum"* approach is used. If the impedance module and phase are not considered as features for the neural network, they should not affect the performance since they are not relevant, but instead, the accuracy has dropped to 67%. Maybe it is due to the reduced number of features; therefore, the other five features were taken off two by two, leading to ten different combinations. In almost all cases, the accuracy remains relatively high to 80%. The accuracy decreases to 65% when one of the two removed features is the soil moisture.

Then three plants are used as training set and the fourth is used for the testing. Four cases can be considered:

- plant 1 as testing (36.5% of accuracy)
- plant 2 as testing (82.7% of accuracy)
- plant 3 as testing (87.1% of accuracy)
- plant 4 as testing (58.6% of accuracy)

The bad performance is mainly due to each plant's different impedance module range for both the first and subsequent networks. For example, during this period, the impedance of plant 2 remains almost constant while the value of plant 4 increases since it is getting dry, and even if they are in two different statuses, their impedance values are very similar. Two different methods are tried to solve this problem. The first one considers the impedance difference between two adjacent samples as an additional feature for the training. The second one is to consider the difference after a polynomial fitting of the data, filtering out the daily cycle variation and considering only the general trend of the impedance. Nevertheless, unfortunately, in both cases, the accuracy remains similar to the first implementation.

The authors have also observed the daily variations of the impedance; in particular, they found out that these variations have two different behaviors when the plant is healthy and getting dry. This information can be implemented with neural networks considering the past values of module and phase as features. For example,

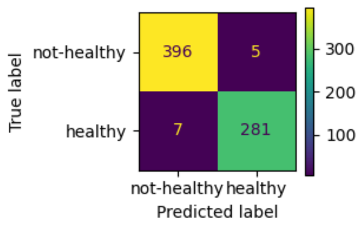


Figure 1: Confusion matrix of the testing dataset

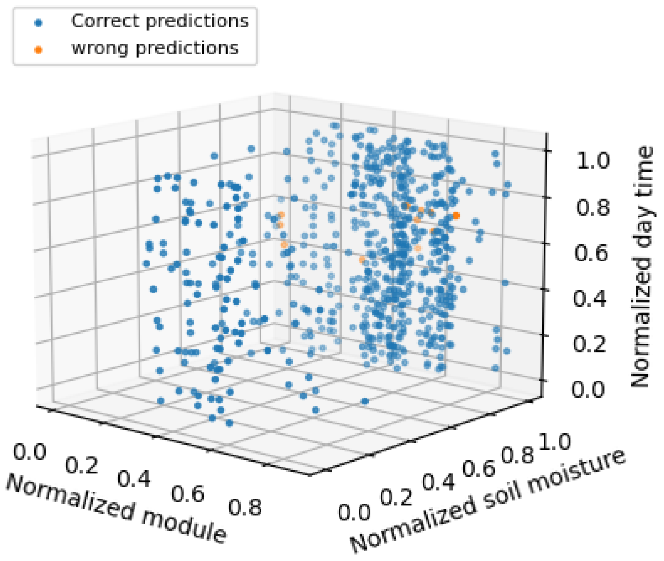


Figure 2: Predictions of the testing dataset

considering 24 times the module and phase, which means 24 hours of samples, the neural network reaches a training accuracy of 98.8% and testing accuracy of 98.3%; the testing results are shown in figures 1 and 2, in the former the confusion matrix and in the latter a simplified 3-D plot with only three features, where the wrong predictions can be observed. This type of network also has excellent performances for future prediction. This approach is also applied to the training with three plants and testing with the fourth one, but the performances are awful, and the problem of different modules' value for different plants is still unsolved.

This thesis work demonstrates the importance of impedance and soil moisture for detecting a tobacco health status; this result is confirmed by a simple SVM implementation with a radial basis function kernel. The neural network has reached significant results, but it is still not ready to be used on an unknown tobacco plant, which means not used during the training phase. Possible solutions may be training with samples over a broader period or introducing more features regarding the plants, such as age or stem radius.

Acknowledgements

"Be Water, My Friend. Empty your mind. Be formless, shapeless, like water. You put water into a cup, it becomes the cup. You put water into a bottle, it becomes the bottle. You put it into a teapot, it becomes the teapot. Now water can flow or it can crash. Be water, my friend."
B. Lee

I am incredibly grateful to my supervisors, professors Martina, Demarchi, and Garlando, who gave me the opportunity to work on this subject. Special thanks to Garlando, who helped, supported, and followed me during all these months.

I'm deeply indebted to my family, my parents, my sisters, and my nephews. No words can express their presence, what they gave me and allowed me to do.

To Elettra, someone special who never let you down and is always present.

To Pietro, the first person that I met at the university. A friend who taught me countless things.

Thanks to my old and new roommates, Antonio Francesco, Pierpaolo, Giacomo, and Paolo. I will never forget the study and gaming nights spent together.

To my old friend Francesco, who I met again at the university after five years of adventures in that old room. To my colleges Francesco, Francesco, Marco, Edoardo, Antonio, Alessandro, Paolo, Gaspare. Thank you for the calls and meetings that lasted an eternity. It was my great pleasure to work with you.

To Fabrizio, with whom you always eat well.

To Malio, Valentino, Federico, Yubo, and Junyi, who are distant but at the same time are always close.

To Huqing, Jiang and Stefano, with whom I have spent my life in the study room.

To my friends, Alessio, Simone, Paola e Federica. Thank you for letting me discover Torino. To Fabrizio, even if you're a physicist.

Table of Contents

List of Tables

List of Figures

Acronyms

ADC

Analog Digital Converter

AI

Artificial Intelligence

ANN

Artificial Neural Network

CNN

Convolutional Neural Network

GHG

Greenhouse Gas

GPIO

General Purpose Input Output

IoT

Internet of Things

I2C

Inter-Integrated Circuit

KNN

k-Nearest Neighbor

MLP

multi-layer percepton

RBF	Radial Basis Function
SVM	Support Vector Machine
UNEP	United Nations Environment Programme
UN	United Nations

Chapter 1

Introduction & Background

In the 21st century, the world faces several environmental issues such as global warming, air pollution, ocean acidification, ocean plastic pollution, GHG emission, intensive food production, hydric stress and water scarcity, desertification, overpopulation, waste management, and more. These problems and others are well described in a foresight report published in 2012 by UNEP[1]; in particular, the UN expects the world population to exceed 9 billion by 2050, which will lead to an increase in demand for food. However, on the other side the effects of desertification cannot be neglected[2]. Desertification refers to the persistent degradation of dryland ecosystems caused mainly by human activities: unsustainable farming that depletes the nutrients in the soil, mining, overgrazing, clear-cutting of land, when the tree and plant cover that binds the soil is removed, urbanization, deforestation, and also excessive watering. Climate change leads to more frequent extreme weather events such as droughts and coastal surges, and warmer average temperatures could amplify this soil degradation. In 1994, the United Nations established the Convention to Combat Desertification (UNCCD), through which 122 countries have agreed to safeguard arable land, repair degraded land, and manage water supplies more effectively. Moreover, technology is often an excellent means to solve problems; today, smart agriculture offers a possible path toward sustainable agriculture.

1.1 Smart agriculture

Smart agriculture leads to the third revolution in the history of agriculture. It refers to introducing modern technologies, such as IoT, sensors, location systems, robots, and artificial intelligence in the farming world. Examples of applications are the measurements of light, soil moisture, temperature, or the precision irrigation of the plant roots only when needed based on the environmental condition. The

goal of smart farming is to increase the quantity and improve the quality of crop harvest, saving all the resources such as water, energy, fertilizers, pesticides, and human resources.[3].
An overview of the IoT applications is described in the article [4]. Many countries such as China, Taiwan, Thailand, Malaysia have developed successful IoT applications for both precision agriculture and greenhouse monitoring. The former is based on environmental monitoring, divided into weather, soil conditions, plant disease, and irrigation. The latter requires high precision controlling, monitoring, and tracking, which is reached thanks to a real-time monitoring system formed by a Wireless Sensor Network and a Smart Control Panel. All of these operations are made possible by the smart agriculture software, examples of open source software are *Tambero*, *FarmOS*, *Trimble*, *Farmathand*, *Tania* and *FarmaRexx*.
An overview of the plant and environment sensors is presented in the article [5]. The most common plant sensors are visual sensors. The application of machine learning algorithms on visual data makes it possible to have high accuracy for classifying diseased plants and detecting fruit defects. Some researches have demonstrated that volatile organic compounds emitted by the plants are related to their current status. For this purpose, some low-cost and mobile sensors have been developed, such as optical or electrochemical ethylene sensors or jasmonate sensors. A new kind of sensor is related to the impedance measurement, on which the work of the current thesis is based. Even if the related researches are at the beginning, it has great potential since the impedance measurement is a low-power procedure, and the system could be small and straightforward.
The environmental measurements, as introduced before, are mainly related to light intensity, temperature, relative humidity, and soil moisture. Knowing these parameters gives information about the plant's surroundings and helps understand the plant's status. Several solutions and commercial sensors are available for these tasks.

1.2 Impedance

As mentioned before, plant sensors are becoming more significant since the environment sensors cannot directly detect the status of a plant, such as a disease, a parasite, or even the death of the plant itself. Among the different kinds of sensors, new studies about electrical impedance are rising. The impedance studies, proposed by the authors in [6], [7], [8] and [9], show how the impedance can be measured, how the watering events are correlated to the plant stem electrical impedance, and how it depends on the daily activity of the plant. This section is an overview and summary of these studies, which will be this thesis's starting point.

1.2.1 Impedance measurement

The method to measure the impedance introduced before is described in detail in [6]. In this experimental setup, the four-point probe measurement is used. In general, this technique is commonly used because it can eliminate the contributions of the contacts. In particular, a known current is injected into the two outer probes, and the voltage is measured on the inner probes.

The article's authors measured the stem impedance of a young tobacco plant, which had 7 mm as diameter. In particular, the four probes, as shown in figure 1.1, were inserted at different positions on the plant stem to a depth of 6.8 mm, reaching in this way the vascular tissues. The first step was to perform a DC analysis by applying a voltage in the range 0 V-20 V and varying the distance **d** of the probes.

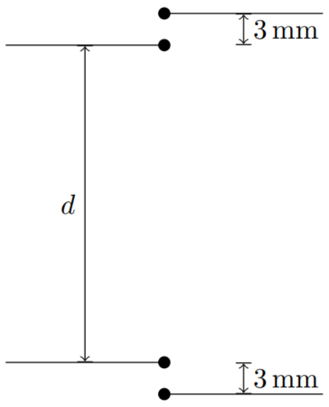


Figure 1.1: Experimental setup with four point probe technique. Image from [6].

The results are shown in figure 1.2, the stem provides a constant value of resistance with the DC analysis, and it increases with the distance d between the inner probes.

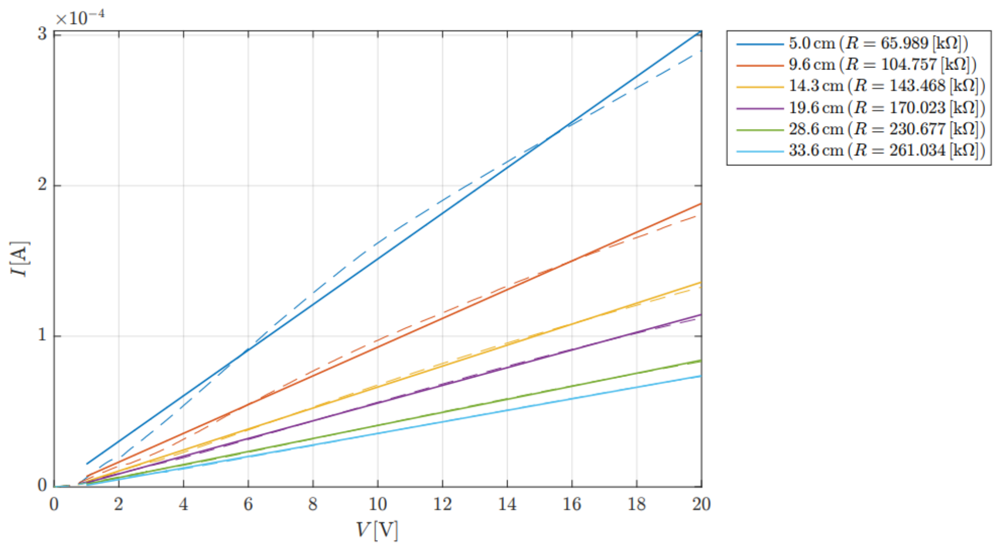


Figure 1.2: Current-Voltage Characteristic of tobacco plant stem. Image from [6].

After that, an AC analysis was performed, and the spectrum, shown in figure 1.3, has a low pass filter-like behavior; the interesting thing is that the data can be reasonably fitted with the Randles model.

Usually, the standard four-point probe setups use a professional impedance analyzer, leading to very high accuracy. Nevertheless, this kind of solution is not feasible for use in the field, especially in large numbers, and it contradicts what is said in section 1.1. Fortunately, the authors have solved this problem by proposing a low-cost end-to-end system. The system is made using Texas Instruments' LMP91000 AFE potentiostat, Analog Devices' AD7896 A/D converter(ADC), and a Raspberry Pi Zero W. The total cost of this setup is around USD 10 (price in 2018).

1.2.2 Experimental setup

The system proposed by the authors in [6] and briefly described in the previous section(1.2.1) is a good starting point, but it is surely not enough for a smart agriculture system. The setup shown before was only to measure an electrical parameter of a plant, but how this parameter is related to its health status and how the environmental parameters can be measured are still unknown. Luckily, these works have been done by other authors in [8], [7] and [9].

A first version of the in-vivo monitoring system is well described in [9]. In particular, the system is made of three generic sensors:

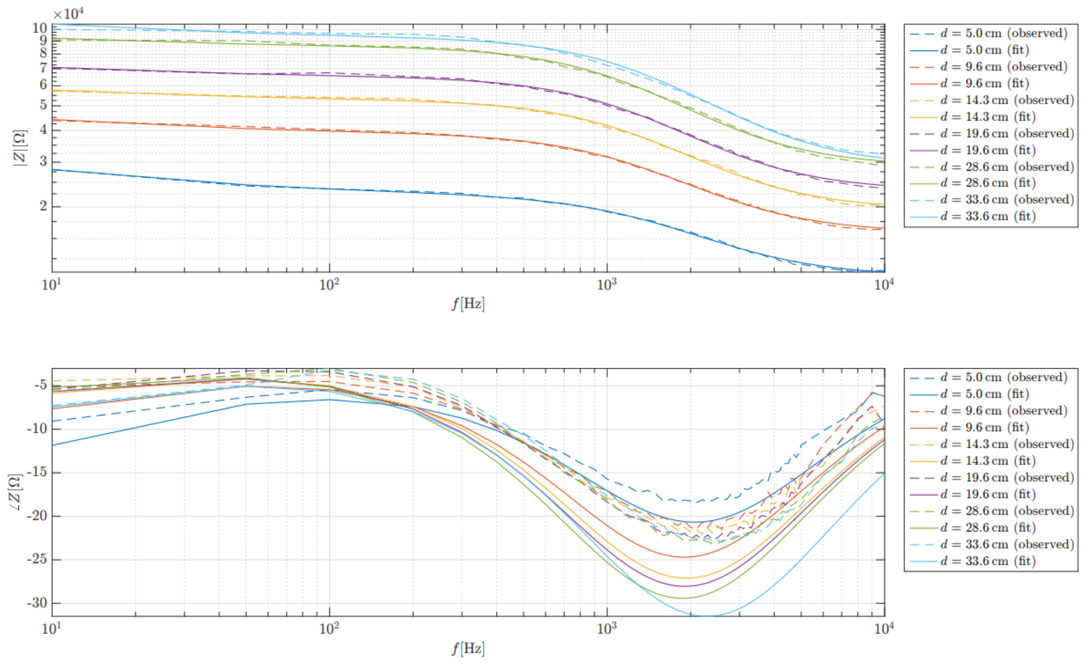


Figure 1.3: Impedance spectrum of a 4 month tobacco plant. Image from [6].

- HDC2080 (Texas Instruments Ltd)[10]. It is a temperature and relative air humidity sensor that provides high accuracy measurements with very low-power consumption. It uses the I2C protocol and the relative humidity is measured from **0 %** to **100 %** with a typical accuracy of ±**2%**, while the operating temperature range is from **-40°C** to **85°C** with typical accuracy of ± **0.2°C**;

- MAX44009 (Maxim Integrated Ltd)[11]. This low-power ambient light sensor with an internal ADC allows the user to get the ambient light value in lux directly. It has an ultra-wide 22-bit dynamic range from **0.045 lux** to **188000 lux**;

- 200SS WATERMARK Sensor (Irrometer Ltd)[12] - It is used for soil moisture monitoring. This kind of sensor acts as an artificial root, which means that it exchanges water with the surrounding soil as a plant does. In this way, the effort required by a plant to extract water from the soil is measured, and no complicated calibrations for individual sites or different soil are needed.

All these sensors are connected to the GPIO port of a Raspberry Pi using the I2C protocol as shown in figure 1.4, while the block diagram is shown in figure 1.5. As can be seen in the figures, the ADS1015 converter is used as a readout circuit for the soil moisture sensor. The system is then controlled using a Python interface developed for data collection and monitoring.

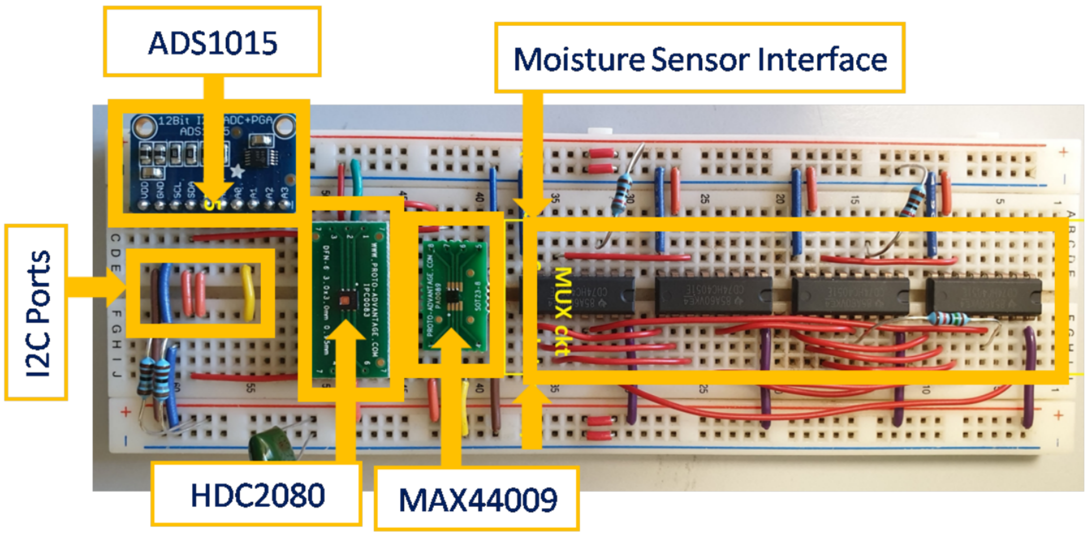


Figure 1.4: Generic sensors setup on breadboard. Image from [9].

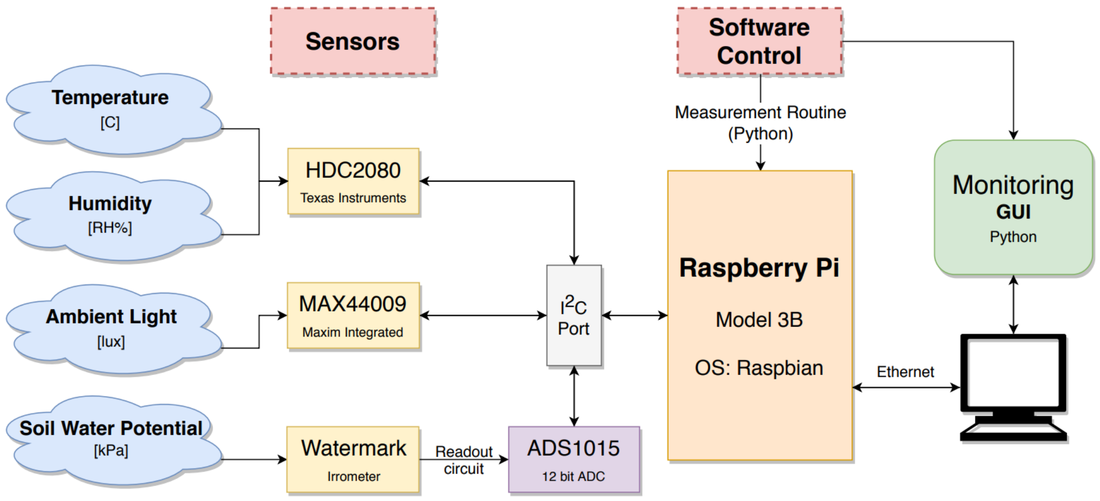


Figure 1.5: Block diagram of the sensor system. Image from [9].

It is possible to notice that the system described before contains only environmental sensors; the impedance measurement is done separately but simultaneously. An impedance analyzer (Agilent 4294a) is connected to the plant stem, and a designed LabView© software interface is used to control the samples as shown in figure 1.6.

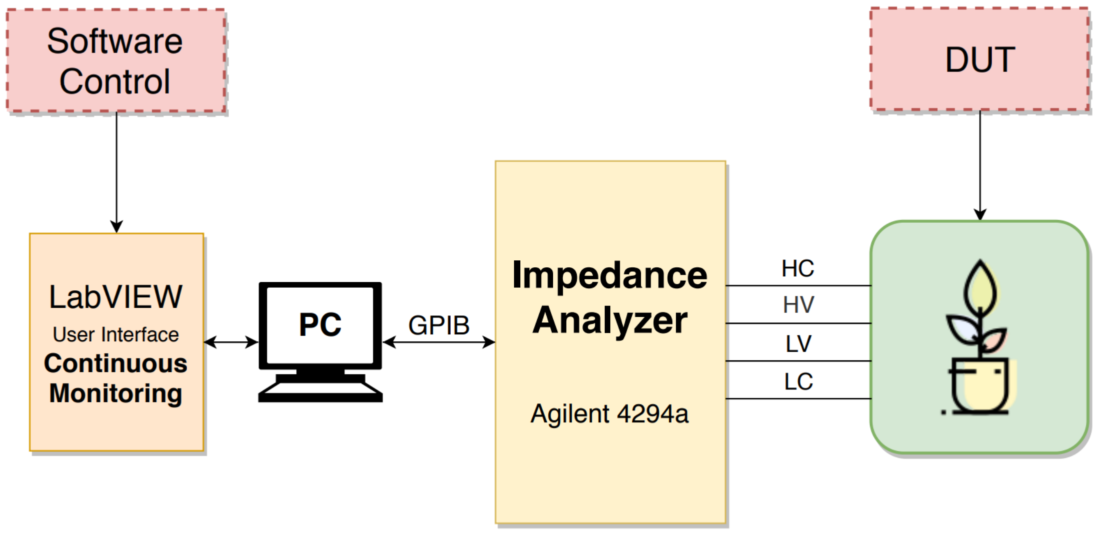


Figure 1.6: Block diagram of impedance measurement system. Image from [9].

With this experimental setup, it is then possible to monitor both the environmental changes and the impedance values and, above all, the dependency of the parameters among them and their variation in time.

1.2.3 Impedance variation

The authors in [8] explored the possibility of optimal green plant irrigation based on the value of impedance and not only on soil moisture. They tried to find a correlation between impedance changes and watering events and then the dependence on the soil moisture value. The experimental setup used by these authors was slightly different from the one described in the previous section(1.2.2). As shown in figure 1.7 a multiplexing circuit has been introduced in order to monitor with the same setup two different tomato plants.

For both the plants the environmental parameters and impedance values are measured, with the difference that the first plant is often watered while the second one is not.

In figure 1.8, it is possible to notice how the changes in impedance depend on the watering events. In particular in figure 1.8.a is shown how the watering events, marked with dashed lines, influence both the value of impedance modulus and soil moisture. While in figure 1.8.b the dependency between impedance modulus and soil moisture is observed. In figures 1.8.c and 1.8.d the phases of the measured impedance are presented. From these graphs it is possible to notice that, in case of

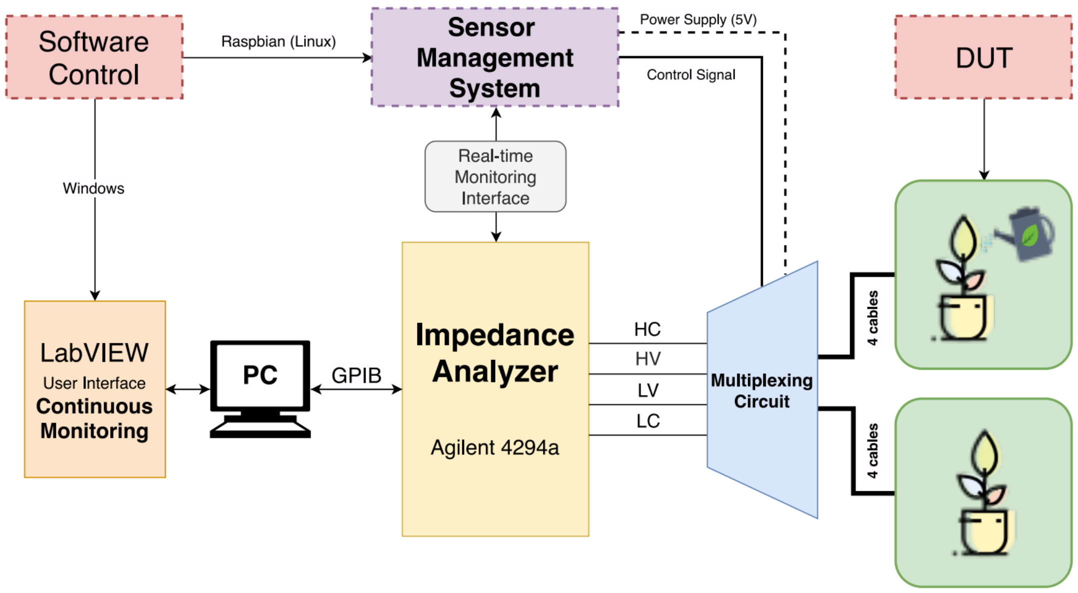
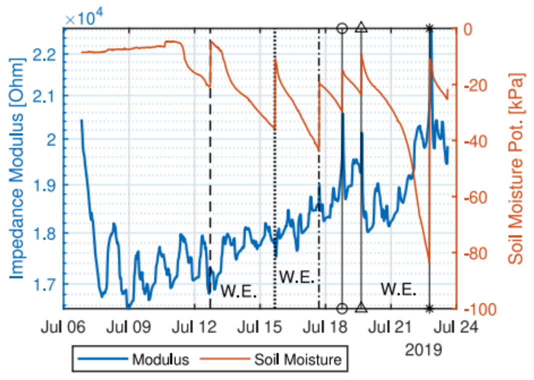


Figure 1.7: Impedance measurements and environment sensors for two different plant. Image from [8].

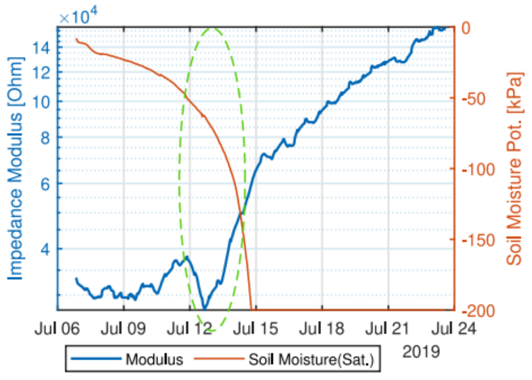
the watered plant, both the modulus (fig. 1.8.a) and the angle (fig 1.8.c) are quite constant(few kΩ for the modulus). The variations, for both modulus and phase, are due to a periodic daily behavior of the plant, which is well explained in article [7]. On the right side, when the plant is not watered, a steady (and mostly monotonic) increase in the impedance up to about one order of magnitude is observed, as well as a significant and identifiable change in the impedance angle.

The correlation between soil moisture and impedance is then demonstrated mathematically, following two different paths. The first one exploits the Pearson correlation coefficients, while the second method is about the Granger causality. The Pearson coefficient ranges from -1 to 1 and evaluates the linear correlation among different data series. A correlation coefficient equal to 1 means the strongest positive correlation, and similarly, the perfect negative correlation is associated with -1. A coefficient near 0 means that the two quantities are completely uncorrelated.

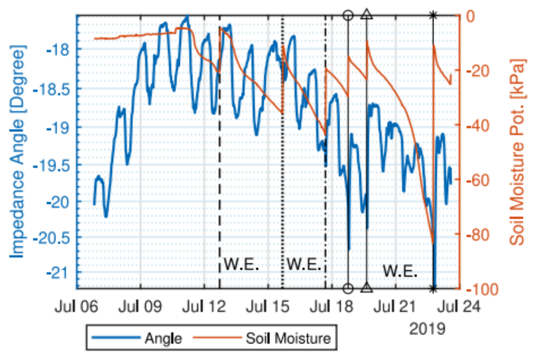
The coefficients were computed by using the Python Pandas package for both watered and not watered plants. The results are shown in figure 1.9, it can be easily noticed that the impedance module and angle are highly correlated. Furthermore, the soil moisture shows very high values in the not watered case, while the correlations with the other ambient measurements are almost zero. On the other side, in the watered case, the correlation with the soil moisture is present but not very strong; in fact, the impedance is also related to the other parameters.



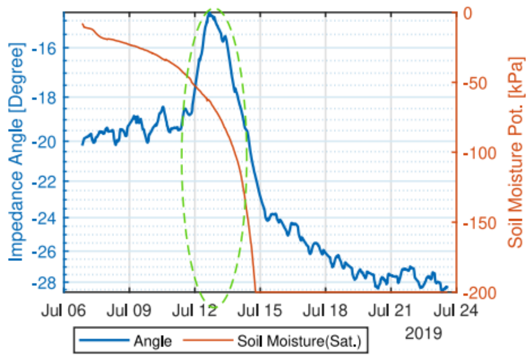
(a) Impedance modulus, watered



(b) Impedance modulus, not watered



(c) Impedance phase angle, watered

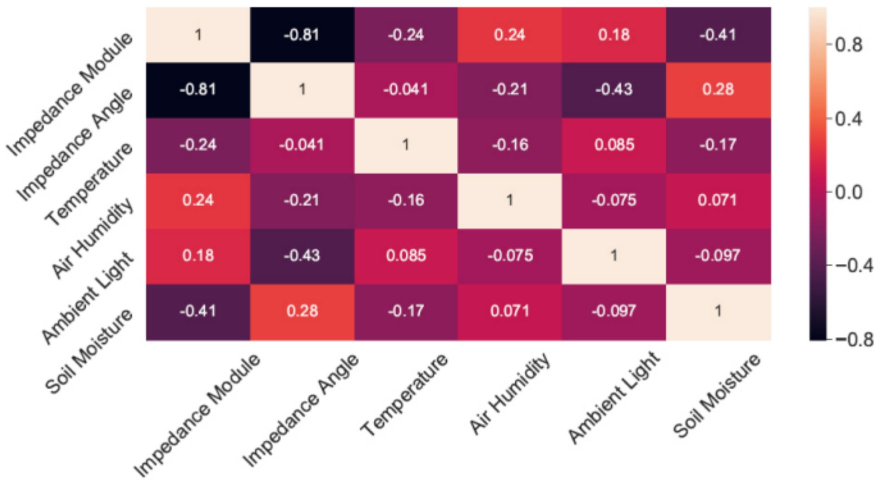


(d) Impedance phase angle, not watered

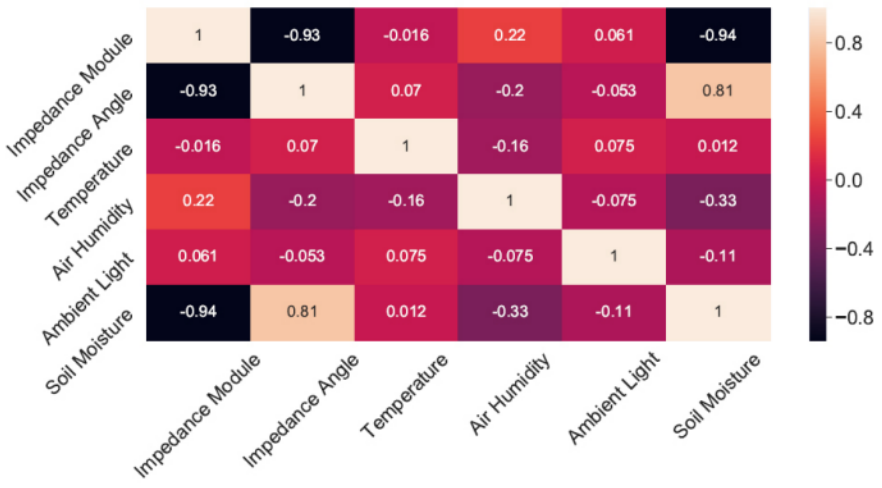
Figure 1.8: Body electrical impedance (modulus and phase angle) for two tomato plants, regularly watered and not watered respectively at 1kHz. Image from [8].

However, in both cases, soil moisture is a crucial parameter for the impedance values. Nevertheless, this relation **doesn't mean** that the stem impedance is highly related to watering events since the relation between two parameters where one is affected by the other one is defined as causality. For this purpose, the Granger causality test has been exploited, but before that, the "difference" technique was applied to prevent the non-stationarity of the data.

The results are shown in figure 1.10, the matrices should be read in this way: if the number in the column is smaller than 0.01, then the quantity corresponding to the column is “Granger causing” the row's quantity with a 99% confidence level. Thanks to the matrices, it can be easily observed that the impedance module and phase are highly dependent on the soil moisture in both watered and not watered



(a) Correlation matrix, watered

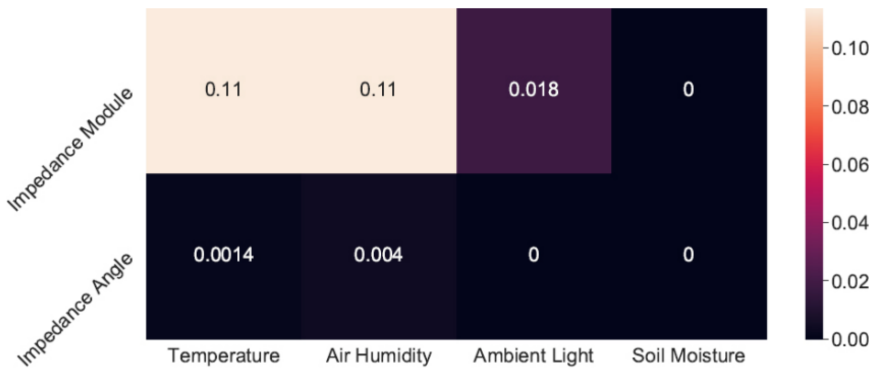


(b) Correlation matrix, not watered

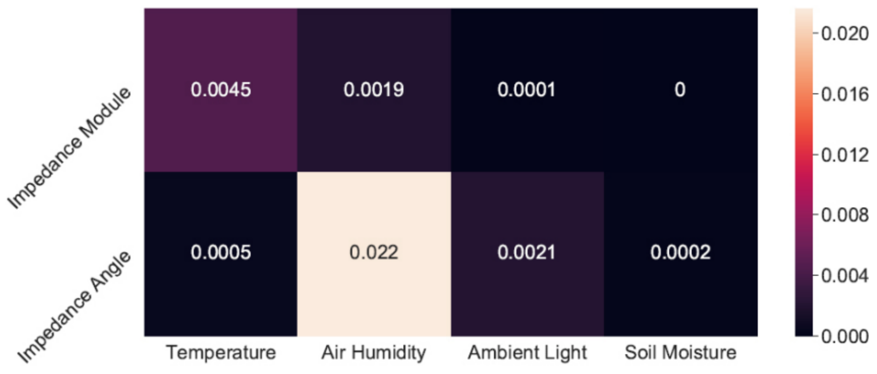
Figure 1.9: Correlation matrices for water and not watered plants for impedance at 1 kHz. Image from [8].

cases, while the other metrics show different behaviors in the two experiments.

These studies demonstrated the correlation between the impedance and the environmental parameters such as soil moisture, ambient light, air humidity, and temperature and how the impedance depends on the other measurements. Furthermore, the dependency between impedance variations and watering events is proved. In this way, it is possible to understand when the plants should be watered, leading to a plant irrigation system.



(a) Granger matrix watered plant



(b) Granger matrix not watered plant

Figure 1.10: Granger matrices for water and not watered plants for impedance at 1 kHz. Image from [8].

1.2.4 Daily cycle variations

As mentioned before, it seems like that the variation of the stem impedance changes daily; the authors well study this behavior in [7]. The experimental setup used this time was upgraded; it can measure the impedance and the surroundings of **4** different plants simultaneously as shown in figure 1.11. In this study, four tobacco plants are monitored; modular, wireless, small, and low-cost data acquisition nodes are used for environmental measurements. Environmental parameters are measured every 30 minutes, while the impedance spectrum is sampled every 15 minutes by a impedance analyzer (Keysight 4294a). Since the same analyzer is used for all the plants, a multiplexing circuit is needed; therefore, each plant stem impedance value is stored every hour. A webcam is also connected to the Raspberry Pi for monitoring visually the plants under test, taking a picture synchronously with the impedance measurements.

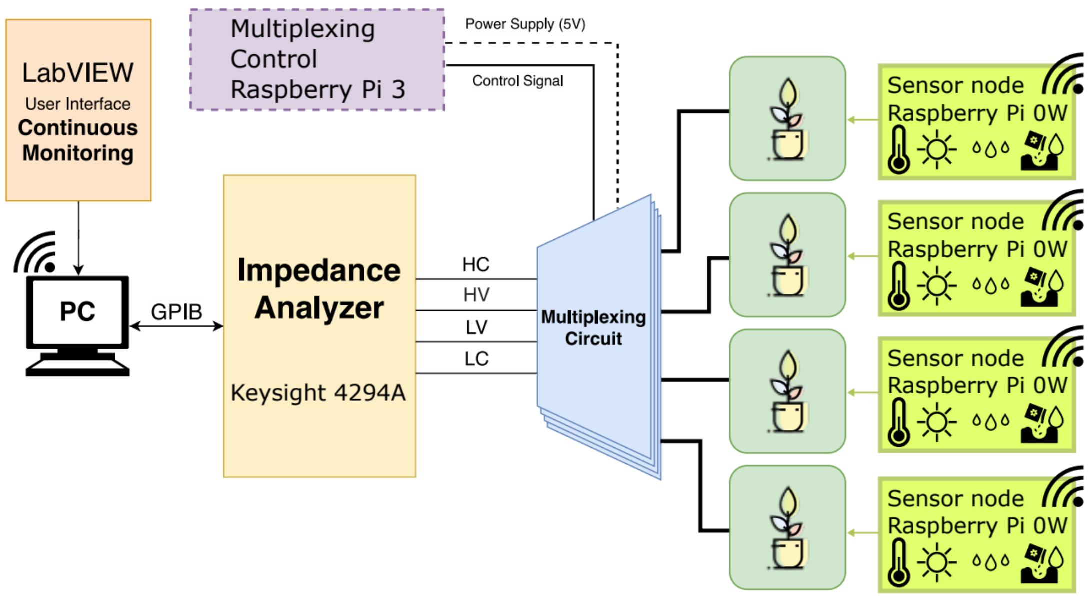

LabVIEW
User Interface
Continuous Monitoring
Multiplexing Control Raspberry Pi 3
Power Supply (5V)
Control Signal
PC
GPIB
Impedance Analyzer
Keysight 4294A
HC
HV
LV
LC
Multiplexing Circuit
Sensor node Raspberry Pi 0W
Sensor node Raspberry Pi 0W
Sensor node Raspberry Pi 0W
Sensor node Raspberry Pi 0W

Figure 1.11: Experimental setup for monitoring 4 plants. Image from [7].

A first analysis has been done in order to find the optimal frequency, figure 1.12 shows impedance modulus and phase variations and environmental data changes in 2 weeks, the watering event happens after ten days from the beginning, and it is clearly visible in the moisture chart: the soil moisture is decreasing over time, which means that it is getting dry. Then a step representing a watering event appears.

As discussed in section 1.2.3, also in this case, when the soil moisture decreases, the impedance module increases slowly, neglecting the “bouncing” effect. However, in this case, the interest is right on this daily "bouncing" effect. It is possible to notice from the impedance charts that lower frequencies are more appropriate for measuring the module, while higher ones are more suitable for monitoring the phase. Therefore a frequency of 10 kHz is chosen as a reasonable trade-off since the variations are visible for both the values, and the noise effect is reduced for the angle.

The plant and environmental data at 10 kHz over a week are then plotted in figure 1.13. Labels are placed in correspondence of minimum and maximum values in all the curves, and impedance modulus and phase are zoomed to highlight their trend. In this way, it is visible that all the trends have a bouncing over a day time.

All the sensors are positioned near a window inside a laboratory; the light intensity reached the maximum peak around midday when the sun hits directly through the window. The weather in this week of measurements was sunny, and temperature data confirm weather information since the average and minimum temperature are relatively high. In addition, the maximum temperature values correspond reasonably to light peaks. On the other side, relative humidity behaves opposite to the temperature, with the maximums in correspondence of temperature minimum and vice versa.

It is possible to notice that the impedance modulus and phase follow the trend on the environmental conditions, but there are two different situations based on the water condition. The first portion is the one before November 8th, in this period, the plant hydration is regular, and the minimum peaks of both modulus and angle correspond to about one hour later the maximum peaks of the temperature, and then they start to increase. In the second phase, when soil moisture is less than -150 kPa, the shapes of modulus and angle change. Narrow negative peaks separate the high flat portion for the modulus and the low flat portion with high peaks for the phase. The modulus remains high during the night, and then a steep fall is present in the early hours of the day when the light starts to appear. The high-temperature impacts the plant condition, and the modulus rises again: this time, the process begins far before the temperature peak due to the water stress condition. On the contrary, the phase shows precisely the opposite behavior, with minimum values corresponding with modulus maximum.

Thanks to this work, the relation between the plant impedance and the daily cycle is proved. Therefore, the changes in the impedance, both modulus and phase, are induced by environmental conditions, in particular, light, temperature, and soil moisture.

1.3 Plants' dataset

The MiNES group, from Politecnico di Torino, provides the data used in this thesis, particularly those working on the sensor and smart systems for Agrifood. The experimental setup used by them is constantly being improved. The final version (November 7th, 2021) can monitor the plant stem impedance and the surrounding parameters. The impedance module and phase are measured utilizing an impedance analyzer(Keysight 4294A). The four-probe technique (section 1.2.1) is applied using two Kelvin's clips, which are connected to the plant's stem through tiny stainless steel needles. The needles, with a diameter of 0.4 mm, are placed at 5 cm of distance, and the bottom one is placed 3 cm above the ground level. The system can monitor different plants' impedance because of a multiplexing circuit, which a Raspberry Pi controls. In particular, the impedance analyzer is measuring four plants, and it is connected to a PC, on which a LabVIEW program manages the measurement procedure and stores impedance spectra. For each plant, a sensor node is used to monitor the environmental values. The integrated circuit HDC2080[10] is used for the temperature and air humidity, while the ambient light is measured with the sensor OPT3001[13]. Both the integrated circuits are mounted on a small PCB connected to a custom PCB, which in turn is placed on top of a Raspberry Pi ZERO W. Thanks to this last wireless communication is possible to configure the nodes and acquire data. The soil moisture is measured with Irrometer WATERMARK[12], in particular through a resistance value, the soil water potential is measured in kPa. Since a DC current may damage the sensor, an AC circuit was developed with a timer in the feedback loop.

In this way, **four** different plants are monitored by this autonomous system where two of them are watered regularly, and the others are left to water stress. The information is sampled every one hour and gathered in four different *.csv* format files, one for each plant. To summarizing, in each file, the following measurements are present:

- air humidity;
- temperature;
- light intensity;
- soil moisture;
- time when the data is sampled;
- impedance modulus
- impedance phase

The impedances are measured at a frequency of 10.145 kHz. In addition to these data, also a camera is used to monitor the plants. A picture is taken every 15 minutes; in this way, it will be possible to visually check the plants' status.

All the studies in this chapter have demonstrated that the stem impedance **is not** a "random" electrical parameter. However, somehow, it is linked to the plant's health; this makes it possible to check the plant status directly by measuring an intrinsic parameter of the plant, which is the electrical stem impedance and the surrounding environmental parameters. It is opportune to find an automatic way to detect the plant status once the environment and plant measurements are available for this task. Machine learning algorithms can be an excellent means for the work of this thesis.

In chapter 2, the world of machine learning is scouted in order to find out the most suitable **machine learning algorithm** for the prediction of the plant status.

In chapter 3, the **neural networks** is explained in detail, then the model is implemented in Python, and finally the results are discussed. After that, simple **SVM** models are implemented in order to confirm the results of the neural networks. Finally, a new approach is explored. In particular, the same feature is considered more times by involving the past samples during the training.

Chapter 4 is regarding the conclusion about this thesis work and possible future works.

In the appendix 5, the instructions to run the implemented Python scripts are shown.

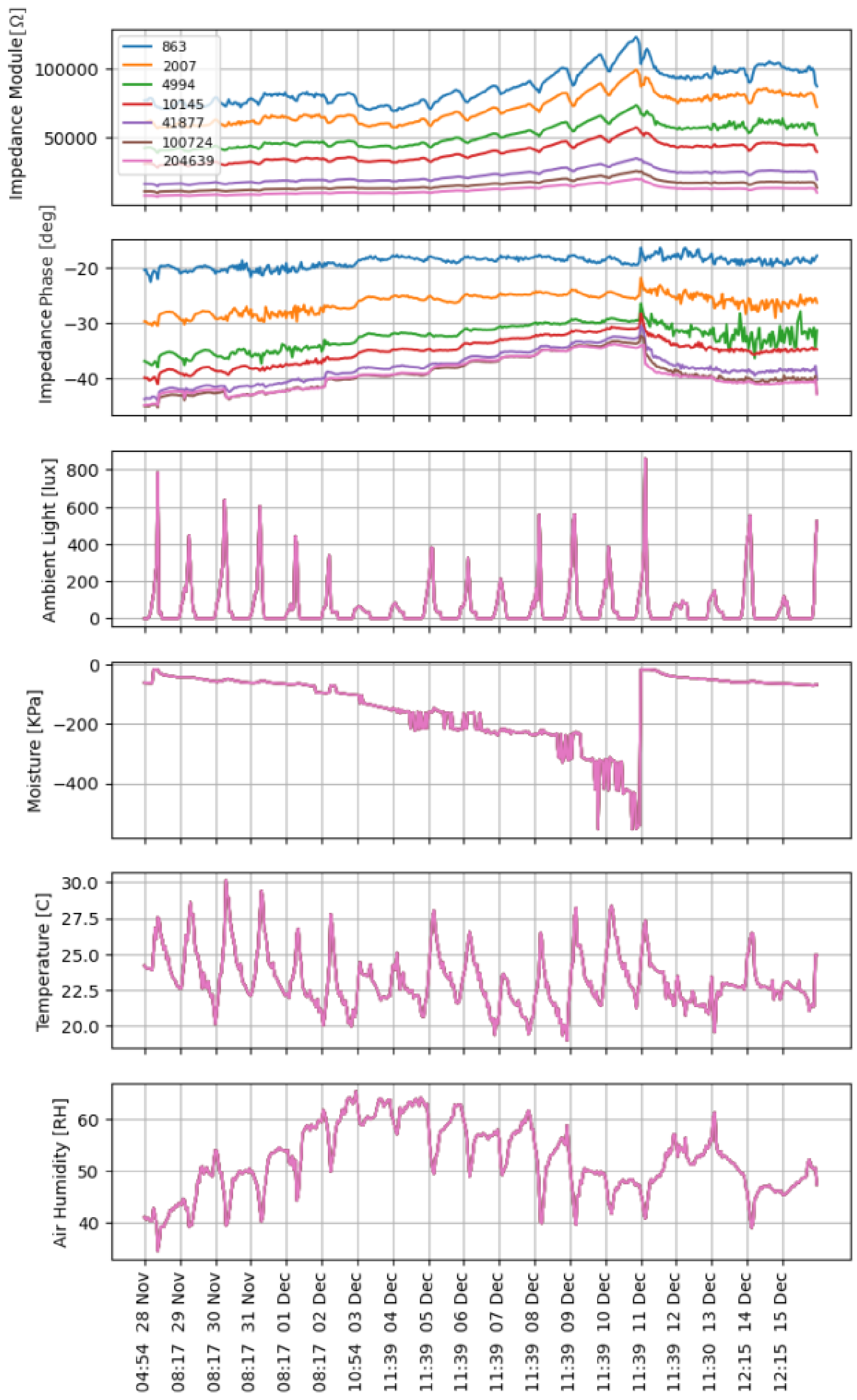


Figure 1.12: Impedance at different frequencies and environmental data over two weeks. Image from [7].

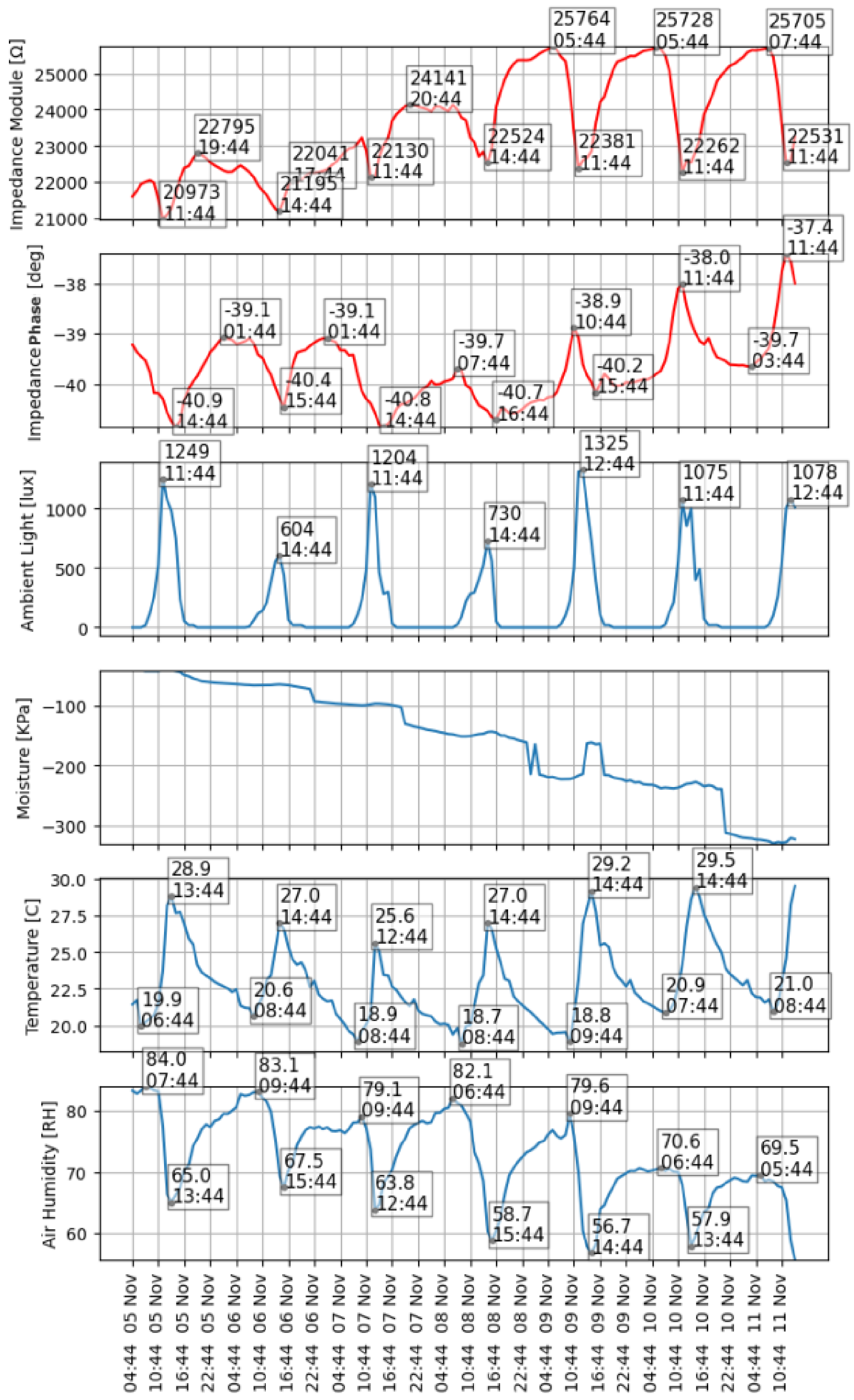


Figure 1.13: Impedance modulus and phase of a tobacco plant over a week with environmental data. Image from [7].

Chapter 2

Machine learning

Machine learning is a wing of artificial intelligence (AI) and computer science. AI refers to all those systems that imitate human intelligence using data and algorithms. While machine learning builds models that make predictions or decisions based on a training set. Nowadays, ML and AI are everywhere in the daily life of humans: personal assistants, for example, Siri or Alexa, self-driving cars, recommendation systems, web search engines, IBM's Watson, and much more. The term machine learning was coined for the first time in history by Arthur Samuel in 1959 with his research about the game of checkers. The research was regarding a checkers playing program that learned what board positions were good or bad by observing if that position would lead to loss or to win. The impressive thing was that in the mid-1970s, his program was able to challenge expert checkers players even if he was not a very good checkers player. He described machine learning as:

"The field of study that gives computers the ability to learn without being explicitly programmed"

Tom Mitchell brings a more modern 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."

So, back to the aim of this thesis, the experience E is represented by the data measured from the plants and the environmental sensors, task T is the prediction of the status of the plant, and the performance P is the accuracy of the machine learning algorithm.

2.1 Classification of machine learning algorithms

There are several types of algorithms around, and the goal is to find the most suitable one to predict the plant status. In general they can be grouped by their learning style[14, 15, 16, 17]:

- supervised learning;
- unsupervised learning;
- semi-supervised learning;
- reinforcement learning.

Supervised learning The machine is trained on labeled data set in order to classify data or predict outcomes accurately. The training process continues until the model reaches the desired level of accuracy on the training data. An inferred function is generated as a result of the training that could be used on new data sets. This type of algorithm resolves mainly regression or classification problems, such as e-mail spam detection, speech recognition, or object recognition from an image.

Unsupervised learning In this case, the input data are not labeled. Therefore this kind of algorithm is able to find out hidden patterns or data structures without expensive human labor. Unsupervised learning algorithms are suitable for anomaly detection, cluster analysis, dimensionality reduction, customer segmentation, and recommender systems.

Semi-supervised learning For this type of algorithms, the input data is a mixture of labeled and unlabelled examples. This type of algorithm is necessary because labeling is relatively expensive in several real situations due to the need for human resources. Therefore, some data are labeled, and others are not.

Reinforcement learning This is an exciting and particular learning model; it is not easy to explain and understand. The main idea is that a reinforcement learning algorithm does not need correct input-output datasets, but it learns continuously from the interaction with the environment. So this model learns using a trial and error approach. Examples of applications are computer-played board games, robotics, and self-driving cars.

2.2 Supervised learning algorithms

The available dataset is composed of samples, each with seven different features, as described in section 1.3. The machine learning algorithm should be able to detect the status of health by analyzing the input information. Suppose that each sample is labeled as healthy plant or not-healthy plant, the prediction of the status of the plant becomes a classification problem, and the most suitable algorithm belongs to the **supervised learning class**. The most common supervised learning algorithms are:

- k-Nearest neighbor algorithm;
- Naive Bayes;
- Decision Tree and Random forest;
- Linear and logistic regression;
- Support vector machine (SVM);
- Neural Networks

A schematic view of a typical supervised learning approach applied to plant status detection problem is presented in figure 2.1. Once the model has been trained, it is ready to predict the output of newly sampled data.

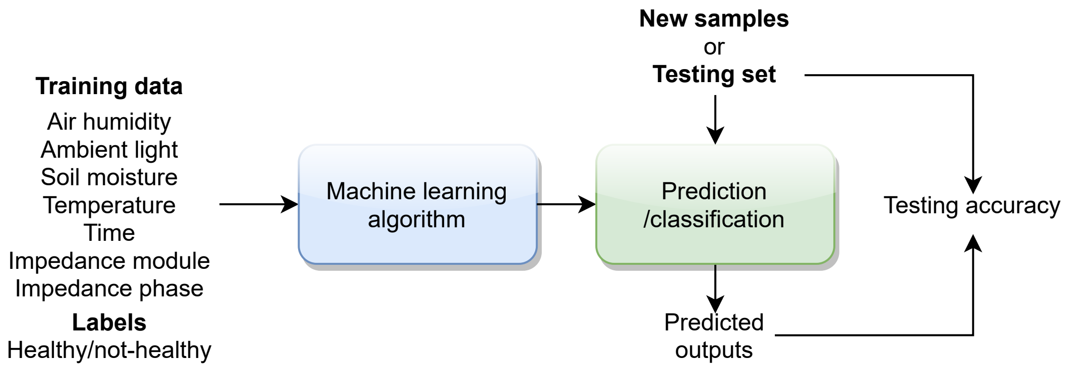


Figure 2.1: Schematic block of supervised learning approach applied to plant's status predictions

So, now the main problem is to choose an algorithm to be implemented that gives the best performance. Considering that a future application of this work will be on-field as an IoT node (fig 2.2), power consumption is not a negligible parameter. The training phase, which is the longest part, could be done on a PC, and then the trained model is used to predict results employing a MicroController. Therefore, the performance should consider both the accuracy of the algorithm, which means the correct prediction rate and its computational cost.

For this purpose, the most common algorithm will be briefly discussed qualitatively in the following sections, considering the advantages and disadvantages and showing some smart agriculture applications or real-world implementations that could be somehow similar to the health detection problem.

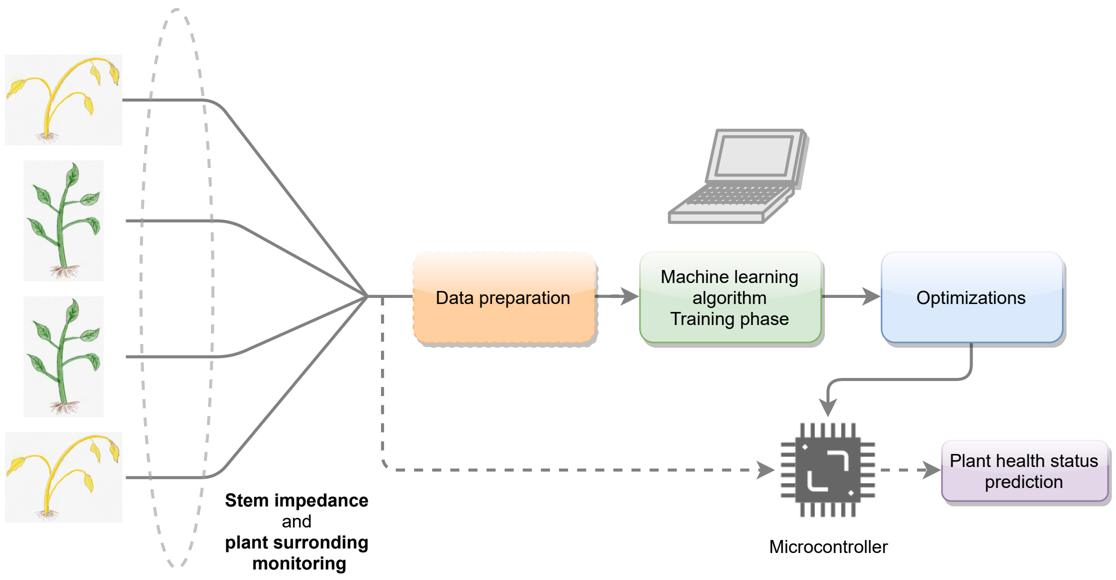

Data preparation
Machine learning algorithm Training phase
Optimizations
Plant health status prediction
Microcontroller
Stem impedance and plant surronding monitoring

Figure 2.2: Block diagram of plant's status predictions

2.2.1 Machine learning applications

Nowadays, machine learning algorithms are already widely used in the farming world. A fascinating overview is presented in article [18]. The authors have grouped some applications around; most use algorithms such as artificial neural networks and support vector machines, but also decision trees and Bayesian models are helpful. The application can be subdivided into four big fields:

- crop management, most of the applications belong to this category, and it includes species recognition, weed detection, crop quality, yield prediction, and disease detection;
- livestock management, which is divided into animal welfare and livestock production;
- water management;
- soil management.

The applications described in this article and others will be cited in the subsequent sections of the algorithms' description.

2.2.2 k-Nearest neighbor algorithm

k-Nearest neighbor is one of the most basic and simple machine learning algorithms based on the supervised learning method, usually used for classification problems.

It is based on the similarity between the new sample and available data. So, this algorithm stores all the sampled data. When new data occurs, it will be classified based on the K nearest data labels by computing the Euclidean distance and counting the number of the data points in each category. Thus, there is no proper training phase for this algorithm, and the training phase is substituted by storing data and postponed to the moment of the prediction. Data must be normalized in order to avoid bias towards any of the inputs features.

In spite of its simplicity, this algorithm has been successful in several classification problems in different fields[19], such as agriculture for simulating daily precipitations or for evaluating the forest inventory, where satellite images are used. A software that exploits KNN for soil water retention has been developed [20]. Thanks to the KNN, additional data can be included, and this improves estimation for the site-specific range of soil properties; finance, such as forecasting the stock market, bank customer profiling, money laundering analyses or credit rating; medical field, for instance, estimation of the level of glucose in the blood of a diabetic patient, based on the infrared absorption spectrum of that person's blood or identification of prostate cancer risk factors, having clinical and demographic information.

Wisconsin-Madison Breast Cancer Problem An exciting application is described in [21] when KNN and fuzzy KNN are used for the Breast Cancer Diagnosis Problem. The presence of a breast mass may indicate (but not always) cancer. *Fine needle aspiration* technique is used for sampling the data such as clump thickness, uniformity of cell, single epithelial cell size, and more. Each sample also contains the patient's ID and is labeled as benign or malign. Fuzzy KNN is a version that also considers the distances of the k nearest samples; as a result, it generates smoother borders between classes. Classification result of test set reached an accuracy above 99%, and results are even better with fuzzy KNN.

To better understand how this algorithm works, a simplified version is shown graphically in figure 2.3. Only two features, impedance module and soil moisture, are displayed in the graph. As mentioned in section 1.2, generally, the impedance module increases when the soil is getting dry, this trend is shown in this figure. When a new data point must be predicted, distances with all the samples data are computed, and only the nearest K (K=3 in this example) are considered. Finally, the prediction is based on the healthy or dry plants among these neighbors(figure 2.4).

KNN is one of the first algorithms implemented in history, all the examples described are 'old' implementations before 2010. It is still used today for the recommendation systems, but it has many disadvantages when the dataset is vast, since there is no proper training phase, and all the data should be stored. Consider

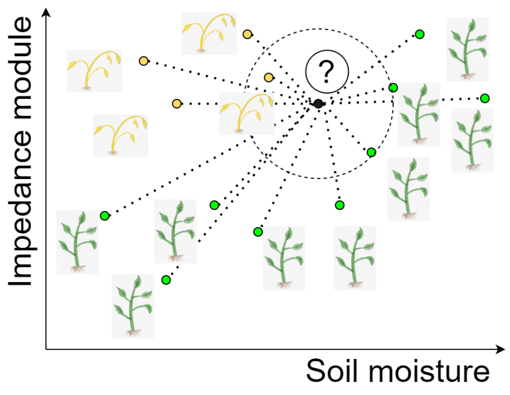


Figure 2.3: Simplified impedance and soil moisture distribution to explain KNN with a new data to be predicted

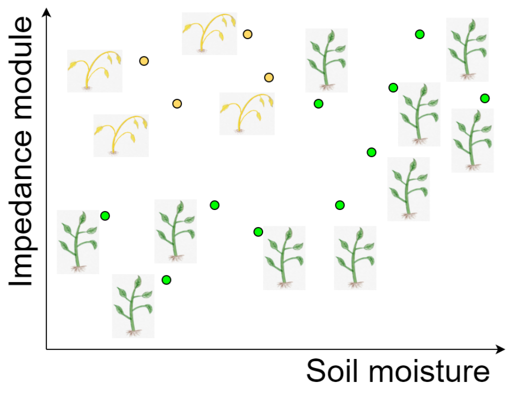


Figure 2.4: The prediction of the status of the new data is based on k=3 nearest neighbor and the result is a healthy plant

N the number of samples and M the number of features. Every time a prediction is needed, N Euclidean distances, each with M dimensions, must be computed, leading to O(NM) computational complexity and poor run-time performance. Additional works are needed to find the best value for K; several simulations should be run, trying different values for K to find the best out of them.

As mentioned before, the final aim is to use the machine learning algorithm on the field as an IoT node, probably on a MicroController. So, it must be a low-power system, and besides, a MicroController has limited memory. Even if this algorithm has very high accuracy in a particular application, it is not the algorithm that best suits the plant status detection.

2.2.3 Naive Bayes

Naive Bayes is a straightforward machine learning algorithm used for classification problems, for instance, spam detection or text classification. Authors in [22] propose an exciting application. They developed a recommendation system based on Naive Bayes, which proposes the best crop to be cultivated in a particular season and region. They considered features like environmental conditions such as soil moisture, temperature, rainfall and found which among cotton, chilies, maize, and rice is most suitable for the cultivation

This algorithm is based on Bayes' theorem, and "naive" indicates the hypothesis of independence among the features. Bayes' theorem says that:

$$\mathrm{P(A|B)} = \frac{\mathrm{P(B|A)} \cdot \mathrm{P(A)}}{\mathrm{P(B)}} \tag{2.1}$$

P(A) and P(B) are the probabilities of events A and B, respectively. P(A|B) and P(B|A) are called conditional probability, the probability of the first event knowing that the second one is true. Applying the Bayes' theorem for a classification problem, the following version can be obtained:

$$\mathrm{P(class|data)} = \frac{\mathrm{P(data|class)} \cdot \mathrm{P(class)}}{\mathrm{P(data)}} \tag{2.2}$$

Therefore, the prediction of new data is based on the probability obtained with the training set. Naive Bayes works well with categorical features. Nevertheless, it is possible to use it for numerical data considering a Gaussian distribution[23], leading to the so-called Gaussian Naive Bayes. In order to do that, the average value (2.3) and standard deviation (2.4)should be computed. the probability is then calculated using the Gaussian distribution formula.

$$\text{Mean:} \qquad \mu = \frac{1}{N}\sum_{k=1}^{N} x_k \tag{2.3}$$

$$\text{Standard deviation:} \qquad \sigma = \sqrt{\frac{1}{N-1}\sum_{k=1}^{N}(x_k-\mu)^2} \tag{2.4}$$

It is helpful to apply this algorithm to a small subset of the plants' data to see how it works. Ten samples of plant 3 are taken; for simplicity, only 4 features are considered, as shown in table 2.1.

Since, the plant 3 is watered regularly, the soil moisture value remains almost constant and relatively high. For both Status cases, the mean and standard variation of all the features should be computed, shown in table 2.2. To predict the status output of a new set of data, the probability density function of Normal distribution(2.5) can be used.

$$\text{PHD:} \qquad f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{2.5}$$

What have to be done is the computation of P(Status=1|X) and P(Status=0|X), where X represents the features' value of the new sampled data, namely the last row of table 2.1.

Given the several probability density functions computed in table 2.3, the probability density function of the new sample is proportional to their product. In this case, it can be observed that P(Status=1|X) is much greater than P(Status=0|X). Therefore, the final prediction of the algorithm is Status = 1, which corresponds to the real health state of the plant.

It is possible to notice that the prediction output is mainly due to the impedance module value, and the algorithm seems to be unbalanced; this is one of the reasons

Temperature [C]	Air Humidity [RH]	Soil moisture [kPa]	Impedance module [Ω]	Status
24.47	62.25	-10.56	34479.96	0
25.45	63.19	-10.56	34839.94	0
25.97	60.95	-10.56	34767.41	0
26.46	62.35	-10.56	34123.81	0
28.96	66.44	-10.55	34616.83	0
27.15	61.45	-10.55	27650.08	1
27.11	61.18	-10.56	27967.37	1
28.11	61.36	-11.01	28050.06	1
30.03	57.97	-10.57	28543.6	1
26.82	58.33	-10.56	27922.81	1
Data to be predicted				
25.33	62.93	-10.57	29527.26	1

Table 2.1: Subset of the plant 3 dataset, considering only 4 features

Status	Value	Temperature [C]	Air Humidity [RH]	Soil moisture [kPa]	Impedance module [Ω]
0	μ	26.26	63.04	-10.56	34565.59
0	σ	1.679	2.065	0.004	283.34
1	μ	27.84	60.06	-10,65	28026.78
1	σ	1.315	1.750	0.201	325.53

Table 2.2: Mean and standard deviation of the features split based on the Status

why **normalization** of the data is always recommended. The normalization is essential not only for Naive Bayes but, in general, for all the machine learning algorithms. Despite that, Naive Bayes seems to be a suitable algorithm, but, as said before, Naive indicates independence assumption; but, authors in [8] have demonstrated precisely the contrary with the Pearson correlation.

2.2.4 Decision Tree and Random forest

Decision Tree is a machine learning algorithm that works very well for classification problems. The prediction is based on decisions rules coming from the training set. The second part of the name derives from its graphic representation. The decision rules can be seen as internal nodes. Starting from the root node, it splits into

Probability density function Status = 0	Probability density function Status = 1
f(T=25.33|Status=0) = 0.289	f(T=25.33|Status=1) = 0.028
f(AH=62.93|Status=0) = 0.480	f(AH=62.93|Status=1) = 0.950
f(Sm=-10.57|Status=0) = 0.00364	f(Sm=-10.57|Status=1) = 0.654
f(|Z|=29527.26|Status=0) $\simeq 3 \times 10^{-72}$	f(|Z|=29527.26|Status=1) $\simeq 1$
P(Status = 0) = 0.5	P(Status = 1) = 0.5

Table 2.3: Probability density function of the features

different branches leading to the decision nodes. When a node does not represent a condition anymore, it is called a leaf node, corresponding to the final prediction. To summarize, decision tree is a set of if-then-else statements, which can be represented with a flowchart. The model is usually trained with the recursive binary splitting method, where all the features are considered to generate all the possible splits. The one with the lowest cost function is chosen. Another problem is regarding when to stop the tree. A huge tree could lead to overfitting, and two criteria could be used to prevent overfitting. The first one is to set a minimum number of training data for each leaf, which means that the splitting stops when the leaf takes less than a certain number of samples. The second one concerns the maximum depth of the tree. Furthermore, the pruning technique can reduce the complexity of the tree, cutting the branches that use less critical features.

Different decision trees are ID3 (Iterative Dichotomiser 3), C4.5, C5.0, and CART (Classification and regression trees). These versions differ in their learning style. For instance, ID3, the oldest one, takes only categorical data as features; trees are left to grow to their maximum possible depth; then, pruning is applied to prevent overfitting. The others are subsequent versions improving accuracy, memory, and training cost, arriving at the latest version CART, which also solves regression problems.

Interesting applications can be found in the medical field. In the article [24], the authors used decision trees to classify patients into four different levels of severity, based on 12 symptoms of Covid-19 and the age of the patient. They used two types of trees, J48 and Hoeffding tree. In both cases, 83% of accuracy is achieved. An example in a completely different context is the identification of stars and cosmic-ray using the pictures taken by the Hubble Space Telescope[25]. Using CART, they have reached an accuracy above 92%.

Back to the problem of plant health prediction, each internal node of the decision tree could be an environmental measurement or the impedance value. Soil moisture and impedance module will be the most critical features surely. A very simple view is shown in figure 2.5 The prediction is based on the impedance module and soil moisture, and the status depends on two threshold values A and B. In the real application one feature could be present mode times, and the tree is certainly more complex.

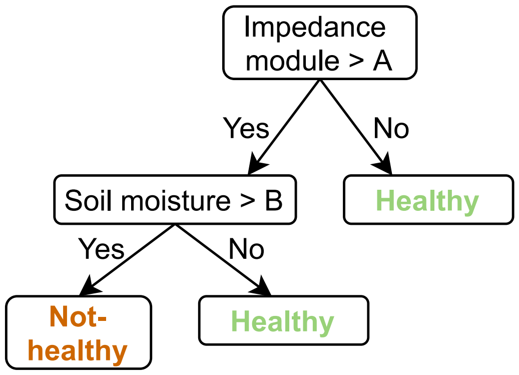


Figure 2.5: Simplified example of decision tree

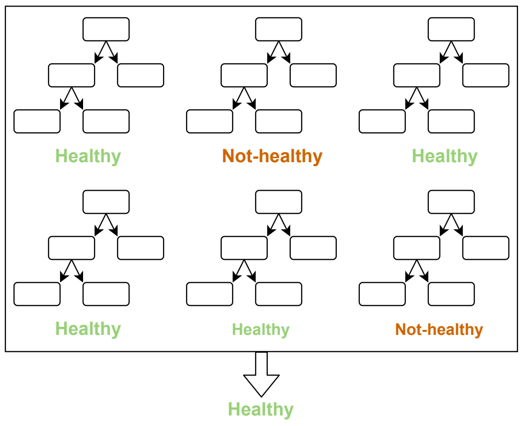


Figure 2.6: Hypothetical example of random forest

As said before, decision trees suffer from overfitting problems. Therefore, the prediction on unseen data may be terrible. Furthermore, another problem is that they are very unstable, which means that the noise of the data would lead to a significant variation of the tree structure. This problem is partially solved using ensemble learning. Multiple uncorrelated trees are trained, and the final prediction is based on the majority. This last technique is called Random Forest (figure 2.6), but it will increase the storage memory and the training cost, which is already high for a single tree compared to other algorithms.

2.2.5 Linear regression

Linear and logistic regression are based on the same concept. The former, as can be deduced from the name, is to solve regression problems. In comparison, the latter is for classification problems. Linear regression takes as input the features and gives as output a continue value. It is nothing else than a fitting model. Considering n features, the output prediction y is obtained with equation 2.6, where x_j are the inputs, and θ_j are the coefficients obtained during the training phase. The sum can also be expressed in a compact form, i.e., $h_\theta(\vec{x})$ hypothesis function, where $\vec{x}$ is a vector containing the features.

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = h_\theta(\vec{x}) \tag{2.6}$$

It is possible to notice that the name linear comes from the linear relation between the output and the inputs since x_k are at first grade. Consider the case of only one feature x_1, the coefficients θ_0 and θ_1 determine a line that best approximates the data of the training set.

A set of optimal θ_k that minimizes the so-called cost function is found during the training phase. The cost function is used to evaluate the accuracy of the hypothesis function, computing the difference between $\hat{y}$ predicted and the actual value of y, considering all the data in the training set. Suppose a dataset with m samples, the cost function, also called "Mean Squared Error", is defined as:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(\vec{x_i}) - y_i)^2 \tag{2.7}$$

In order to minimize $J(\theta)$, an algorithm called gradient descent is used; it consists of repeating several steps many times until the cost function converges to a certain value. In each cycle, new values of θ are updated, as showing in equation 2.8, where θ_j corresponds to the coefficient to be multiplied to the feature x_j. The symbol ":=" indicates that the value on the right is assigned to what is on the left, so it is not the equal symbol.

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) = \theta_j - \alpha\frac{\partial}{\partial\theta_j}\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(\vec{x_i}) - y_i)^2 \tag{2.8}$$

Doing the partial derivative in 2.8, equation 2.9 is obtained. A new notation is used here, $x_j^{(i)}$ means the j-th features of the i-th sample. Therefore, $x^{(i)}$ indicates the i-th sample. This procedure is repeated for all the θ_j for j from zero up to n. It is possible to notice that there are $n+1$ different j. But the number of features is only n, $x_0^{(i)}$ is equal to 1 for every samples per definition.

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for} \quad j := 0{,}1...n-1, n \tag{2.9}$$

With equation 2.9 it is possible to update the value of θ, and after each cycle the value of the cost function decreases. However, two problems arise, when to stop the iterations and what value α should have. The interruption could be decided by a given value ϵ. Therefore the values of θ should be re-computed until the cost function becomes smaller than ϵ. Another more practical way is to plot the cost function over the number of iterations and see when the cost function stops to decrease. Alpha is called learning rate, and it influences the step size of gradient descent in each iteration. If α is set too small, the algorithm may have slow convergence, which means a long training time. Otherwise, if it is too large, the algorithm may not converge since the step is too large in every iteration, and it may never find the minimum.

From equation 2.9, it can be noticed that in every iteration all the data (from data one to data m) are used to update θ. This procedure is called Batch gradient descent. However, when the training set is very large, using this version of gradient descent could be very slow and expensive. Another algorithm, called Stochastic gradient descent, is more suitable for large datasets. This type of gradient descent requires a random shuffle of the training set, and then only one training sample is considered a time. Therefore, in each iteration, the θ are updated using only one sample, and this is repeated for all the data, which means m times. If this is not enough, the m iterations are repeated more times, as shown below:

```
Repeat more times if needed {
    for i=1,...,m {
        $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$    for  $j := 0,1...n-1, n$
        }
    }
```

There is no for loop for j because all the *theta* must be updated simultaneously considering the same hypothesis function. The final result is a cost function near the global minimum since the data are never considered together. However, it is much faster than Batch gradient descent. A third version, called Mini-Batch gradient descent, exists. Basically, b samples are considered in each iteration, where b is between 1 and m. This algorithm can be much faster than the others if a good vectorized implementation is used.

A helpful technique that can be applied during the data pre-processing step is data normalization. The problem is that if the features are on very different ranges, each feature will converge with a different step size during the iterations in the gradient descent. Therefore, if all the features have the same range of values, the training will be much faster. Normalization of the data to a range from 0 to 1 can be done with the formula:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.10}$$

When more complex hypothesis functions are needed, the behavior of the fitting function can be changed using polynomial regression. It consists of adding to the cost function other terms with a grade greater than one or other functions, such as sine, cosine, square root functions.

2.2.6 Logistic regression

Logistic regression solves classification problems, so the output prediction is a binary value that expresses the belonging class. Therefore, the hypothesis function used in linear regression is not good. The hypothesis function for logistic regression should give as output a value between 0 and 1. For this purpose, a sigmoid function is used. A scalar product between two vectors can express the multiplication between features and coefficients. If both θ and x are row vectors, their product can be expressed as $\theta^T x$. There are several functions in this class, an example is the logistic function shown in equation 2.11.

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \tag{2.11}$$

The graphical representation of the logistic function is shown in figure 2.7. It is possible to notice that some decision boundaries are needed to get binary output since the hypothesis function gives continuous values from 0 to 1.

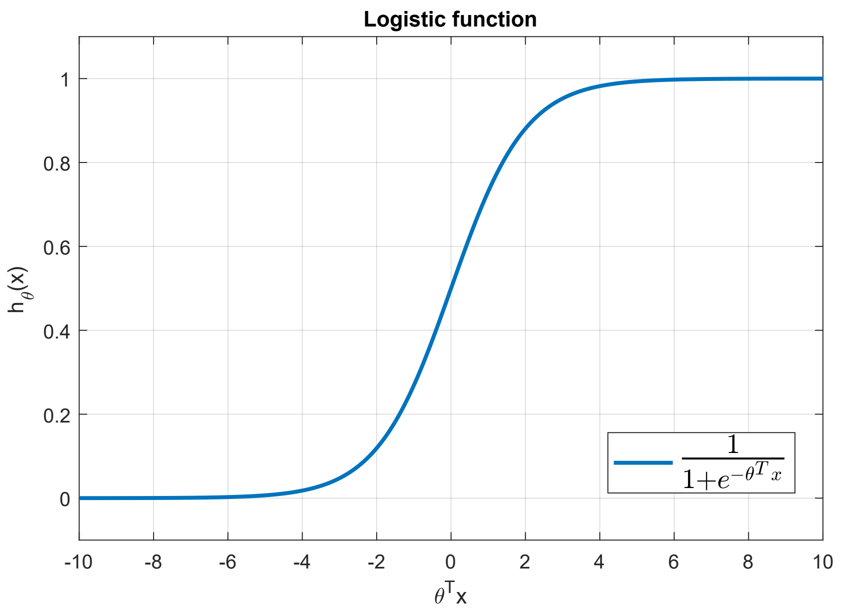


Figure 2.7: Graphical representation of the logistic function. Plotted with MATLAB.

The binary output can be obtained with the following relations:

$$h_\theta(x) \geq 0.5 \quad \text{if} \quad \theta^T \geq 0 \Rightarrow y = 1 \tag{2.12}$$

$$h_\theta(x) < 0.5 \quad \text{if} \quad \theta^T < 0 \Rightarrow y = 0 \tag{2.13}$$

Another thing to be changed is the cost function, it is defined in 2.14. In particular, two different cost function are considered based on the value of the output y. When $y^i = 1$, $\log(h_\theta(x^{(i)}))$ is taken as cost function otherwise $\log(1 - h_\theta(x^{(i)}))$ is considered.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))] \tag{2.14}$$

Logistic and linear regressions are straightforward algorithms. Their applications are minimal due to the linearity assumption between output and input. However, it was important to describe them because many concepts are familiar to SVM and Neural Network, such as cost function, gradient descent, sigmoid functions; others are shared by all the algorithms in general, for instance, normalization of the data, overfitting, and random shuffle of the data.

2.2.7 Support Vector Machine

SVM or large margin classifier has excellent performance for classification problems since it can learn complex non-linear functions. The prediction of the output is based on hyperplanes that separate the two classes of data samples. The dimension of the hyperplanes depends on the number of features. In figure 2.8 2-dimensional hyperplanes are shown in the plant status detection problem, where only two features are considered. Therefore, the hyperplanes are simply lines. It would be hard to represent hyperplanes graphically with a dimension greater than three.

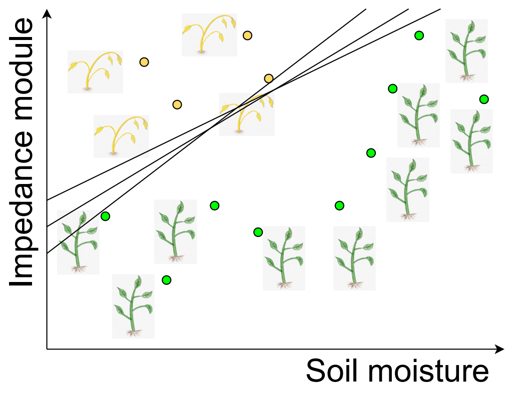


Figure 2.8: Hyperplanes in plant health prediction problem. Only two features are considered.

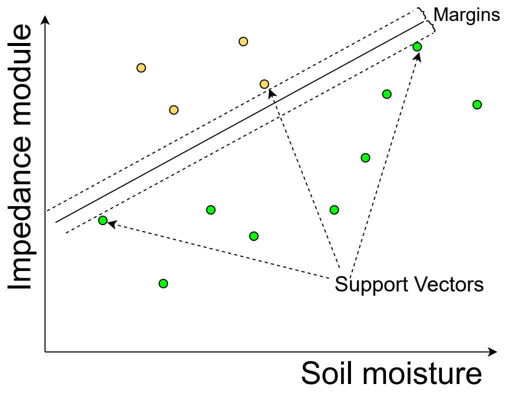


Figure 2.9: Hyperplane with maximized margin, determined by support vectors.

An infinite number of hyperplanes are suitable to divide the data into two classes. The one obtained during the train is the one with the maximum margin for data samples of both classes (figure 2.9). The margin is the distance between data points and the hyperplane. The orientation and position of this last depend on the closest data points, namely the support vectors.

$$J(\theta) = C\sum_{i=1}^{m}[y^{(i)}cost_1(\theta^T x^{(i)}) + (1-y^{(i)})cost_0(\theta^T x^{(i)})] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \tag{2.15}$$

The hypothesis function does not indicate the probability of belonging to a class but gives binary values as output. This is possible for the cost function, shown in equation 2.15, that assumes different forms as in the case of logistic regression. But instead of the logarithms, two peace wise linear functions are used. If particular:

$$\text{cost}_0(z) = \begin{cases} 0 & \text{if} \quad z \leq -1 \\ 1 + z & \text{otherwise} \end{cases} \qquad \text{cost}_1(z) = \begin{cases} 1 - z & \text{if} \quad z \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

The second term in equation 2.15 is due to a technique called regularization. This last prevent that the coefficients assume extreme value and significantly reduces the variance, i.e., the overfitting problem. Therefore, if a data point that highly reduces the margins is present, setting the parameter C to a low value, this data is considered an outlier. It is then possible to increase the margin and reduce the error for unseen data. A possible scenario of applying regularization to status prediction is shown in figure 2.10.

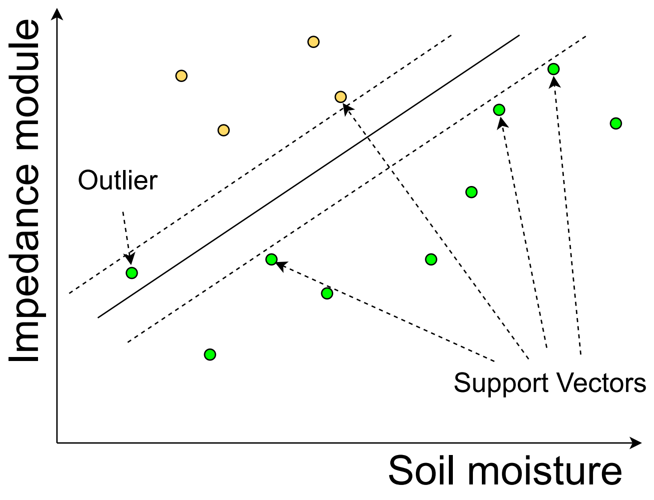


Figure 2.10: Hypothetical application of SVM with regularization of plant status detection

The algorithm seen so far can solve only linear problems. In order to manage complex non-linear functions, a concept called kernel is introduced. The argument of the cost function is a linear combination between coefficient and data examples. The idea of SVM kernels is to introduce new features to be used during the train, called similarities. Basically, $\theta^T f(x)$ is used as features, where $f(x)$ is a vector containing the similarities between the data to be predicted and the data of the training set. The i-th element f_i is defined as $f_i = \text{similarity}(x, x^{(i)})$. The vector $f(x)$ can be computed in several manners, such as polynomial, Gaussian Radial

Basis Function, sigmoid and more. For example, Gaussian RBF kernel functions are defined as follows:

$$f_i = e^{(-\gamma \|x - x^{(i)}\|^2)} \tag{2.17}$$

According to the article [18], SVM is widely used in the agriculture field for both regression and classification problems. Examples of applications are identification of immature green citrus (80.4% of accuracy); detection of Bakanae disease in rice seedlings (87.9%); identification of Korla fragrant pears into deciduous-calyx or persistent-calyx categories (more than 93%); other applications are for livestock management. For most of the applications, colored digital images or hyperspectral images are used.

2.2.8 Neural Networks

Neural networks have been around for a long time; in 1958, Frank Rosenblatt created the first perceptron. Nevertheless, they have become popular recently. Artificial Neural Network is a machine learning algorithm that is based on brain information processing. In a human brain, the information is processed as electrical signals through billions of neurons. A biological neuron's architecture is very complicated, but it can be divided into three parts: a cell body with two extensions, a dendrite, and an axon. The dendrites receive the electrical impulses, elaborated in the body, and transmitted with the axons. Therefore, two adjacent neurons can communicate between them through axons and dendrites. In the same way, artificial neurons process the inputs and give the result as output to other neurons, the one shown in figure 2.11 is a sigmoid neuron since an activation function is present. A bias unit with a value of 1 is always present. The inputs are multiplied by the correspondent weights and sum together. The result is then given to an activation function generating the output of the neuron.

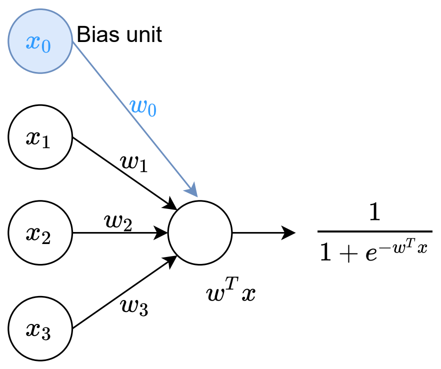


Figure 2.11: Artificial neuron with logistic activation function

More neurons can be put together, forming a neural network. There are many different types of neural networks, each of them is suitable for particular problems:

- Feedforward neural networks also called MLPs(multi-layer perceptrons);
- CNNs (Convolutional Neural Networks), usually applied to solve problems such as computer vision and image recognition;
- RNNs (Recurrent neural networks). They exploit feedback loops to predict future outcomes using time-series data.

Today's well-known term is deep learning; they are nothing else than a neural network made of more than three total layers.

The description of neural networks will focus on MLPs since they are the most suitable for plant status detection. MLPs layers can be classified as:

- input layer: it takes the input variables, and the number of nodes, i.e., neurons, corresponds to the number of features used during the training phase;
- hidden layer: internal layers of the network. There may be one or more of these layers, each one with a different number of nodes;
- output layer: a layer of nodes that produce the outputs.

An example of multi-layer perceptron is shown in figure 2.12. This network takes three inputs, consists of two hidden layers, each with four neurons, and gives two outputs. Since more layers form this neural network, and each layer contains several neurons, a matrix notation is needed.

The terms $a_i^{(j)}$ indicate the output of node i in layer j. In this case, the bias unit is not shown, but it is always present. Also, the weights are expressed in a matrix way; the matrix of weights controlling function mapping from layer j to layer $j+1$ is $W^{(j)}$. The elements of this last are $w_{ik}^{(j)}$, which indicates the weight applied to element k in the layer j to form the node in i in layer $j+1$. Therefore, assuming $g(\cdot)$ a generic activation function, in this example:

$$\begin{aligned}
a_1^{(2)} &= g(w_{10}^{(1)}x_0 + w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3)\\
a_2^{(2)} &= g(w_{20}^{(1)}x_0 + w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3)\\
a_3^{(2)} &= g(w_{30}^{(1)}x_0 + w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3)\\
a_4^{(2)} &= g(w_{40}^{(1)}x_0 + w_{41}^{(1)}x_1 + w_{42}^{(1)}x_2 + w_{44}^{(1)}x_3)
\end{aligned}$$

The same thing can be done for layer 3 and 4, for example the outputs of the network are:

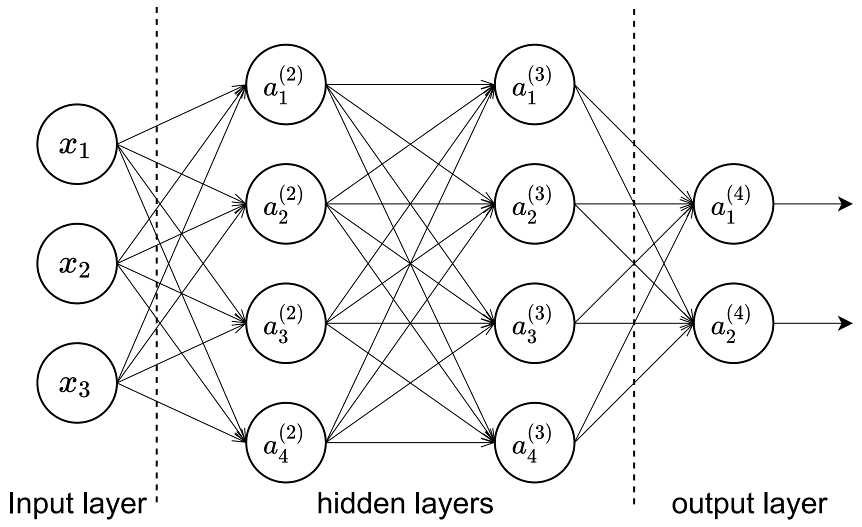


Figure 2.12: Example of multi-layer percepton

$$a_1^{(4)} = g(w_{10}^{(4)}a_0^{(3)} + w_{11}^{(4)}a_1^{(3)} + w_{12}^{(4)}a_2^{(3)} + w_{13}^{(4)}a_3^{(3)} + w_{14}^{(4)}a_4^{(3)})$$

$$a_2^{(4)} = g(w_{20}^{(4)}a_0^{(3)} + w_{21}^{(4)}a_1^{(3)} + w_{22}^{(4)}a_2^{(3)} + w_{23}^{(4)}a_3^{(3)} + w_{14}^{(4)}a_4^{(3)})$$

All these expressions can be written in a compact matrix form such as $a^{(j+1)} = g(W^{(j)}a^{(j)})$, considering the inputs x as $a^{(1)}$. It is possible to notice that, calling s_j the number of nodes in the j-th layer, the generic dimension of a matrix $W^{(j)}$ is $s_{j+1} \times s_j + 1$.

The training phase aims to find the weights for the neural network that minimizes the cost function. The cost function is more complicated than in logistic regression since multiple output nodes should be considered. In order to do that, an algorithm called backpropagation is exploited. Basically: the weights are initialized randomly to avoid symmetry during the training; forward propagation is performed, computing the outputs of the neural network using the training set; the output error can be computed using the data of the training set; now it is possible to compute the errors for the previous layers with the transpose matrix of the weight and derivative of the active function; finally, it is possible to update the weights, and these steps are repeated several times. A complete mathematical description is present in Tom Mitchell's book "Machine Learning" [26].

Following the article [18], ANN is widely used in agriculture. Many exciting applications are presented; many are regarding detecting diseases exploiting both ANNs and CNNs, all of which use hyperspectral or typical images. One application is about identifying and classifying soybean and red and white bean, using vein leaf images and CNN. Other exciting usages concern environment parameters prediction and estimation, such as temperature, soil temperature, soil moisture, and more.

After this exploration into the world of machine learning, general knowledge of the various algorithms, their advantages and disadvantages is acquired. The most suitable algorithms for plant status detection are ANNs and SVMs for their ability to manage non-linear relations of the data. As introduced in section 1.1, most of the applications regarding plant health status detection exploit visual data. Applications that use numerical data concern environmental data prediction and classification problems in other fields.

In the next chapter, the neural network implementation is described, and a simple SVM is used to confirm the results obtained with MLPs.

Chapter 3

Neural networks

This chapter describes the implementation of neural networks, particularly MLPs, to detect the plant status. The most important thing is to prepare a reliable training data set. After that, the number of layers and the number of nodes in each layer should be decided. Several networks will be implemented in order to see the influence of different features. Some results of the neural network will also be confirmed by SVM implementation.

3.1 Number of layers and nodes

The most critical choice for a neural network is to decide the number of hidden layers and the number of nodes for each layer. Generally, two problems will occur: using too few neurons in the hidden layer would lead to the so-called underfitting. Using too many neurons in the hidden layers may lead to overfitting and significantly increase the computational cost. When overfitting occurs, the network may result very well for the training set, but it is too specific and could perform poorly for general test and validation sets.
Some common approaches to choose the optimal number of nodes and layers are described in article [27]. There is no analytic formula to compute them; it is only possible to use an experimental approach or trial and error method since every real application is unique. However in [28] some rules of thumb are presented. It has been demonstrated that two hidden layers are enough for creating decision boundaries of any shape. Moreover, for the number of nodes, the following rules of thumb are shown:

- The number of nodes in the hidden layers should be between the input layer's size and the size of the output layer.
- The number of nodes in the hidden layers should be 2/3 the size of the input

layer, plus the size of the output layer.

- The number of nodes in the hidden layers should be less than twice the input layer dimension.

By applying these three rules to the plants' problem the following numbers are obtained:

- The number of neurons should be between 7 and 2
- The number of neurons should be 6
- The number of neurons should be less than 14

Starting from these computations and theory background, a neural network with two hidden layers, each with six neurons, is decided to implement.

3.2 Networks to be implemeted

As mentioned before, several networks will be implemented. In most cases, the networks will be formed by two hidden layers, each with six nodes. In particular, the work consists of:

- training of a neural network with **2** hidden layers of sizes **6** and **6**. Different situations will be considered by changing some parameters and removing various features;
- training of a neural network with **2** hidden layers of sizes **6** and **6**. The data used for the training come from **3** plants from 2021-03-24 to 2021-05-04, and the testing is done of the fourth plant;
- training of a neural network with **2** hidden layers of sizes **6** and **6**. The data used for the training are from 2021-03-24 to 2021-05-04. The testing is done of the future plants' data, i.e., from 2021-05-06 to 2021-06-04;
- the operations above are repeated considering the difference in time of the impedance as other features.
- the operations above are repeated with impedance module and phase after a polynomial fitting;

3.3 Generally labeled dataset

As said in section 1.3, the available data set is regarding **4** different plants, where **2** of them are watered regularly everyday and **2** are not. In order to apply a supervised machine learning algorithm, labeling of the data is necessary; the most simple procedure is to label as healthy all the samples from the two watered plants (Status = 1) and not-healthy the others (Status = 0). The watered plants are plants 2 and 3, while the ones letting to dry are enumerated as 1 and 4.

3.3.1 Implementation

The neural network is implemented in Python by exploiting the open source library **TensorFlow**. For this part of program the file *nn_training.py* should be run, this Python script can be divided into four parts:

1. some parameters are set in order to choose the data files to be imported and the features to be considered during the training;

2. preparation of the data imported from the files for the training. The dataset is divided into the training set and testing set;

3. creation of the neural network and training;

4. the network trained is saved in a *.h5* format file. Then the accuracy is computed and saved, with the model information, in a *.txt* output file.

Parameters setting In the first part of the code, some parameters are defined, in particular:

- the first parameter is set_by_hand which is set in order to choose how will be the labeling. If it is equal to 0, the labelling method used is the one described in 3.3, otherwise a more specific method is used, as described in section 3.4. The choice of input data files is between two different group of files, based on set_by_hand;

- the second one is saturation_moisture. When the soil is left to dry, the soil moisture measured could reach very high negative value, of order of 10^9, this could lead to an asymmetry between dry and wet soil. Furthermore, from the datasheet of watermark, -200kPa is the lowest values that the sensor can precisely detect. So if this parameter is 1, all the soil moisture with value less than -200 kPa are saturated to -200kPa, otherwise these values are filtered out;

- the other parameters decide which features will be used during the training of the neural network, such as temperature, airhumidity, ambientlight, moisture, date_time, impedance_module, impedance_phase. Other parameters such as module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit are not used in this part for the training phase, they will be explained later. Therefore they are set to 0 for now.

Listing 3.1: Parameters setting

```
# *****Parameters to decide the input files*****
set_by_hand = 0 # First parameter to decide if the plants used is
    with exported data or with status set by hand
saturation_moisture = 1 # If this parameter is equal to zero the
    moisture <-200kPa will be
# filtered out else it will be saturated to -200kPa
# **Decide which features should be used in the training phase**
temperature = 1
airhumidity = 1
ambientlight = 1
moisture = 1
date_time = 1
impedance_module = 1
impedance_phase = 1
module_difference = 0
phase_difference = 0
module_fit=0
phase_fit= 0
module_diff_fit= 0
phase_diff_fit = 0
# Import of the correct files names
if set_by_hand == 0:
    filename_plant1 = '../data_of_the_plants/data_export_plant1.csv'
    filename_plant2 = '../data_of_the_plants/data_export_plant2.csv'
    filename_plant3 = '../data_of_the_plants/data_export_plant3.csv'
    filename_plant4 = '../data_of_the_plants/data_export_plant4.csv'
else:
    filename_plant1 = '../data_of_the_plants/
    data_plant1_03_05_diff_withfit.csv'
    filename_plant2 = '../data_of_the_plants/
    data_plant2_03_05_diff_withfit.csv'
    filename_plant3 = '../data_of_the_plants/
    data_plant3_03_05_diff_withfit.csv'
    filename_plant4 = '../data_of_the_plants/
    data_plant4_03_05_diff_withfit.csv'
```

Data preparation For now, only the file names are known, so some operations are needed in order to import the data and prepare them for the training phase.

For this purpose, two functions are exploited. Both are written in the Python script *import_data.py*. The first function, called `import_plants()`,takes as inputs the names of the four *.csv* files and the two parameters set_by_hand and saturation_moisture; it gives as output the data of the four plants as Pandas DataFrame. The following operations are done inside this function:

- the data are imported as Pandas DataFrame by exploiting the function `pd.read_csv()` from Pandas library;
- the plants are labelled generally if set_by_hand is equal to 0, a Status column is added and it contains **zeros** if the plant is not watered regularly and **ones** otherwise. If the parameter is one, this operation is not done since the Status column is already present;
- the parameter soil_moisture decides if the values under -200 kPa should be saturated or filtered out;
- date conversion. The column of the time is in the format yyyy-mm-dd hh:mm:ss, in order to use this information, the data is converted to a numeric format by exploiting **datetime** library, and in particular only the hours and the minutes are considered. The final value corresponds to $hh \times 60 + mm$.

The second function `prepare_samples()` takes as inputs the four outputs of the previous one and the parameters, which decide the features to be used during the training phase. It gives eight outputs as NumPy data type, one sample data, and one label for each plant. Inside the function, the features that will not be used for the training are deleted, the data are converted into Numpy arrays and divided into labels and samples.

After that, the four plant data are concatenated, shuffled among them, and normalized to a number from 0 to 1. All these operations are done with **Scikit-learn** library. Finally, 80% of the data is used as training set and the remaining as testing set.

Listing 3.2: Concatenation shuffle and normalization of the data

```
from sklearn.utils import shuffle
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

# Here the 4 dataset from 4 different plants are concatenated
plant_samples = np.concatenate((plant_1_samples, plant_2_samples,
    plant_3_samples, plant_4_samples))
plant_labels = np.concatenate((plant_1_labels, plant_2_labels,
    plant_3_labels, plant_4_labels))

#Shuffle of the data set
```

```
plant_labels, plant_samples = shuffle(plant_labels, plant_samples)

#Normalization of the samples
scaler = MinMaxScaler(feature_range=(0,1)) #the data are normalized
    to the range 0,1
scaled_plant_samples = scaler.fit_transform(plant_samples)
plant_samples_train, plant_samples_test, plant_labels_train,
    plant_labels_test = \
                train_test_split(scaled_plant_samples, plant_labels,
    test_size = 0.2, random_state = 4)
```

Creation and training As mentioned before, the neural network is implemented by using **TensorFlow** library. Thanks to this library, it will not be hard to do this step. The model is first created, defining the number of layers, how many nodes are in each layer, and the activation function. Then the neural network is trained, after deciding different parameters, such as the optimizer, the loss computation, and the metrics to be displayed during the training. For the creation of the model `tf.keras.Sequential` is exploited; it allows the addition of a linear stack of layers to `tf.keras.Model`. For each layer, it is possible to define an activation function that transforms the weighted sum to an output, adding non-linearity to the network. Usually, the activation functions of the hidden layers are the same, and they differ from the one of the output layer. Many types of activation functions are available as described in [29]: relu (rectified linear unit) function, sigmoid function, softmax function, tanh function and more. The relu function is a piecewise function; it converts all the negative values to zero. While the softmax function gives a probability distribution as output, the values are between 0 and 1, and the sum equals 1. In this first implementation, the relu function is assigned to the hidden layer and the softmax to the output layer. The two outputs indicate the probability of the plant to be in the two states, respectively. With the `summary` method, it is possible to print the summary of the defined model.

In the second part, there is the training of the neural network. The `compile` method is used to configure the model for training. The main arguments to be decided are the optimizer, the loss function, and the metrics. The available optimizers are described in [30]: SGD, RMSprop, Adam, Adadelta, Adagrad, Adamax, Nadam, Ftrl. The Adam (Adaptive Moment Estimation) optimizer [31] is the common choice for deep learning cases since it works empirically very well. It is a stochastic gradient descent method that takes into account also the moments of first-order and second-order. The idea comes from a physic concept. Image a 2-D curve surface, and a ball represents the cost function. If the ball is left to fall, in the end, it will reach the bottom part of the surface, which corresponds to the global minimum for the cost function. The gradient descent, once it reaches the minimum, will be stuck there. In contrast, the ball will oscillate several times and stop at the bottom for friction. This idea is applied to gradient descent by considering the gradient update of the previous step. A mathematical description is presented in [32]. The previous gradient descent step, also called momentum, is considered as:

$$sum\_of\_gradient = previous\_sum\_of\_gradient \times \beta_1 + gradient \times (1 - \beta_1) \tag{3.1}$$

RMSprop(Root Mean Square Propagation) is also present in Adam optimizer, in order to accelerate the converge procedure:

$$\begin{split} sum\_of\_gradient\_squared = previous\_sum\_of\_gradient\_squared \times \beta_2 \\ + gradient^2 \times (1 - \beta_2) \end{split} \tag{3.2}$$

Finally the step to update the weights is computed as:

$$\delta = -learning\_rate \times \frac{sum\_of\_gradient}{\sqrt{(sum\_of\_gradient\_squared)}} \tag{3.3}$$

$$weights := weights + \delta \tag{3.4}$$

The root mean square is used to shrink the relevance of the previous gradient. Therefore, compared to stochastic gradient descent, the Adam optimizer is much faster, and it gives the possibility to escape from local minima. Default values are used for $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$, which is used to prevent divisions by zero. While the learning rate is set to 0.0001.

The second parameter to be decided is the loss function [33]. Loss functions are divided into probabilistic and regression losses. Usually, the cross-entropy loss is used for classification problems. It can compute the difference between the probability distribution of the predicted result and the sample label. For this case, both binary_crossentropy or sparse_ categorical_crossentropy can be used, and the difference is that the second one is also able to manage multi-class classification problems.

Finally, the metrics to be displayed during the training have to be chosen [34]. A metric function is used to evaluate the performance of the model, it is similar to the loss function, but it is not used for the training. The accuracy metric, which computes how often predictions match labels, is usually chosen for a classification problem.

The `fit` method is then used to train the model after setting the algorithm hyperparameters, such as the validation split, the batch size, which indicates the number of samples per gradient update, and the number of epochs, where an epoch is an iteration over the entire training set.

Listing 3.3: Creation and training of the neural network

```python
## Creation of the neural network to be trained
nn_layers = [6, 6, 2]
model = Sequential([
    Dense(units = nn_layers[0], input_shape=(feature_size,), 
   activation='relu'), #this represents the second layer
    Dense(units = nn_layers[1], activation = 'relu'),
    Dense(units = nn_layers[2], activation = 'softmax')
])
#
#model.summary()
#
## Training of the neural network
epochs_num = 300
model.compile(optimizer=Adam(learning_rate = 0.0005), loss='
   sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x=plant_samples_train, y=plant_labels_train,
   validation_split=0.1, batch_size=10, epochs=epochs_num, shuffle=
   True, verbose=2)
```

Running the program, during the training phase, the loss and accuracy of both training set and validation set are displayed as shown in .

```
Epoch 296/300
255/255 - 0s - loss: 0.2336 - accuracy: 0.9007 -
               val_loss: 0.2006 - val_accuracy: 0.9225
Epoch 297/300
255/255 - 0s - loss: 0.2335 - accuracy: 0.9011 -
               val_loss: 0.1996 - val_accuracy: 0.9261
Epoch 298/300
255/255 - 0s - loss: 0.2330 - accuracy: 0.9015 -
               val_loss: 0.2003 - val_accuracy: 0.9225
Epoch 299/300
255/255 - 0s - loss: 0.2329 - accuracy: 0.8984 -
               val_loss: 0.1992 - val_accuracy: 0.9261
Epoch 300/300
255/255 - 0s - loss: 0.2323 - accuracy: 0.9004 -
               val_loss: 0.2003 - val_accuracy: 0.9225
The training has finished, the model is saved in
../Neural_network_trained/plant_01_1111111000000_7662.h5
and the results are saved in
../output_files/output_neuralnetwork.txt
```

At the end, the model and the results are saved.

Saving model and results Once the training is finished, the model is saved in a *.h5* format file in the directory **Neural_network_trained**. The filename is something like $plant\_a_0a_1\_a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}a_{12}a_{13}a_{13}\_a_{14}a_{15}a_{16}a_{17}.h5$, where each char is a number with a particular meaning, as explained in table 3.1. Then the results, such as train accuracy, test accuracy and confusion matrices are saved in a *.txt* file named *output_neuralnetwork.txt* in the directory **output_files**. The results for this part of implementation are:

```
The network is saved as:
../Neural_network_trained/plant_01_1111111000000_7662.h5
The overall accuracy of this network is: 0.8983625070581592
The train accuracy is: 0.9015178256265443
The test accuracy is: 0.8857545839210155
The overall confusion matrix is [[1527  245]
                                 [ 115 1655]]
The train confusion matrix is [[1231  189]
                               [ 90 1323]]
The test confusion matrix is [[296  56]
                              [ 25 332]]
```

3.3.2 Results discussion

The test accuracy is similar to the train one, and overall accuracy of 89.8% is reached. Furthermore, the confusion matrices are also considered since accuracy will produce misleading results in the case of an unbalanced data set, which means that the numbers of observations in different classes vary greatly. It is possible to notice that the number of false positives and false negatives are almost the same. The accuracy in the two cases, with moisture saturation and not, is almost the same. Therefore, the soil moisture will always be saturated for future implementations since this will lead to more data for the training set.

Accuracy of 89.8% is not an excellent result, but it is definitively something to say that it may be possible to predict the plant status with a neural network. A doubt that might come to mind is if the labeling method is correct. Reminding that in this first step, the labels, i.e., the Status column, have been added based on the watering condition on the plants, the pictures of the plants can be observed to check if the labels are appropriate.

For this purpose, the picture of the four plants taken on 31/03/2021 at 10:28:34 is shown in figure 3.1. Plant 1, which is not watered, has yellow leaves, while plant 2, which should be healthy, has big green leaves with bright colors. However, plant 3, which was labeled as healthy, and plant 4, which should be dry, are opposite. That means that the general labeling done in this section is not correct since the watering events have no immediate effect on the plant status. Therefore, a more precise and correct labeling method is needed.

Character	Corresponding parameter	value	meaning
a_0	set_by_hand	0/1	it indicates if the data used for the training is set by hand or not
a_1	saturation_moisture	0/1	it indicates if the soil moisture with high negative value is filtered out or saturated
a_2	temperature	0/1	if these parameters are 1, the corresponding measurement is used as a feature during the training phase
a_3	airhumidity	0/1	
a_4	ambientlight	0/1	
a_5	moisture	0/1	
a_6	date_time	0/1	
a_7	impedance_module	0/1	
a_8	impedance_phase	0/1	
a_9	module_difference	0/1	
a_{10}	phase_difference	0/1	
a_{11}	module_fit	0/1	
a_{12}	phase_fit	0/1	
a_{13}	module_diff_fit	0/1	
a_{14}	phase_diff_fit	0/1	
$a_{15}, a_{16}, a_{17}, a_{18}$	nn_layers[ ]	positive integer number	these numbers represent the number of nodes in each layer, input layer, hidden layers and output layer

Table 3.1: $plant\_a_0a_1\_a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}a_{12}a_{13}a_{13}\_a_{14}a_{15}a_{16}a_{17}.h5$, explanation of this filename

3.4 Labeling by hand

Preparing the dataset for the training is one of the most challenging parts, not only because the sampled data, the inputs, must make sense, but also the labels, i.e., the outputs, must be correct. There have been many more or less severe disasters of machine learning applications in the real world due to a bad learning data set.

In March 2016, Microsoft released on Twitter an AI chatbot called Tay. The training data was some anonymized public data and material written by comedians.

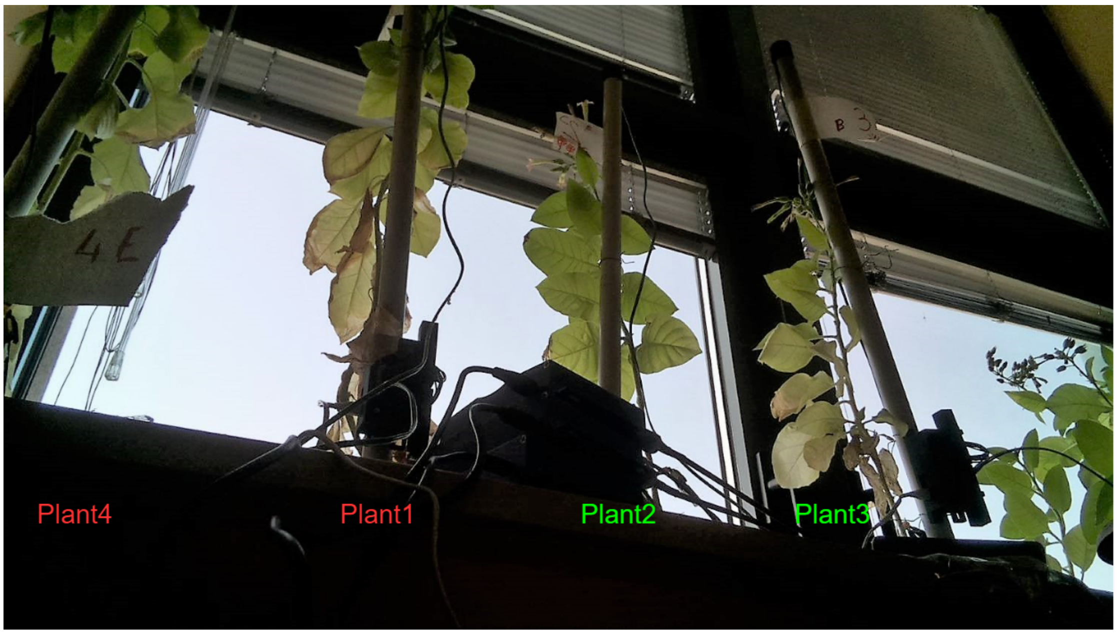


Figure 3.1: Picture of the four plants. Date: 31/03/2021 10:28:34

Then the bot, pretending to be a teen girl, learned and evolved continuously from its interactions with other users on the social media platform. The result is that within 16 hours, 95000 tweets were published that quickly became openly racist, misogynistic, and anti-Semitic.

In 2018, an Uber self-driving car killed a woman that was walking her bike. The test driver on the car, which had not been properly vetted or trained, was watching her phone until half a second before the crash. The sensors on the car had detected the woman six seconds before the collision. However, it could not identify her as a pedestrian, and the system did not make the correct decision for action. The system was able to identify a pedestrian or a rider, but it had never trained for this kind of situation. Machine learning is a powerful tool, helping and improving humans' life. Nevertheless, mistakes can lead to useless algorithms or even irreparable errors.

In order to label the plants correctly, the pictures of the plants in this period are exploited and looked at one by one, and the health status is determined based on the leaves conditions of the tobacco plants.

Starting from April 9th, as shown in figure 3.2, the leaves of plant 4 start becoming yellow, while plant 3 still seems to like dry. From April 23rd, plant 3 has new green leaves growing on the bottom, so it can be considered healthy from now. From this date, all the plants' states are coherent with their watering conditions:

plants 2 and 3 are labeled with Status equal to 1 and the others with Status equal to 0.

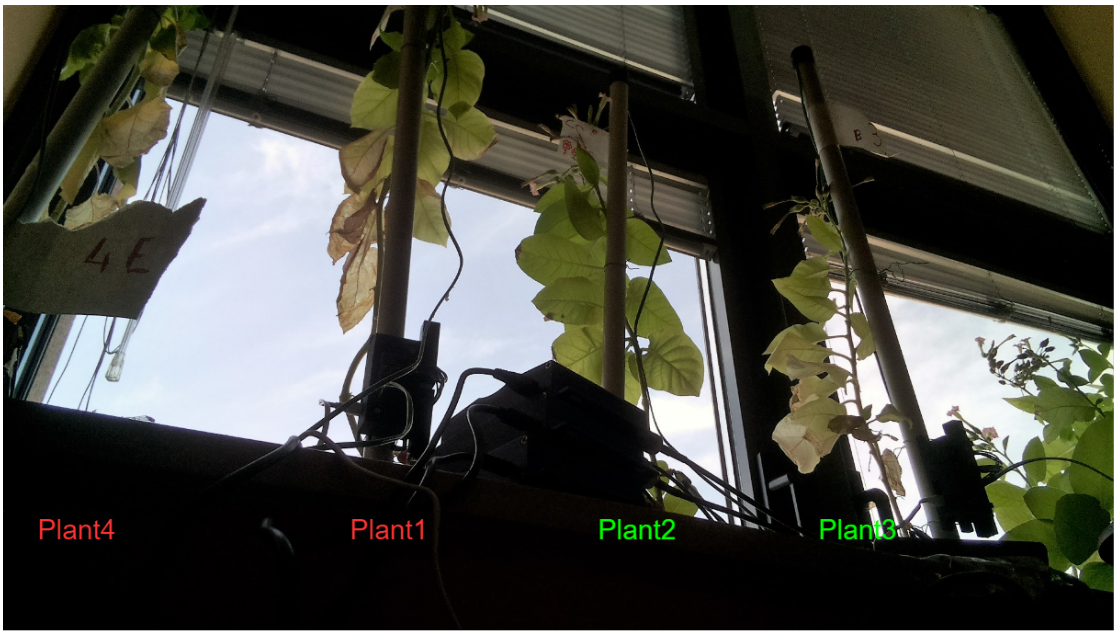


Figure 3.2: Picture of the four plants. Date: 09/04/2021 11:28:34

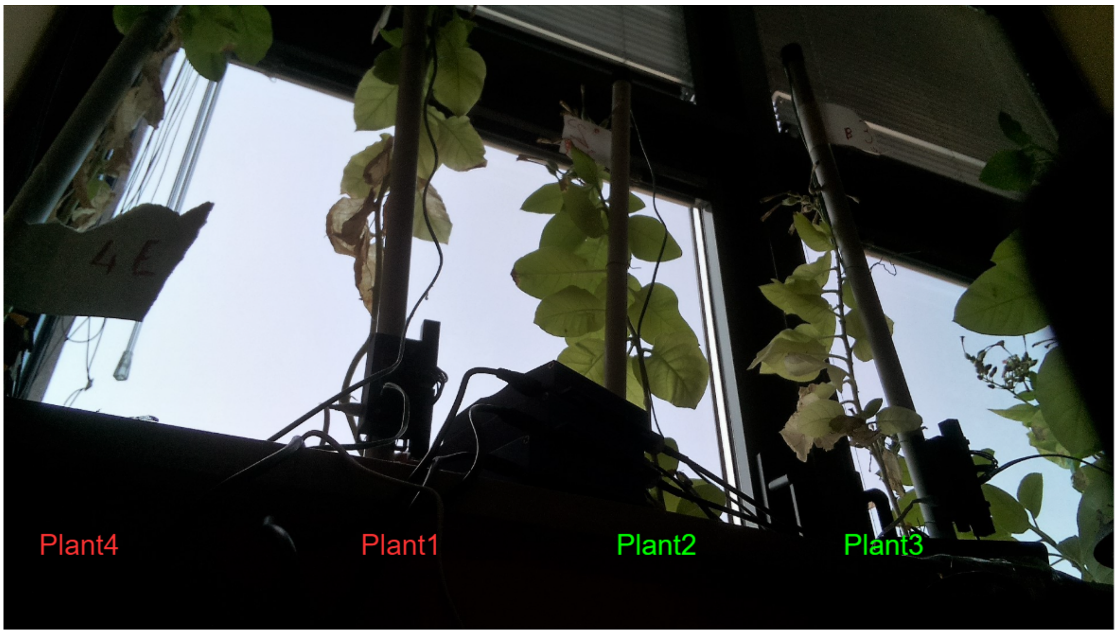


Figure 3.3: Picture of the four plants. Date: 23/04/2021 10:45:48

3.4.1 Implementation

Once the correct labels are obtained, the neural network model has been trained again. For this part, the same Python script as the one described in section 3.3.1 can be used, except in this case, the parameter `set_by_hand` is set to one. Therefore, the plants' data files imported already contain the Status column. Also, in this case, the trained model is saved in a *.h5* format file, and the performance is saved in the same output file as before, which is *output_neuralnetwork.txt*.

3.4.2 Results discussion

Here too, 20% of the data are used for the testing and 80% for the train. The performances reached are 78.6% for the training accuracy and 80.8% for the test accuracy. It is possible to notice that with the correct labeling, the performance has even fallen by 10% concerning the previous case; that is because each plant has a different range of impedance module value, and in the previous case, since each plant was always labeled with one state, it was as if the label was referring to which plant the data belongs since it did not correctly indicate the plant status.

3.4.3 Optimal model and algorithm hyperparameters

Until now, neural networks with two hidden layers, each with six neurons, have been used. A question that arises is if these are the optimal model hyperparameters for this problem. In order to find the optimal hyperparameters, the `GridSearchCV` of **sklearn** library is exploited. Starting from the computations described in section 3.1, several networks with **two** hidden layers have been built with the internal nodes from **two** to **seven**. This function trains the neural networks for all the combinations of the hyperparameters passed as argument, and the one with the best performance is chosen. The Python script *optimal_N_of_layers.py* was run several times, but each time, it gives a different result:

```
0.60524730637305
{'batch_size': 10, 'epochs': 120, 'nn_layer_1': 6, 'nn_layer_2': 5}
0.55472052693367
{'batch_size': 10, 'epochs': 120, 'nn_layer_1': 5, 'nn_layer_2': 4}
0.562247359752655
{'batch_size': 10, 'epochs': 120, 'nn_layer_1': 6, 'nn_layer_2': 2}
0.6126938939094544
{'batch_size': 10, 'epochs': 300, 'nn_layer_1': 6, 'nn_layer_2': 6}
0.5764867194571285
{'batch_size': 10, 'epochs': 200, 'nn_layer_1': 5, 'nn_layer_2': 5}
```

The numbers represent a parameter called F1 Score, used to evaluate the performance of a machine learning algorithm, it is defined as:

$$\text{F1 score} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} \tag{3.5}$$

Therefore, F1 score is the harmonic mean of precision and recall. Precision indicates the truly positive percentage out of the positive predicted, TP/(TP + FP). At the same time, recall refers to the rate between true positive and the total number of positives, corresponding to TP/(TP+FN). Each time the script indicates a different number of nodes as optimal values. This is due to the local minima. In each training, a network may stall in a different local minimum, giving a different performance. In order to solve this problem, the script should be run many times, but each time 36 neural networks should be trained. Suppose that the optimal network is decided based on the number of times it results from the script; for instance, the optimal network is the one that appears five times. In the worst scenario, the script should be run $36 \times 4 + 1$ times, supposing each model results as optimal four times; the 145-th time is needed to decide the optimal parameters. Each time the script is run, 36 neural networks are trained, so a total of 5220 neural networks are trained. Suppose an average training time of five minutes for each network, 26100 minutes are required to finish the training of all the networks, which means 435 hours or more than 18 days. It is not impossible to do that, but it is not necessary. In order to have an idea of how the number of layers and nodes affect the neural network performance, some neural networks are trained by hand using the script *nn_training.py*:

- a network with two hidden layers, the first one with six nodes and the second one with four nodes;
- a network with two hidden layers, the first one with four nodes and the second one with three nodes;
- a network with two hidden layers, the first one with fourteen nodes and the second one with seven nodes;
- a network with three hidden layers, the first one with fourteen nodes, the second one with ten nodes, and the third one with six layers;

Each model is trained six times in order to see the effects of local minima; the results are summarized in table 3.2. It is not very meaningful to compute the average accuracy of the models trained more times. Therefore, the final accuracy considered is the maximum testing accuracy.

It is possible to notice that generally, the accuracy increases with the number of nodes. With a different number of nodes, the accuracy ranges from 71.9% to 82.1%. However, the maximum accuracy is also reached by the network with layers

model name	training accuracy	testing accuracy	maximum testing accuracy
$plant\_11\_1111111000000\_7642.h5$	62.6%	61.1%	73.4%
	73.6%	72.1%	
	67.0%	68.7%	
	72.2%	71.1%	
	65.4%	65.2%	
	73.5%	73.4%	
$plant\_11\_1111111000000\_7432.h5$	59.9%	58.8%	71.9%
	69.0%	67.4%	
	71.5%	71.9%	
	63.1%	64.7%	
	69.2%	65.5%	
	56.6%	54.6%	
$plant\_11\_1111111000000\_71472.h5$	56.4%	56.2%	77.8%
	78.7%	74.7%	
	77.6%	77.8%	
	73.9%	70.9%	
	70.7%	71.4%	
	77.6%	76.2%	
$plant\_11\_1111111000000\_7141062.h5$	78.0%	75.3%	81.4%
	70.8%	71.1%	
	78.5%	76.7%	
	81.2%	81.4%	
	77.3%	77.1%	
	72.6%	69.6%	

Table 3.2: Training of different version of neural networks

7-6-6-2. Furthermore, using too many layers and nodes will lead to overfitting. Therefore, it is not convenient to train a more complex neural network for the same performance. From now the neural network with layers 7-6-6-2 will always be used. Note that the 7 is the dimension of the input layer, which depends on the number of features used during the training phase. Therefore it can change.

Another parameter that could affect the performance is the activation function for the hidden layers. Till now, the relu function is used; it will be helpful to train a neural network with a sigmoid activation function. The model is trained several times, and the following testing accuracy is obtained: 56.4%, 60.8%, 60.7%, 60.1%. Furthermore, it can be observed that the loss is stuck from the first epoch, which means that the accuracy is not improved during the training. So, the relu activation function is chosen for future models.

The question is if the algorithm hyperparameters, such as the chosen number of epochs and the learning rate, are suitable values. Now, the learning rate is set to 0.0001 and the number of epochs to 300. If the learning rate is too low, the training would be prolonged; otherwise, it could not reach a convergence point. In order to evaluate these parameters, the cross-validation technique is exploited. In particular, a part of the training set is used as validation set. Based on the validation accuracy, the hyperparameters can be tuned. With the intention to see if the hyperparameters are suitable, the training accuracy and the validation accuracy are plotted over the epoch. In figure 3.4, the accuracy obtained with learning rate set to 0.0001 and 300 epochs is shown. It is possible to observe that

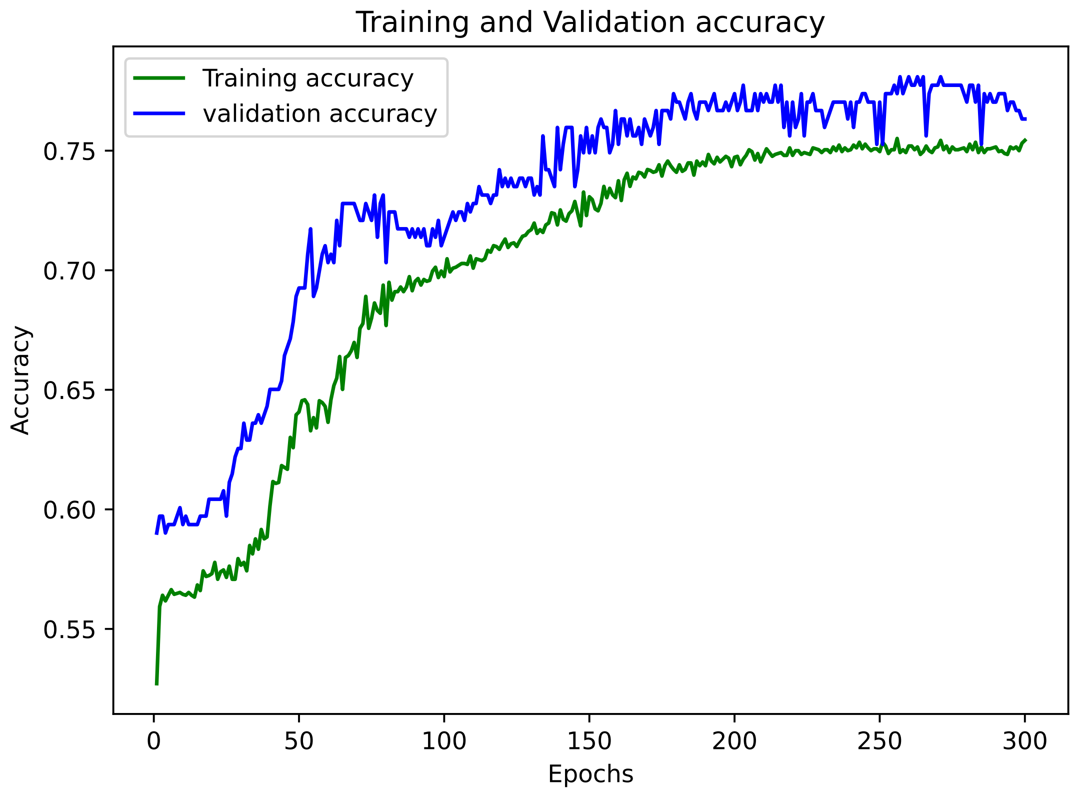


Figure 3.4: Training accuracy and validation accuracy. Learning_rate = 0.0001, epochs = 300

the accuracy increase is relatively fast, so the learning rate is set to a good value. Therefore, the accuracy becomes flat at the end, which means that the number of epochs is enough. The learning rate and epochs are set to other values to see their effects on the training. In figure 3.5 the accuracy obtained with learning rate set to 0.00001 and 300 epochs is shown. It is visible that the accuracy is almost constant, and the number of epochs is not enough. In figure 3.6 the number of epochs is increased to 1000, and it is still not enough.

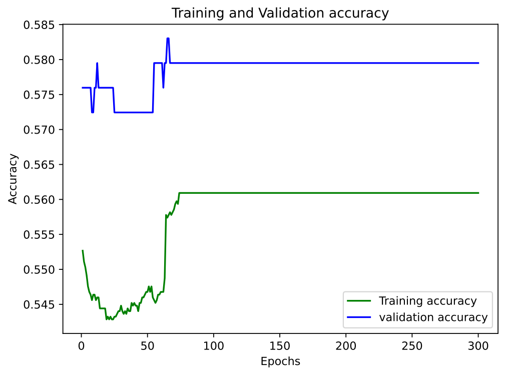


Figure 3.5: Training accuracy and validation accuracy. Learning_rate = 0.0001, epochs = 300.

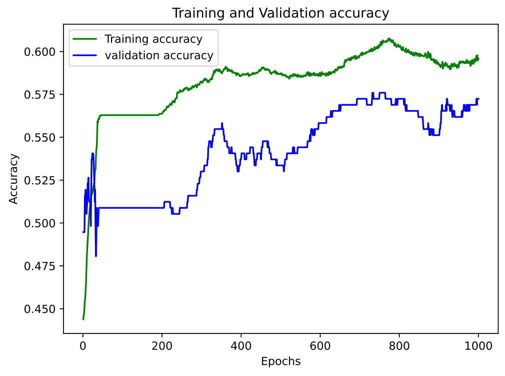


Figure 3.6: Training accuracy and validation accuracy. Learning_rate = 0.0001, epochs = 1000.

To summarize, the hyperparameters chosen are good values. Therefore the learning rate is left to 0.0001; two hidden layers are kept, each with six nodes, and each model will be trained several times to avoid local minima.

3.5 Future predictions

A possible application of the trained model is to predict the plant's status for future data. The idea is to train a neural network with the data of some plants and then use the trained network to monitor the health status of the same plants. It is not an optimal method because it means that the neural network should be trained again when monitoring new plants are required. However, it is an excellent way to evaluate the performance of the model and to detect overfitting. The trained model is used to test the data from 2021-05-06 to 2021-06-04.

3.5.1 Implementation

In this step, the script *future_predictions.py* is exploited, which description can be divided into four parts:

1. some parameters are set in order to choose the trained model to be imported and the data file names are defined;

2. preparation of the imported data for the testing. A proper normalization for the data is need;

3. import of the neural network to be tested and computation of the accuracy;

4. the results are saved is a *.txt* output file.

Parameters setting The parameters which decide the features and the number of nodes in the two layers are set. These parameters are used to select the trained neural network to be imported. After that, the file names are defined. Two groups of file names are needed; the first group is regarding the data used for the training, the old data. The second one is the file names of the data to be tested;

Data preparation Once the file name are known, two functions, `import_plants` and `prepare_samples`, are exploited to import both the old and new data. In this case, the normalization is tricky. The normalization of the new samples can not be done using the data set itself, but the data set used for the training should be considered since the normalization during the training phase was done in that way. Otherwise, the neural network will give wrong predictions. Therefore, the normalization can be expressed as:

$$x' = \frac{x - x_{old-min}}{x_{old-max} - x_{old-min}} \tag{3.6}$$

For this reason also the old dataset are imported in this program.

Listing 3.4: Normalization of the testing set

```
scaled_1_samples = (plant_1_samples - np.min(data_plants_old, axis=0))/(np.max(data_plants_old, axis=0) - np.min(data_plants_old, axis=0))
scaled_2_samples = (plant_2_samples - np.min(data_plants_old, axis=0))/(np.max(data_plants_old, axis=0) - np.min(data_plants_old, axis=0))
scaled_3_samples = (plant_3_samples - np.min(data_plants_old, axis=0))/(np.max(data_plants_old, axis=0) - np.min(data_plants_old, axis=0))
scaled_4_samples = (plant_4_samples - np.min(data_plants_old, axis=0))/(np.max(data_plants_old, axis=0) - np.min(data_plants_old, axis=0))
```

Import of the model and accuracy computation To import the trained model it is possible to use `load_model` from Keras. Once the model is obtained , the accuracy and confusion matrices of the four plants are computed separately.

Saving the testing results Finally, the results are saved in a *.txt* file called *output_future_prediction.txt* in the folder **output_files**:

```
The following accuracy is computed with the neural network trained with the
name../Neural_network_trained/plant_11_1111111000000_7662.h5
 on the data sampled in the period 06/05/2021 to 07/06/2021
The accuracy of this network on plant 1 in the month of May is: 0.904
The confusion matrix is[[452  48]
 [  0   0]]
The accuracy of this network on plant 2 in the month of May is: 0.994
The confusion matrix is[[  0   0]
 [  3 497]]
The accuracy of this network on plant 3 in the month of May is: 0.923
The confusion matrix is[[  0   0]
 [ 14 179]]
The accuracy of this network on plant 4 in the month of May is: 0.777
The confusion matrix is[[383 110]
 [  0   0]]
```

3.5.2 Results discussion

The results reached for future data prediction are outstanding, with more than 90% for plants 1, 2, and 3. For plant 2, only three wrong predictions out of 500 sampled data. While plant 4 reaches 78% of accuracy. All these numbers suggest that the neural network trained in this way can predict the health condition for the same plants used during the training phase. Nevertheless, the algorithm trained in this way is not practical in an actual application. However, this is a small achievement since this method can be used to evaluate the performance of a network detecting overfitting.

3.6 Relevance of the impedance

As has always been said, this thesis aims to find a way to predict the plant health status with the impedance value. Other authors have demonstrated the correlation between impedance and other environmental parameters, but the importance of the impedance for the machine learning algorithm still needs to be proved. With this intention, a neural network is trained with only two features: impedance module and phase.

3.6.1 Implementation with only impedance

The same Python script described in section 3.3.1 can be used. All the parameters indicating the features are set to zero except the impedance module and impedance phase. The result is saved in the same output file, i.e., *output_neuralnetwork.txt*:

```
The network has saved as:
../Neural_network_trained/plant_11_0000011000000_2662.h5
The overall accuracy of this network is: 0.7058159232072275
The train accuracy is: 0.7042004941757853
The test accuracy is: 0.7122708039492243
The overall confusion matrix is[[1529  461]
 [ 581  971]]
The train confusion matrix is[[1214  367]
 [ 471  781]]
The test confusion matrix is[[315  94]
 [110 190]]
```

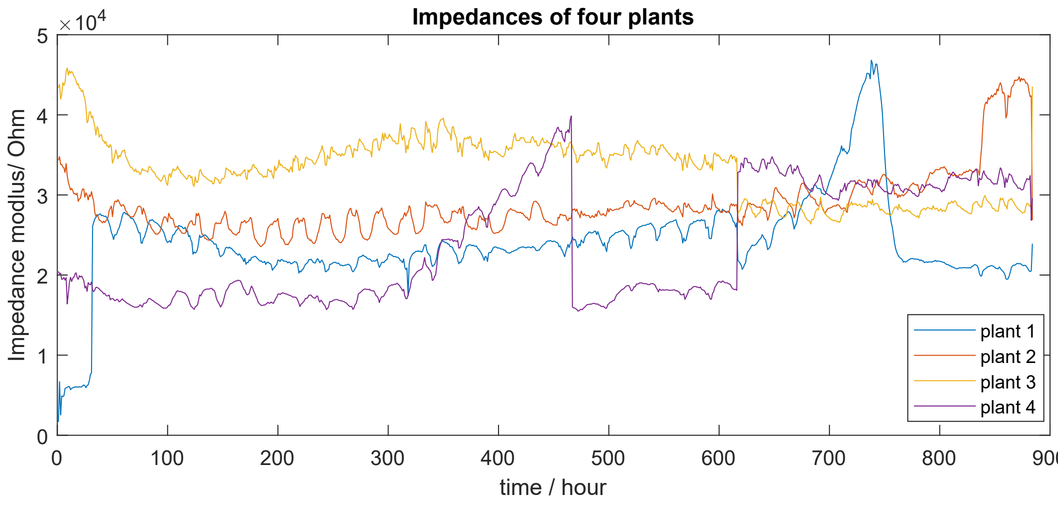


Figure 3.7: Impedance module of the four plants during the period from 2021-03-24 to 2021-05-04

The neural network has an accuracy of 70%. It has reduced by about 10%, but it is still something to say that detecting the plant status is possible with only impedance. However, this result could be tricky. It is possible to notice that the impedance of the plants overlaps in specific periods by observing the impedance module and phase in figures 3.7 and 3.8. For instance, plants 2 and 4 have very similar modules even if the two plants are in different health conditions. At the same time, the phases overlap almost all the time. Therefore, the relatively good performance reached by the neural network could be due to overfitting. The model is tested on future data in order to verify if there is really overfitting. The Python script *future_predictions.py*, which is described in section 3.5.1 is exploited. Only the parameters impedance_module and impedance_phase are set to one in order to import the correct model, and the following results are obtained:

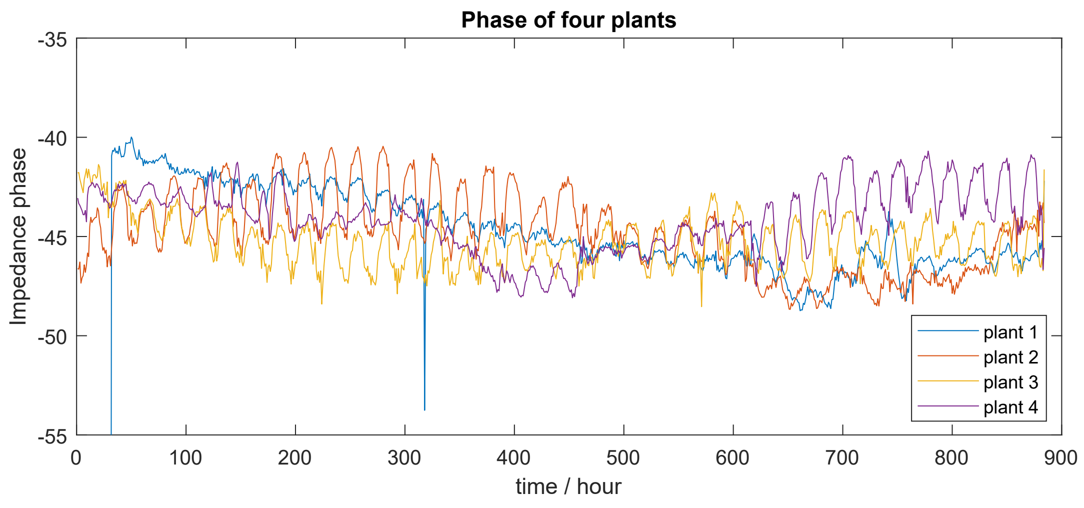


Figure 3.8: Impedance phase of the four plants during the period from 2021-03-24 to 2021-05-04

```
The following accuracy is computed with the neural network trained with the
name ../Neural_network_trained/plant_11_0000011000000_2662.h5
 on the data sampled in the period 06/05/2021 to 07/06/2021
The accuracy of this network on plant 1 in the month of May is: 0.992
The confusion matrix is[[496   4]
 [  0   0]]
The accuracy of this network on plant 2 in the month of May is: 0.208
The confusion matrix is[[  0   0]
 [396 104]]
The accuracy of this network on plant 3 in the month of May is: 0.057
The confusion matrix is[[  0   0]
 [182  11]]
The accuracy of this network on plant 4 in the month of May is: 0.351
The confusion matrix is[[173 320]
 [  0   0]]
```

The accuracy of the prediction for plants 2, 3, and 4 is very low. Therefore, it is possible to conclude that it is impossible to train a neural network with only impedance since overfitting is present even if the training accuracy is not relatively high. Another method or approach is necessary to prove the importance of the impedance and that a neural network with good performance can not be built only with environmental data. For this purpose, the “*reductio ad absurdum*” approach is used. Suppose that the impedance is not essential for the plant status detection. Therefore, if it is not used during the training phase, the neural network’s performance will remain almost the same.

3.6.2 Implementation without impedance

As said before, training without impedance values is done. Therefore, the features are only 5, which are the environmental data and the time. The usual script, i.e., *nn_training.py*, is used. The following results are obtained:

```
The network has saved as:
../Neural_network_trained/plant_11_1111100000000_5662.h5
The overall accuracy of this network is: 0.6716544325239977
The train accuracy is: 0.6703141546064243
The test accuracy is: 0.6770098730606487
The overall confusion matrix is[[1496  494]
 [ 669  883]]
The train confusion matrix is[[1190  398]
 [ 536  709]]
The test confusion matrix is[[306  96]
 [133 174]]
```

The accuracy has dropped by 10%, which means that eliminating the impedance module and phase from the training has hardly affected the neural network performance. However, this performance loss could be due to the reduced number of features used during the train. Therefore, several neural networks have been trained using five features. Two features among soil moisture, temperature, air humidity, and light intensity, are removed, leading to ten combinations. The results are shown in figure 3.9. The numbers in the bar graph correspond to the trained

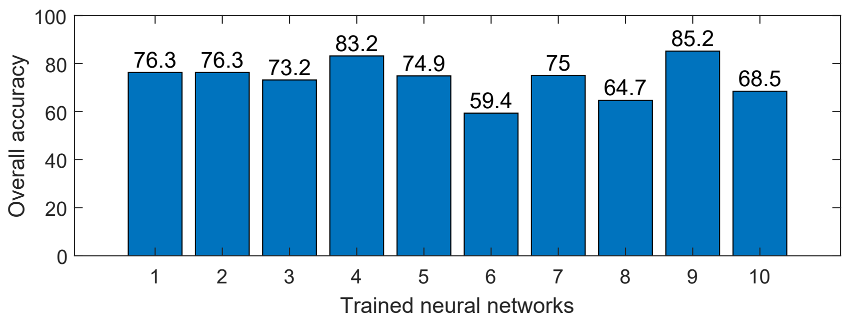


Figure 3.9: Results of the ten different neural networks

neural network, in particular:

1. temperature and air humidity are removed from the features;
2. temperature and ambient light are removed from the features;
3. temperature and soil moisture are removed from the features;

4. temperature and time are removed from the features;
5. air humidity and ambient light are removed from the features;
6. air humidity and soil moisture are removed from the features;
7. air humidity and time are removed from the features;
8. ambient light and soil moisture are removed from the features;
9. ambient light and time are removed from the features;
10. soil moisture and time are removed from the features;

It is possible to observe that the accuracy remains relatively high to 80% in almost all the cases. The accuracy decreases to 65% in some cases: when one of the two features removed is the soil moisture.

Therefore, it is possible to conclude that the impedance module and phase are two relevant features in a plant detection problem. If they are removed, the accuracy of the trained neural network drops significantly. Moreover, also the soil moisture hardly influences the network performance.

3.7 Three plants for the training phase

An optimal way to use a machine learning algorithm is to prepare the model for use in any situation. It will be beneficial if the model trained with the four plants can be directly used on-field on any other plants. Since the available data set is not unlimited, this situation is emulated by training a neural network with three plants, and then the model is tested on the fourth one.

3.7.1 Implementation

The training and testing are done with Python scripts in the folder **Machine learning**. For the training *three_plants_train.py* is used, this script trains four models, each one is trained with three different plants. Once the models are obtained, the script *three_plants_accuracy* is exploited in order to evaluate the models' performance.

Training phase The procedure is very similar to the training of other networks. In the first part, the data are imported and prepared for the training. Then the models are defined and trained. Finally, the models are saved as *.h5* format files. The only difference is that now four different datasets are considered, obtained taking three plants out of four:

Listing 3.5: Four different testing set are obtained

```
data_123 = pd.concat([data_plant1, data_plant2, data_plant3]) #data of the plants 1 2 3
data_124 = pd.concat([data_plant1, data_plant2, data_plant4]) #data of the plants 1 2 4
data_134 = pd.concat([data_plant1, data_plant3, data_plant4]) #data of the plants 1 3 4
data_234 = pd.concat([data_plant2, data_plant3, data_plant4]) #data of the plants 2 3 4
```

Therefore, for each training set a neural network is trained:

Listing 3.6: Four different networks are trained and saved

```
model123.compile(optimizer=Adam(learning_rate = 0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model123.fit(x=scaled_123_samples, y=data_123_labels, validation_split=0.2, batch_size=10, epochs=200, shuffle=True, verbose=2)

model124.compile(optimizer=Adam(learning_rate = 0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model124.fit(x=scaled_124_samples, y=data_124_labels, validation_split=0.2, batch_size=10, epochs=200, shuffle=True, verbose=2)

model134.compile(optimizer=Adam(learning_rate = 0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model134.fit(x=scaled_134_samples, y=data_134_labels, validation_split=0.2, batch_size=10, epochs=200, shuffle=True, verbose=2)

model234.compile(optimizer=Adam(learning_rate = 0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model234.fit(x=scaled_234_samples, y=data_234_labels, validation_split=0.2, batch_size=10, epochs=200, shuffle=True, verbose=2)

filename_neuralnetwork_1 = 'Neural_network_trained/plant_neural_network_123_bh.h5'
filename_neuralnetwork_2 = 'Neural_network_trained/plant_neural_network_124_bh.h5'
filename_neuralnetwork_3 = 'Neural_network_trained/plant_neural_network_134_bh.h5'
filename_neuralnetwork_4 = 'Neural_network_trained/plant_neural_network_234_bh.h5'
```

At the end of the training, each model is saved in a *.h5* format file in the folder **Neural_network_trained**.

Testing phase Once the models are trained, they are tested using the script *three_plants_accuracy.* The testing is done on the fourth plant, which is the one not used during the training. For example, if the model is trained with plants 1, 2, and 3, then the model is tested on plant 4. A proper normalization is necessary; the data of plant 4 should be normalized using the minimum and maximum values of the other plants since they are used during the training phase. The results are saved in the file **output_threeplants.txt** in the folder **output_files**:

```
Using for the training a network of dimension 7 6 6 2:
Plants 1 2 3 as training
The accuracy of this network is: 0.5857787810383747
The confusion matrix is[[384 135]
 [232 135]]
Plants 1 2 4 as training
The accuracy of this network is: 0.8710407239819005
The confusion matrix is[[524  92]
 [ 22 246]]
Plants 1 3 4 as training
The accuracy of this network is: 0.827313769751693
The confusion matrix is[[  0   0]
 [153 733]]
Plants 2 3 4 as training
The accuracy of this network is: 0.36455981941309257
The confusion matrix is[[292 563]
 [  0  31]]
```

3.7.2 Results discussion

The testings, done on plants 2 and 3, when plants 1, 3, 4, and plants 1,2,4 are used for training, respectively, give pretty good performance. In both cases, the accuracy is greater than 82%. It means that the idea of preparing a machine learning algorithm for plant health status detection ready to be used for other tobacco plants is feasible. However, when plant 4 is used for testing, the accuracy is only 58.6%, even worse for plant 1. This bad performance is because each plant has its range of impedance. This problem must be solved in order to reach a reliable system. For this purpose, two data transformation techniques can be exploited. The first one is to consider the impedance difference between two adjacent samples in time as additional features for the training. The second one is regarding a polynomial fitting that filters out the daily cycle variation of the data, considering only the general trend of the impedance. Also, in this case, the difference can be considered.

3.8 Impedance difference

The problem of different impedance ranges can be reduced by considering the impedance variation instead of the value itself. Therefore, the other two columns, i.e., two new features, should be added to the original plant data. These new features are the difference of the impedance module and the difference of impedance phase. Then, it is possible to create new *.csv* files containing these new columns. Once the new data are obtained, the usual *nn_training.py* can be used.

3.8.1 Adding the impedance difference

With the intention to do this step, a new Python script is exploited, which is *diff_considering.py*. This script imports the data, computes the difference and saves the results in new data files. The data are imported exploiting the `pd.read_csv` function of Pandas library. Then, the differences are computed and saved with a custom function called `add_difference`. This function takes two inputs, the data frame imported with the Pandas function and the file's name where the data wants to be saved. It takes the data frame and generates two NumPy data frames, one without the first row and one without the last row. After that, they have converted again into Pandas data frame. Furthermore, two new columns, called module_diff and phase_diff are computed, making simply the difference between the impedance modulus columns of the two data frames and the difference between the impedance phase columns of the two data frames. Finally, the data with new columns are saved into a new *.csv* file.

Listing 3.7: Function used to add the difference to the data

```python
def add_difference(data_plant1, filename_data1_tobesaved):
    data_plant_1_n = data_plant1.to_numpy() #convert the data to numpy
    plant1_Nplus1 = np.delete(data_plant_1_n, 0 , 0)#delete the first row
    plant1_N = np.delete(data_plant_1_n, -1 , 0)#delete the last row
    #Now the two string are converted again into pandas dataframe
    data_plant_1_N = pd.DataFrame(data=plant1_N, columns= ["unnamed:0", "unnamed1","Temperature [C]", "Air Humidity [RH]", "Ambient Light [lux]", "Moisture [KPa]","Date","impedance_modlus","impedance_phase", "Status"])
    #print(data_plant_1_N) #print to see if the data is correct
    data_plant_1_Nplus1 = pd.DataFrame(data=plant1_Nplus1, columns= ["unnamed:0", "unnamed1","Temperature [C]", "Air Humidity [RH]", "Ambient Light [lux]", "Moisture [KPa]","Date","impedance_modlus","impedance_phase", "Status"])
    # The impedance difference is computed
    data_plant_1_Nplus1['module_diff'] = data_plant_1_Nplus1['impedance_modlus'] - data_plant_1_N['impedance_modlus']
    data_plant_1_Nplus1['phase_diff'] = data_plant_1_Nplus1['
   impedance_phase'] - data_plant_1_N['impedance_phase']
    print(data_plant_1_Nplus1) #print to see if the data is correct
    #the useless column are deleted
    data_plant_1_Nplus1.drop("unnamed1", inplace=True, axis=1)
    print(data_plant_1_Nplus1) #print to see if the data is correct
    #The data with the difference is saved in a new file
    data_plant_1_Nplus1.to_csv(filename_data1_tobesaved)
```

This function is called four times, one for each plant. Note that if the original data frame has N rows, it contains only N-1 rows after the difference computation since nothing can be subtracted from the first row.

3.8.2 Implementation

Once the data files are ready, the usual script *nn_learning.py* is exploited, this time the parameters module_difference and phase_difference shown in table 3.1 are set to one. The neural network is trained twice. Firstly, using only the difference computed, the second time considering the impedance module and phase values as features. The results are saved in the usual file *output_neuralnetwork.txt*. The following results are obtained considering only the differences as features:

```
The network is saved as:
../Neural_network_trained/plant_11_1111100110000_7662.h5
The overall accuracy of this network is: 0.6836445953593662
The train accuracy is: 0.690484612663601
The test accuracy is: 0.6562942008486563
The overall confusion matrix is [[1385  599]
 [ 519 1031]]
The train confusion matrix is [[1123  470]
 [ 405  829]]
The test confusion matrix is [[262 129]
 [114 202]]
```

Considering only the environmental parameters and impedance difference, the overall accuracy reached by the trained model is around 68.3%. The performance has deteriorated concerning the case with impedance module and phase as features. It may be helpful to consider also the impedance module and phase values. Therefore, the number of features becomes nine, and the following performance is reached:

```
The network is saved as:
../Neural_network_trained/plant_11_1111111110000_9662.h5
The overall accuracy of this network is: 0.7560837577815507
The train accuracy is: 0.7569862044570216
The test accuracy is: 0.7524752475247525
The overall confusion matrix is [[1285  699]
```

```
 [ 163 1387]]
The train confusion matrix is [[1024  552]
 [ 135 1116]]
The test confusion matrix is [[261 147]
 [ 28 271]]
```

The accuracy obtained with this network is 75.6%, higher than the previous case since impedance module and phase are considered. However, it is still lower than the case with only impedance and environmental data. From these results, it is possible to conclude that considering the difference as features for the model training is not possible to solve the problem of a different range of impedance values. In the next section, another idea is exploited; a polynomial fitting is applied to the data to neglect the daily variations.

3.9 Polynomial fitting

It is possible to observe the daily variation of impedance described in [7] in figures 3.8 and 3.7. Perhaps these variations are not useful for plant health condition detection. Maybe the plant status depends only on the general impedance trend, and these daily variations are more like additional noise that negatively influences the performance. With the intention to verify if it is really like this, the data are fitted with a polynomial function. And then, the difference is computed again.

3.9.1 Polynomial fitting implementation

In order to fit the data, the script *curve_fitting.py* is exploited. This script is composed by four parts:

1. Import of the plant data;
2. polynomial fitting of the data;
3. plotting of the figures after the fitting;
4. saving the values in new *.csv* files.

Import of the data In this part, the names of the data files and the names of the files to save the results are declared. The data are extracted from the files by using the usual two functions`import_plants` and `prepare_samples`. In this case, the labels and the environmental parameters are not needed. The impedance module and impedance phase are in the 5-th and 6-th columns, respectively.

Polynomial fitting of the data Once the impedance module and phase are available, a custom function `fitting_poly` is used to compute the fitted "y". In particular, it inputs the grade of the polynomial function for the fitting, the x and y to be fitted, and the number of the plant to which the data belongs. It returns two values, x_line which corresponds to x, and the fitting values y_line.

Plots The results obtained in the previous step are used for a graphical view of the functions. In figure 3.10, it is possible to see how a second-degree polynomial fits the impedance module; this curve is not sufficient to represent the impedance variation correctly.

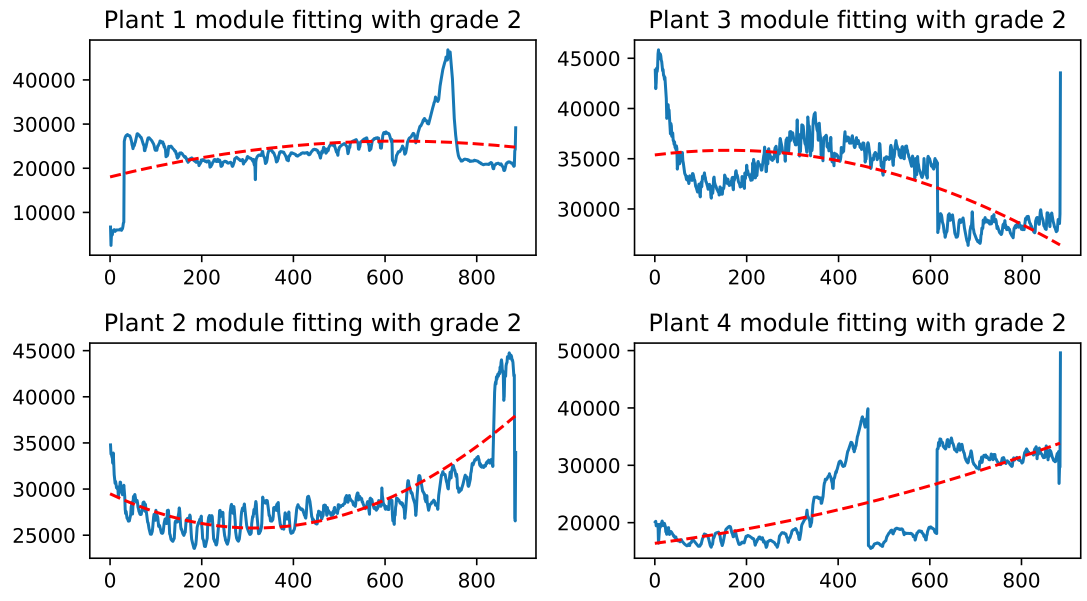


Figure 3.10: Module fitting with a second degree polynomial function. In blue is the module, and in red, the fitting function

Therefore, the polynomial degree has increased up to 8 as shown in figure 3.11. The situation has improved, but the fitting function is not optimal. When the impedance variation is speedy, the polynomial function cannot follow its trend, such as for plants 1 and 4.

In figures 3.12 and 3.13, two 49-th degree polynomial function are used to fitting the impedance module and phase respectively. A polynomial function with this degree is suitable for the data fitting since it can reasonably follow the impedance's general trend but neglect the daily variation.

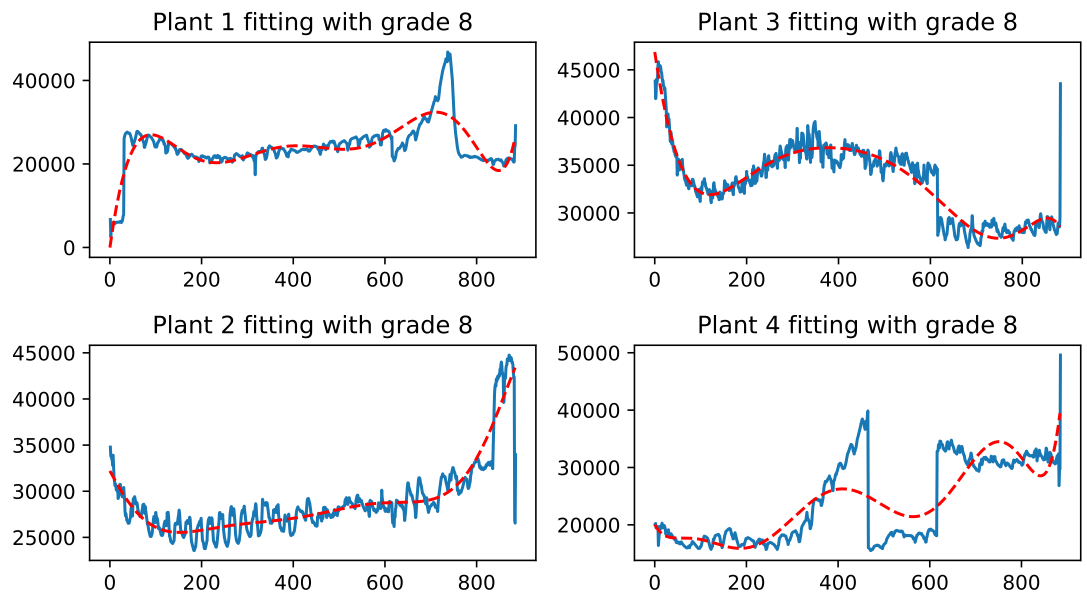


Figure 3.11: Module fitting with a 8-th degree polynomial function. In blue is the module, and in red, the fitting function

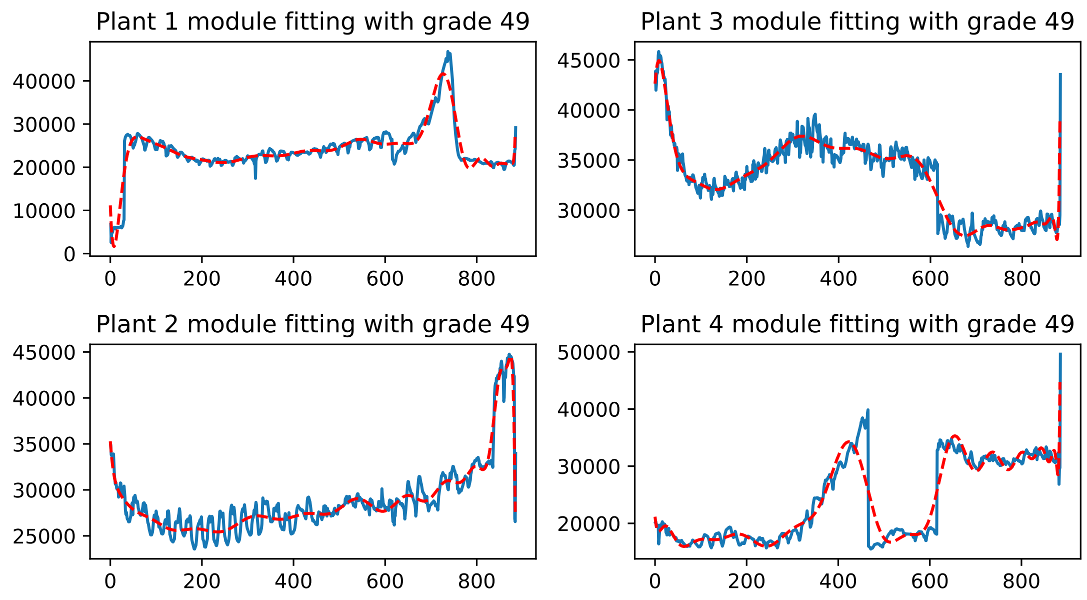


Figure 3.12: Module fitting with a 49-th degree polynomial function. In blue is the module, and in red, the fitting function

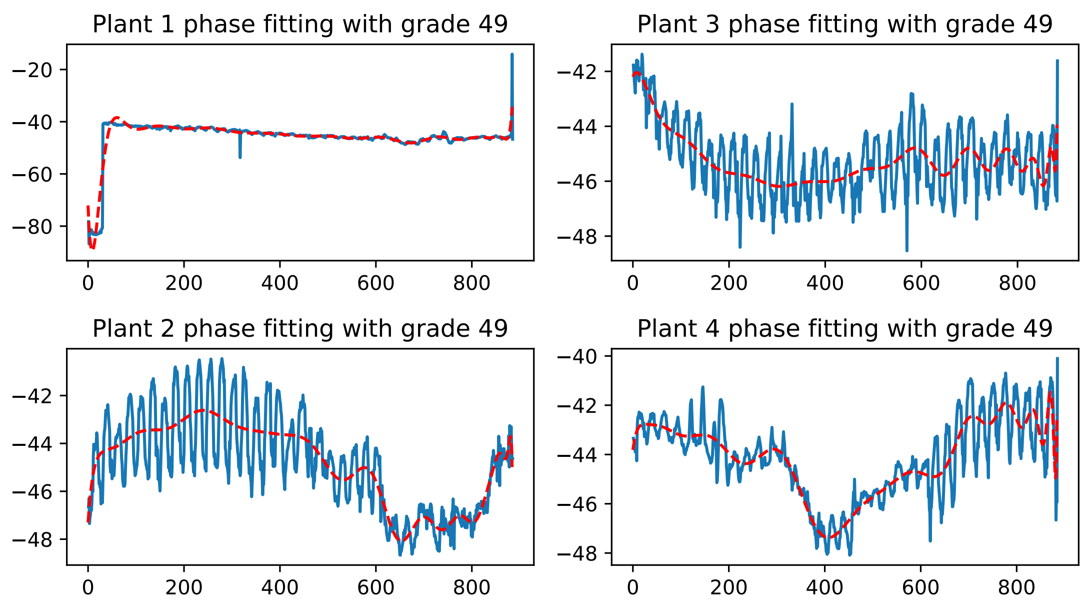


Figure 3.13: Phase fitting with a 49-th degree polynomial function. In blue is the phase, and in red, the fitting function

Saving of the data Finally, the fitting data is saved as additional columns to the *.csv* files. With this purpose a custom function `add_fit` is exploited. It inputs the Pandas data frame of the plant data, the name of the file where the new data will be saved, the module, and the phase after the fitting. It simply adds two new columns module_fitting and phase_fitting containing the values obtained with the fitting to this Dataframe. And then, the data are saved into a *.csv* file with the name passed to the function.

Listing 3.8: Function used to add the data to the files

```python
def add_fit(data_plant1, filename_data1_tobesaved, y_plant_module,
    y_plant_phase):
    data_plant_1_N = data_plant1
    data_plant_1_N['module_fitting'] = y_plant_module
    data_plant_1_N['phase_fitting'] = y_plant_phase
    print(data_plant_1_N) #print to see if the data is correct
    #An useless column is deleted
    data_plant_1_N.drop("unnamed1", inplace=True, axis=1)
    print(data_plant_1_N) #print to see if the data is correct
    #The data after the fitting is saved in a new file
    data_plant_1_N.to_csv(filename_data1_tobesaved)
```

Once the new data files are available, another script, *diff_considering_fitting.py*, is exploited in order to compute the difference of the module and phase after the fitting. This script is almost equal to the one described in section 3.8.1.

3.9.2 Implementation

Now, it is possible to train neural networks considering the fitting values and their difference as features. The usual script *nn_training.py* is used, this time the parameters set to one are module_fit, phase_fit, module_diff_fit and phase_diff_fit.

Firstly, only module_fit, phase_fit and the environmental parameters are set to one, the following results are obtained:

```
The network is saved as:
../Neural_network_trained/plant_11_1111100001100_7662.h5
The overall accuracy of this network is: 0.7940011318619129
The train accuracy is: 0.7955429784223559
The test accuracy is: 0.7878359264497878
The overall confusion matrix is [[1469  515]
 [ 213 1337]]
The train confusion matrix is [[1187  411]
 [ 167 1062]]
The test confusion matrix is [[282 104]
 [ 46 275]]
```

The overall accuracy is 79.4%, compared to the first implementation described in section 3.4, it is almost the same. It means that the fitting is not much helpful for improving the performance. It is just more effort for the same performance. After that, a second attempt is done setting also the parameters module_diff_fit and phase_diff_fit to one, the performance is the following:

```
The network is saved as:
../Neural_network_trained/plant_11_1111100001111_9662.h5
The overall accuracy of this network is: 0.7439162422184493
The train accuracy is: 0.7467279801910152
The test accuracy is: 0.7326732673267327
The overall confusion matrix is [[1431  553]
 [ 352 1198]]
The train confusion matrix is [[1145  442]
 [ 274  966]]
The test confusion matrix is [[286 111]
 [ 78 232]]
```

In this case, the performance is even worse.

Although the performance is similar to the implementation with standard impedance, this method is not particularly useful because using this algorithm on-field will be a problem. Suppose that the detection of the status of a plant is required. The data given to the model is the data after a fitting, but the number of samples used for the fitting should be defined. It means that the prediction is not possible before collecting a certain number of samples. Moreover, every time that new data is sampled, the fitting values should be computed again. Unfortunately, it is possible to conclude that this method does not help improve plant health status detection's accuracy. Furthermore, it will also increase the computational cost due to the data preparation in both the training phase and the prediction moment.

3.10 SVM approach

As explained in section 2.2.7, Support Vector Machine is a machine learning algorithm that exploits separative hyperplanes to solve classification problems. Perhaps this algorithm may improve the performance since the number of data may not be enough to train a neural network properly.

3.10.1 Implementation

The SVM is implemented in Python by exploiting the open-source library **Scikit-learn**. In particular, two SVM are implemented, the first one using the linear kernel and the second one using the Radial Basis Function kernel. A Python script, *svm_train.py*, in the folder **SVM_training**, is exploited. The description is divided into four parts:

1. some parameters are set in order to choose the data files to be imported and the features to be considered during the training;
2. preparation of the data imported from the files for the training. The dataset is divided into the training set and testing set;
3. choose the model to be trained between linear and RBF, creation and training of the model;
4. the network trained is saved in a *.sav* format file. Then the accuracy is computed and saved, with the model information, in a *.txt* output file.

Parameters setting Firstly, the usual parameters `set_by_hand`, `saturation_moisture` and the features to be used for the training are set. Then the data file names are obtained.

Data preparation The next thing to do is to import the plants' data and prepare them for the training. The usual custom functions `import_plants` and `prepare_samples` are exploited. Then the data are shuffled, normalized, and divided into training and testing sets.

Creation and training The model is implemented using the **Scikit-learn** library. In particular the `svm.SVC` (Support Vector Classification) method is used. The kernel should be chosen among linear, RBF, sigmoid, poly, and precomputed. In this implementation, only linear and RBF kernels will be considered:

Listing 3.9: Parameters setting

```python
model_type = 0
if model_type == 0:
    model_type_name = "linear"
elif model_type == 1:
    model_type_name = "rbf"

# Creation of the model to be trained
svm_train = svm.SVC(kernel= model_type_name)
# Training of the model
svm_train.fit(plant_samples_train , plant_labels_train)
```

Saving model and results Once the model is trained, the accuracy can be computed. After that, both the model and the performance are saved. The trained SVM is saved in a *.sav* format file in the folder **SVM_trained**, with a filename like *plant_*a_0a_1*_*$a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}a_{12}a_{13}$*_*$a_{14}$*_*$a_{15}$*.sav*. The first two coefficients refer to the parameters set_by_hand and saturation_moisture, the coefficients from a_2 to a_{13} indicate the features used during the training phase, a_{14} is the number of features considered, and a_{15} shows which kernel is used,linear kernel or an rbf one. At the end, the results are saved in the folder **output_files** in the file *output_svm.txt*. The results obtained with a linear kernel are:

```
The network has saved as: ../SVM_trained/plant_11_1111111000000_7_linear.sav
The overall accuracy of this network is: 0.5645161290322581
The train accuracy is: 0.561726211531659
The test accuracy is: 0.5756718528995757
The overall confusion matrix is[[1474  510]
 [1029  521]]
The train confusion matrix is[[1170  409]
 [ 830  418]]
The test confusion matrix is[[304 101]
 [199 103]]
```

With a linear kernel, only 56.4% of accuracy is reached. This poor performance can be expected since a linear function cannot describe the relationship between the inputs and the output. Therefore, an RBF kernel is used. For RBF kernels, two parameters must be taken into consideration: C (equation 2.15) and gamma (equation 2.17). The first parameter defines the so-called regularization; low C means that the decision surface is smooth while a high value of C provides a more precise classification for the training examples, which means the possibility of overfitting. The second one determines how much influence a single training example has. For this implementation, standard values are used for both the parameters [35]. The following results are obtained with an RBF kernel:

```
The network has saved as: ../SVM_trained/plant_11_1111111000000_7_rbf.sav
The overall accuracy of this network is: 0.7880588568194681
The train accuracy is: 0.7905907322249734
The test accuracy is: 0.7779349363507779
The overall confusion matrix is[[1413  571]
 [ 178 1372]]
The train confusion matrix is[[1142  452]
 [ 140 1093]]
The test confusion matrix is[[271 119]
 [ 38 279]]
```

The accuracy reached with RBF is 78.8%, now it is possible to train several models with different combinations of features. A summary is presented in table 3.3.

model name	overall accuracy	training accuracy	testing accuracy
plant_11_1111111000000_7_linear.sav	56.4%	56.2%	57.6%
plant_11_1111111000000_7_rbf.sav	78.8%	79.1 %	77.8%
plant_11_1111100000000_5_rbf.sav	66.4%	66.6%	65.6%
plant_11_1111100110000_7_rbf.sav	67.8%	67.7%	68.2%
plant_11_1111111110000_9_rbf.sav	78.1%	78.0%	78.6%
plant_11_1110100000000_4_rbf.sav	59.9%	60.8%	56.6%
plant_11_1111100001100_7_rbf.sav	77.9%	77.8%	78.5%
plant_11_1111100001111_9_rbf.sav	77.9%	77.6%	78.4%

Table 3.3: Several versions of SVM are trained, considering different combination of features

The accuracy of the model trained with the impedance values is almost equal to that obtained with the neural network. Around 80% of accuracy decreases to 66% if the impedance module and phase are not used during the training phase, and the performance is even worse when soil moisture is removed. When only impedance difference and the environmental parameters are used for the training, only 67.8% of accuracy is reached. Furthermore, in the cases where both impedance and impedance difference are used, before and after the fitting, the accuracy is slightly lower than 80%. As explained before, it will not be convenient to use impedance difference and the data after the fitting since they increase the computational cost in the data processing without improving the performance.

It is possible to conclude that the SVM has not led to better performance of neural networks. However, it was worth implementing Support Vector Machine models since they have further confirmed the importance of using the impedance values for the training and the results about the impedance difference and polynomial fitting found before.

3.11 Neural Network, samples in time

Till now, the impedance values are exploited in order to predict the plants' status. It means that each environmental parameter and the impedance value were considered once. Therefore what influences the output prediction is the actual value of the impedance. In the article [7], the authors have demonstrated that the plants have two different behaviors when it is watered and when it is under water-stress condition. The new idea is to exploit the different variations of the impedance in these two conditions. For this purpose, more samples of impedance should be considered during the training. It is like that there is a time window that considers all the samples in this period, and then the window slides one sample at a time. Therefore, the environmental values are considered once; the impedance, the impedance difference, and the impedance after the polynomial fitting can be considered more times.

Firstly, the performance of the neural network is evaluated. Then, how these new features will affect the prediction of future data will be observed. Finally, the neural network will be trained using only three plants and tested on the fourth one.

3.11.1 Implementation

For the first part the Python script *nn_training_intime.py* in the folder **Neural_network_training_intime** is exploited. The description can be divided into four parts, which are very similar to the previous implementation. The only difference is that in the data preparation phase, more columns are added, where each column contains the past samples of a certain feature. In particular:

1. the environmental and impedance parameters are set in order to decide which values will be used during the training phase and how many times these values will be taken into consideration;
2. preparation of the data imported from the files for the training. Then, the dataset is divided into the training and test sets;
3. creation and training of the neural network;
4. the network trained is saved in a *.h5* format file. Then the accuracy is computed and saved, with the model information, in a *.txt* output file.

Parameters setting The usual parameters set_by_hand and soil_moisture are set to one. Now, the environmental parameters temperature, airhumidity, ambientlight, moisture, date_time, and the impedance values impedance_module, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit could be an any integer value. This value represents how many times these features will be used during the training phase. For example, if impedance_module is set to 3, then it is used as features three times, which means the n-th sample, the sample n-1 of the previous hour and the sample n-2 of two hours ago. After that, the file names are imported.

Data preparation The input size of the neural network is computed; it is just the sum of all the parameters described before. Then, the data are imported as Pandas data frame using the custom function `import_plants`. In order to obtain the new features a custom function `add_feature_intime` is exploited. This function takes four inputs: the plant data, the value of the parameter to be added, the name of the parameter to be added, and the list of the features. It generates two outputs: the plant data with new features and the updated list of the features.

Listing 3.10: Function add_feature_intime

```python
def add_feature_intime(data_plant, feature_value, feature_name,
    list_of_features):
    feature_name_added = feature_name
    if feature_value == 0:
        data_plant_np = data_plant.to_numpy()
        data_plant = pd.DataFrame(data=data_plant_np, columns=[
    list_of_features])
        data_plant.drop(feature_name_added, inplace=True, axis=1)
        list_of_features.remove(feature_name_added)
    elif feature_value > 1:
        for i in range(1,feature_value):
            #print("Cycle number: " + str(i) +'\n')
            feature_column = data_plant[[feature_name_added]]
            feature_column_np = feature_column.to_numpy()
            feature_column_np = np.delete(feature_column_np, -1 , 0)#
    delete the last row
            feature_column_np = np.insert(feature_column_np,0,0)
            data_plant_np = data_plant.to_numpy()
            data_plant = pd.DataFrame(data=data_plant_np, columns= [
    list_of_features])
            feature_name_added = feature_name + '_minus' + str(i)
            list_of_features.append(feature_name_added)
            data_plant[feature_name_added] = feature_column_np
    return data_plant, list_of_features
```

If feature_value is zero, it means that the feature indicated by feature_name will not be used during the training. Therefore, it is eliminated from the data frame, and the corresponding name is removed from the feature list. If feature_value is one, then nothing is done since it is already present once in the data frame and the feature list. If this parameter is bigger than one, a *for* cycle is exploited in order to add the new columns.

Considering the past samples means that the rows of the corresponding features should be shifted by one and then added as a new column to the data frame. Enumerate the rows from 1 to N. The new column is formed by the rows from 1 to N-1, the second new column from 1 to N-2, and so on. However, in this way, the new columns have fewer rows. Zeros fill the empty rows. A graphical view can be seen in figure 3.14, in this example feature_value is set to 4. At the same time, the feature list is updated, adding the name of the past features.

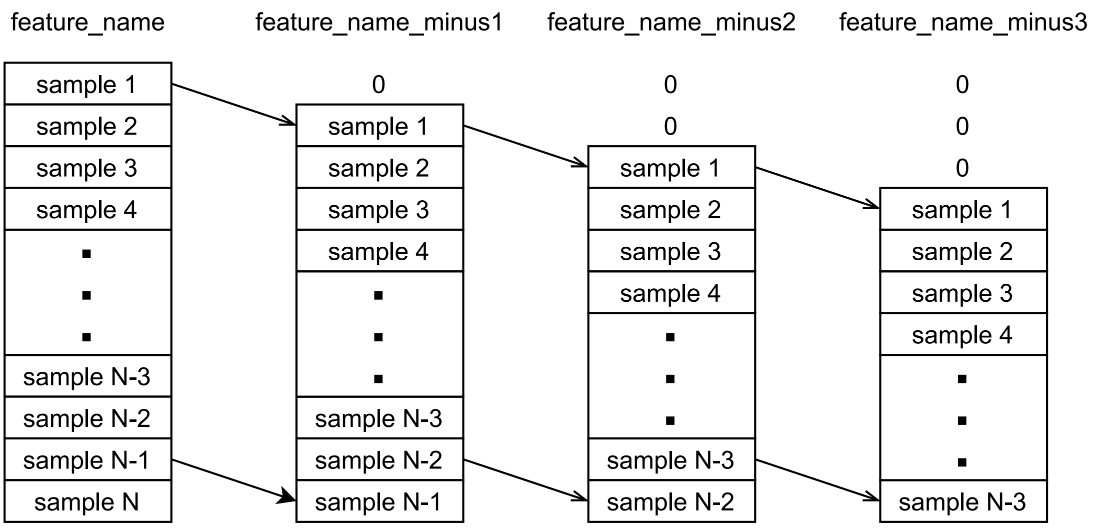


Figure 3.14: Considering a parameter more times. Adding new columns with past samples of the parameter.

This function can be called also for adding past samples of the environmental parameters. But in this case only the impedance values are considered more times. Therefore, this function is called as much as is necessary. Notice that now some rows are filled by zeros, these zeros are meaningless for the training, so they have to be removed. The number of rows to be removed corresponds to maximum value among the parameters temperature, airhumidity, ambientlight, moisture, date_time, impedance_module, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit minus one. Now, the data are ready to be shuffled, normalized and divided into training and testing sets.

Creation and training A neural network with two hidden layers is used. Since now the number of features is much higher than seven, more nodes are used in each layer. However, in order to have a comparison between the neural networks trained with different number of features, a fixed number of nodes is used, for instance, twenty-five and fifteen:

Listing 3.11: Creation of the model

```
 # Creation of the neural network
 nn_layers =[25,15,2]
 model = Sequential()
 model.add(Dense(nn_layers[0], activation='relu', input_shape=(
    feature_size,)))
 model.add(Dense(nn_layers[1],activation='relu'))
 model.add(Dense(nn_layers[2], activation='softmax'))
#Training of the model
model.compile(optimizer=Adam(learning_rate = 0.0005), loss='
    sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x=plant_samples_train, y=plant_labels_train,
    validation_split=0.1, batch_size=10, epochs=300, shuffle=True,
    verbose=2)
```

Saving the model and the results The computed accuracy is saved in the file *output_neuralnetwork_intime.txt* in the usual folder **output_files**. While the trained model is saved in the folder **Neural_network_trained_intime** with the following name:

Listing 3.12: Model filename

```
filename_neural_network = '../Neural_network_trained_intime/plant_'+
    str(set_by_hand) + str(saturation_moisture) + '_' + str(
    temperature) + str(airhumidity) + str(ambientlight) + str(moisture
    ) + str(date_time) + '_' + str(impedance_module) + '_' + str(
    impedance_phase) + '_' + str(module_difference) + '_' + str(
    phase_difference) + '_' + str(module_fit) + '_' + str(phase_fit) +
     '_' + str(module_diff_fit) +'_' + str(phase_diff_fit) + '_' +
    str(feature_size) + '_' + str(nn_layers[0]) + '_' + str(nn_layers
    [1]) + '_' + str(nn_layers[2]) + '.h5'
```

Several neural networks have been trained considering different numbers of samples in time. For a better comparison, the results obtained with the different neural networks are summarized in table 3.4.

It is possible to observe how the accuracy is slightly increased in the case with only one impedance module and one impedance phase concerning the first implementation with six nodes in each hidden layer. This improvement is because more nodes are used in this case. Then, it is possible to notice that both the training accuracy and testing accuracy increase with the number of samples considered. When 48 samples, which means 48 hours, are considered, the training accuracy reaches 99.7%, which is an excellent result. Implementing the neural networks in

model name	training accuracy	testing accuracy
plant_11_11111_1_1_0_0_0_0_0_0_53_25_15_2.h5	94.0%	92.6%
plant_11_11111_6_6_0_0_0_0_0_0_53_25_15_2.h5	95.6%	94.0%
plant_11_11111_12_12_0_0_0_0_0_0_53_25_15_2.h5	96.7%	95.0%
plant_11_11111_24_24_0_0_0_0_0_0_53_25_15_2.h5	97.9%	95.8%
plant_11_11111_36_36_0_0_0_0_0_0_53_25_15_2.h5	99.3%	98.2%
plant_11_11111_48_48_0_0_0_0_0_0_53_25_15_2.h5	99.7%	99.4%

Table 3.4: Several neural network are trained considering different numbers of past samples

this way leads to excellent results. This is another step toward a successful system for detecting plant health status. However, the testing accuracy is always smaller than the training accuracy; this brings the possibility of overfitting. To prove that, the trained models, as usual, are used for testing the future data.

3.11.2 Future predictions implementation

In this step, the Python script in the folder **Neural_network_training_intime** is exploited. As usual, the description can be divided into four parts: parameters setting, import and data preparation, import of the neural network to be tested and computation of the accuracy, and results saving.

Parameters setting The parameters which decide how many times a value should be considered are set. Then, these parameters are used to get the data frame with the added features. They also define the name of the neural network to be imported. After that, the filenames are defined. Two groups of filenames are needed; the first group is regarding the data used for the training, the old data. The second one is the filenames of the data to be saved.

Data preparation After the plants' data are imported as Pandas data frame, the function `add_feature_intime` is exploited several times in order to add the features to the data frame. This procedure is done for both the old and new data. The old data is used to properly normalize the new data as shown in equation 3.6.

Import of the model and accuracy computation Once the data set is ready; the model is loaded by using the name defined by the parameters set before. Then, the testing accuracy on the new data is computed and saved in a *.txt* format file in the folder **output_files**, called *output_future_prediction_intime.txt.*

Now, the six models trained before are used for testing the future data. In order to have a better comparison, the results are plotted with bar graphs, shown in figure 3.15. The number on the x-axis corresponds to the version of the trained neural networks:

1. model trained with impedance module and phase considered 1 time;

2. model trained with impedance module and phase considered 6 times;

3. model trained with impedance module and phase considered 12 times;

4. model trained with impedance module and phase considered 24 times;

5. model trained with impedance module and phase considered 36 times;

6. model trained with impedance module and phase considered 48 times;

The accuracy is computed separately on the four plants. It is possible to notice that the performance is relatively good for the first four implementations. While in the case of 36 and 48 samples in time, the accuracy of plants 1 and 3 drops significantly.

As expected, the excellent performance of 99.7% and 99.3% of accuracy is mainly due to overfitting. However, the neural networks trained with 12 and 24 samples are already an outstanding achievement since they also have excellent performance on the future data. Therefore, there is no need to implement other neural networks considering the impedance difference and the fitted values. However, other implementations are necessary to see if it is possible to use the neural network on unseen tobacco plants. These implementations are explained in the next section.

An interesting to do is to plot the correct and wrong predictions. However, it is impossible to use all the features since they are much greater than three. Considering the case of 24 samples, it is possible to plot the testing predictions by choosing three of the features as the three-axis. In figure 3.17, impedance module, soil moisture and time are considered. Normalized values are used, and in figure 3.16, the confusion matrix is shown.

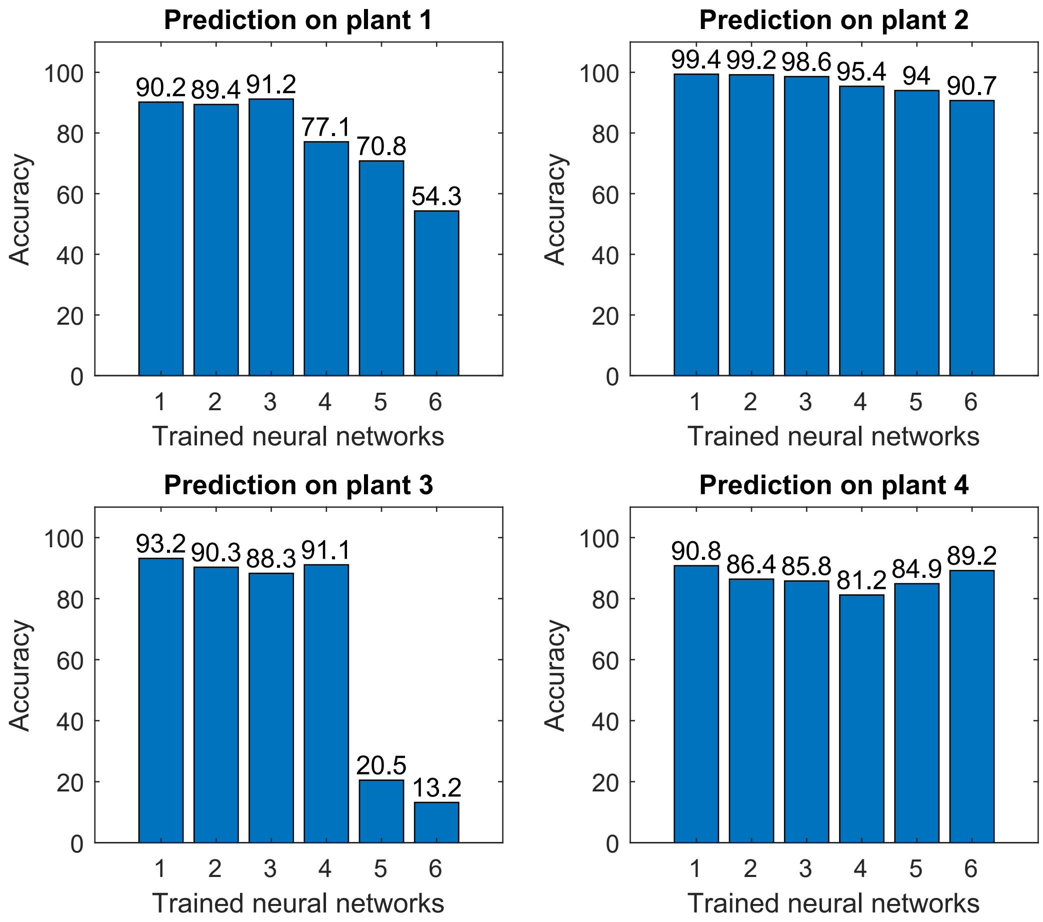


Figure 3.15: Accuracy of future predictions

3.11.3 Three plants training

For this part the Python script *Three_plants_training.py* in the folder **Neural_network_training_intime** is exploited. This script implements four models taking as training set three plants out of four. And then, predict the health status of the fourth one, which is the one not used during the training phase. The description of this script can be divided into four parts:

1. parameters setting and data import;
2. data preparation;
3. models creation, training, and saving;
4. accuracy computation and results saving.

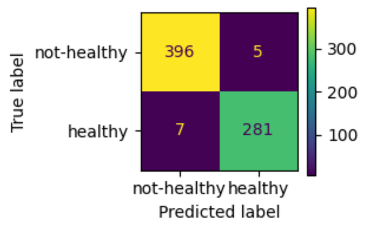


Figure 3.16: Confusion matrix of the testing set

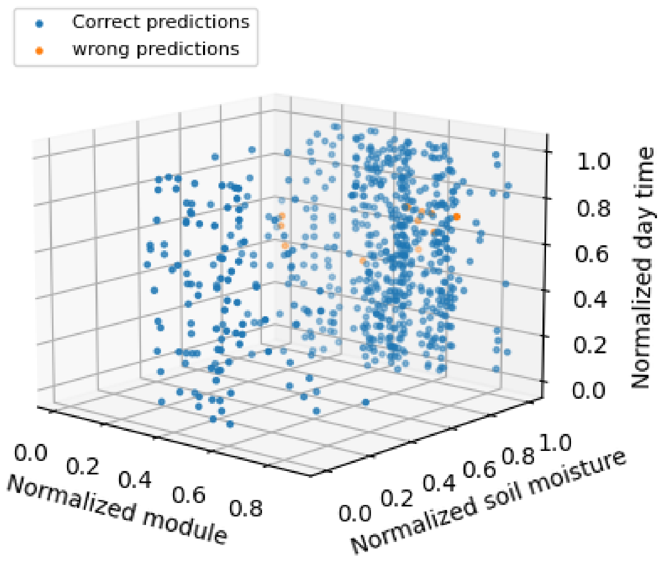


Figure 3.17: Predictions of the testing set

Parameters setting As usual, the first step to do is to set some parameters that import correctly the input files and decide the features to be used for the training. The two parameters set_by_hand and saturation_moisture are left to one. Also the environmental parameters temperature, airhumidity, ambientlight, moisture and date_time are set to 1. What are changed in the different implementations are the values of **impedance_module**, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit. Each of them can be any integer value. Then, the correct filenames are imported.

Data preparation The data are imported from the files. Then, the parameters defined before are used to add the correct past samples and to remove the unused features by using the custom function `add_feature_intime`. After that, four training sets are obtained, choosing three plants out of four. The four training sets are formed by plants 1, 2, and 3; plants 1, 2, and 4; plants 1, 3, and 4; and plants 2, 3, and 4. These data are shuffled and normalized:

Listing 3.13: Function add_feature_intime

```
# Concatenation of the data, in order to obtain 4 training sets
plant_samples_123 = np.concatenate((plant_1_samples, plant_2_samples,
     plant_3_samples))
plant_labels_123 = np.concatenate((plant_1_labels, plant_2_labels,
    plant_3_labels))

plant_samples_124 = np.concatenate((plant_1_samples, plant_2_samples,
     plant_4_samples))
```

```
plant_labels_124 = np.concatenate((plant_1_labels, plant_2_labels,
    plant_4_labels))

plant_samples_134 = np.concatenate((plant_1_samples, plant_3_samples,
     plant_4_samples))
plant_labels_134 = np.concatenate((plant_1_labels, plant_3_labels,
    plant_4_labels))

plant_samples_234 = np.concatenate((plant_2_samples, plant_3_samples,
     plant_4_samples))
plant_labels_234 = np.concatenate((plant_2_labels, plant_3_labels,
    plant_4_labels))

#Shuffle of the 4 data sets
plant_labels_123, plant_samples_123 = shuffle(plant_labels_123,
    plant_samples_123)
plant_labels_124, plant_samples_124 = shuffle(plant_labels_124,
    plant_samples_124)
plant_labels_134, plant_samples_134 = shuffle(plant_labels_134,
    plant_samples_134)
plant_labels_234, plant_samples_234 = shuffle(plant_labels_234,
    plant_samples_234)

#Normalization of the samples
scaler = MinMaxScaler(feature_range=(0,1)) #the data are normalized
    to the range 0,1
scaled_plant_samples_123 = scaler.fit_transform(plant_samples_123)
scaled_plant_samples_124 = scaler.fit_transform(plant_samples_124)
scaled_plant_samples_134 = scaler.fit_transform(plant_samples_134)
scaled_plant_samples_234 = scaler.fit_transform(plant_samples_234)
```

Model creation Four models are created, each one is trained with one of the four datasets obtained before. All the neural networks are composed of two hidden layers, the first one with 25 nodes and the second one with 15 neurons. After the models have been trained, they are saved in *.h5* format files in the folder **Neural_network_trained_intime**. The name used for the model is like $plantxyz\_a_0\_a_1\_a_2\_a_3\_a_4\_a_5\_a_6\_a_7\_a_8\_a_9\_a_{10}\_a_{11}.h5$, the meaning of the coefficients are explained in table 3.5.

Accuracy computation Once the models have been trained; they are used to predict the health status of the plant that is not used for the training. For instance, if plants 1,2 and 3 are used to train the neural network, the model is used to predict the status of plant 4, if plants 1, 2, and 4 are used to train the neural network, the model is used to predict the status of plant 3 and so on. The data of the testing plant should be properly normalized using the data of the other plants, which are

Character	Corresponding parameter	value	meaning
xyz	-	1, 2, 3, 4	they indicate which three plants are used during the training phase
a_0	impedance_module	positive integer number	these parameters indicate which are used as features for the training, and how many times
a_1	impedance_phase	positive integer number	
a_2	module_difference	positive integer number	
a_3	phase_difference	positive integer number	
a_4	module_fit	positive integer number	
a_5	phase_fit	positive integer number	
a_6	module_diff_fit	positive integer number	
a_7	phase_diff_fit	positive integer number	
a_8, a_9, a_{10}, a_{11}	nn_layers[ ]	positive integer number	these numbers represent the number of nodes in each layer, input layer, hidden layers and output layer

Table 3.5: $plantxyz\_a_0\_a_1\_a_2\_a_3\_a_4\_a_5\_a_6\_a_7\_a_8\_a_9\_a_{10}\_a_{11}.h5$, explanation of this filename

used during the training. After that, both the training and testing accuracy is computed, and saved in the file *output_neuralnetwork_intime_threeplants.txt* in the folder **output_files**.

In sections 3.8 and 3.9, the impedance difference and polynomial fitting are introduced. They are used to try to improve the neural networks' performance, but without success. Now, these values are used to see how they affect the performace of the networks trained with three plants. Several neural networks are implemented by changing the parameters impedance_module, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit. The following neural networks are trained, when the parameters are not mentioned, they are set to zero, remember that the environmental parameters are always set to one:

- impedance_module = 1, impedance_phase = 1, figure 3.18;
- impedance_module = 6, impedance_phase = 6, figure 3.19;
- impedance_module = 12, impedance_phase = 12, figure 3.20;
- impedance_module = 24, impedance_phase = 24, figure 3.21;
- impedance_module = 36, impedance_phase = 36, figure 3.22;
- module_difference = 1, phase_difference = 1, figure 3.23;
- impedance_module = 1, impedance_phase = 1, module_difference = 1, phase_difference = 1, figure 3.24;
- impedance_module = 6, impedance_phase = 6, module_difference = 6, phase_difference = 6, figure 3.25;
- impedance_module = 12, impedance_phase = 12, module_difference = 12, phase_difference = 12, figure 3.26;

Since there are many implementations, the performance is shown graphically, in particular with bar graphs. Each bar graph represents a version of the trained model; the model's name can be found in the caption. There are four groups of bars, each group composed of two bars. Each group represents the accuracy of the model trained with a different group of plants. In blue is the training accuracy, and in red is the testing accuracy. For instance, the first group is the accuracy of the model obtained with plants 1, 2, and 3, and the testing is done on plant 4 and so on.

From figures 3.18, 3.19, 3.20, 3.21 and 3.22, it is possible to notice that increasing the number of past samples used during the training phase, the training accuracy increases. However, the testing accuracy remains below 50% in almost all cases. As

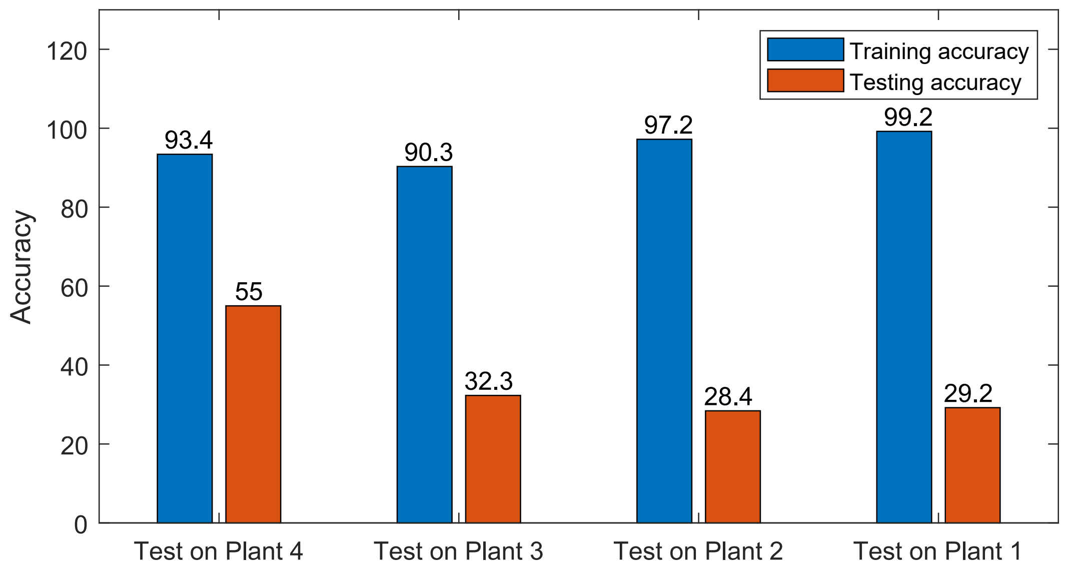


Figure 3.18: Accuracy of *plantxyz*_1_1_0_0_0_0_0_0_7_25_15_2.*h*5

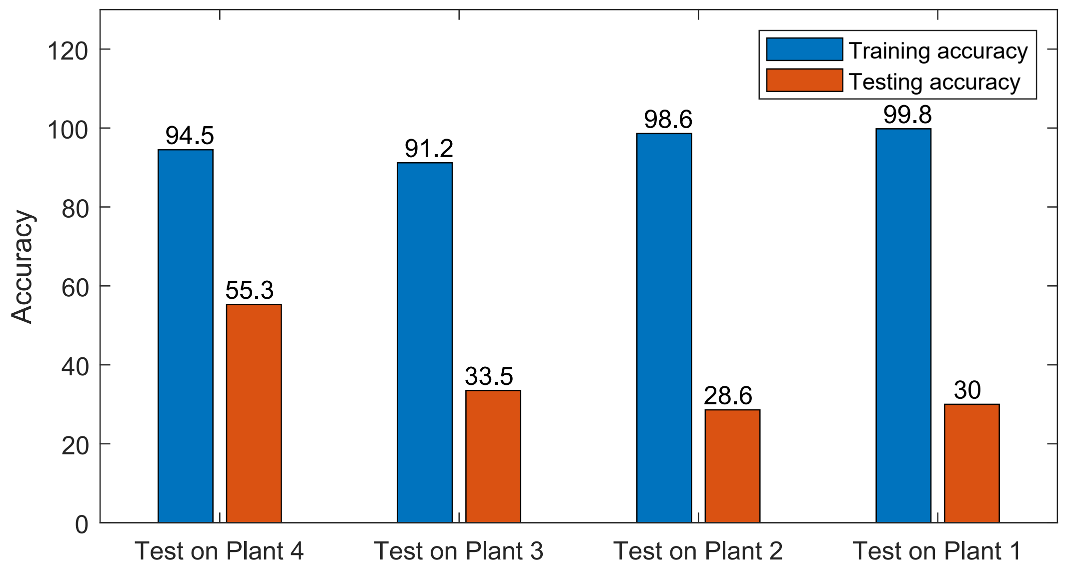


Figure 3.19: Accuracy of *plantxyz*_6_6_0_0_0_0_0_0_17_25_15_2.*h*5

explained before, this is mainly due to the different impedance ranges of the plants. Therefore, the impedance difference has been introduced in order to see if there may be any improvements. Unfortunately, the impedance difference did not improve the accuracy of the neural network. However, the impedance difference is not used in the case of three plants for the training yet. The performance of the neural network

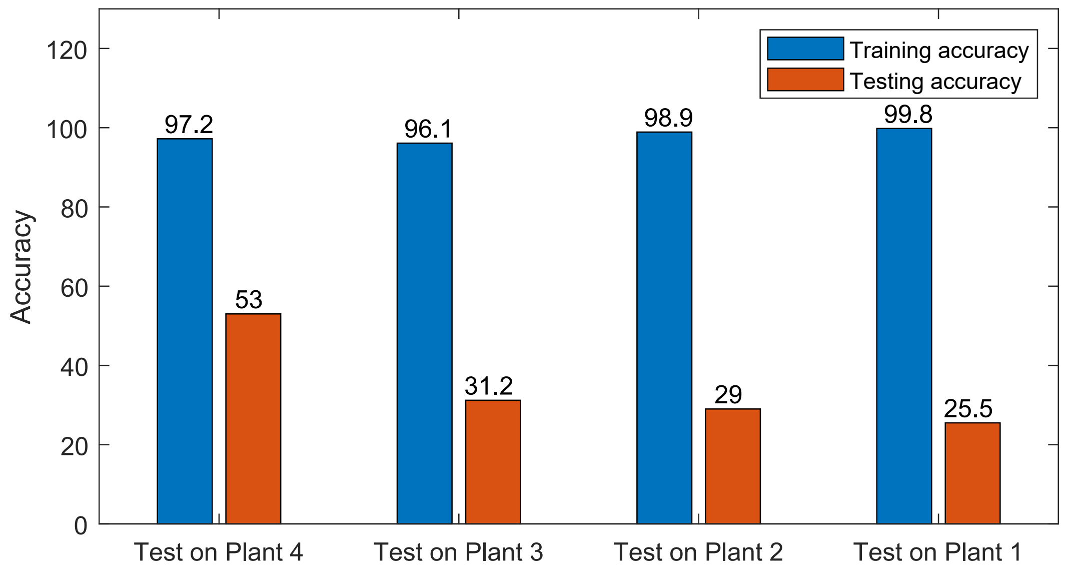


Figure 3.20: Accuracy of *plantxyz*_12_12_0_0_0_0_0_0_29_25_15_2.*h*5

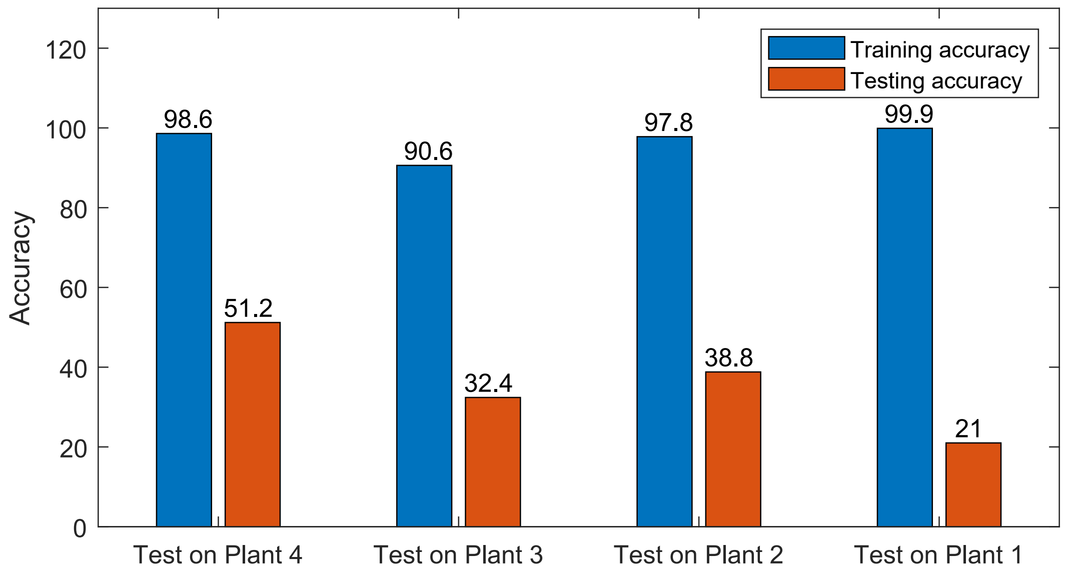


Figure 3.21: Accuracy of *plantxyz*_24_24_0_0_0_0_0_0_53_25_15_2.*h*5

with trained with only the difference values is shown in figure 3.23. As expected, the training accuracy is relatively low and so is the testing accuracy. Therefore also the impedance module and phase are taken into consideration during the training phase. The results are shown in figure 3.24. The performance is slightly better than previous implementations but without significant improvements. The number of

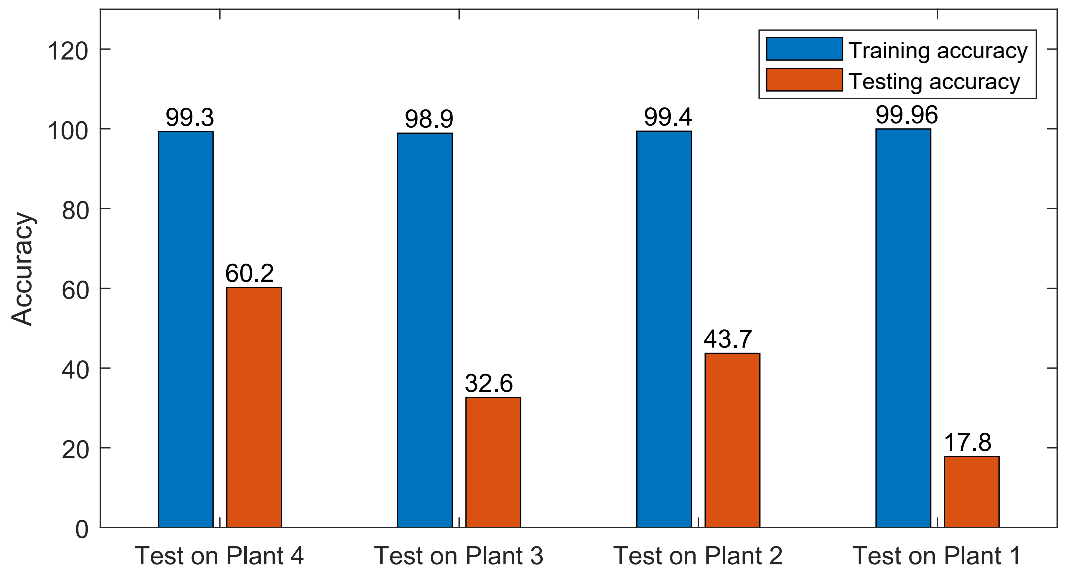


Figure 3.22: Accuracy of $plantxyz\_36\_36\_0\_0\_0\_0\_0\_0\_77\_25\_15\_2.h5$

past samples is increased up to 6 and 12. The results are shown in figures 3.25 and 3.26. Unlucky, also in these cases, the performance is terrible. The last attempt can be made considering the values after the polynomial fitting. However, remember that the polynomial fitting has been introduced to filter out the daily variations of the plant impedance, obtaining a general trend. On the other hand, the past samples are introduced to exploit the daily variations; these two methods conflict. Therefore, it is not very meaningful to implement a neural network applying both methods.

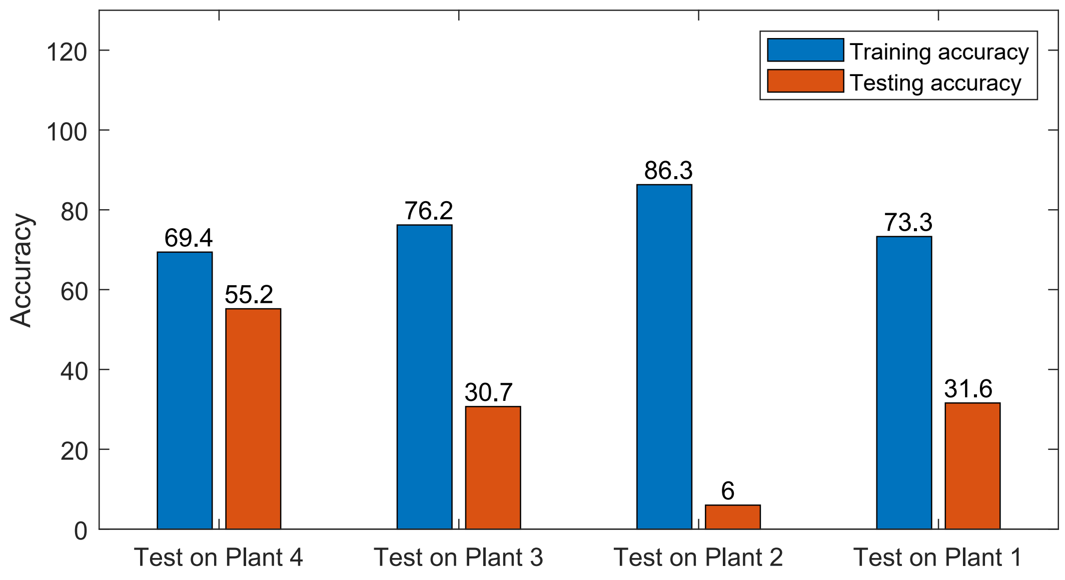


Figure 3.23: Accuracy of *plantxyz*_0_0_1_1_0_0_0_0_7_25_15_2.*h*5

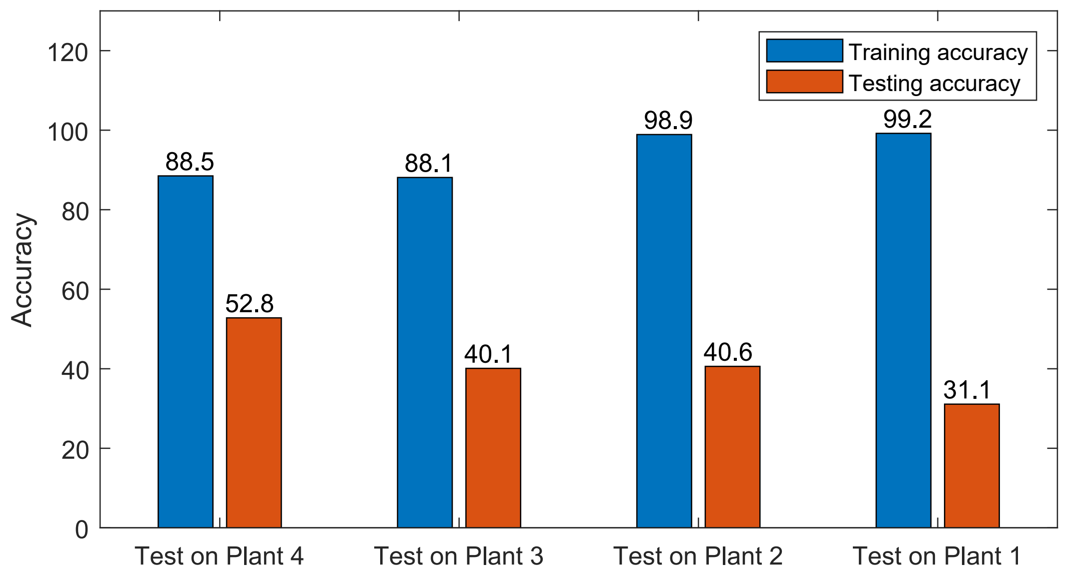


Figure 3.24: Accuracy of *plantxyz*_1_1_1_1_0_0_0_0_7_25_15_2.*h*5

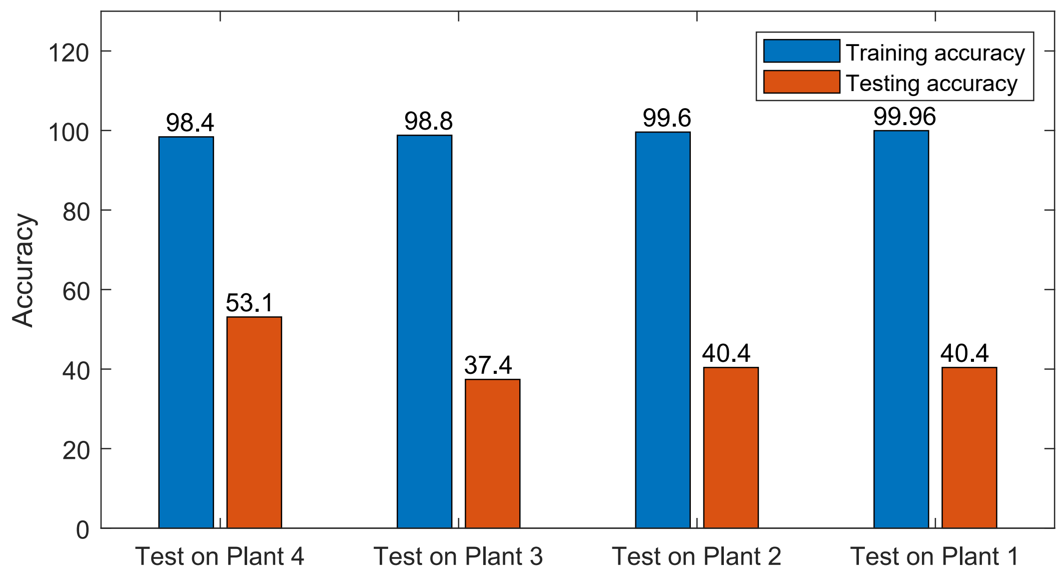


Figure 3.25: Accuracy of $plantxyz\_6\_6\_6\_6\_0\_0\_0\_0\_7\_25\_15\_2.h5$

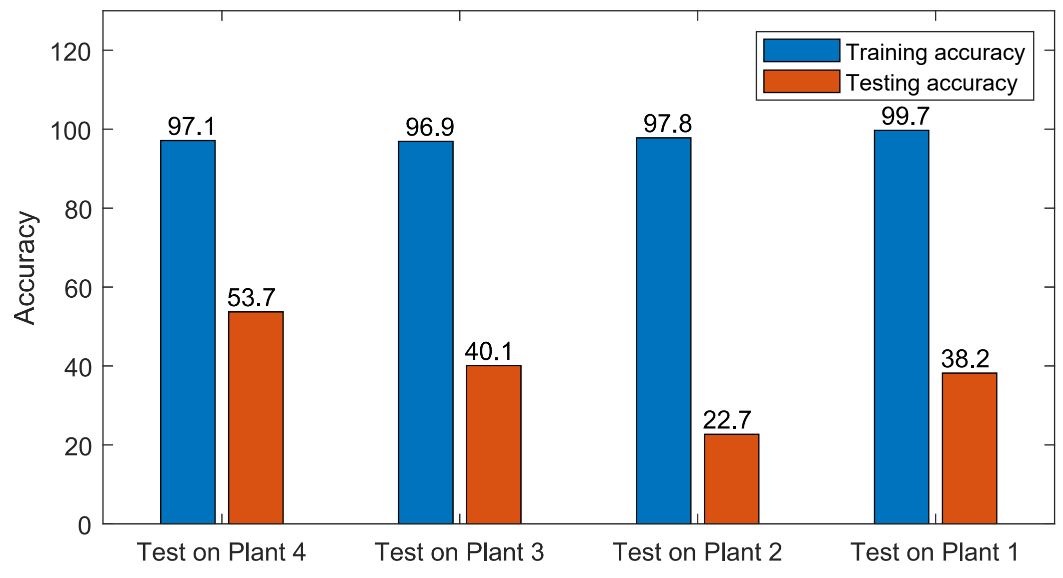


Figure 3.26: Accuracy of $plantxyz\_12\_12\_12\_12\_0\_0\_0\_0\_7\_25\_15\_2.h5$

Chapter 4

Conclusion and Future Perspective

In this dissertation, a way of detecting tobacco plants' health status is explored. It was possible thanks to the researches of students and professors from Politecnico di Torino and Tel Aviv University. They found that the stem electrical impedance is a relevant quantity related to watering events and the environmental values. The idea is then to find a way to predict the plant health status by exploiting these measurements. The machine learning algorithms are exploited, in particular neural networks and SVM. The dataset provided by the MiNES group has led to excellent results for the neural networks. Outstanding performance is also obtained for predictions of future data. It is possible to increase further the accuracy found in section 3.4 of 10% by tuning the learning rate of the Adam optimizer from 0.0001 to 0.001.

Hopefully, the neural network will be part of an autonomous monitoring system of the plants in the future. Therefore, the predictions may be used to set the optimal conditions for the plants automatically. It is beneficial to develop a prediction system that can be used for any tobacco plant. Since, for now, the dataset is quite limited, this situation was emulated by training a model with three plants and testing the trained model on the fourth one. Unfortunately, the performance reached is abysmal, mainly due to the different impedance ranges of the plants. Some techniques, such as computing the difference, fitting the data with a polynomial function, using more samples, were exploited to solve this problem and improve the accuracy, but it was unsuccessful.

In addition, in this thesis work, the relevance of impedance and soil moisture for predicting the tobacco plants' health status is demonstrated by the neural network implementations and then confirmed by SVM models. Despite the significant results, the neural network is still not ready to be used on unknown tobacco plants.

More works should be done, and other solutions to the different impedance ranges problem should be found. A suitable solution is to collect more data both from a broader period and from more plants. Another solution could be introducing more features regarding the plants, such as age or stem radius.

A thing to be improved is the labeling method used for the plants. For now, the plants are classified based on their leaves conditions, in particular their colors. However, the drying of the leaves and their growth are prolonged processes. Therefore it is pretty tricky to decide the boundary between healthy and not-healthy status.

Nevertheless, all in all, this work is a slight step toward a low-power, straightforward, economical, smart system. It is hoped that it could be a starting point for future studies and applications.

Chapter 5

Manual of the scripts

5.1 What the code does

Several operations can be done with the Python scripts implemented for this thesis:

1. Add to the *.csv* files other features, such as impedance difference, impedance after fitting, impedance difference after fitting;
2. Train a neural network having the data of the plants;
3. Predictions using the model trained before with a new dataset sampled in a different period;
4. Training with three plants and testing on the forth;
5. Train an SVM model;
6. Training with more features shifted in time;
7. Future prediction with more features shifted in time;
8. Training with three plants and testing of the fourth with features shifted in time;

5.2 Add other features to the *.csv* files

When the plants' data are exported for the first time using the script *main.py*, four *.csv* format files are obtained, one file for each plant. Each file contains seven columns corresponding to seven values: temperature, air humidity, ambient light, soil moisture, time, impedance module, and impedance phase. It is possible to do some data transformation, such as compute the difference between two adjacent impedance values, fit the impedance with a polynomial function, compute the difference between two adjacent impedance values after the fitting. Three Python script in the folder **Neural_network_training** are exploited: *diff_considering.py*, *curve_fitting.py* and *diff_considering_fitting.py*. These three scripts should be run in order.

- *diff_considering.py*: to run this script, the input file names and the output file names should be decided;
- *curve_fitting.py*: the input file names are the output file names of the previous script, output file names, and the grade of the fitting polynomial function should be decided. Suggested value: grade = 49;
- *diff_considering_fitting.py*: the input files names should correspond to the output file names of the previous script. Then, the output file names should be defined.

After running these scripts, four new *.csv* files are obtained, one for each plant. Each file now contains fourteen columns: seven columns for the initial features; two for impedance module and phase difference; two for the fitted impedance module and phase, and two for their differences; a first column indicating the number of samples, this column will be eliminated before the training since it is not helpful.

5.3 Train a neural network having the data of the plants

With the four data files obtained as described in the previous section, now, it is possible to train neural networks. With the intention to do that, the script *nn_training.py* in the folder **Neural_network_training** can be used. This script can train neural networks changing the features used during the training phase, the structure of the neural networks, such as the number of layers and nodes, and the algorithm hyperparameters. The parameters are well explained in table 3.1, to summarize:

- the first two parameters to be set are set_by_hand and saturation_moisture. Both should be set to one, because with set_by_hand = 0 a wrong labeling is done for the data, and for saturation_moisture = 1, more samples are considered.
- the parameters temperature, airhumidity, ambientlight, moisture, date_time, impedance_module, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit decide which are

the features to be used for training the neural network. Their values can be 0 or 1, 1 indicates that they will be used for the training and 0 the contrary;

- after that the model hyperparameters can be set, such as the number of layers and the number of nodes in each layer. Suggested values: number of hidden layers = 2; first hidden layer with six nodes with 'relu' activation function; second hidden layer with six nodes with 'relu' activation function; output layer with two nodes with 'softmax' activation function, since two class are present;

Once the model has been defined, the neural network can be trained, other parameters called algorithm hyperparameters should be defined. Suggested values:

- optimizer = Adam with learning rate from 0.0005 to 0.001;
- loss = 'sparse_categorical_crossentropy';
- metrics = ['accuracy'];
- number of epochs around 300 with the learning rate suggested above. A typical value for batch_size is 10.

After that the model is trained, it is saved as a *.h5* format file, and the performance are saved in a file in the path **'../output_files/output_neuralnetwork.txt'**. It is possible to plot the accuracy over epochs, a 3-D graph with right and wrong prediction of the training and testing sets, and the confusion matrix by setting plot_accuracy, plot_train_predictions, plot_test_predictions and display_cm respectively.

Example of application Suppose that the neural network is wanted to be trained with temperature, air humidity, soil moisture, time, light intensity, and module and phase after fitting, the parameters are set as follows:

Listing 5.1: Parameters setting

```
 # *****Parameters to decide the input files*****
set_by_hand = 1 # First parameter to decide if the plants used is
    with exported data or with status set by hand

saturation_moisture = 1 # If this parameter is equal to zero the
    moisture <-200kPa will be
# filtered out else it will be saturated to -200kPa

# *****Decide which features should be used in the training phase
    *****
temperature = 1
airhumidity = 1
ambientlight = 1
```

```
moisture = 1
date_time = 1
impedance_module = 0
impedance_phase = 0
module_difference = 0
phase_difference = 0
module_fit = 1
phase_fit = 1
module_diff_fit= 0
phase_diff_fit = 0
```

Then, the input filenames are obtained, the parameter set_by_hand choose which group of files is imported.

Listing 5.2: Filenames of the data

```
# Import of the correct files names
if set_by_hand == 0:
    filename_plant1 = '../data_of_the_plants/data_export_plant1.csv'
    filename_plant2 = '../data_of_the_plants/data_export_plant2.csv'
    filename_plant3 = '../data_of_the_plants/data_export_plant3.csv'
    filename_plant4 = '../data_of_the_plants/data_export_plant4.csv'
else:
    filename_plant1 = '../data_of_the_plants/
   data_plant1_03_05_diff_withfit.csv'
    filename_plant2 = '../data_of_the_plants/
   data_plant2_03_05_diff_withfit.csv'
    filename_plant3 = '../data_of_the_plants/
   data_plant3_03_05_diff_withfit.csv'
    filename_plant4 = '../data_of_the_plants/
   data_plant4_03_05_diff_withfit.csv'
```

Once the filenames are obtained, the data are imported, the correct features are selected, and the data is converted to a Numpy array. After that, the samples are shuffled, normalized, and divided into test and training sets. Then, the model is created. It is possible to define the number of hidden neurons or add new hidden layers. In this example, two hidden layers, each with 6 neurons, are used. The activation function chosen for hidden layers is the relu function and softmax for the output layer.

Listing 5.3: Creation of the model

```
nn_layers = [6, 6, 2]
model = Sequential([
    Dense(units = nn_layers[0], input_shape=(feature_size,),
   activation='relu'), #this represents the second layer
    Dense(units = nn_layers[1], activation = 'relu'),
    Dense(units = nn_layers[2], activation = 'softmax')
])
```

Now, the model is trained with the training samples and training labels obtained before, the following hyperparameters are used:

Listing 5.4: Training of the model

```python
## Training of the neural network
epochs_num = 300
model.compile(optimizer=Adam(learning_rate = 0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x=plant_samples_train, y=plant_labels_train, validation_split=0.1, batch_size=10, epochs=epochs_num, shuffle=True, verbose=2)
```

The filename where the model will be saved is obtained automatically with the parameters set before:

Listing 5.5: Saving of the model

```python
filename_neural_network = '../Neural_network_trained/plant_'+ str(set_by_hand) + str(saturation_moisture) + '_' + str(temperature) + str(airhumidity) + str(ambientlight) + str(moisture) + str(date_time) + str(impedance_module) + str(impedance_phase) + str(module_difference) + str(phase_difference) + str(module_fit) + str(phase_fit) + str(module_diff_fit) + str(phase_diff_fit) + '_' + str(feature_size) + str(nn_layers[0]) + str(nn_layers[1]) + str(nn_layers[2]) + '.h5'

## Saving the trained neural network
save_new_model = 1 #If this parameter is 1, the trained model will overwrite the previously trained model. Otherwise, it will be saved only if the filename is not present yet.
if save_new_model == 1:
    model.save(filename_neural_network)
else:
    if os.path.isfile(filename_neural_network) is False:
        model.save(filename_neural_network)
```

The training and testing results are then saved in a *.txt* file:

Listing 5.6: Saving of the results

```python
# Name of the .txt file where the results will be saved
filename_output = '../output_files/output_neuralnetwork.txt'
if os.path.isfile(filename_output) is False:
    with open(filename_output, 'w') as f:
        f.write('The network is saved as: ' + filename_neural_network+'\n')
        f.write('The overall accuracy of this network is: ' + str(accuracy) + '\n')
        f.write('The train accuracy is: ' + str(accuracy_train) + '\n')
        f.write('The test accuracy is: ' + str(accuracy_test) + '\n')
        f.write('The overall confusion matrix is' + str(cm) +'\n')
        f.write('The train confusion matrix is' + str(cm_train) +'\n')
        f.write('The test confusion matrix is' + str(cm_test) +'\n')
        f.write('****************************************************** \n')
else:
    with open(filename_output, 'a') as f:
        f.write('The network is saved as: ' + filename_neural_network +'\n')
        f.write('The overall accuracy of this network is: ' + str(accuracy) + '\n')
        f.write('The train accuracy is: ' + str(accuracy_train) + '\n')
        f.write('The test accuracy is: ' + str(accuracy_test) + '\n')
        f.write('The overall confusion matrix is ' + str(cm) +'\n')
        f.write('The train confusion matrix is ' + str(cm_train) +'\n')
        f.write('The test confusion matrix is ' + str(cm_test) +'\n')
        f.write('****************************************************** \n')
```

Three plots are generated if the corresponding parameters are set to one, and inside the *.txt* file, the following results are saved:

```
The network is saved as:
../Neural_network_trained/plant_11_1111100001100_7662.h5
The overall accuracy of this network is: 0.6994906621392191
The train accuracy is: 0.7021577644145738
The test accuracy is: 0.6888260254596889
The overall confusion matrix is [[1566  418]
 [ 644  906]]
The train confusion matrix is [[1267  330]
 [ 512  718]]
The test confusion matrix is [[299  88]
 [132 188]]
```

5.4 Predictions using the model trained before with a new dataset

It is possible to test the model trained and saved in the previous section with a new data set. In particular, the data of the same plants but in a different future period is used. In order to do that the Python script future_predictions.py in the folder **Neural_network_training** is used. The model is imported by setting the usual parameters, set_by_hand, saturation_moisture, temperature, etc. With these parameters, the model's name is obtained. After that, the new file names are needed. However, the data used for training the model is also needed because the data should be properly normalized. With all this information, the testing is done, and the results are saved in the path **'../output_files/output_future_prediction.txt'**.

Example of application The model trained in the previous section is used now to predict the status of future data. The features' parameters and the model hyperparameters are used to generate the corresponding model name:

Listing 5.7: Parameters setting

```
# *****Parameters to decide the input files*****
set_by_hand = 1
saturation_moisture = 1
# *****Decide which features should be used in the training phase
    *****
temperature = 1
airhumidity = 1
ambientlight = 1
moisture = 1
date_time = 1
impedance_module = 0
impedance_phase = 0
module_difference = 1
phase_difference = 1
module_fit = 0
phase_fit = 0
module_diff_fit = 0
phase_diff_fit = 0
feature_size = temperature + airhumidity + ambientlight + moisture +
    date_time + impedance_module + impedance_phase + module_difference
     + phase_difference + module_fit + phase_fit + module_diff_fit +
    phase_diff_fit

nn_layers = [6, 6, 2]
```

Two groups of data have to be imported. The first group regards the old data, namely those used for the training, and the second one is the data to be predicted:

Listing 5.8: Data filenames

```
filename_plant1_old = "../data_of_the_plants/
    data_plant1_03_05_diff_withfit.csv"
filename_plant2_old = "../data_of_the_plants/
    data_plant2_03_05_diff_withfit.csv"
filename_plant3_old = "../data_of_the_plants/
    data_plant3_03_05_diff_withfit.csv"
filename_plant4_old = "../data_of_the_plants/
    data_plant4_03_05_diff_withfit.csv"
```

```

filename_plant1 = "../data_of_the_plants/
    data_plant1_05_06_diff_withfit.csv"
filename_plant2 = "../data_of_the_plants/
    data_plant2_05_06_diff_withfit.csv"
filename_plant3 = "../data_of_the_plants/
    data_plant3_05_06_diff_withfit.csv"
filename_plant4 = "../data_of_the_plants/
    data_plant4_05_06_diff_withfit.csv"
```

Once all is ready, the accuracy is computed and saved in a *.txt* file:

```
The following accuracy are computed with the neural network trained with the
name../Neural_network_trained/plant_11_1111100110000_7662.h5
on the data sampled in the period 06/05/2021 to 07/06/2021
The accuracy of this network on plant 1 in the month of May is: 0.98996
The confusion matrix is[[493   5]
 [  0   0]]
The accuracy of this network on plant 2 in the month of May is: 0.09237
The confusion matrix is[[  0   0]
 [452  46]]
The accuracy of this network on plant 3 in the month of May is: 0.16230
The confusion matrix is[[  0   0]
 [160  31]]
The accuracy of this network on plant 4 in the month of May is: 0.86762
The confusion matrix is[[426  65]
 [  0   0]]
```

5.5 Training with three plants and testing on the fourth;

An interesting to do is to test a model on a plant that it has never seen. In order to do that, three plants are used as the training set, and the fourth one is used for the testing. For this step two scripts in the folder **Machine learning** are exploited: *three_plants_train.py* and *three_plants_accuracy.py*. The former is used to train the models, and the latter to test them.

For the script *three_plants_train.py* the input file names are needed. The training uses all seven features: temperature, air humidity, ambient light, soil moisture, time, impedance module, and impedance phase. Three plants out of four are used for the training. Therefore four different models can be trained. All the four neural networks are saved in the folder **Network_network_trained**.

After the training, the script *three_plants_accuracy.py* is used to test the models. The input file names and the model names are needed. After that, the testing is done, and the results are saved in the file **output_threeplants.txt** in the folder **Neural_network_trained**. This is a first version for this type of implementation; an upgraded version is described in section 5.9.

5.6 Train a SVM model

With the script *svm_train.py* in the folder **SVM_training**, it is possible to implement SVM models. The input data file names are needed, and after that the usual parameters are set:

- the first two parameters to be set are set_by_hand and saturation_moisture. Both should be set to one, because with set_by_hand = 0 a wrong labeling is done for the data, and for saturation_moisture = 1, more samples are considered.
- the parameters temperature, airhumidity, ambientlight, moisture, date_time, impedance_module, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit decide which are the features to be used for training the neural network. Their values can be 0 or 1, 1 indicates that they will be used for the training and 0 the contrary;

It is possible to choose the kernel for the SVM to be implemented; only linear and RBF kernels are present in this script. However, it is possible to add other kernels inside the if statement. The available kernels are described in [35]. The hyperparameters such as C and gamma for the RBF kernel are left to standard values. Once the model is trained, it is saved in the folder **SVM_trained**, and the performance is saved in the txt file *output_svm.txt* in the folder **output_files**.

5.7 Training with more features shifted in time

The idea is to exploit more impedance data as features, so the N-th impedance sampled and the one sampled one hour before (sample N-1), two hours before (N-2), and so on. The Python script *nn_training_intime.py* in the folder **Neural_network_training_intime** is exploited. Since the measurement is exploited more times, the parameters that decide if the corresponding measurement will be used for the training can assume any positive integer value. As usual, the parameters set_by_hand and saturation_moisture are set to one. In this case, also the environmental parameters are set to one. Nevertheless, it is possible to change these values to any integer number by adding in the code a part that calls the custom function `add_feature_intime`. The parameters that can be changed in order to see their effect to the performance on the neural networks are impedance_module, impedance_phase, module_difference, phase_difference, module_fit, phase_fit, module_diff_fit and phase_diff_fit.

Before creating the model, it is possible to change the number of layers and nodes by changing the values of nn_layers[..]. Finally, the trained model is saved in the folder **Neural_network_trained_intime** and the results in the file *output_neuralnetwork_intime* in the usual folder **outputfiles**.

At the end, the accuracy over epochs, a 3-D graph with right and wrong prediction of the training and testing sets, and the confusion matrix are plotted.

Example of application The network is wanted to be trained with the environmetal parameters and 24 samples of impedance module and phase. Therefore the parameters should be set as follows:

Listing 5.9: Parameters setting

```
# *****Parameters to decide the input files*****
set_by_hand = 1 # First parameter to decide if the plants used is
    with exported data or with status set by hand
saturation_moisture = 1 # If this parameter is equal to zero the
    moisture <-200kPa will be filtered out else it will be saturated
    to -200kPa

# *****Decide which features and how many times they will be used in
    the training phase*****
temperature = 1
airhumidity = 1
ambientlight = 1
moisture = 1
date_time = 1
impedance_module = 24
impedance_phase = 24
module_difference = 0
phase_difference = 0
module_fit = 0
phase_fit = 0
module_diff_fit= 0
phase_diff_fit = 0
```

After that, the data are imported and prepared adding the new features. Then, the model is created and trained:

Listing 5.10: Parameters setting

```
# Creation of the neural network
nn_layers =[25,15,2]
model = Sequential()
model.add(Dense(nn_layers[0], activation='relu', input_shape=(
    feature_size,)))
model.add(Dense(nn_layers[1],activation='relu'))
model.add(Dense(nn_layers[2], activation='softmax'))
model.compile(optimizer=Adam(learning_rate = 0.0005), loss='
    sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x=plant_samples_train, y=plant_labels_train,
    validation_split=0.1, batch_size=10, epochs=300, shuffle=True,
    verbose=2)
```

Finally, the results are saved in a *.txt* file:

```
The network has saved as:
../Neural_network_trained_intime/plant_11_11111_24_24_0_0_0_0_0_0_53_25_15_2.h5
The overall accuracy of this network is: 0.9869262056943637
The train accuracy is: 0.9880130766436614
The test accuracy is: 0.9825834542815675
The overall confusion matrix is[[1941   20]
 [  25 1456]]
The train confusion matrix is[[1545   15]
 [  18 1175]]
The test confusion matrix is[[396   5]
 [  7 281]]
```

5.8 Future prediction with more features shifted in time

A trained model can be used to test the data in a different period. The script *future_prediction.py* in the folder **Neural_network_training_intime** is used. The parameters are set to generate the model filename and add the features to the imported data. After the old and new data filenames are obtained, the data are prepared, and the model is imported. The predictions are saved in the file *output_future_prediction_intime.txt* in the folder **output_files**.

5.9 Training with three plants and testing of the fourth with features shifted in time

For this part, the script *Three_plants_training.py* is used. The data filenames are needed. The usual parameters are used to select which and how many features to be used for the training. Four models are saved in the folder **Neural_network_trained_intime** and the results are saved in a *.txt* file called *output_neuralnetwork_intime_threeplants.txt* in the folder **output_files**.

Example of application A neural network is wanted to be trained considering the environmental parameters, the impedance module and phase, and the module and phase difference. The environmental parameters are used once, while what regards the impedance is considered twelve times. The parameters should be set as follows:

Listing 5.11: Parameters setting

```
# *****Parameters to decide the input files*****
set_by_hand = 1 # First parameter to decide if the plants used is
    with exported data or with status set by hand
saturation_moisture = 1 # If this parameter is equal to zero the
    moisture <-200kPa will be
# filtered out else it will be saturated to -200kPa

# *****Decide which features and how many times they will be used in
    the training phase*****
temperature = 1
airhumidity = 1
ambientlight = 1
moisture = 1
date_time = 1
impedance_module = 12
impedance_phase = 12
module_difference = 12
phase_difference = 12
module_fit = 0
phase_fit = 0
module_diff_fit = 0
phase_diff_fit = 0
```

Then, the model and algorithm hyperparameters are defined, and the four neural networks are created and trained:

Listing 5.12: Parameters setting

```
# Creation of the neural network
nn_layers = [25, 15, 2]
#nn_layers = [6, 6, 2]
#First model, it will be trained with the plants 123
model123 = Sequential()
model123.add(Dense(nn_layers[0], activation='relu', input_shape=(
    feature_size,)))
model123.add(Dense(nn_layers[1],activation='relu'))
model123.add(Dense(nn_layers[2], activation='softmax'))

#First model, it will be trained with the plants 124
model124 = Sequential()
model124.add(Dense(nn_layers[0], activation='relu', input_shape=(
    feature_size,)))
model124.add(Dense(nn_layers[1],activation='relu'))
model124.add(Dense(nn_layers[2], activation='softmax'))

```

```python
#First model, it will be trained with the plants 134
model134 = Sequential()
model134.add(Dense(nn_layers[0], activation='relu', input_shape=(
    feature_size,)))
model134.add(Dense(nn_layers[1],activation='relu'))
model134.add(Dense(nn_layers[2], activation='softmax'))

#First model, it will be trained with the plants 234
model234 = Sequential()
model234.add(Dense(nn_layers[0], activation='relu', input_shape=(
    feature_size,)))
model234.add(Dense(nn_layers[1],activation='relu'))
model234.add(Dense(nn_layers[2], activation='softmax'))
```

Finally, the four models are tested and the results are saved in a *.txt* file:

```
The network saved as:
 ../Neural_network_trained_intime/plant123_12_12_12_12_0_0_0_0_53_25_15_2.h5
 is traing with plants 123
The train accuracy is: 0.9705769965609476
The train confusion matrix is[[1454    4]
 [  73 1086]]
The test is done on the plant 4:
The test accuracy is: 0.5372279495990836
The test confusion matrix is[[469  46]
 [358   0]]

The network saved as:
 ../Neural_network_trained_intime/plant124_12_12_12_12_0_0_0_0_53_25_15_2.h5
 is traing with plants 124
The train accuracy is: 0.9690721649484536
The train confusion matrix is[[1358   12]
 [  69 1180]]
The test is done on the plant 3:
The test accuracy is: 0.4006888633754305
The test confusion matrix is[[ 82 521]
 [  1 267]]

The network saved as:
 ../Neural_network_trained_intime/plant134_12_12_12_12_0_0_0_0_53_25_15_2.h5
 is traing with plants 134
The train accuracy is: 0.9782193351165457
The train confusion matrix is[[1946   27]
 [  30  614]]
The test is done on the plant 2:
The test accuracy is: 0.2268041237113402
The test confusion matrix is[[  0   0]
 [675 198]]
```

```
The network saved as:
 ../Neural_network_trained_intime/plant234_12_12_12_12_0_0_0_0_53_25_15_2.h5
 is traing with plants 234
The train accuracy is: 0.9973251815055407
The train confusion matrix is[[1114    4]
 [   3 1496]]
The test is done on the plant 4:
The test accuracy is: 0.38258877434135163
The test confusion matrix is[[334 521]
 [ 18   0]]
```

Bibliography

[1] United Nations Environment Programme (UNEP). *21 Issues for the 21st Century: Result of the UNEP Foresight Process on Emerging Environmental Issues.* UNEP, available here (cit. on p. 1).

[2] Christina Nunez. «Desertification, explained». In: (MAY 31, 2019). URL: `https://www.nationalgeographic.com/environment/article/desertification` (cit. on p. 1).

[3] Svetla Markova. *What is smart agriculture and why smart agriculture is the future?* `https://ondo.io/what_is_smart_agriculture/`. May 20th, 2020 (cit. on p. 2).

[4] Mohamed Rawidean Mohd Kassim. «IoT Applications in Smart Agriculture: Issues and Challenges». In: *2020 IEEE Conference on Open Systems (ICOS)*. 2020, pp. 19–24. DOI: `10.1109/ICOS50156.2020.9293672` (cit. on p. 2).

[5] U. Garlando, L. Bar-On, A. Avni, Y. Shacham-Diamand, and D. Demarchi. «Plants and Environmental Sensors for Smart Agriculture, an Overview». In: *2020 IEEE SENSORS*. 2020, pp. 1–4. DOI: `10.1109/SENSORS47125.2020.9278748` (cit. on p. 2).

[6] L. Bar-on, A. Jog, and Y. Shacham-Diamand. «Four Point Probe Electrical Spectroscopy Based System for Plant Monitoring». In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2019, pp. 1–5. DOI: `10.1109/ISCAS.2019.8702623` (cit. on pp. 3–5).

[7] Umberto Garlando, Lee Bar-On, Paolo Motto Ros, Alessandro Sanginario, Stefano Calvo, Maurizio Martina, Adi Avni, Yosi Shacham-Diamand, and Danilo Demarchi. «Analysis of In Vivo Plant Stem Impedance Variations in Relation with External Conditions Daily Cycle». In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: `10.1109/ISCAS51556.2021.9401242` (cit. on pp. 3, 4, 8, 11, 12, 16, 17, 64, 72).

[8] U. Garlando, L. Bar-On, P. M. Ros, A. Sanginario, S. Peradotto, Y. Shacham-Diamand, A. Avni, M. Martina, and D. Demarchi. «Towards Optimal Green Plant Irrigation: Watering and Body Electrical Impedance». In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9181290 (cit. on pp. 3, 4, 7–11, 25).

[9] Lee Bar-On, Sebastian Peradotto, Alessandro Sanginario, Paolo Motto Ros, Yosi Shacham-Diamand, and Danilo Demarchi. «In-Vivo Monitoring for Electrical Expression of Plant Living Parameters by an Impedance Lab System». In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 178–180. DOI: 10.1109/ICECS46596.2019.8964804 (cit. on pp. 3, 4, 6, 7).

[10] *HDC2080 Low-Power Humidity and Temperature Digital Sensor data sheet.* Available here. Texas Instruments, May 2018, revised July 2021 (cit. on pp. 5, 14).

[11] *MAX44009 Industry's Lowest Power Ambient Light Sensor with ADC data sheet.* Available here. Maxim Integrated (cit. on p. 5).

[12] *WATERMARK Soil Moisture Sensor - Model 200SS.* Available here. Irrometer (cit. on pp. 5, 14).

[13] *OPT3001 Ambient Light Sensor (ALS).* Available here. Texas Instruments (cit. on p. 14).

[14] Jason Brownlee. «A Tour of Machine Learning Algorithms». In: (August 12, 2019). URL: https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/ (cit. on p. 19).

[15] «Types of Machine Learning Algorithms You Should Know». In: () (cit. on p. 19).

[16] Rares Ivan. «What is Machine Learning?» In: (June 30, 2017). URL: https://www.cognizantsoftvision.com/blog/what-is-machine-learning/ (cit. on p. 19).

[17] IBM Cloud Education. «Machine Learning». In: (15 July 2020). URL: https://www.ibm.com/cloud/learn/machine-learning#toc-machine-le-K7VszOk6 (cit. on p. 19).

[18] Konstantinos G. Liakos, Patrizia Busato, Dimitrios Moshou, Simon Pearson, and Dionysis Bochtis. «Machine Learning in Agriculture: A Review». In: *Sensors* 18.8 (2018). ISSN: 1424-8220. DOI: 10.3390/s18082674. URL: https://www.mdpi.com/1424-8220/18/8/2674 (cit. on pp. 21, 33, 35).

[19] S.B. Imandoust and Mohammad Bolandraftar. «Application of K-nearest neighbor (KNN) approach for predicting economic events theoretical background». In: *Int J Eng Res Appl* 3 (Jan. 2013), pp. 605–610 (cit. on p. 22).

[20] A. Nemes, R.T. Roberts, W.J. Rawls, Ya.A. Pachepsky, and M.Th. van Genuchten. «Software to estimate 33 and 1500kPa soil water retention using the non-parametric k-Nearest Neighbor technique». In: *Environmental Modelling Software* 23.2 (2008), pp. 254–255. ISSN: 1364-8152. DOI: `https://doi.org/10.1016/j.envsoft.2007.05.018`. URL: `https://www.sciencedirect.com/science/article/pii/S1364815207001193` (cit. on p. 22).

[21] M Sarkar and T Y Leong. «Application of K-nearest neighbors algorithm on breast cancer diagnosis problem». In: *Proceedings. AMIA Symposium* (2000), pp. 759–63. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2243774/pdf/procamiasymp00003-0794.pdf` (cit. on p. 22).

[22] Rashmi Priya, Dharavath Ramesh, and Ekaansh Khosla. «Crop Prediction on the Region Belts of India: A Naïve Bayes MapReduce Precision Agricultural Model». In: *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2018, pp. 99–104. DOI: `10.1109/ICACCI.2018.8554948` (cit. on p. 23).

[23] Soumo Chatterjee. «Use Naive Bayes Algorithm for Categorical and Numerical data classification». In: (Nov 24, 2019). URL: `https://medium.com/analytics-vidhya/use-naive-bayes-algorithm-for-categorical-and-numerical-data-classification-935d90ab273f` (cit. on p. 24).

[24] Naim Rochmawati, Hanik Badriyah Hidayati, Yuni Yamasari, Wiyli Yustanti, Lusia Rakhmawati, Hapsari P. A. Tjahyaningtijas, and Yeni Anistyasari. «Covid Symptom Severity Using Decision Tree». In: *2020 Third International Conference on Vocational Education and Electrical Engineering (ICVEE)*. 2020, pp. 1–5. DOI: `10.1109/ICVEE50212.2020.9243246` (cit. on p. 26).

[25] Steven Salzberg, Rupali Chandar, Holland Ford, Sreerama K. Murthy, and Richard White. «Decision trees for automated identification of cosmic-ray hits in Hubble Space Telescope images». In: 107 (Mar. 1995), p. 279. DOI: `10.1086/133551`. URL: `https://doi.org/10.1086/133551` (cit. on p. 26).

[26] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2 (cit. on p. 35).

[27] Jason Brownlee. *How to Configure the Number of Layers and Nodes in a Neural Network*. `https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/`. July 27, 2018 (cit. on p. 37).

[28] Jeff Heaton. *The Number of Hidden Layers*. `https://www.heatonresearch.com/2017/06/01/hidden-layers.html`. June 01, 2017 (cit. on p. 37).

[29] *Layer activation functions*. Available here. Keras (cit. on p. 42).

[30] *Optimizers*. Available here. Keras (cit. on p. 42).

[31] *Adam*. Available here. Keras (cit. on p. 42).

[32] Lili Jiang. «A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam)». In: (Jun 7, 2020). URL: `https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c` (cit. on p. 43).

[33] *Losses*. Available here. Keras (cit. on p. 43).

[34] *Metrics*. Available here. Keras (cit. on p. 43).

[35] *sklearn.svm.SVC*. Available here. scikit-learn (cit. on pp. 71, 98).